Memory Consistency Models

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@cs.rice.edu

From Coherence to Consistency

- **Coherence**
  - —focus: visible values of an individual variable
  - —problems can arise if multiple actors (e.g., multiple cores) have access to multiple copies of a datum (e.g., in multiple caches) and at least one access is a write
    - – must appear to be one and only one value per memory location
  - —access to stale data (incoherence) is prevented using a <u>coherence protocol</u>
    - – set of rules implemented by the distributed actors within a system
- **Consistency models**
  - —focus: visible values for multiple variables
  - —define correct shared memory behavior in terms of loads and stores (memory reads and writes)
    - – independent of caches or coherence
  - —can stores be seen out of order? if so, under what conditions?
    - – a spectrum of alternatives
      - sequential consistency to weak memory models

Example: What Can a Programmer Expect?

```
Initially all pointers = null, all integers = 0.

P1

while (there are more tasks) {
  Task = GetFromFreeList();
  Task → Data = ...;
  insert Task in task queue
}
Head = head of task queue;
```

```
P2, P3, ..., Pn

while (MyTask == null) {
  Begin Critical Section
  if (Head != null) {
    MyTask = Head;
    Head = Head → Next;
  }
  End Critical Section
}
...   = MyTask → Data;
```

Memory Consistency Model

- Memory model
  - —formal specification of how shared memory will appear to programmers
- Consistency
  - —restricts values that can be returned by a read during execution
- Why memory consistency models? Eliminate gap between
  - —expected behavior
  - —behavior supported by a system

Impact of Memory Models

- **Programmability**
  - —programmers must reason about allowable behaviors
    - surprisingly subtle!
- **Performance**
  - —determines what reorderings of loads and stores are legal
    - hardware
    - compiler
- **Portability**
  - —different systems implement different memory models

Multiple Levels of Memory Models

- **Machine level**
  - —affects hardware design (processor, memory, interconnect)
  - —affects assembly-code programmer
- **Language level**
  - —affects both designers and users of high-level languages

Memory Models for Uniprocessors

- **Memory operations**
  - —occur one at a time
  - —in order specified by program (program order)
- **Simple, intuitive sequential semantics for memory**
- **Expectation**
  - —read of X will return value of last write (in program order) to X

In practice: a uniprocessor can relax strict ordering
- suffices to maintain control and data dependences
- order constrained only when
  - same location
  - one controls execution of other

Why Relax Strict Ordering?

Overlapping and reordering memory accesses enables a range of hardware and software optimizations

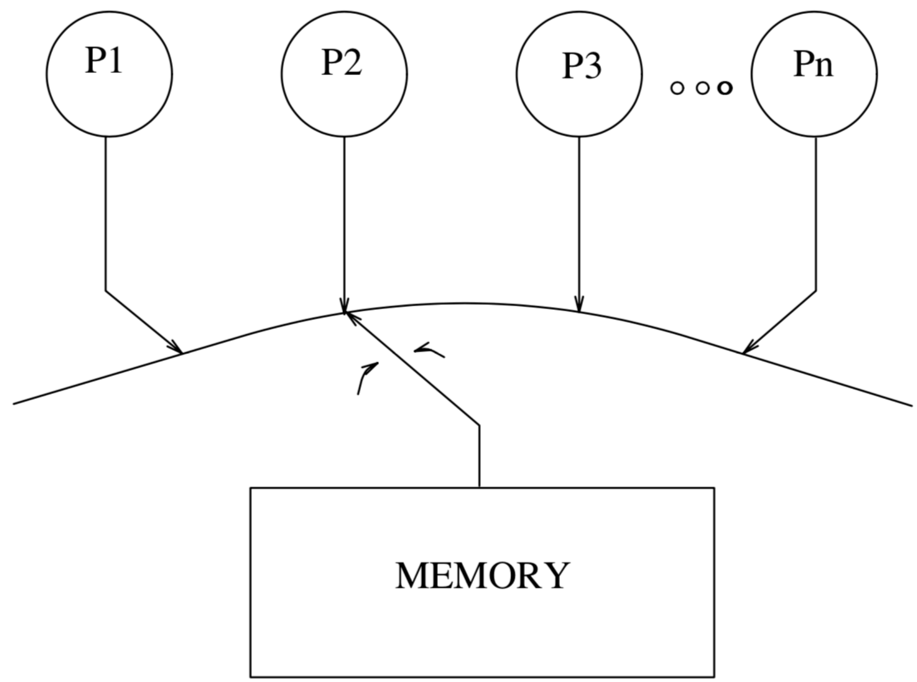
- **Compiler optimizations**
  - —register allocation
  - —code motion
  - —loop transformations
- **Hardware optimizations**
  - —pipelining
  - —multiple issue
  - —write buffer bypassing
  - —forwarding a value from one cache to another
  - —lockup-free caches: don’t delay accesses that follow a miss

A Memory Model for Multiprocessors?

- Intuitively, a read of a memory location should return the value of its “last” write
- Natural for uniprocessors
- Not obvious what this means for multiprocessors with concurrent operations
- Idea: require that all memory operations appear to execute one at a time, and the operations of a single processor appear to execute in the order described by that processor’s program

Sequential Consistency

- **Intuitive memory model defined by Lamport [1979]**
- **Result of an execution appears as if**
  - **all operations appear as if executed in some sequential order**
  - **memory operations of each thread appear in program order**



simple memory system:
no caches, no write buffers

Consider the Following …

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

After all statements execute, could r2 == 2 and r1 == 1?

Consider the Following …

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

After all statements execute, could r2 == 2 and r1 == 1?

Possible interleavings

Consider the Following …

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

After all statements execute, could r2 == 2 and r1 == 1?

Possible interleavings

– 1, 2, 3, 4 ?

Consider the Following …

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

After all statements execute, could r2 == 2 and r1 == 1?

Possible interleavings

– 1, 2, 3, 4 ? ✓

Consider the Following …

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

After all statements execute, could r2 == 2 and r1 == 1?

Possible interleavings

- 1, 2, 3, 4 ? ✓
- 1, 3, 2, 4 ?

Consider the Following …

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

After all statements execute, could r2 == 2 and r1 == 1?

Possible interleavings

- 1, 2, 3, 4 ? ✓
- 1, 3, 2, 4 ? ✓

Consider the Following …

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

After all statements execute, could r2 == 2 and r1 == 1?

Possible interleavings

- 1, 2, 3, 4 ? ✓
- 1, 3, 2, 4 ? ✓
- 3, 4, 1, 2 ?

Consider the Following …

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

After all statements execute, could r2 == 2 and r1 == 1?

Possible interleavings

- 1, 2, 3, 4 ? ✓
- 1, 3, 2, 4 ? ✓
- 3, 4, 1, 2 ? ✓

Consider the Following …

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

After all statements execute, could r2 == 2 and r1 == 1?

Possible interleavings

– 1, 2, 3, 4 ? ✓
– 1, 3, 2, 4 ? ✓
– 3, 4, 1, 2 ? ✓
– 4, 1, 2, 3 ?

Consider the Following …

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

After all statements execute, could r2 == 2 and r1 == 1?

Possible interleavings

- 1, 2, 3, 4 ? ✓
- 1, 3, 2, 4 ? ✓
- 3, 4, 1, 2 ? ✓
- 4, 1, 2, 3 ? Sequential consistency would not allow this

Does Program Order Really Matter?

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Both threads could enter critical section

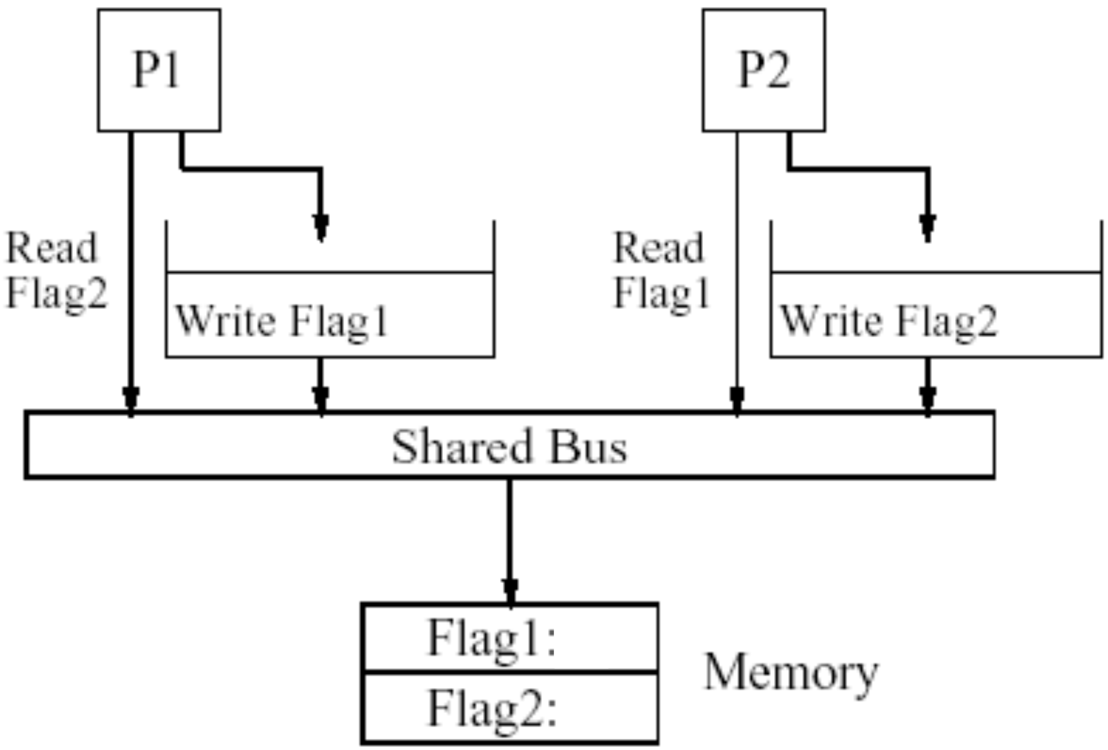
- **— if the hardware allows a thread's read to complete before a prior write completes**
- **— if the compiler reorders the thread's read and write**

Implications of Sequential Consistency

- Assumption: memory atomicity
  - —memory operations cannot overlap
- Impact
  - —limits aggressive hardware designs
  - —limits compiler optimizations
- Result: severely hampers performance

Write Buffers (without Caches)

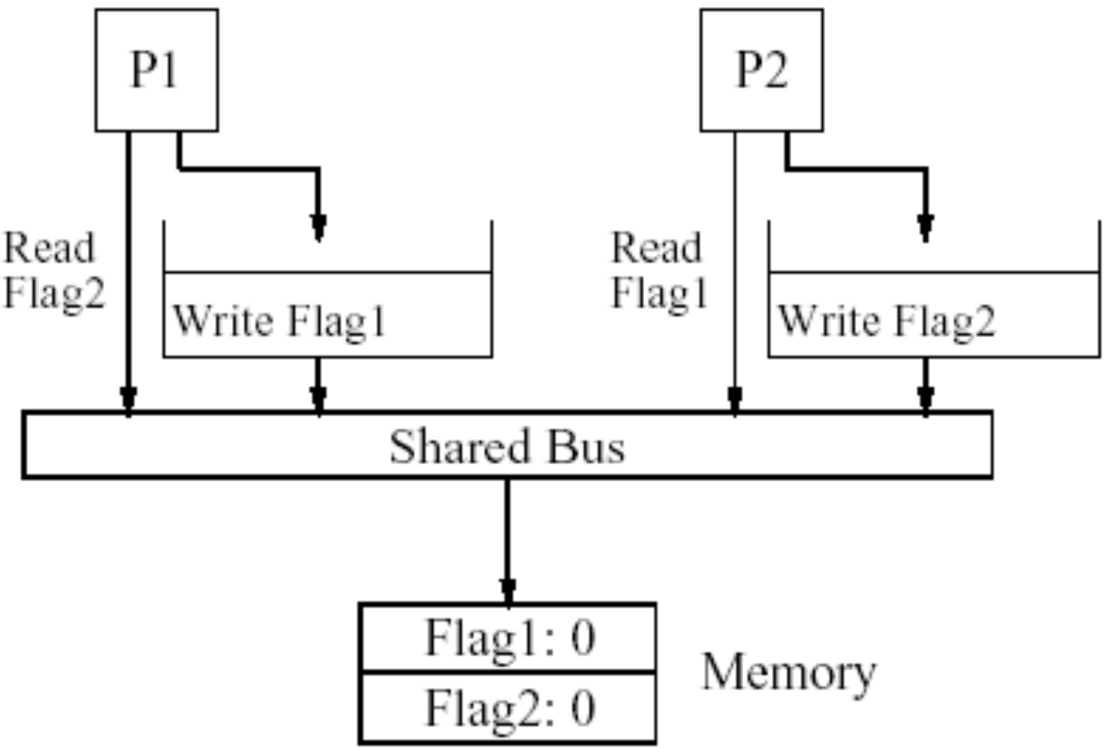
1. W=>R order using <u>write buffers</u>
   - —write buffer with bypassing hides latency of writes
   - —reads of different locations can bypass pending writes



Write Buffers (without Caches)

1. W=>R order using <u>write buffers</u>

- —write buffer with bypassing hides latency of writes
- —reads of different locations can bypass pending writes



Initially: Flag1=Flag2=0.

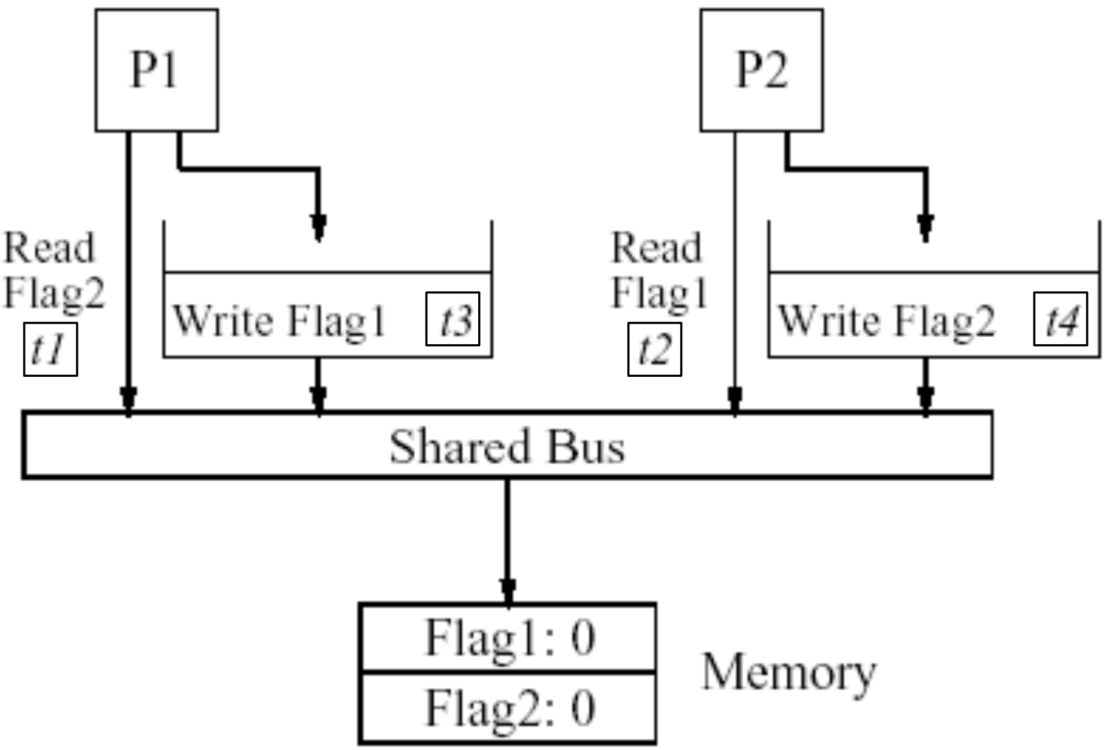
Can write buffers allow an ordering that violates sequential consistency?

<u>P1</u>	<u>P2</u>
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Write Buffers (without Caches)

1. W=>R order using <u>write buffers</u>

- **—write buffer with bypassing hides latency of writes**
- **—reads of different locations can bypass pending writes**



t: *completion* order

Initially: Flag1=Flag2=0.

Can write buffers allow an ordering that violates sequential consistency?

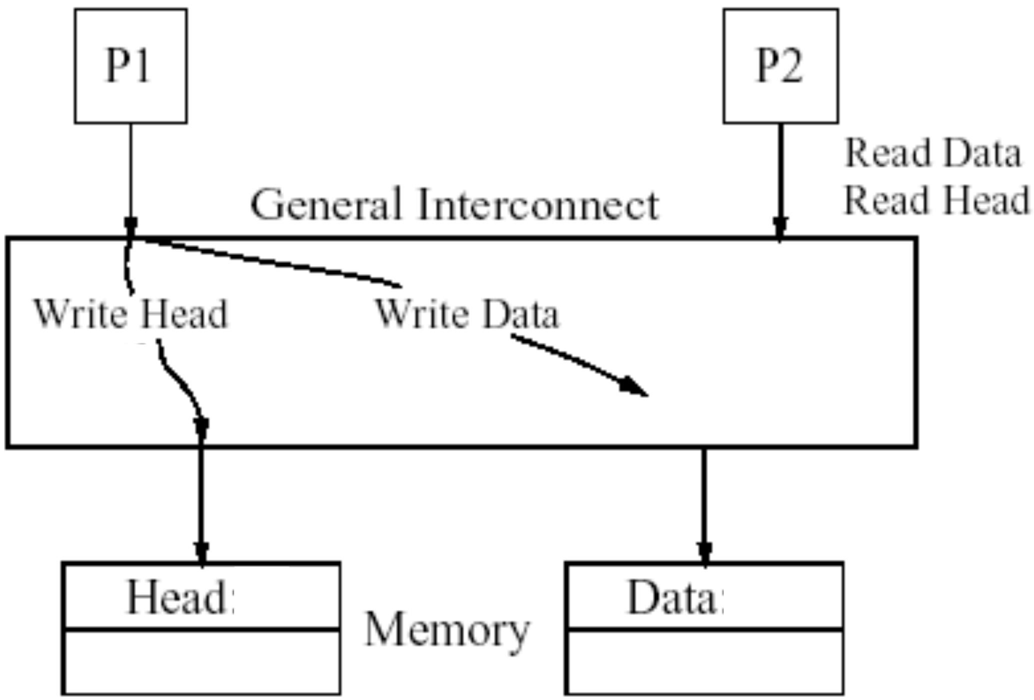
Yes: 1, 2, 3, 4

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Overlapping Writes (without Caches)

2. W=>W order using <u>overlapping writes</u>

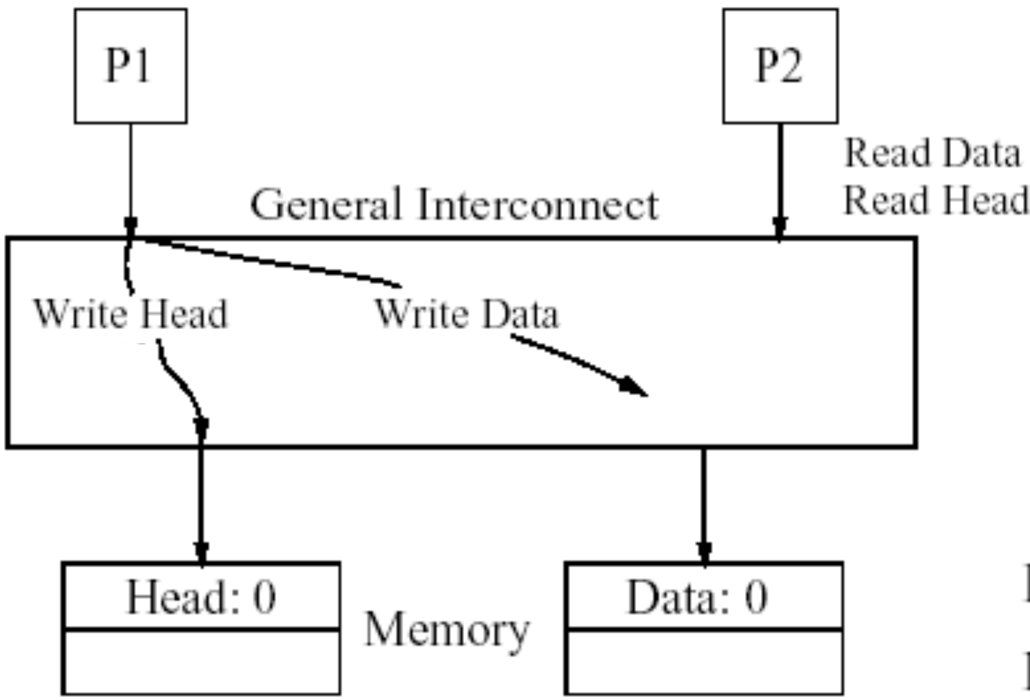
- —general interconnect vs. bus (memory parallelism)
- —writes to different memory locations issued by same processor handled by different memory modules



Overlapping Writes (without Caches)

2. W=>W order using <u>overlapping writes</u>

- —general interconnect vs. bus (memory parallelism)
- —multiple writes to different locations issued by same processor handled by different memory modules



Initially: Data=Head=0

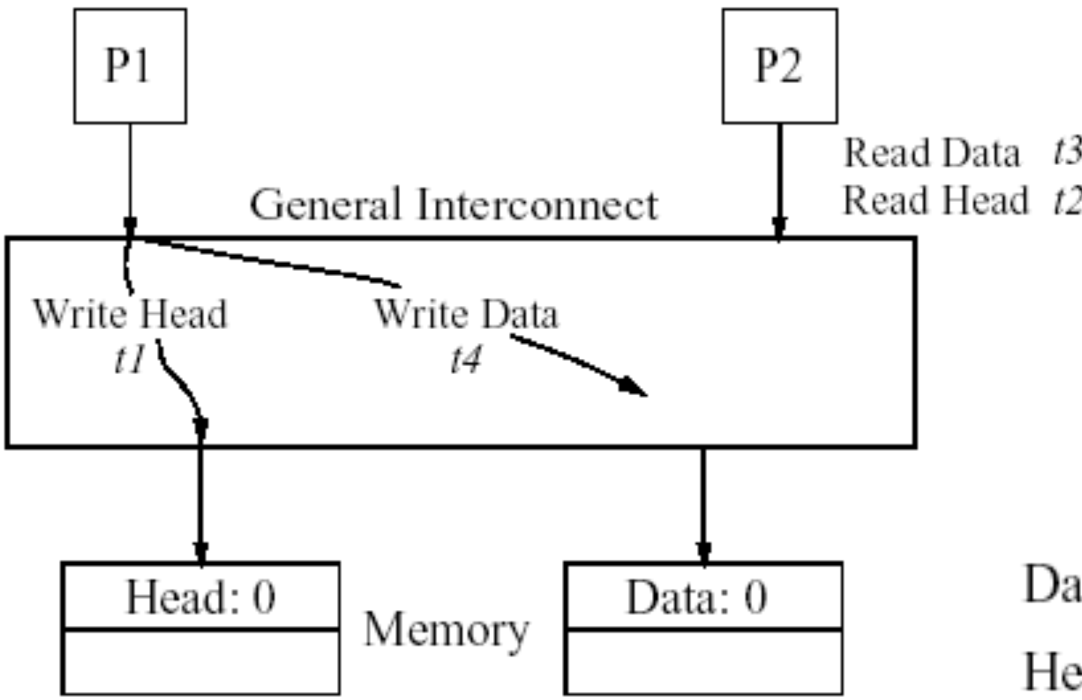
Can overlapping writes violate sequential consistency?

P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Overlapping Writes (without Caches)

2. W=>W order using <u>overlapping writes</u>

- **—general interconnect vs. bus (memory parallelism)**
- **—multiple writes to different locations issued by same processor handled by different memory modules**



t: *completion* order

Initially: Data=Head=0

Can overlapping writes violate sequential consistency?

Yes: 1, 2, 3, 4

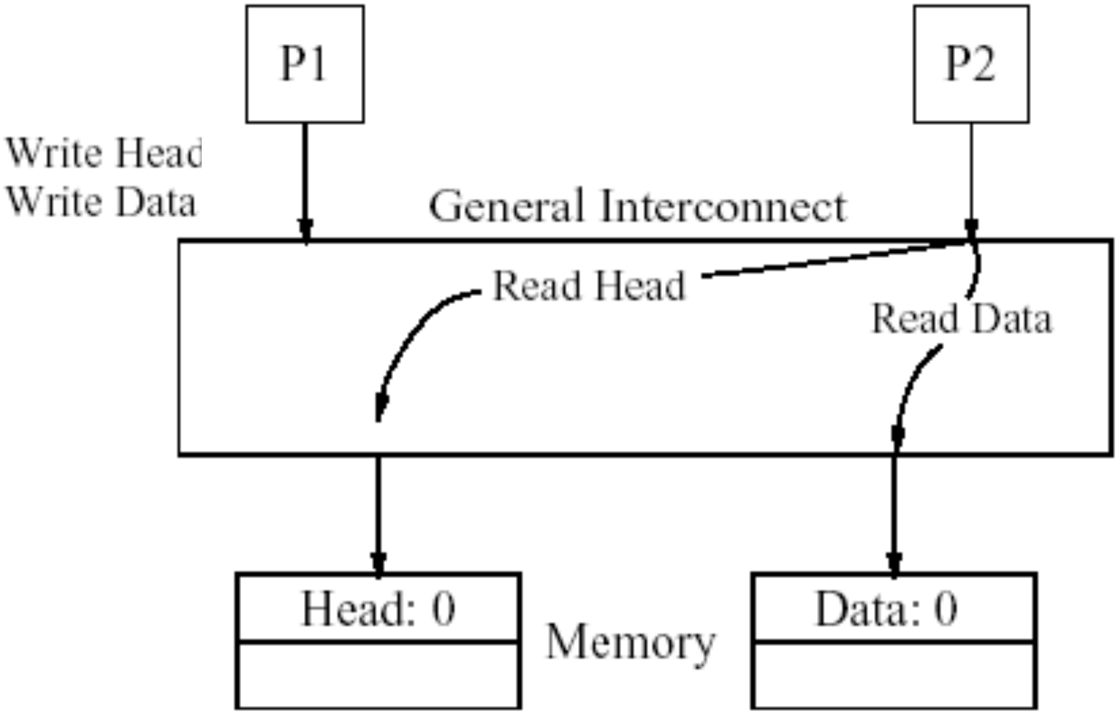
P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Non-blocking Reads (without Caches)

3. R=>R|W order using <u>non-blocking reads</u>

—non-blocking reads + general memory interconnect

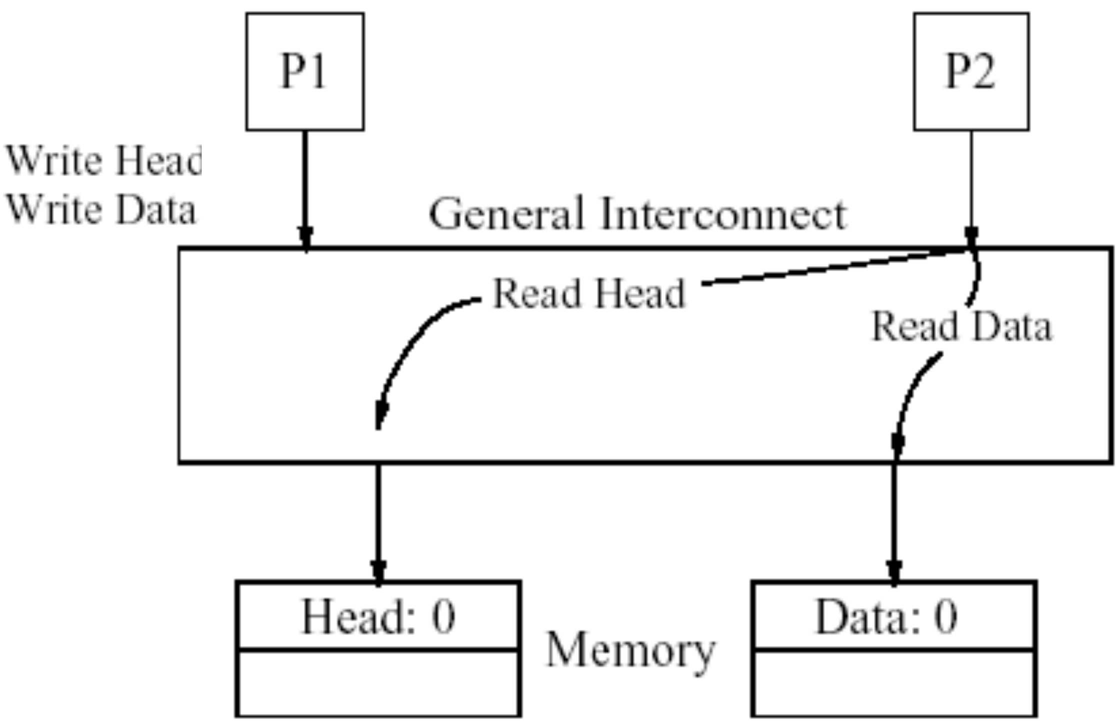
—non-blocking caches, dynamic scheduling, speculative execution



Non-blocking Reads (without Caches)

3. R=>R|W order using <u>non-blocking reads</u>

- —non-blocking reads + general memory interconnect
- —non-blocking caches, dynamic scheduling, speculative execution



Initially: Data=Head=0

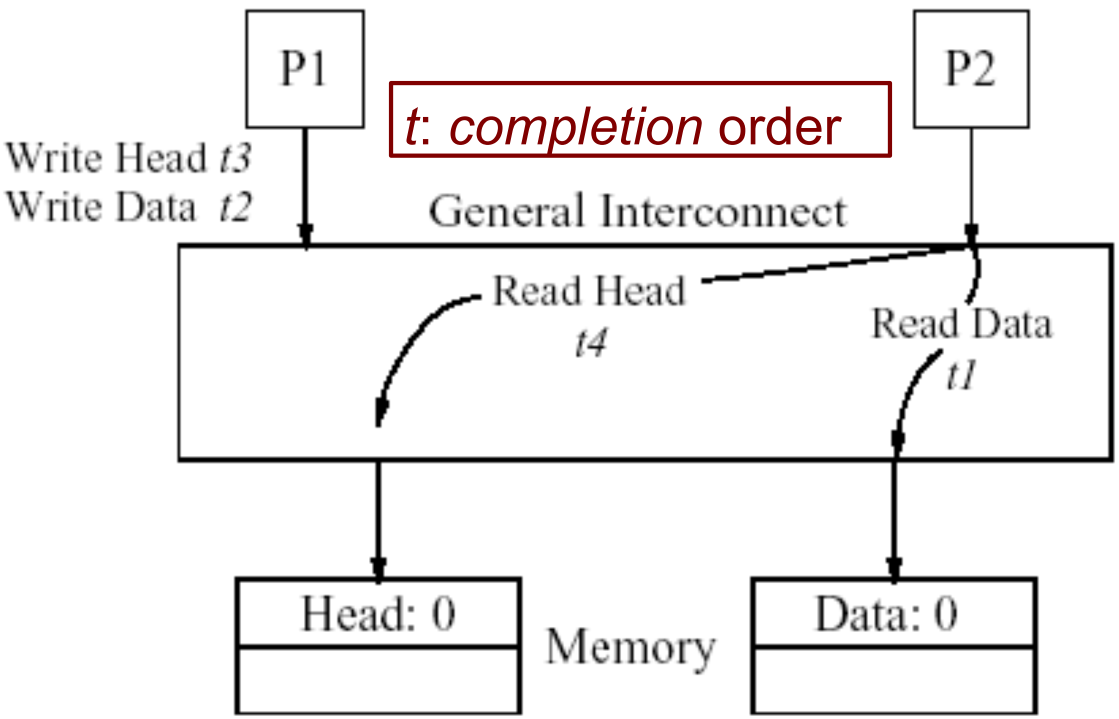
Can non-blocking reads violate sequential consistency?

P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Non-blocking Reads (without Caches)

3. R=>R|W order using <u>non-blocking reads</u>

- **—non-blocking reads (+ same memory interconnect)**
- **—non-blocking caches, dynamic scheduling, speculative execution**



Initially: Data=Head=0

Can non-blocking reads violate sequential consistency?

Yes: [spec] 1, 2, 3, 4

P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Hardware Optimization Effects Summary

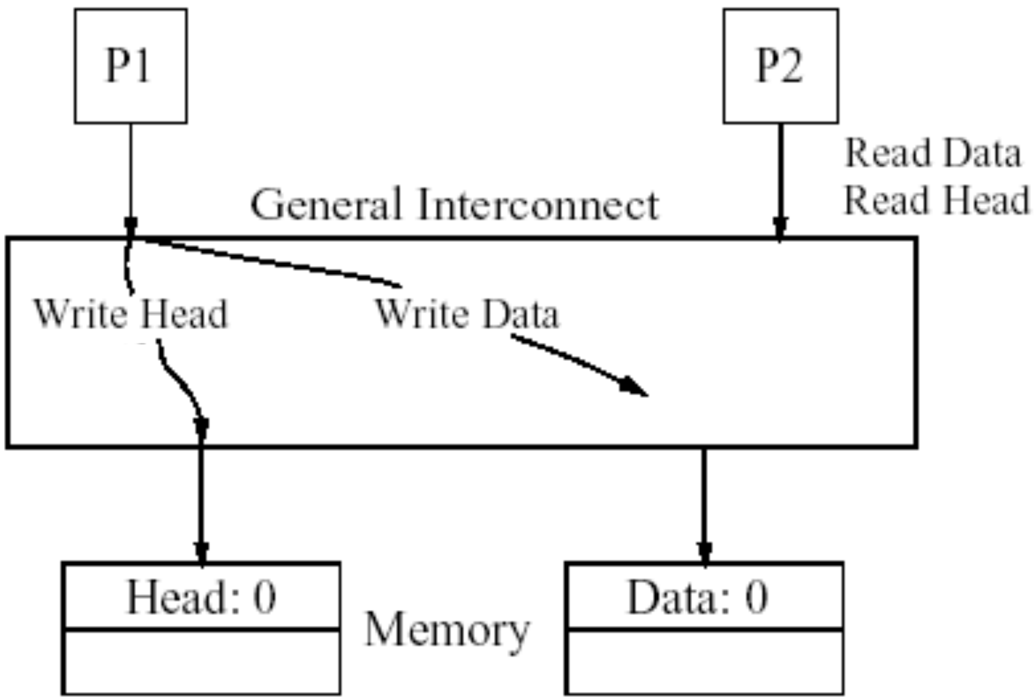
- **Even without caches, hardware optimizations can**
  - **—violate program order**
  - **—violate sequential consistency**

Adding Caches

- **Multiple caches can result in multiple copies of data values**
- **Copies induce three requirements**
  - **—coherence protocol: ensure any copies are up to date**
    - **typical strategies**
      - **invalidate protocol: invalidate copies**
      - **update protocol: update copies**
    - **memory consistency bounds interval when values must propagate**
  - **—detecting when a write is complete**
    - **harder with copies present**
  - **—propagating changes to copies is non-atomic**
    - **requires acknowledgments**
    - **… and you thought things were hard to reason about before!**

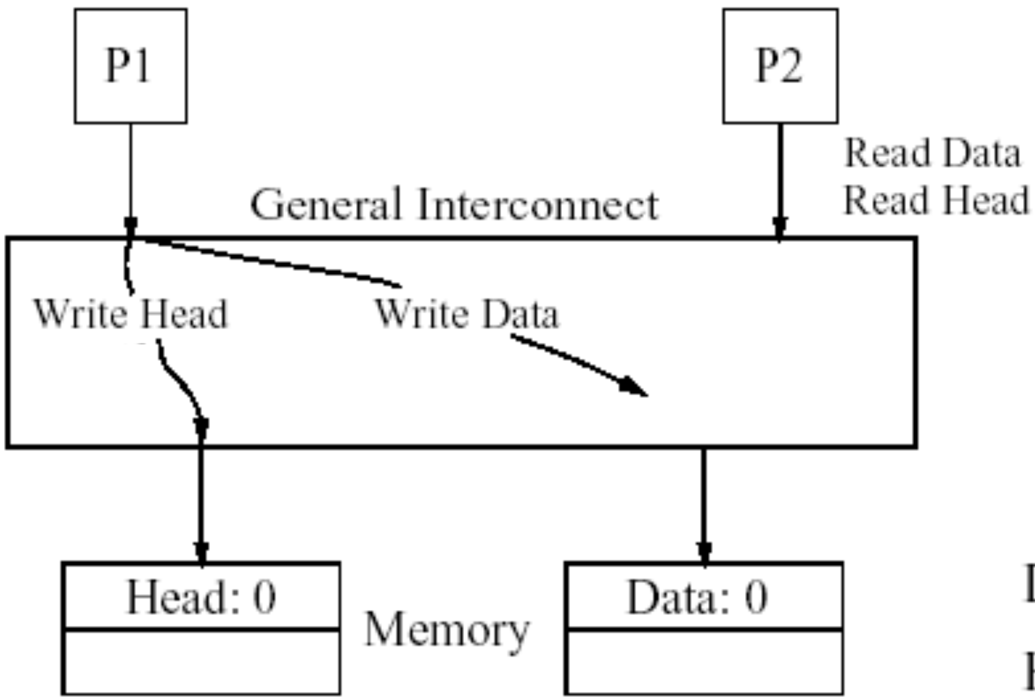
Caches

- **W=>W order using <u>write-through cache</u>**
  - —general interconnect instead of bus (memory parallelism)
  - —write-through cache for each processor (cache not shared)



Caches

- **W=>W order using <u>write-through cache</u>**
  - —general interconnect instead of bus (memory parallelism)
  - —write-through cache for each processor (cache not shared)



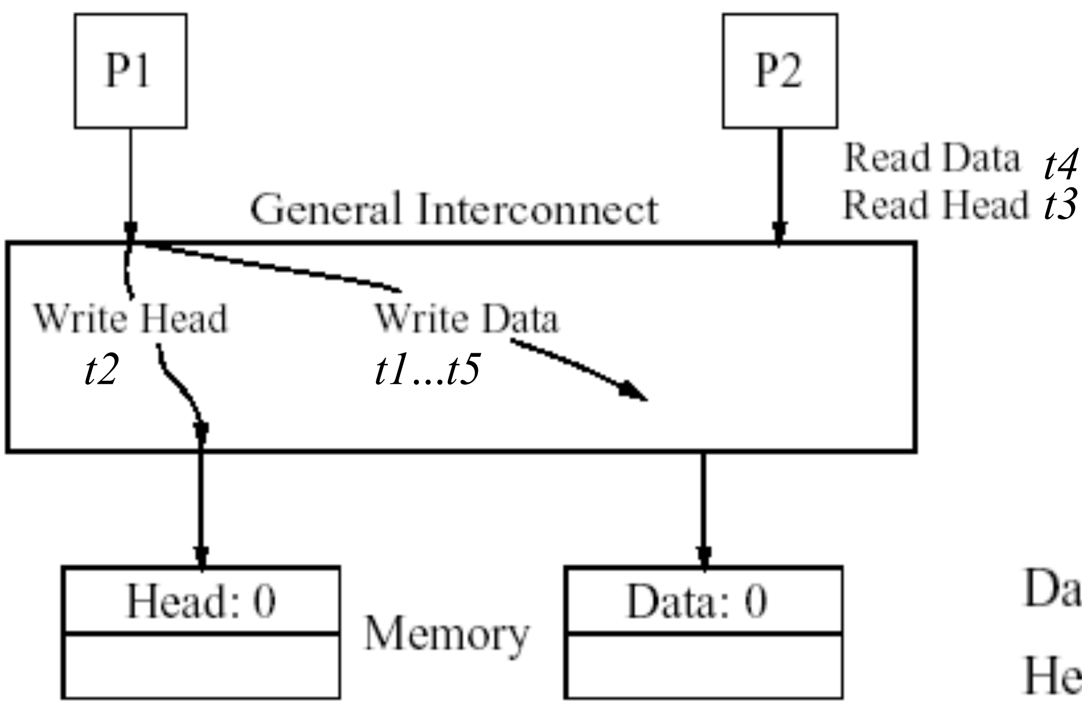
Initially: Data=Head=0
P2 cache has Data

Can write-through caches violate sequential consistency?

<u>P1</u>	<u>P2</u>
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Caches

- **W=>W order using <u>write-through cache</u>**
  - —general interconnect instead of bus (memory parallelism)
  - —write-through cache for each processor (cache not shared)



t: *completion* order

Initially: Data=Head=0
P2 cache has Data

Can write-through caches violate sequential consistency?

Yes: 1, 2, 3, 4, 5

P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Caching Effects Summary

- **Caches can**
  - **—violate memory atomicity**
  - **—violate sequential consistency**

Compilers

- **Reordering accesses to different locations can be problematic**
- **Register allocation of what should be a volatile is bad**
- **Assuming data is not shared can cause a variety of problems**
- **In the absence of analysis, must preserve order among memory operations**
  - **—conflicts with code motion, register allocation, CSE, tiling, software pipelining …**

Compiler Effects Summary

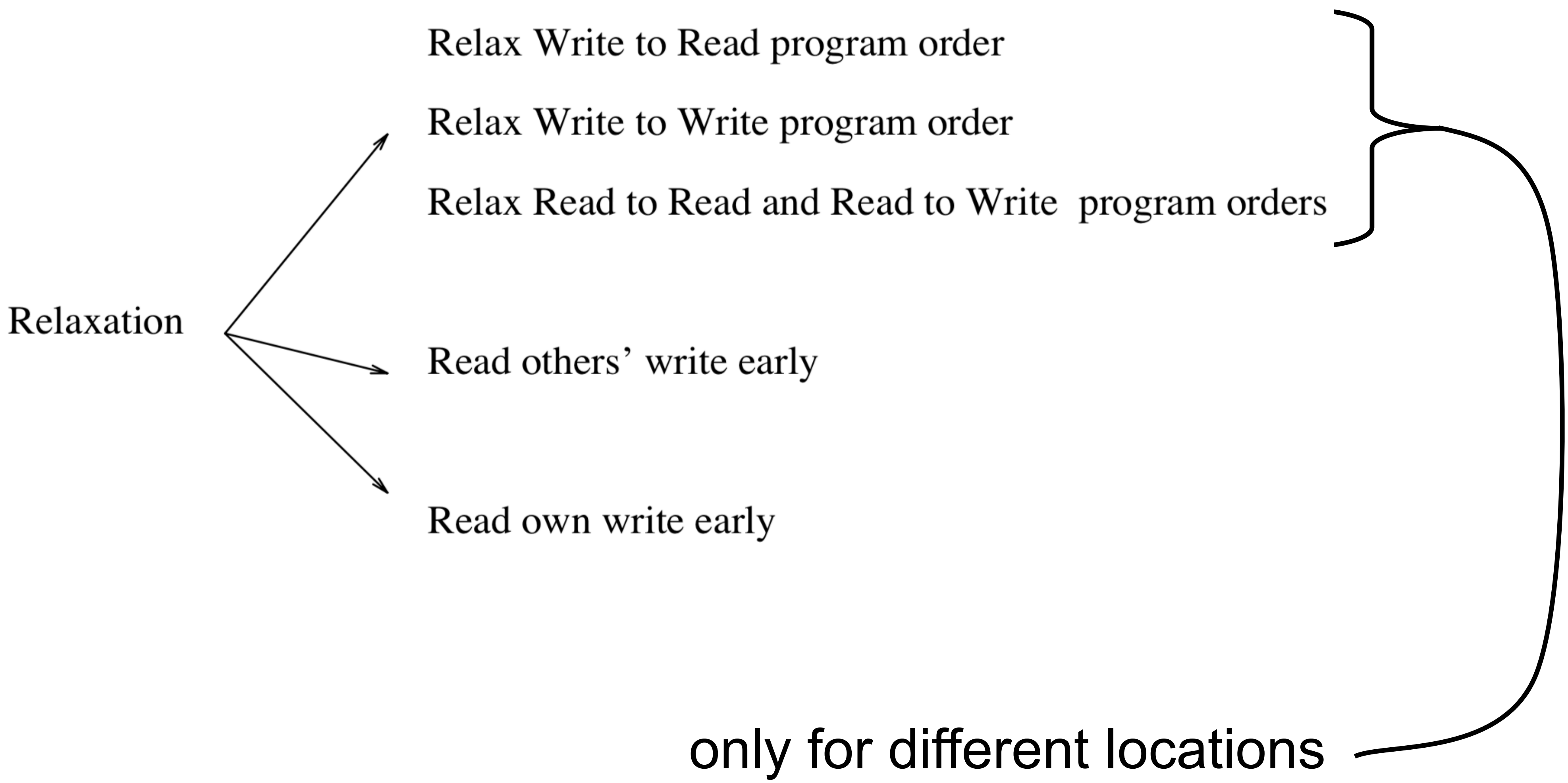
- **Compiler optimizations can**
  - **—violate program order, and thus**
  - **—violate sequential consistency**

Summary

- **Everybody violates sequential consistency!**
  - —hardware optimizations
  - —caches
  - —compilers

Relaxed Memory Models

Relaxed orderings allowed by relaxed memory models

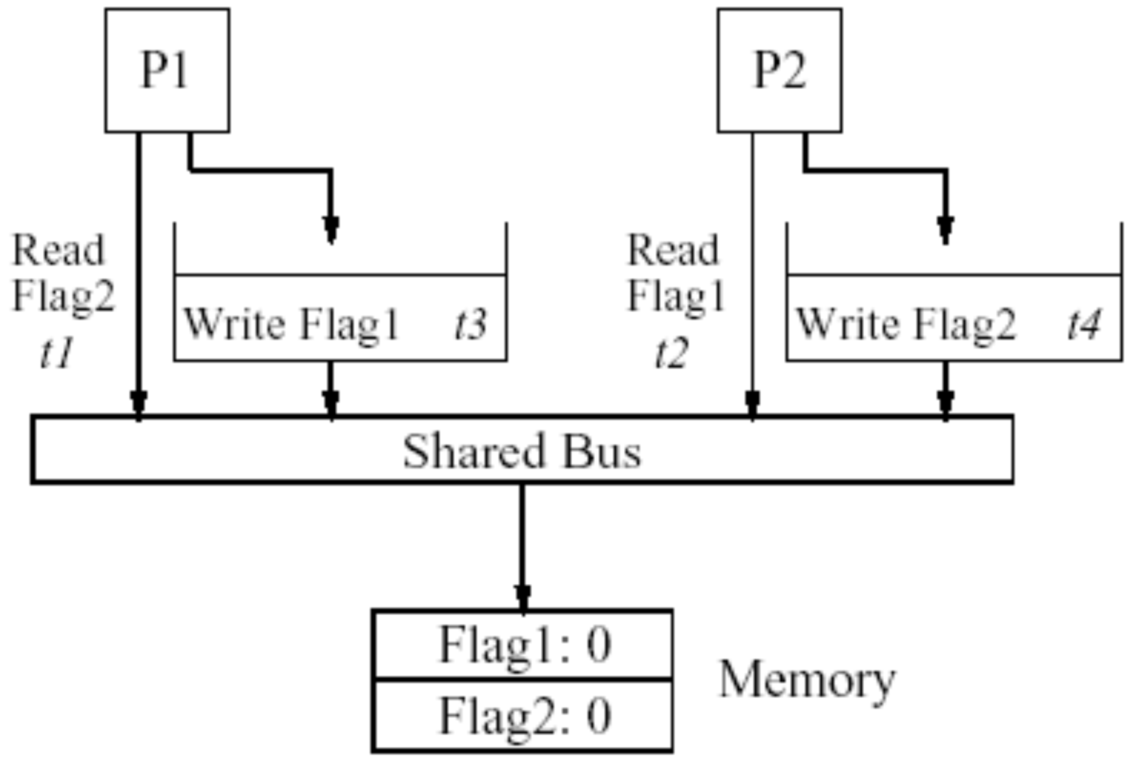


Relaxing W→R Order

Allows write buffers

- —write buffer with bypassing hides latency of writes
- —reads to different locations can bypass pending writes

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

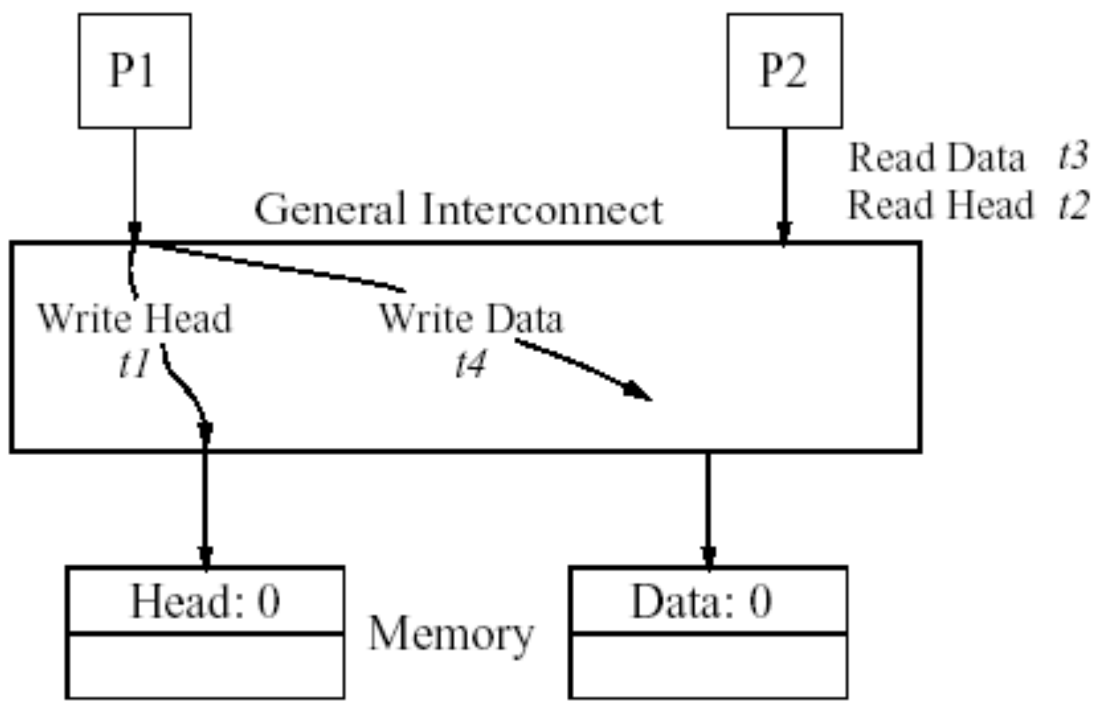


Relaxing W→W Order

Allows overlapping writes

- —general interconnect vs. bus (memory parallelism)
- —multiple writes to different issued by same processor handled by different memory modules

P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

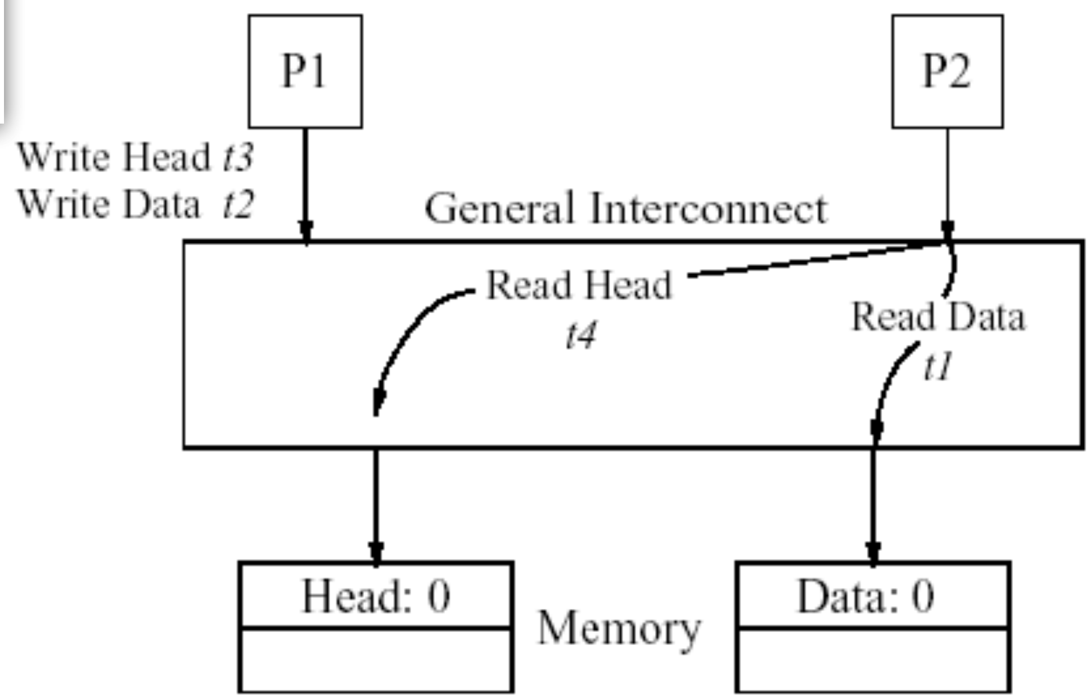


Relaxing R→R|W Order

Allows non-blocking reads

- —non-blocking reads + general memory interconnect
- —non-blocking caches, dynamic scheduling, speculative execution

P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data



Relaxing Write Atomicity

- **Allow a processor to return value of its own write before all cached copies of data are invalidated or updated**
  - —allows read to return value before
    - – write is serialized with other writes to same location
    - – before invalidates or updates reach other processors
  - —how?
    - – forward value in write buffer to a later read
    - – let read following write in write-through-cache return before write completes
- **Allow a thread to return value of another thread's write before all cached copies of data are invalidated or updated**

Benefits of Relaxed Orderings

- Permits high performance hardware
- Permits compiler optimizations
  —reorder instructions between synchronization instructions

Darker Side of Relaxed Ordering

- Complicated safety nets
- "Explaining how [to precisely preserve the atomicity of a write] is difficult within the simple framework presented in this article" (p. 16)

Weak Ordering

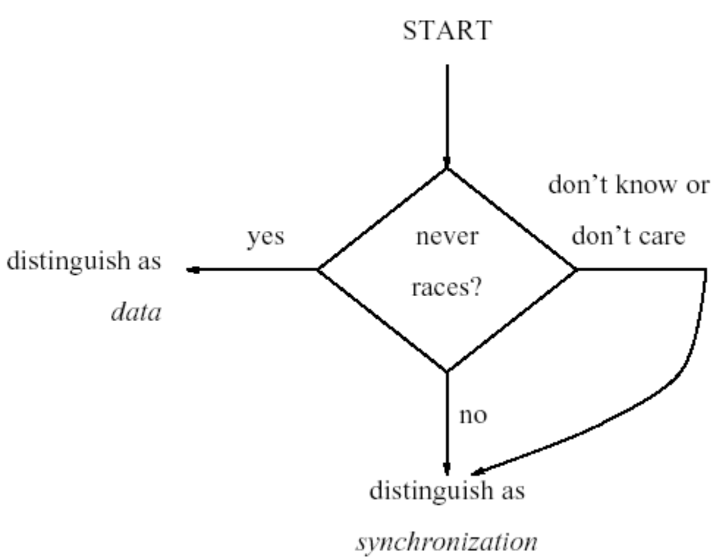
- Assume two types of operations
  - —synchronization
  - —data (i.e., ‘everything else’)
- Observation: typically, reordering data accesses between synchronization operations does not affect correctness

Weak Ordering

- **Assume two types of operations.**
  - —synchronization
  - —data (i.e., ‘everything else’)
- **Observation: typically, reordering data accesses between synchronization operations does not affect correctness**

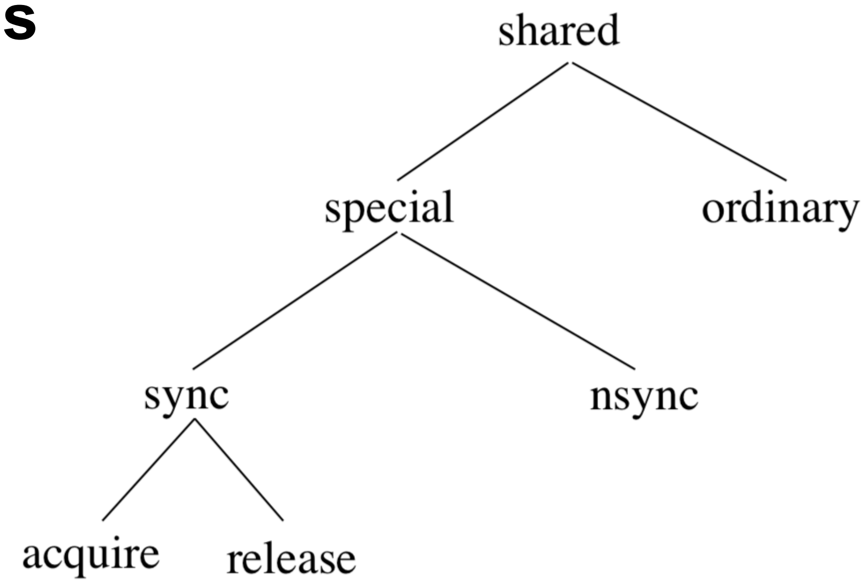
Approach

- allow reordering among normal data accesses
- require stricter ordering constraints for accesses to synchronization variables



Release Consistency

- **Types of operations**



—ordinary ~ data accesses in weak ordering

—special

- **syncs: two types**
  - **acquire: e.g. lock operation to gain access to a CS**
  - **release: write to grant permission to access CS**
- **nsyncs: asynchronous operations that are not synchronization ops**

“Safety Net” Mechanisms

- Serialization instructions (IBM 370)
  - —e.g. compare-and-swap, branches
  - —placing serialization instruction after write guarantees SC
- Atomic read-modify-write operations
  - —e.g. SPARC TSO: program order appears to be preserved between W and following R if one of them is part of a RMW operation
    - – can replace R with “identity” RMW to force ordering on R
    - – can replace W with “oblivious” RMW to force ordering on W
  - —preserving R → W ordering with PC: replace R with “identity” RMW
- Fence instructions
  - —memory barrier: fence for all memory ops
  - —store barrier: fence for writes only
  - —SPARC MEMBAR: can enforce orderings between access types selectively

Relaxed Ordering in Practice

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Coping with Relaxed Models

- What if programmers had to keep all these details in mind?
  - relaxed program order + relaxed memory atomicity...

- Abstraction: we want a memory model for a language
  —general enough
  - to permit performance
  - to be widely used

  —simple enough
  - to reason about
  - to be portable

Examples of Language Level Models

- Java
  - —detailed memory model specification for security, portability
- C++
  - —simple to understand defaults to simplify development
  - —full control for top performance
  - —no concern for security
- Unified Parallel C
  - —supports both “strict” and “relaxed” memory models
  - —simplicity vs. performance
  - —default and per access choices

Take-away Points

- Memory models, which describe the semantics of shared variables, are crucial to both correct multithreaded applications and the entire underlying implementation stack
- Major programming languages are converging on a model that guarantees simple interleaving-based semantics for "data-race-free" programs and most hardware vendors have committed to support this model
- This process has exposed fundamental shortcomings in our languages and a hardware-software mismatch
  - —semantics for programs that contain data races seem fundamentally difficult, but are necessary for concurrency safety and debuggability.
  - —call upon software and hardware communities to develop languages and systems that enforce data-race-freedom, and co-designed hardware that exploits and supports such semantics

References

- **Chapter 3: Memory Consistency Motivation A Primer on Memory Consistency and Cache Coherence, Daniel J. Sorin, Mark D. Hill, David A. Wood Synthesis Lectures on Computer Architecture. Morgan Claypool. 2011.**
- **Shared Memory Consistency Models: A Tutorial, Sarita V. Adve. Kourosh Gharachorloo. Technical Report 95-7, Digital Western Research Laboratory, Palo Alto, CA.**
- **Memory Models: A Case for Rethinking Parallel Languages and Hardware, Sarita V. Adve, Hans-J. Boehm. Communications of the ACM, Vol. 53 No. 8, Pages 90-101, August, 2010. 10.1145/1787234.1787255**
- **The Java Memory Model, J. Manson, W. Pugh, and S. V. Adve, in Proceedings of the Symposium on Principles of Programming Languages (PoPL), January 2005.**

References - II

- **Foundations of the C++ Concurrency Memory Model**, H. Boehm, and S. V. Adve. In Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (Tucson, AZ, USA, June 07 - 13, 2008). PLDI '08. ACM, New York, NY, 68-78.
- **UPC Language Specifications Version 1.3**, UPC Consortium November 16, 2013, https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf