

**Lecture 15:**

# **Memory Coherency and Consistency**

---

**Parallel Computing  
Stanford CS149, Fall 2025**

# **Midterm Tuesday Nov 18, 6-8pm, Hewlett 200 and 201**

**No class on Tuesday, Nov 18<sup>th</sup>**

## **Topic Coverage**

- The midterm exam will cover all topics up to cache coherence (Lectures 1–14)**
- Closed everything**

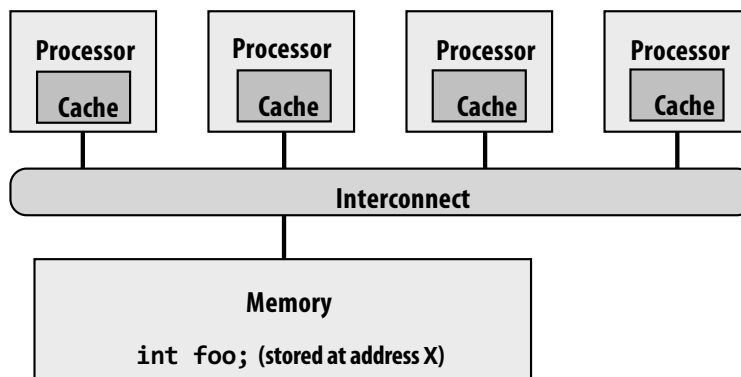
## **Review session (this Friday!)**

- Friday Nov 14<sup>th</sup>, from 4-6pm in Room 420-040 (main quad)**
- CAs will cover practice problems**

# The cache coherence problem

Modern processors replicate contents of memory in local caches

Problem: processors can observe different values for the same memory location



Is this a mutual exclusion problem?

Can you fix the problem by adding locks to your program?

**NO!**

This is a problem created by replicating the data stored at address X in local caches

The chart at right shows the value of variable foo (stored at address X) in main memory and in each processor's cache

Assume the initial value stored at address X is 0

Assume write-back cache behavior

How could we fix this problem?

Action	P1 \$	P2 \$	P3 \$	P4 \$	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (assume this load causes eviction of X)		0	2		1

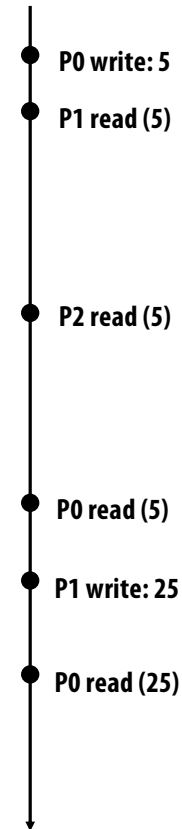
# Definition: Coherence

A memory system is coherent if:

The results of a parallel program's execution are such that for each memory location, there is a hypothetical **serial order** of all program operations (executed by all processors) to the location that is consistent with the results of execution, and:

1. Memory operations issued by any one processor occur in the order issued by the processor
2. The value returned by a read is the value written by the last write to the location... as given by the serial order

Chronology of  
operations on  
address X



# MSI write-back invalidation protocol

## Key tasks of protocol

- Ensuring processor obtains exclusive access for a write
- Locating most recent copy of cache line's data on cache miss

## Three cache line states

- Invalid (I): same as meaning of invalid in uniprocessor cache
- Shared (S): line valid in one or more caches, memory is up to date
- Modified (M): line valid in exactly one cache (a.k.a. "dirty" or "exclusive" state)

## Two processor operations (triggered by local CPU)

- PrRd (read)
- PrWr (write)

## Three coherence-related bus transactions (from remote caches)

- BusRd: obtain copy of line with no intent to modify
- BusRdX: obtain copy of line with intent to modify
- BusWB: write dirty line out to memory

# MSI Invalidate Protocol

Read obtains block in “shared”

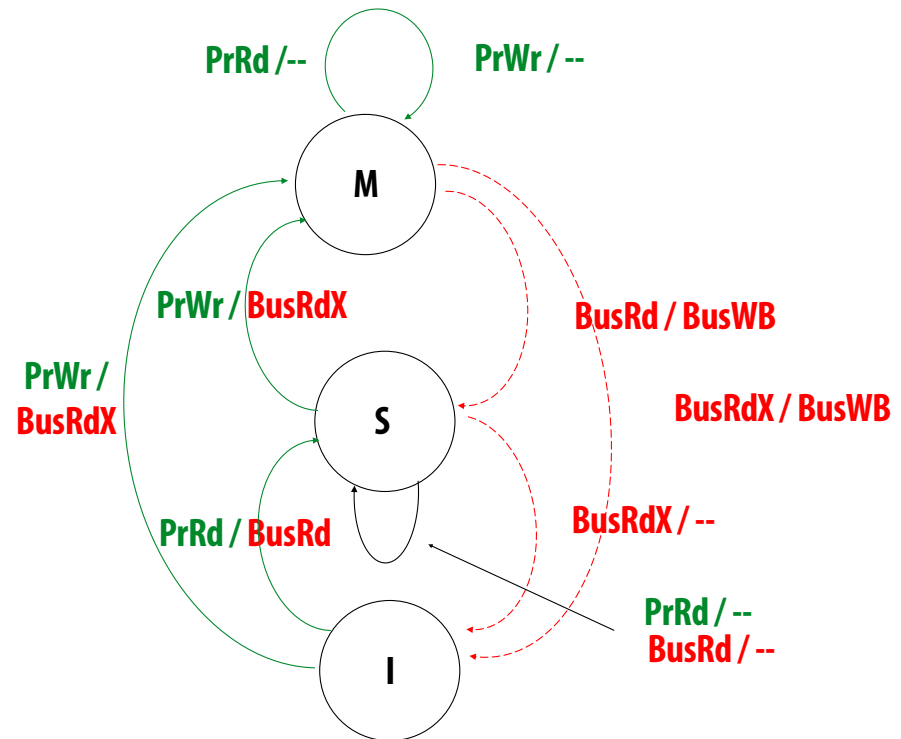
- even if only cached copy

Obtain exclusive ownership before writing

- BusRdX causes others to invalidate
- If M in another cache, will cause writeback
- BusRdX even if hit in S
  - promote to M (upgrade)

A / B: if action A is observed by cache controller, action B is taken

— Processor initiated  
- - - Bus initiated



\* Remember, all caches are carrying out this logic independently to maintain coherence

# A Cache Coherence Example

Proc Action	P1 \$-state	P2 \$-state	P3 \$-state	Bus Trans	Data from
P1 read x	S	--	--	BusRd	Memory
P3 read x	S	--	S	BusRd	Memory
P3 write x	I	--	M	BusRdX	Memory
P1 read x	S	--	S	BusRd	P3 \$
P1 read x	S	--	S		P1 \$
P2 write x	I	M	I	BusRdX	Memory

**Single writer, multiple reader protocol**

**Why do you need Modified to Shared?**

**Communication increases memory latency**

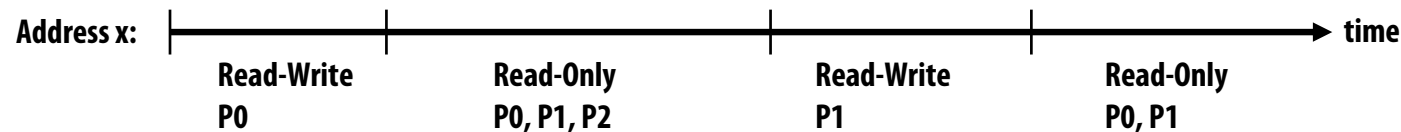
# How Does MSI Satisfy Cache Coherence Invariants?

## 1. Single-Writer, Multiple-Read (SWMR) Invariant

- Only one cache can be in M-state all others get invalidation message
- Multiple caches can be in read-only S-state

## 2. Data-Value Invariant (write serialization)

- On BusRd and BusRdx data is provided by cache with line in M-state
- Bus serializes all transactions



# MESI invalidation protocol

**MSI requires two interconnect transactions for the common case of reading an address, then writing to it**

- Transaction 1: BusRd to move from I to S state
- Transaction 2: BusRdX to move from S to M state



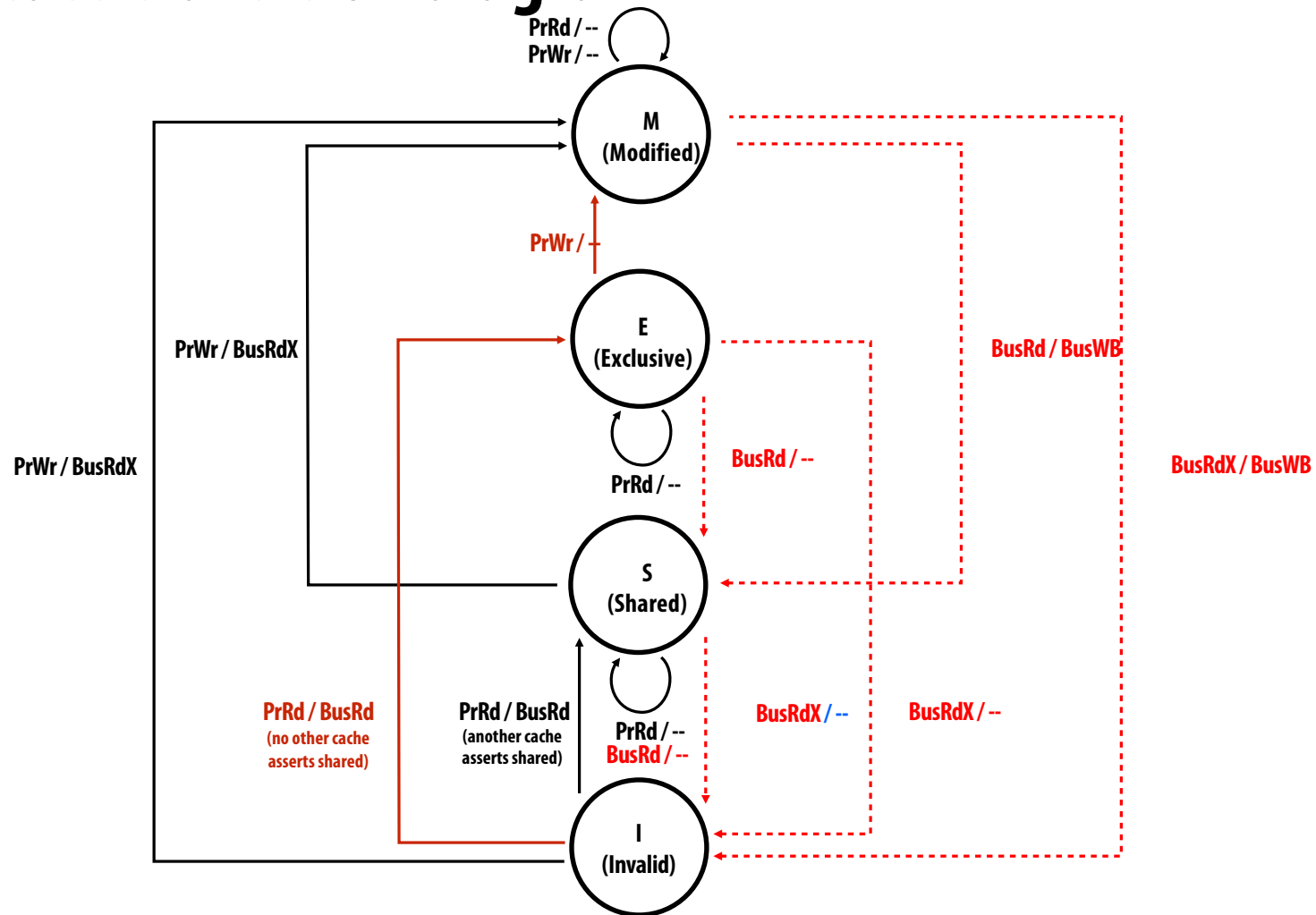
**MESI, not Messi!**

**This inefficiency exists even if application has no sharing at all**

**Solution: add additional state E (“exclusive clean”)**

- Line has not been modified, but only this cache has a copy of the line
- Decouples exclusivity from line ownership (line not dirty, so copy in memory is valid copy of data)
- Upgrade from E to M does not require an bus transaction

# MESI state transition diagram



# Scalable cache coherence using directories

Snooping schemes broadcast coherence messages to determine the state of a line in the other caches: not scalable and too restrictive

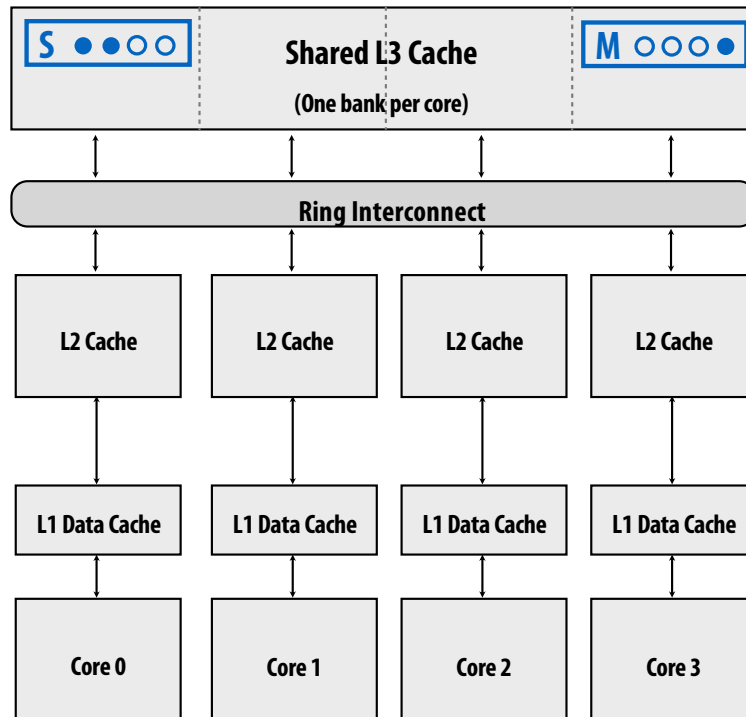
Alternative idea: avoid broadcast by storing information about the status of the line in one place: a “directory”

- The directory entry for a cache line contains information about the state of the cache line in all caches.
- Caches look up information from the directory as necessary
- Cache coherence is maintained by point-to-point messages between the caches on a “need to know” basis (
- Directory instead of bus serves as coherence serialization mechanism

## ■ Still need to maintain invariants

- SWMR
- Write serialization

# Directory coherence in Intel Core i7 CPU



**L3 serves as centralized directory for all lines in the L3 cache**  
- **Serialization point**

(Since L3 is an inclusive cache, any line in L2 is guaranteed to also be resident in L3)

**Directory maintains list of L2 caches containing line**  
**Instead of broadcasting coherence traffic to all L2's, only send coherence messages to L2's that contain the line**

(Core i7 interconnect is a ring, it is not a bus)

**Directory dimensions:**

- **P = 4**
- **M = number of L3 cache lines**

# **Implications of cache coherence to the programmer**

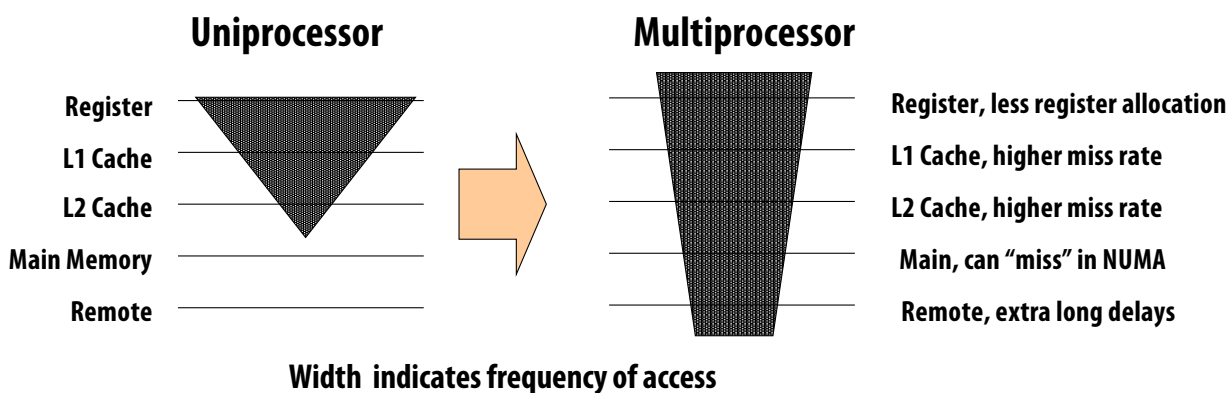
# Communication Overhead

Communication time is a key parallel overhead

- Appears as increased memory access time in multiprocessor
  - Extra main memory accesses in UMA systems
    - Must determine increase in cache miss rate vs. uniprocessor
  - Some accesses have higher latency in NUMA systems
    - Only a fraction of a % of these can be significant!

$$AMAT_{\text{Multiprocessor}} > AMAT_{\text{Uniprocessor}}$$

Average Memory Access Time (AMAT) =  $\sum_0^n \text{frequency of access} \times \text{latency of access}$



## Core i7 Xeon 5500 Series Data Source Latency (approx.)

L1 hit, ~4 cycles  
 L2 hit, ~10 cycles  
 L3 hit, line unshared ~40 cycles  
 L3 hit, shared line in another core ~65 cycles  
 L3 hit, modified in another core ~75 cycles remote  
 Local DRAM ~30 ns (~120 cycles)  
 Remote DRAM ~100 ns (~400 cycles)

## Use system tools to optimize cache performance

## Memory Access Analysis for Cache Misses and High Bandwidth Issues

Use the Intel® VTune™ Profiler's Memory Access analysis to identify memory-related issues, like NUMA problems and bandwidth-limited accesses, and attribute performance events to memory objects (data structures), which is provided due to instrumentation of memory allocations/de-allocations and getting static/global variables from symbol information.

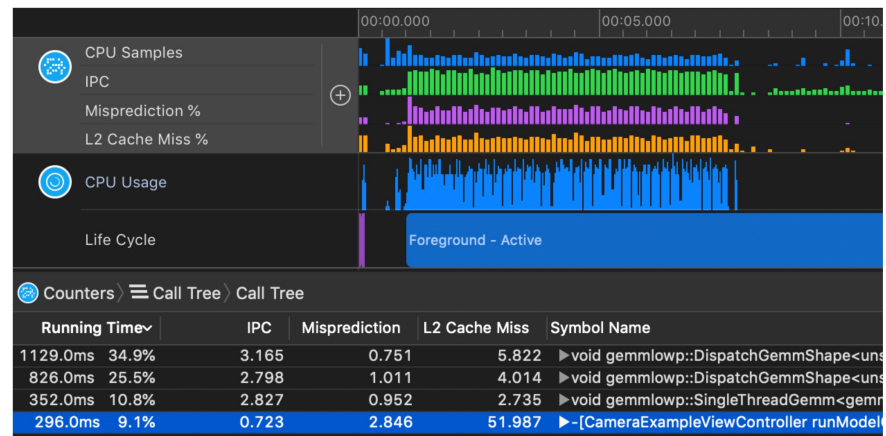
**NOTE:**  
Intel® VTune™ Profiler is a new renamed version of the Intel® VTune™ Amplifier.

## How It Works

Grouping	Bandwidth/Domain/ Bandwidth Utilization %/ Function / Call Stack	CPU Time	Memory Borrow	Loads	Latencies	LLC Miss Count	Average Latency (cycles)
DATA GBtree		9.703%	64.3%	6,517.0%	4,141.2%	11,811,508	92
HWRA		425.5%	56.6%	2,345.0%	2,112.3%	11,807,140	115
Paging	v_xref_scsi_req_memory	0.117%	100.0%	175,000%	3,150,340%	0	233.0
	v_b_soforting	0.021%	0.0%	0.0%	0.0%	0	0.0
	run_timer_sorting	0.002%	0.0%	0.0%	0.0%	0	0.0
	v_b_page_fault	0.001%	0.0%	0.0%	0.0%	0	0.0
	trunc_migrate_page	0.001%	0.0%	0.0%	0.0%	0	0.0
	task_count	0.0%	0.0%	1,400,021%	0.0%	0	0.0
	P Medium	2.880%	70.3%	2,765.0%	1,401.4%	52,853,171	83

Memory Access analysis type uses **hardware event-based sampling** to collect data for the following metrics:

- **Loads and Stores** metrics that show the total number of loads and stores
- **L1 Miss Count** metric that shows the total number of last-level cache misses
  - **Local DRAM Access Count** metric that shows the total number of LLC misses serviced by the local memory
  - **Remote DRAM Access Count** metric that shows the number of accesses to the remote socket memory
  - **Remote Cache Access Count** metric that shows the number of accesses to the remote socket cache
- **Memory Bound** metric that shows a fraction of cycles spent waiting due to demand load or store instructions
  - **L1 Bound** metric that shows how often the machine was stalled without missing the L1 data cache
  - **L2 Bound** metric that shows how often the machine was stalled on L2 cache
  - **L3 Bound** metric that shows how often the CPU was stalled on L3 cache, or contended with a sibling core
  - **L3 Latency** metric that shows a fraction of cycles with demand load accesses that hit the L3 cache under unloaded scenarios (possibly L3 latency limited)
- **NUMA: % of Remote Accesses** metric shows percentage of memory requests to remote DRAM. The lower its value is, the better.
- **DRAM Bound** metric that shows how often the CPU was stalled on the main memory (DRAM). This metric enables you to identify **DRAM Bandwidth Bound**, **UPI Utilization Bound** issues, as well as **Memory Latency** issues with the following metrics:
  - **Remote / Local DRAM Ratio** metric that is defined by the ratio of remote DRAM loads to local DRAM loads
  - **Local DRAM** metric that shows how often the CPU was stalled on loads from the local memory
  - **Remote DRAM** metric that shows how often the CPU was stalled on loads from the remote memory
  - **Remote Cache** metric that shows how often the CPU was stalled on loads from the remote cache in other sockets
- **Average Latency** metric that shows an average load latency in cycles



## Apple Xcode Instruments

# Intel VTune

# Unintended communication via false sharing

**What is the potential performance problem with this code?**

```
// allocate per-thread variable for local per-thread accumulation
int myPerThreadCounter[NUM_THREADS];
```

**Why might this code be more performant?**

```
// allocate per thread variable for local accumulation
struct PerThreadState {
    int myPerThreadCounter;
    char padding[CACHE_LINE_SIZE - sizeof(int)];
};
PerThreadState myPerThreadCounter[NUM_THREADS];
```

# Demo: false sharing

```
void* worker(void* arg) {  
    volatile int* counter = (int*)arg;  
    for (int i=0; i<MANY_ITERATIONS; i++)  
        (*counter)++;  
    return NULL;  
}
```

threads update a per-thread counter  
many times

```
void test1(int num_threads) {  
    pthread_t threads[MAX_THREADS];  
    int counter[MAX_THREADS];  
    for (int i=0; i<num_threads; i++)  
        pthread_create(&threads[i], NULL,  
                        &worker, &counter[i]);  
    for (int i=0; i<num_threads; i++)  
        pthread_join(threads[i], NULL);  
}
```

Execution time with  
num\_threads=8 on 4-core system:  
**14.2 sec**

```
struct padded_t {  
    int counter;  
    char padding[CACHE_LINE_SIZE - sizeof(int)];  
};  
  
void test2(int num_threads) {  
    pthread_t threads[MAX_THREADS];  
    padded_t counter[MAX_THREADS];  
    for (int i=0; i<num_threads; i++)  
        pthread_create(&threads[i], NULL,  
                        &worker, &(counter[i].counter));  
    for (int i=0; i<num_threads; i++)  
        pthread_join(threads[i], NULL);  
}
```

Execution time with  
num\_threads=8 on 4-core system:  
**4.7 sec**

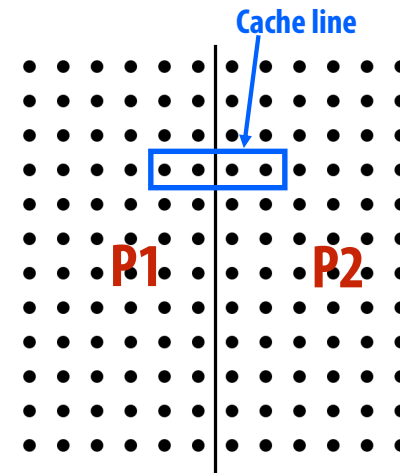
# False sharing

Condition where two processors write to different addresses, but addresses map to the same cache line

Cache line “ping-pongs” between caches of writing processors, generating significant amounts of communication due to the coherence protocol

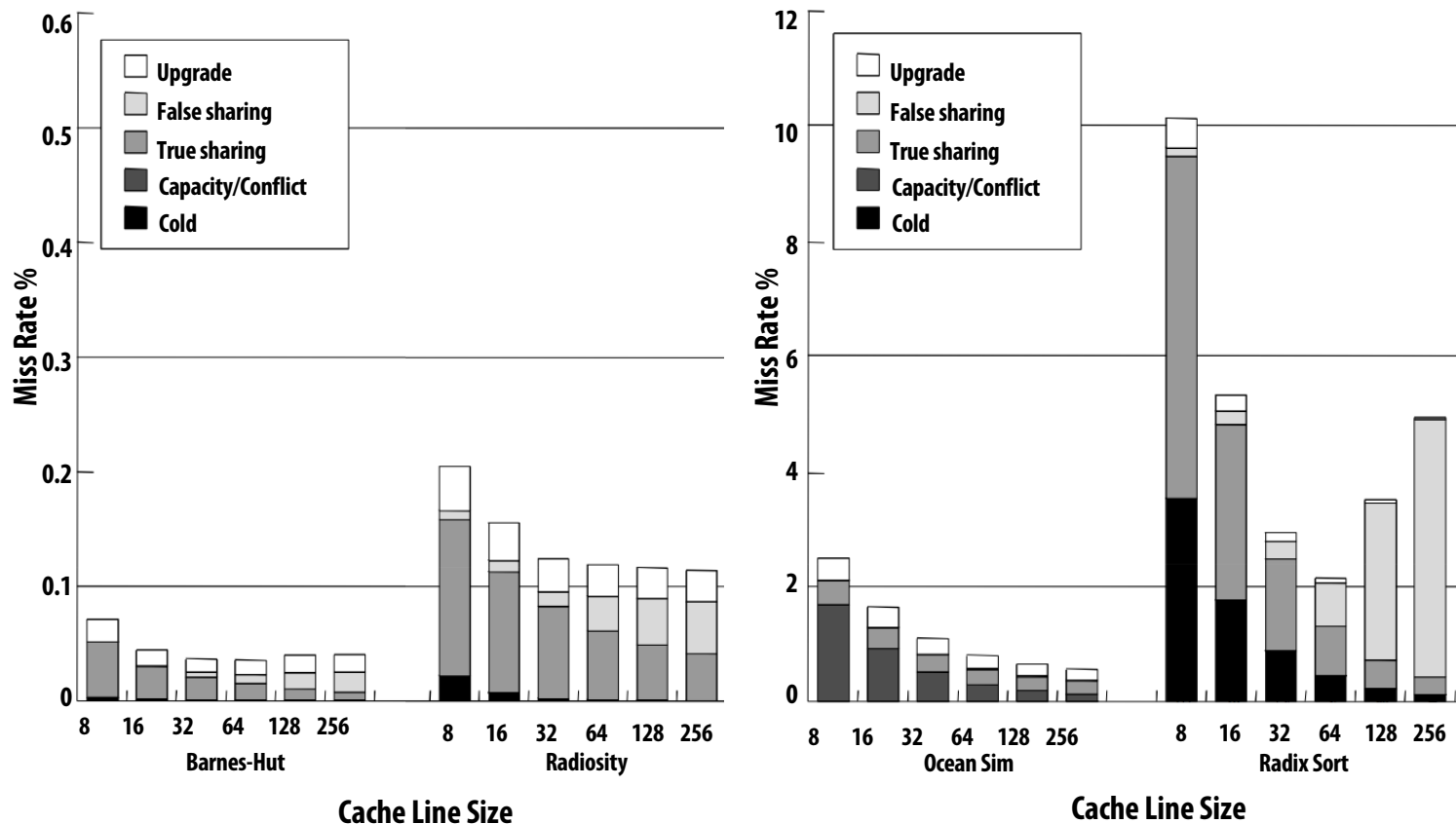
No inherent communication, this is entirely artifactual communication (cachelines > 4B)

False sharing can be a factor in when programming for cache-coherent architectures



# Impact of cache line size on miss rate

Results from simulation of a 1 MB cache (four example applications)



\* Note: I separated the results into two graphs because of different Y-axis scales

Figure credit: Culler, Singh, and Gupta

# Summary: Cache coherence

The cache coherence problem exists because the abstraction of a single shared address space is not implemented by a single storage unit

- Storage is distributed among main memory and local processor caches
- Data is replicated in local caches for performance

Main idea of snooping-based cache coherence: whenever a cache operation occurs that could affect coherence, the cache controller **broadcasts a notification to all other cache controllers in the system**

- Challenge for HW architects: minimizing overhead of coherence implementation
- Challenge for SW developers: be wary of artifactual communication due to coherence protocol (e.g., false sharing)

Scalability of snooping implementations is limited by ability to broadcast coherence messages to all caches!

- Scaling cache coherence via directory-based approaches

**Lecture 15:**

# **Memory Consistency**

---

**Parallel Computing**  
**Stanford CS149, Fall 2025**

# Shared Memory Behavior

**Intuition says loads should return latest value written**

- **What is latest?**
- **Coherence: only one memory address**
- **Consistency: apparent ordering for all addresses**
  - **Order in which memory operations performed by one thread become visible to other threads**

## **Affects**

- **Programmability: how programmers reason about program behavior**
  - **Allowed behavior of multithreaded programs executing with shared memory**
- **Performance: limits HW/SW optimizations that can be used**
  - **Reordering memory operations to hide latency**

# Memory Consistency: Who Should Care

**Anyone who:**

- Wants to implement a synchronization library**
- Will ever work a job in kernel (or driver) development**
- Seeks to implement lock-free data structures \***

**\* Topic of a later lecture**

# Memory coherence vs. memory consistency

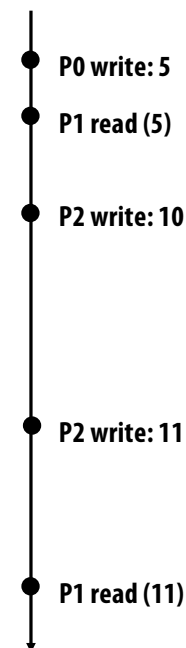
**Memory coherence** defines requirements for the observed behavior of reads and writes to the same memory location

- All processors must agree on the order of reads/writes to X
- In other words: it is possible to put all operations involving X on a timeline such that the observations of all processors are consistent with that timeline

**Memory consistency** defines the behavior of reads and writes to different locations (as observed by other processors)

- Coherence only guarantees that writes to address X will eventually propagate to other processors
- Consistency deals with when writes to X propagate to other processors, relative to reads and writes to other addresses

Observed chronology of operations on address X



# Coherence vs. Consistency

(said again, perhaps more intuitively this time)

**The goal of cache coherence is to ensure that the memory system in a parallel computer behaves as if the caches were not there**

- Just like how the memory system in a uni-processor system behaves as if the cache was not there

**A system without caches would have no need for cache coherence**

**Memory consistency defines the allowed behavior of loads and stores to different addresses in a parallel system**

- The allowed behavior of memory should be specified whether or not caches are present (and that's what a memory consistency model does)

# Memory Consistency

## **TL:DR:**

- **Multiprocessors reorder memory operations in unintuitive and strange ways**
- **This behavior is necessary for performance**
- **Application programmers rarely see this behavior**
- **Systems (OS and compiler) developers see it all the time**

# Memory operation ordering

**A program defines a sequence of loads and stores  
(this is the “program order” of the loads and stores)**

**Four types of memory operation orderings**

- $W_X \rightarrow R_Y$ : write to X must commit before subsequent read from Y \***
- $R_X \rightarrow R_Y$ : read from X must commit before subsequent read from Y**
- $R_X \rightarrow W_Y$ : read to X must commit before subsequent write to Y**
- $W_X \rightarrow W_Y$ : write to X must commit before subsequent write to Y**

**\* To clarify: “write must commit before subsequent read” means:**

**When a write comes before a read in program order, the write must commit (its results are visible) by the time the read occurs.**

# Multiprocessor Execution

Initially  $A = B = 0$

**Proc 0**

(1)  $A = 1$

(2) print B

**Proc 1**

(3)  $B = 1$

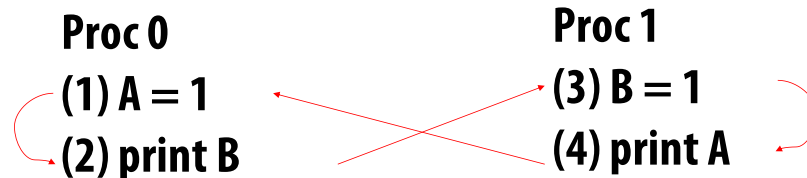
(4) print A

**What can be printed?**

- "01"?
- "10"?
- "11"?
- "00"?

# Orderings That Should Not Happen

Initially  $A = B = 0$



**The program should not print “00” or “10”**

**A “happens-before” graph shows the order in which events must execute to get a desired outcome**

**If there’s a cycle in the graph, an outcome is impossible—an event must happen before itself!**

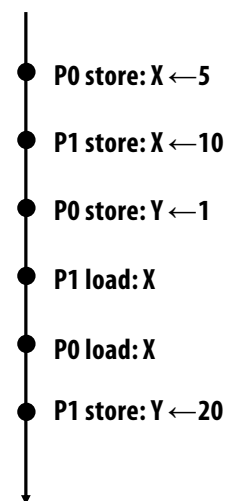
# What Should Programmers Expect

## Sequential Consistency

- Lamport 1976 (Turing Award 2013)
- All operations executed in some sequential order
  - As if they were manipulating a single shared memory
- Each thread's operations happen in program order

A sequentially consistent memory system maintains all four memory operation orderings ( $W_X \rightarrow R_Y$ ,  $R_X \rightarrow R_Y$ ,  $R_X \rightarrow W_Y$ ,  $W_X \rightarrow W_Y$ )

There is a chronology of all memory operations that is consistent with observed values

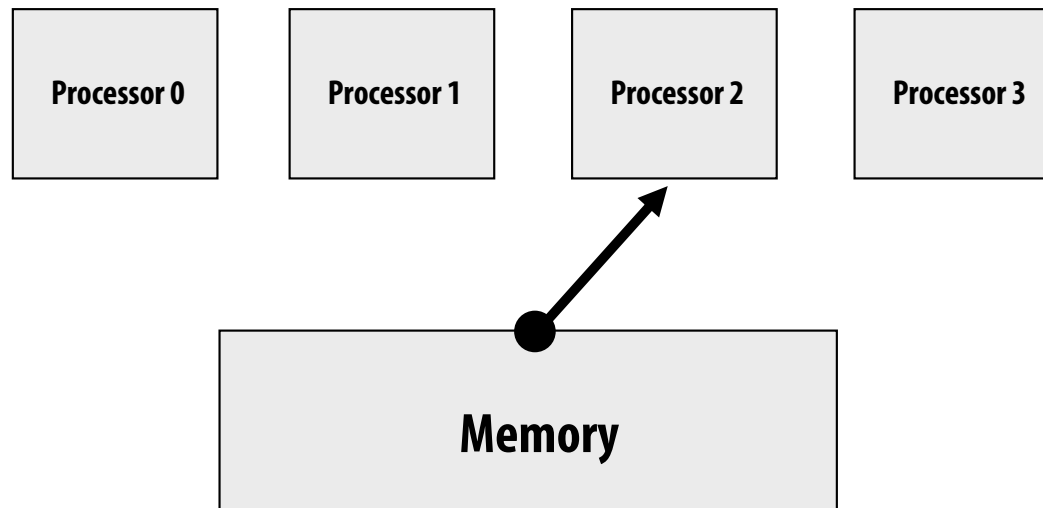


Note, now timeline lists operations to addresses X and Y

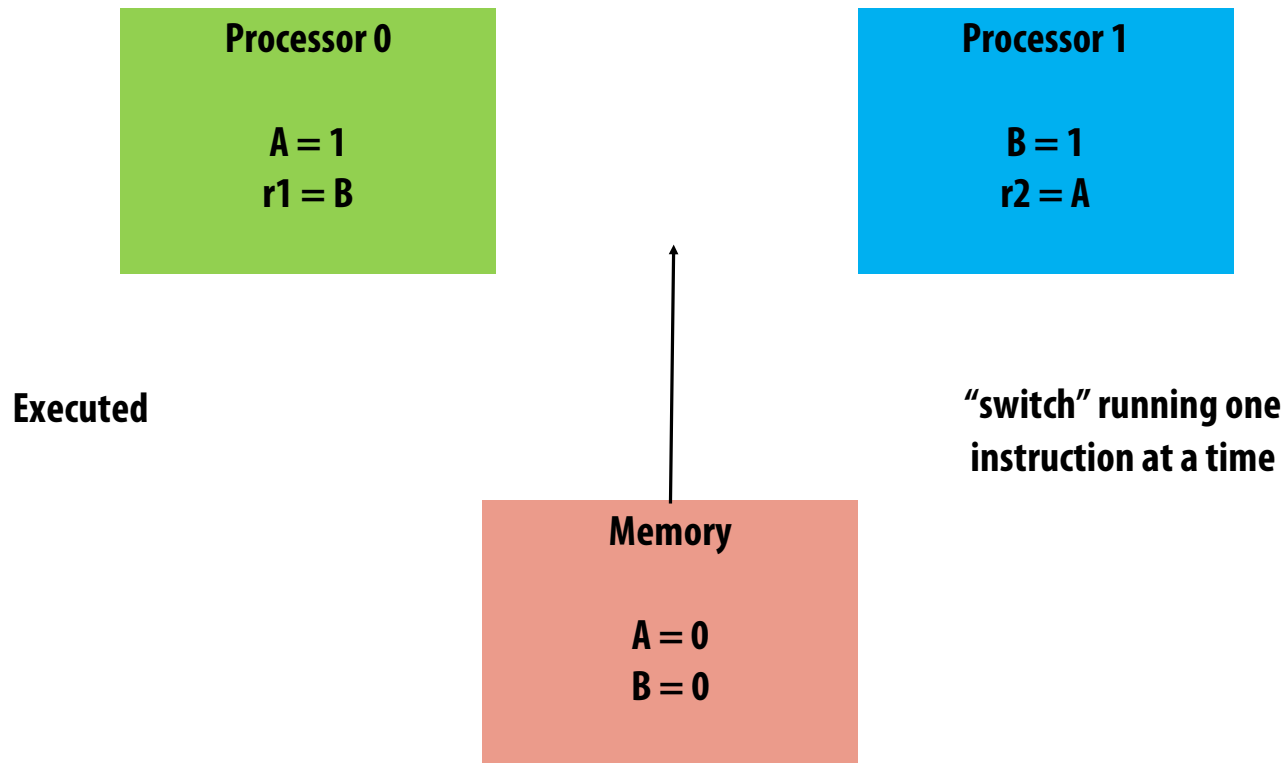
# Sequential consistency (switch metaphor)

**All processors issue loads and stores in program order**

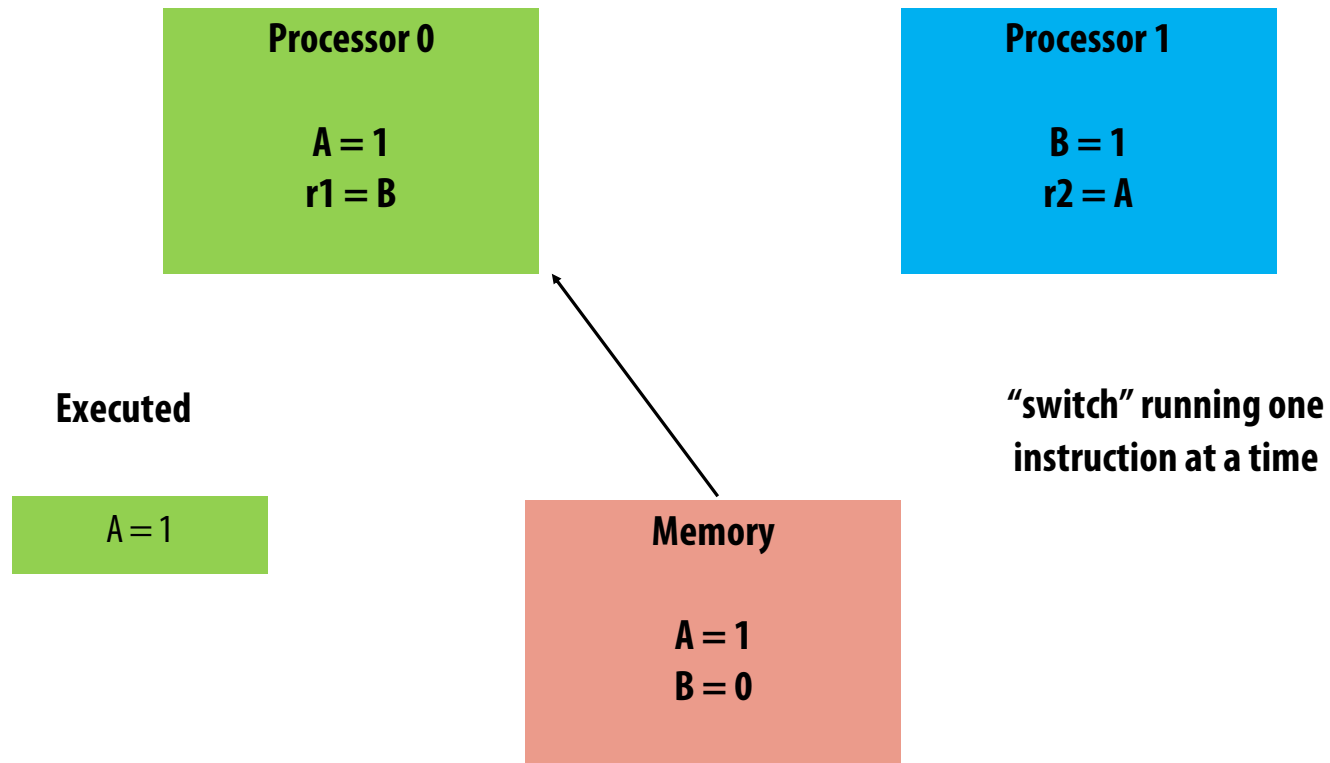
**Memory chooses a processor at random, performs a memory operation to completion, then chooses another processor, ...**



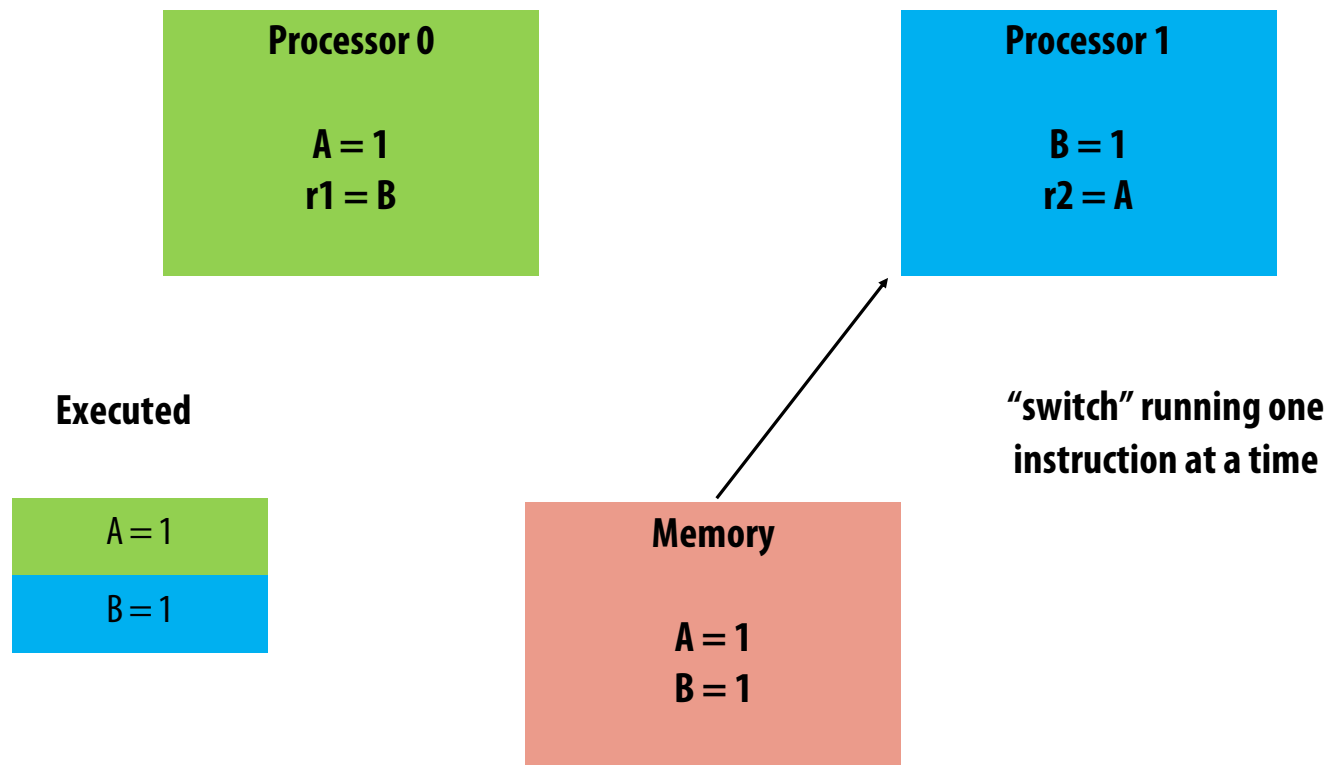
# Sequential Consistency Example



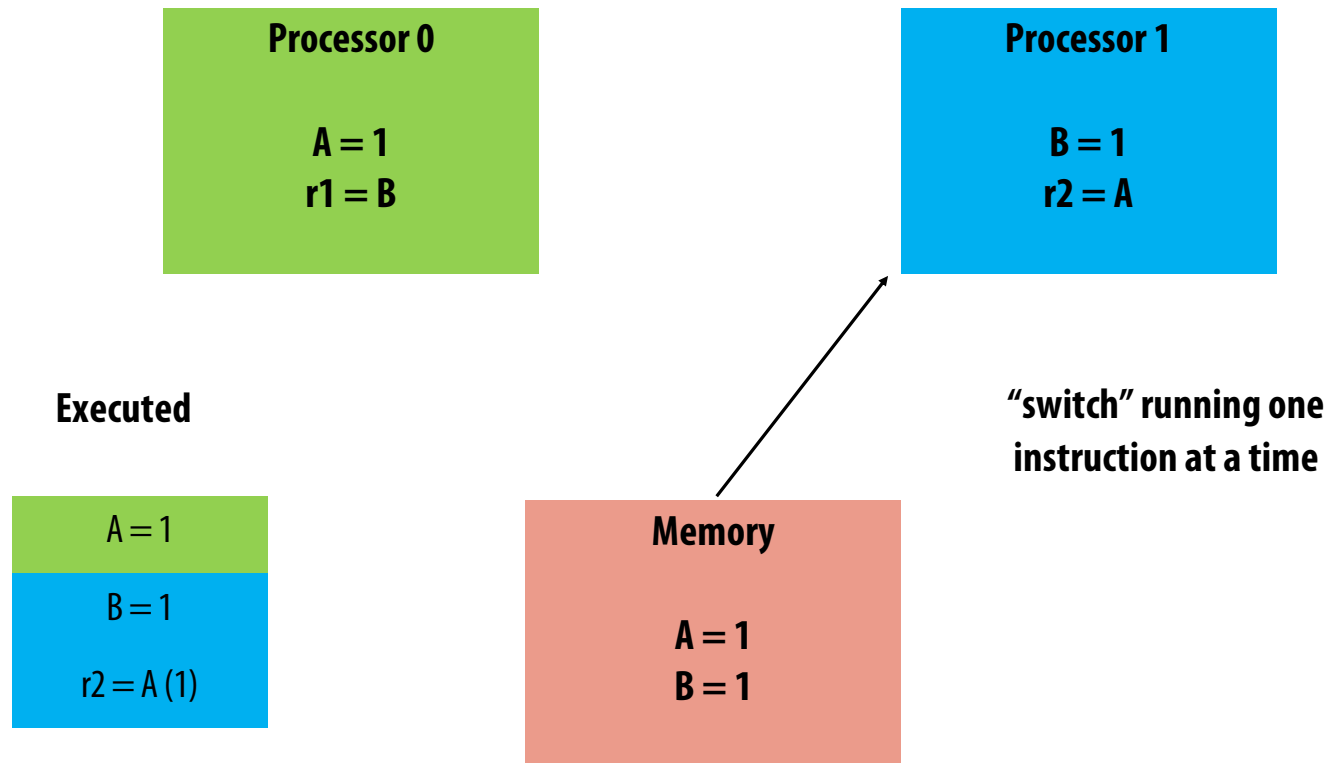
# Sequential Consistency Example



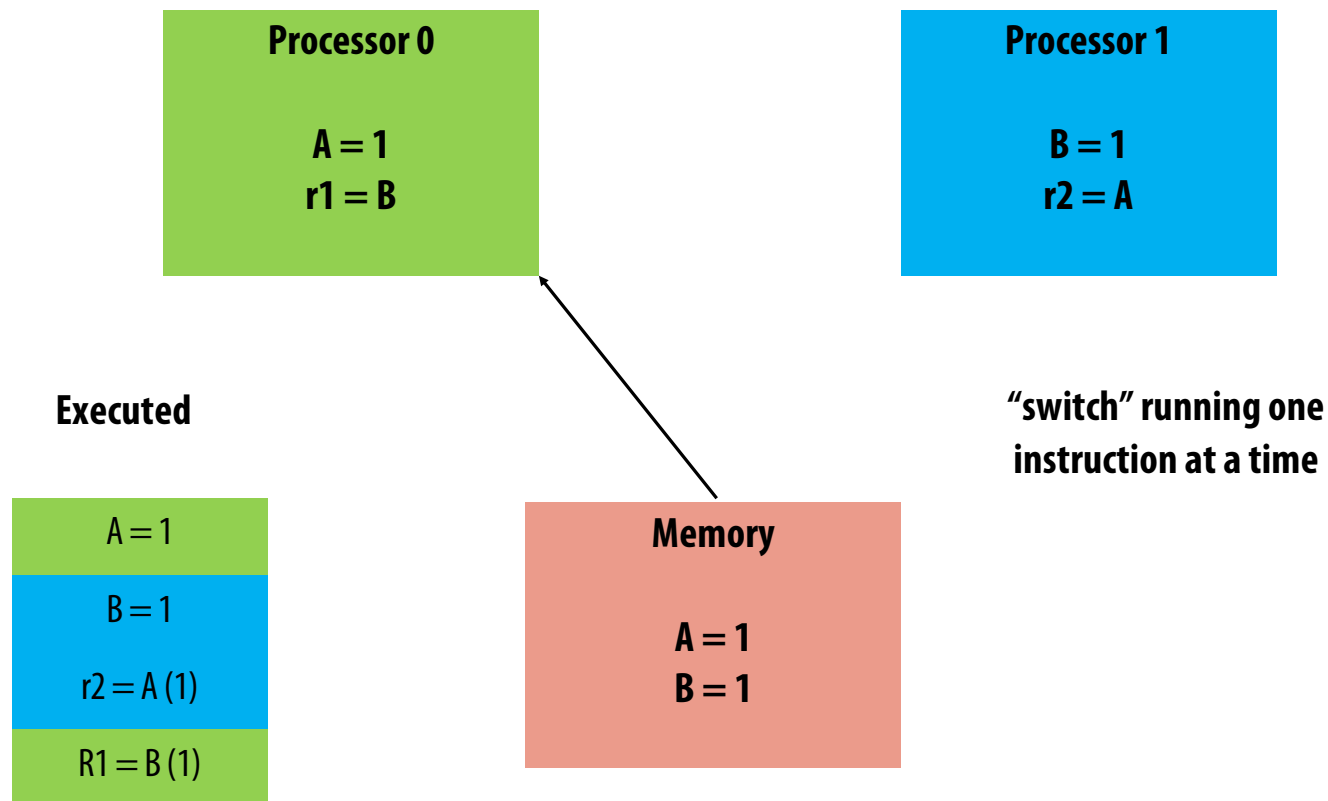
# Sequential Consistency Example



# Sequential Consistency Example



# Sequential Consistency Example



# Relaxing memory operation ordering

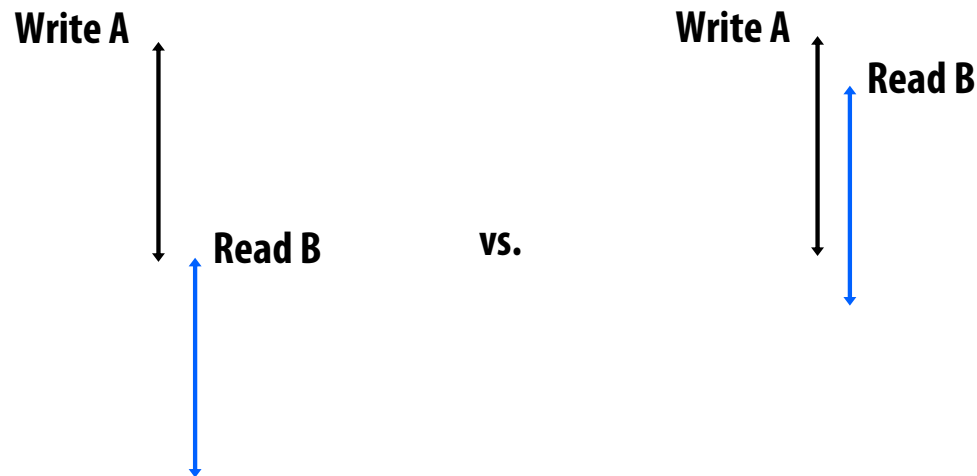
A sequentially consistent memory system maintains all four memory operation orderings ( $W_X \rightarrow R_Y$ ,  $R_X \rightarrow R_Y$ ,  $R_X \rightarrow W_Y$ ,  $W_X \rightarrow W_Y$ )

Relaxed memory consistency models allow certain orderings to be violated

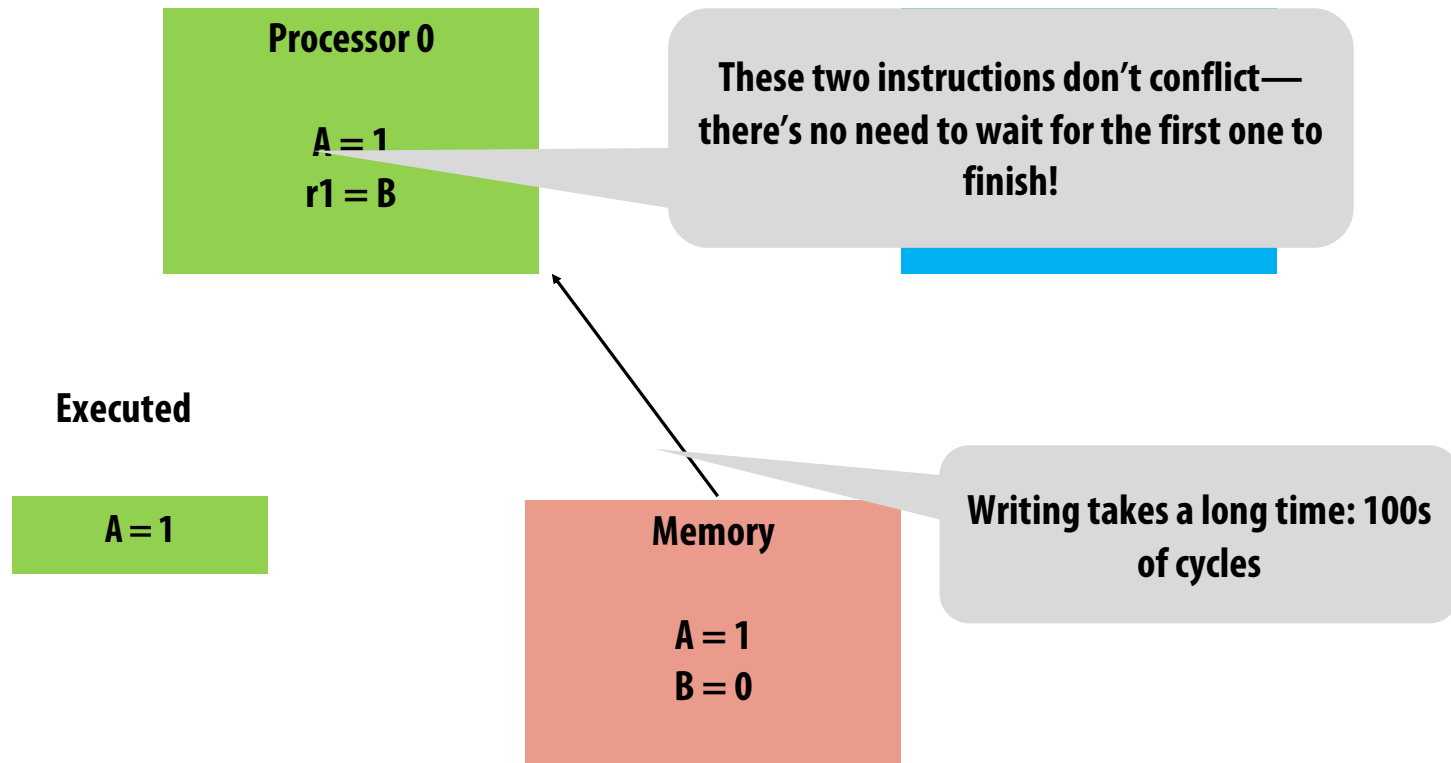
# Motivation for relaxed consistency: hiding latency

## Why are we interested in relaxing ordering requirements?

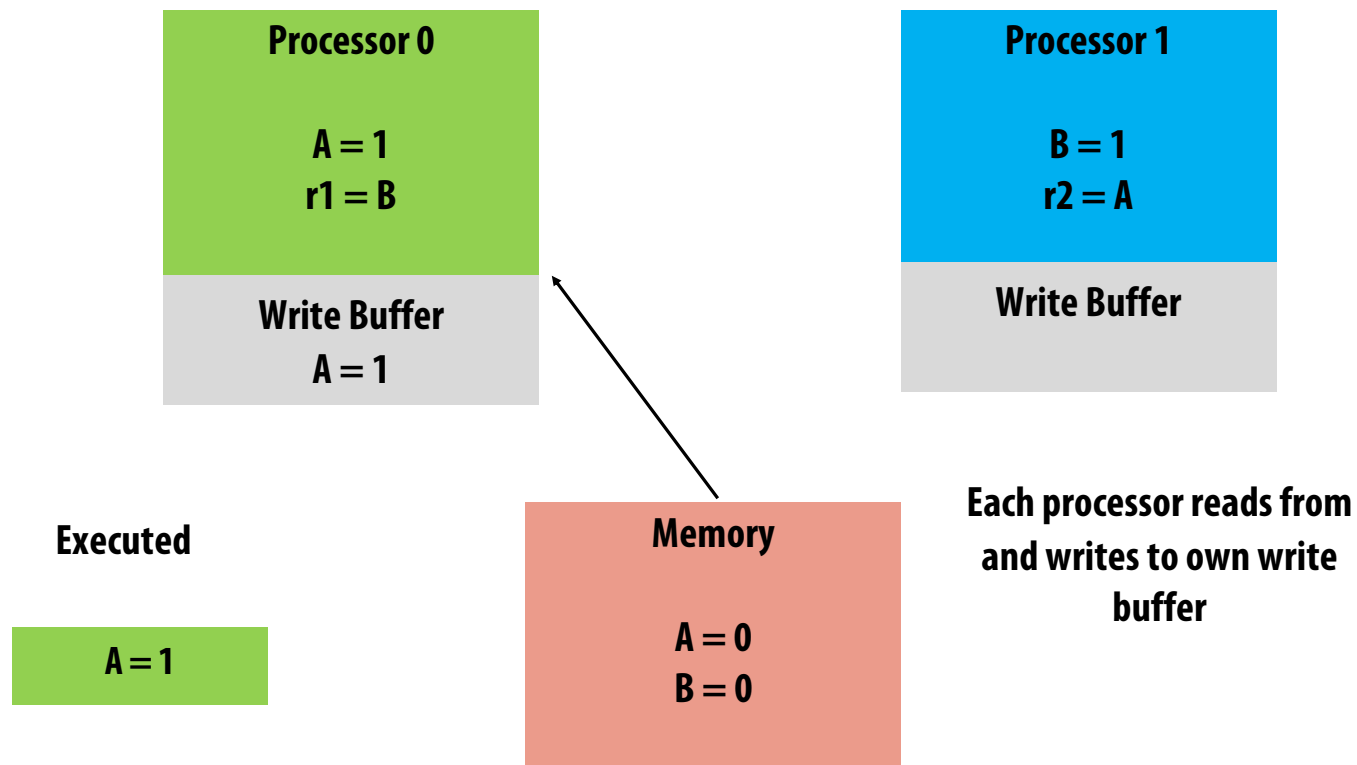
- To gain performance
- Specifically, hiding memory latency: overlap memory access operations with other operations when they are independent
- Remember, memory access in a cache coherent system may entail much more work than simply reading data from memory (finding data, sending invalidations, etc.)



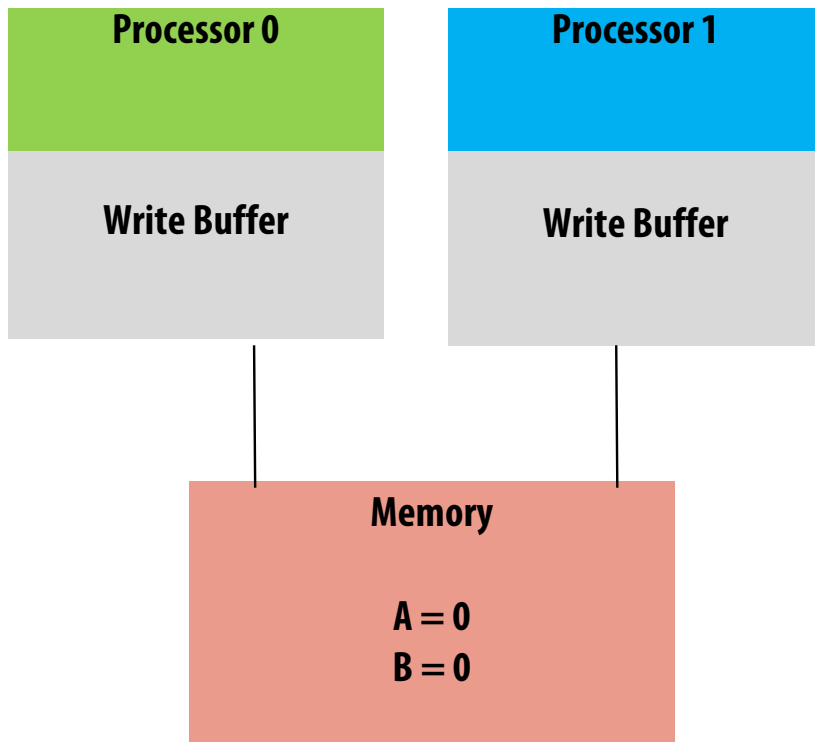
# Problem with SC



# Optimization: Write Buffer



# Write Buffers Change Memory Behavior



**Initially  $A = B = 0$**

**Proc 0**

**(1)  $A = 1$**

**(2)  $r1 = B$**

**Proc 1**

**(3)  $B = 1$**

**(4)  $r2 = A$**

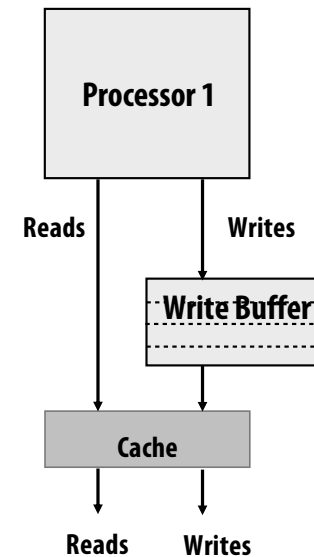
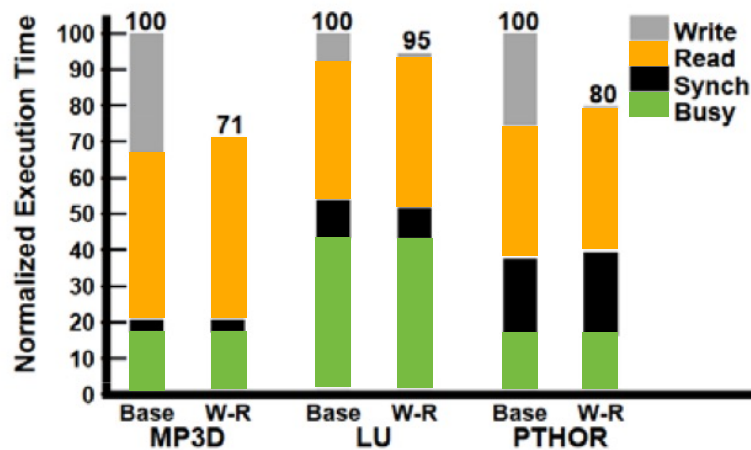
---

**Can  $r1 = r2 = 0$ ?**

**SC: No**

**Write buffers:**

# Write Buffer Performance



**Base:** Sequentially consistent execution. Processor issues one memory operation at a time, stalls until completion

**W-R:** relaxed  $W \rightarrow R$  ordering constraint (write latency almost fully hidden)

# Write Buffers: Who Cares?

**Performance improvement**

**Every modern processor uses them**

- **Intel x86, ARM, RISC-V**

**Need a weaker memory model**

- **TSO: Total Store Order**
- **Slightly harder to reason about than SC**
- **x86 uses an incompletely specified form of TSO**

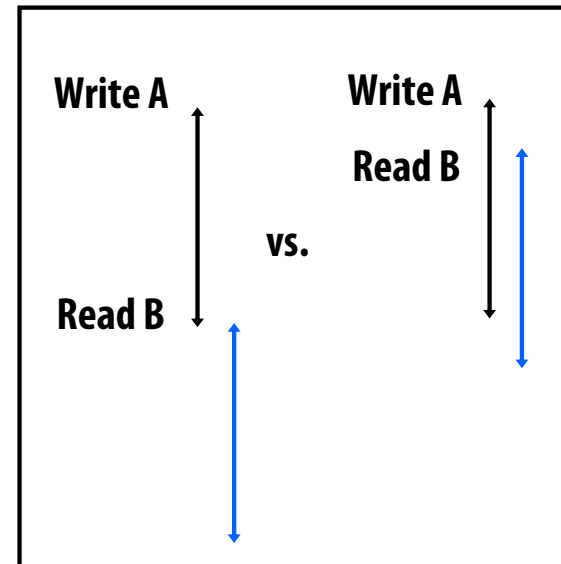
# Allowing reads to move ahead of writes

## Four types of memory operation orderings

- ~~$W_x \rightarrow R_y$~~ : ~~write must complete before subsequent read~~
- $R_x \rightarrow R_y$ : read must complete before subsequent read
- $R_x \rightarrow W_y$ : read must complete before subsequent write
- $W_x \rightarrow W_y$ : write must complete before subsequent write

## Allow processor to hide latency of writes

- Total Store Ordering (TSO)
- Processor Consistency (PC)



# Allowing reads to move ahead of writes

## Total store ordering (TSO)

- Processor P can read B before its write to A is seen by all processors  
(processor can move its own reads in front of its own writes)
- Reads by other processors cannot return new value of A until the write to A is observed by all processors

## Processor consistency (PC)

- Any processor can read new value of A before the write is observed by all processors

In TSO and PC, only  $W_x \rightarrow R_y$  order is relaxed. The  $W_x \rightarrow W_y$  constraint still exists. Writes by the same thread are not reordered (they occur in program order)

# **Clarification (make sure you get this!)**

**The cache coherency problem exists because hardware implements the optimization of duplicating data in multiple processor caches. The copies of the data must be kept coherent.**

**Relaxed memory consistency issues arise from the optimization of reordering memory operations. (Consistency is unrelated to whether or not caches exist in the system)**

# Allowing writes to be reordered

## Four types of memory operation orderings

- ~~$W_x \rightarrow R_y$ : write must complete before subsequent read~~
- $R_x \rightarrow R_y$ : read must complete before subsequent read
- $R_x \rightarrow W_y$ : read must complete before subsequent write
- ~~$W_x \rightarrow W_y$ : write must complete before subsequent write~~

## Partial Store Ordering (PSO)

- Execution may not match sequential consistency on program 1  
(P2 may observe change to `flag` before change to `A`)

### Thread 1 (on P1)

```
A = 1;  
flag = 1;
```

### Thread 2 (on P2)

```
while (flag == 0);  
print A;
```

# Why might it be useful to allow more aggressive memory operation reorderings?

$W_X \rightarrow W_Y$ : processor might reorder write operations in a write buffer (e.g., one is a cache miss while the other is a hit)

$R_X \rightarrow W_Y, R_X \rightarrow R_Y$ : processor might reorder independent instructions in an instruction stream (out-of-order execution)

Keep in mind these are all valid optimizations if a program consists of a single instruction stream

# Allowing all reorderings

## Four types of memory operation orderings

- ~~$W_x \rightarrow R_y$ : write must complete before subsequent read~~
- ~~$R_x \rightarrow R_y$ : read must complete before subsequent read~~
- ~~$R_x \rightarrow W_y$ : read must complete before subsequent write~~
- ~~$W_x \rightarrow W_y$ : write must complete before subsequent write~~

## No guarantees about operations on data!

- Everything can be reordered

## Motivation is increased performance

- Overlap multiple reads and writes in the memory system
- Execute reads as early as possible and writes as late as possible to hide memory latency

## Examples:

- Weak ordering (W0)
- Release Consistency (RC)

# Synchronization to the Rescue

**Memory reordering seems like a nightmare (it is!)**

**Every architecture provides synchronization primitives to make memory ordering stricter**

**Fence (memory barrier) instructions prevent reorderings, but are expensive**

- All memory operations complete before any memory operation after it can begin

**Other synchronization primitives (per address):**

- read-modify-write/compare-and-swap, transactional memory, ...

reorderable reads  
and writes here

...

MEMORY FENCE

...

reorderable reads  
and writes here

...

MEMORY FENCE

# Example: expressing synchronization in relaxed models

## Intel x86/x64 ~ total store ordering

- Provides sync instructions if software requires a specific instruction ordering not guaranteed by the consistency model
  - `mm_lfence` ("load fence": wait for all loads to complete)
  - `mm_sfence` ("store fence": wait for all stores to complete)
  - `mm_mfence` ("mem fence": wait for all mem operations to complete)

## ARM processors: very relaxed consistency model

A cool post on the role of memory fences in x86:

<http://bartoszmilewski.com/2008/11/05/who-ordered-memory-fences-on-an-x86/>

ARM has some great examples in their programmer's reference:

[http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier\\_Litmus\\_Tests\\_and\\_Cookbook\\_A08.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier_Litmus_Tests_and_Cookbook_A08.pdf)

A great list of academic papers:

<http://www.cl.cam.ac.uk/~pes20/weakmemory/>

# **Problem: Data Races**

**Every example so far has involved a data race**

- Two accesses to the same memory location**
- At least one is a write**
- Unordered by synchronization operations**

# Conflicting data accesses

**Two memory accesses by different processors conflict if...**

- They access the same memory location
- At least one is a write

## Unsynchronized program

- Conflicting accesses not ordered by synchronization (e.g., a fence, operation with release/acquire semantics, barrier, etc.)
- Unsynchronized programs contain data races: the output of the program depends on relative speed of processors (non-deterministic program results)

# Synchronized Programs

**Synchronized programs yield SC results on non-SC systems**

- Synchronized programs are data-race-free

**If there are no data races, reordering behavior doesn't matter**

- Accesses are ordered by synchronization, and synchronization forces sequential consistency

**In practice, most programs you encounter will be synchronized (via locks, barriers, etc. implemented in synchronization libraries)**

- Rather than via ad-hoc reads/writes to shared variables like in the example programs

# Summary: Relaxed Consistency

**Motivation: obtain higher performance by allowing reordering of memory operations (reordering is not allowed by sequential consistency)**

**One cost is software complexity: programmer or compiler must correctly insert synchronization to ensure certain specific operation orderings when needed**

- **But in practice complexities encapsulated in libraries that provide intuitive primitives like lock/unlock, barrier (or lower-level primitives like fence)**
- **Optimize for the common case: most memory accesses are not conflicting, so don't design a system that pays the cost as if they are**

**Relaxed consistency models differ in which memory ordering constraints they ignore**

# Languages Need Memory Models Too

Thread 1

```
X = 0  
for i=0 to 100:  
  X = 1  
  print X
```



Thread 1

```
X = 1  
for i=0 to 100:  
  print X
```

# Languages Need Memory Models Too

Optimization not visible to programmer

Thread 1

```
X = 0
for i=0 to 100:
    X = 1
    print X
```

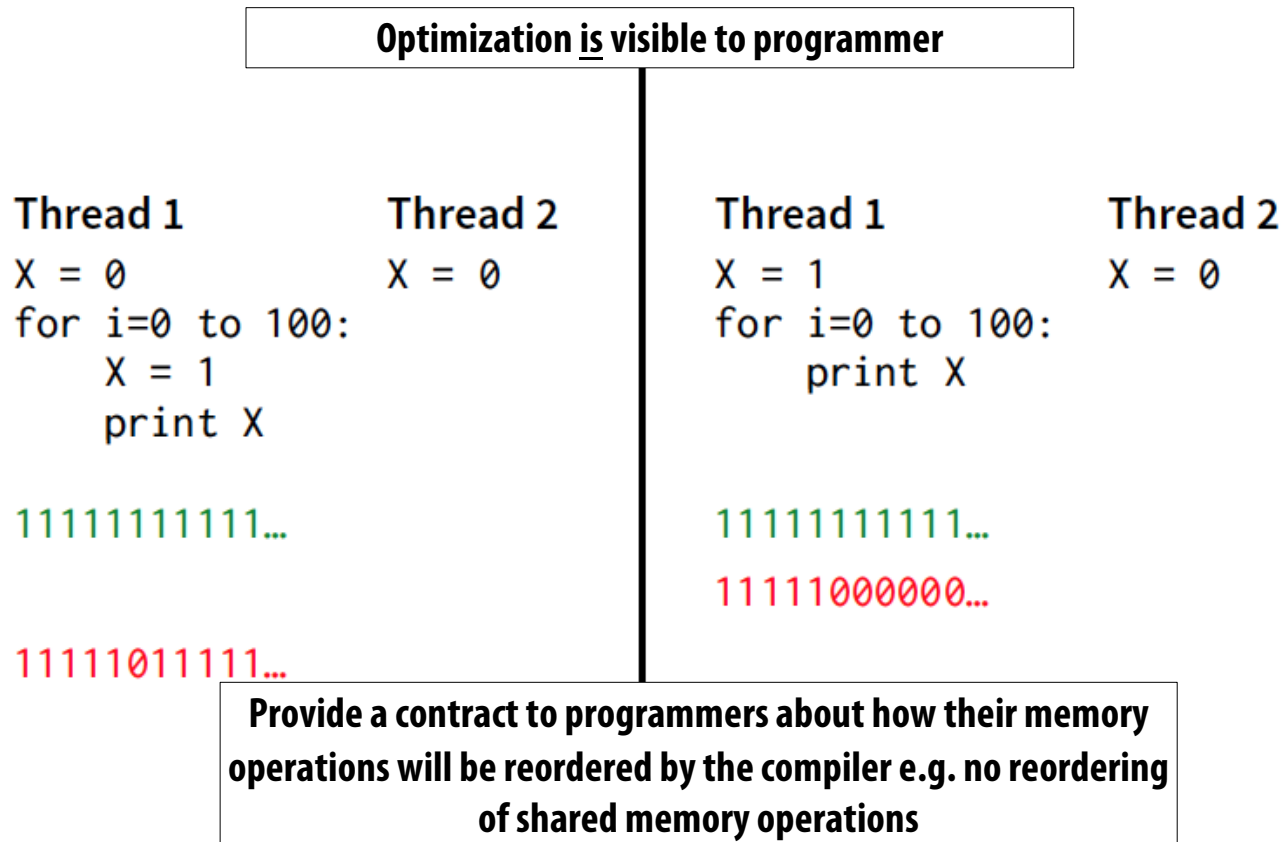
1111111111...

Thread 1

```
X = 1
for i=0 to 100:
    print X
```

1111111111...

# Languages Need Memory Models Too



# Language Level Memory Models

**Modern (C11, C++11) and not-so-modern (Java 5) languages guarantee sequential consistency for data-race-free programs (“SC for DRF”)**

- Compilers will insert the necessary synchronization to cope with the hardware memory model**

**No guarantees if your program contains data races!**

- The intuition is that most programmers would consider a racy program to be buggy**

**Use a synchronization library!**

# Memory Consistency Models Summary

**Define the allowed reorderings of memory operations by hardware and compilers**

**A contract between hardware or compiler and application software**

**Weak models required for good performance?**

- **SC can perform well with many more resources**

**Details of memory model can be hidden in synchronization library**

- **Requires data race free (DRF) programs**