Parallel Computer Architecture

Ryan Hou

Invalid Date

Table of contents

Preface

My notes on ????.

Resources

Some relevant resources:

- Resource Name

Textbooks:

- Book 1

1 Introduction

1.1 Perspective

Note 1: Definition - Some definition

Term is defined as blah blah blah...

This note does ...

1.2 High Level Ideas

2 Parallel Programming Models

Programming Model Elements

- Processes and Threads
  - Process
  - Thread
  - Task
- Communication
- Synchronization

2.1 Message Passing

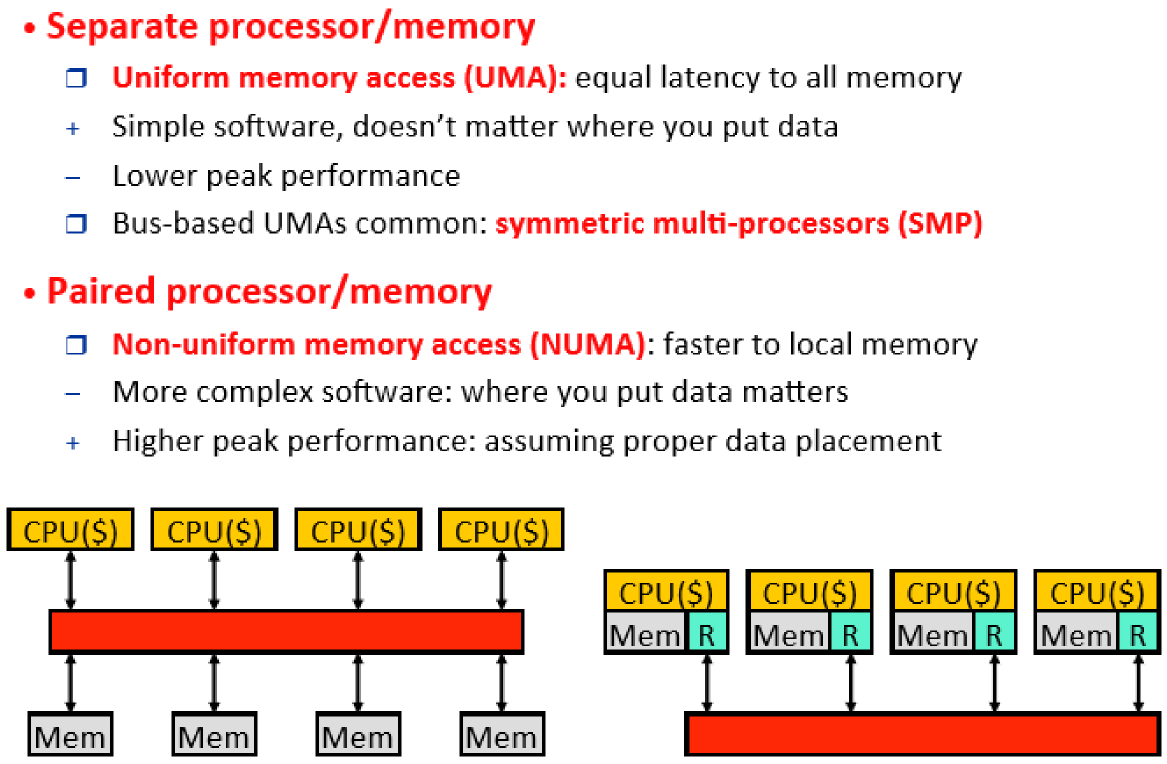
- Each process has their own local address space
- Cannot directly access memory of another node
- Use explicit send/receive operations

2.1.1 Synchronous vs Asynchronous

2.2 Shared Memory

- Multiple execution contexts shares single address space
- Communicate via loads and stores to shared memory

2.2.1 Paired vs Separate Processor/Memory



- **Separate processor/memory**
  - ❐ **Uniform memory access (UMA):** equal latency to all memory
  - + Simple software, doesn't matter where you put data
  - – Lower peak performance
  - ❐ Bus-based UMAs common: **symmetric multi-processors (SMP)**
- **Paired processor/memory**
  - ❐ **Non-uniform memory access (NUMA)**: faster to local memory
  - – More complex software: where you put data matters
  - + Higher peak performance: assuming proper data placement

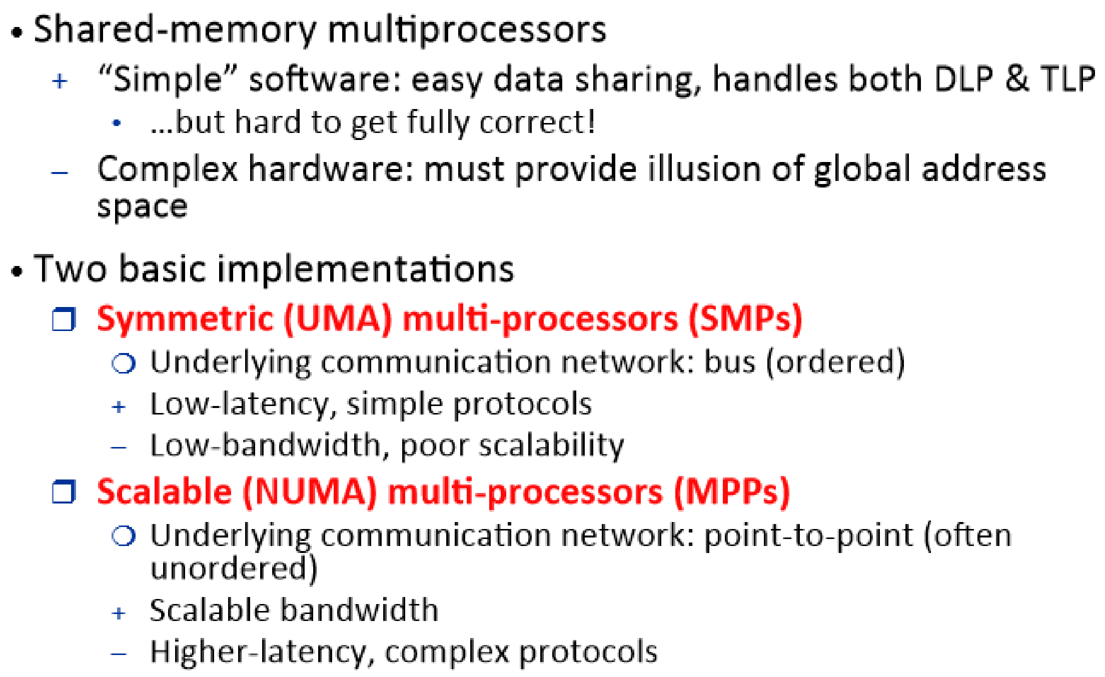
•

2.3 Uniform Memory Access (UMA)

•

2.4 Non-Uniform Memory Access (NUMA)

2.4.1 Shared Memory - Summary



- Shared-memory multiprocessors
  - + "Simple" software: easy data sharing, handles both DLP & TLP
    - ...but hard to get fully correct!
  - – Complex hardware: must provide illusion of global address space
- Two basic implementations
  - ❒ Symmetric (UMA) multi-processors (SMPs)
    - ❍ Underlying communication network: bus (ordered)
    - + Low-latency, simple protocols
    - – Low-bandwidth, poor scalability
  - ❒ Scalable (NUMA) multi-processors (MPPs)
    - ❍ Underlying communication network: point-to-point (often unordered)
    - + Scalable bandwidth
    - – Higher-latency, complex protocols

3 DLP, GPUs

Data-Level Parallelism -

3.1 Vector Architectures

3.2 GPUs

4 Synchronization

5 Cache Coherence

Cache coherence ensure that all processors in a system sees a consistent view of memory, even when multiple caches store copies of the same data.

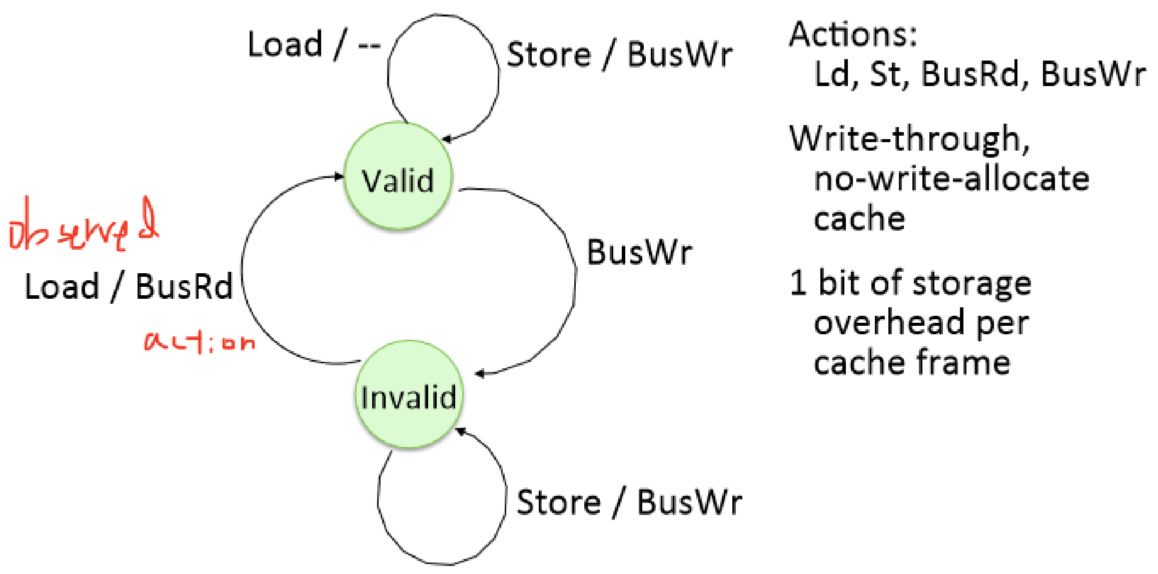
Coherence can be thought of as two invariants:

- Single-Writer Multiple Readers (SWMR)
  - At any (logical) time, there is either only one writer or zero or more readers of a cache line
  - i.e. SWMR enforces that there is only one writer (if there is a writer at all) and that there can be as many readers when there is no writer
- Data Value Invariant (DVI)
  - All cores see the values of the address/line update in the same order
  - Everybody needs to agree on the ordering of the changes on the cache line

Snooping Cache-Coherence Protocols

- The bus provides serialization point
- Each cache controller snoops all bus transactions
  - then takes action to ensure coherence

5.1 Valid-Invalid Snooping Protocol

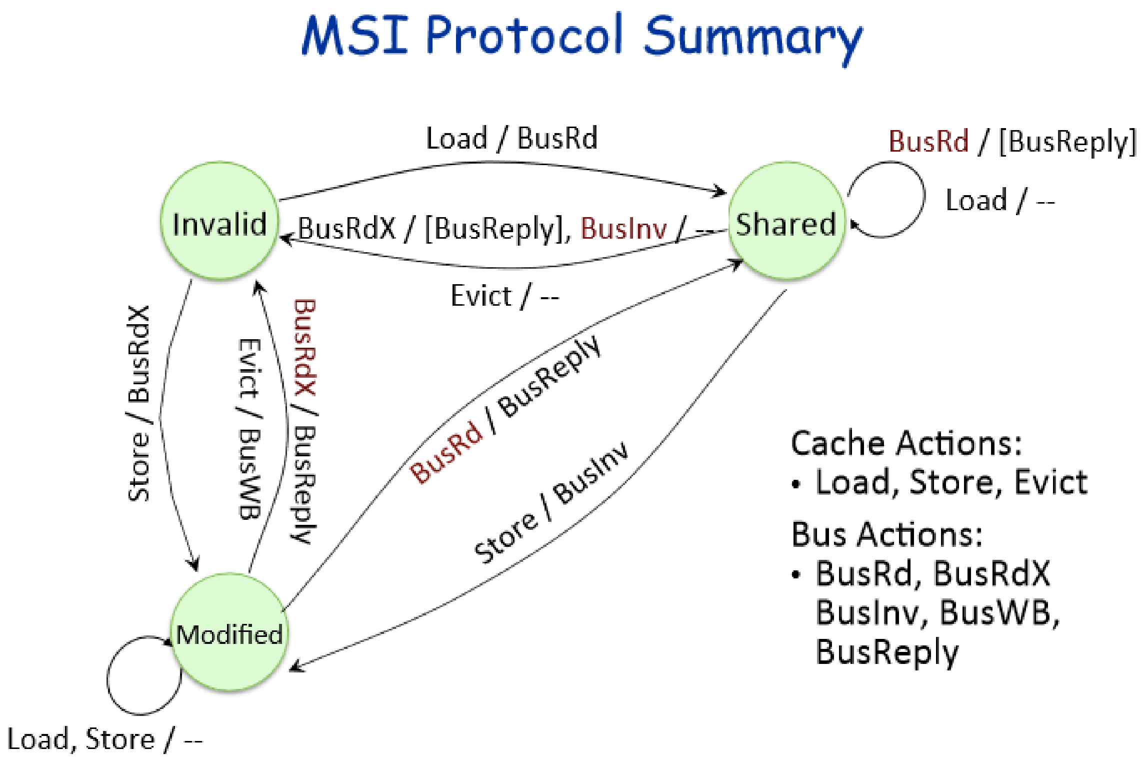


Actions:
Ld, St, BusRd, BusWr

Write-through, no-write-allocate cache

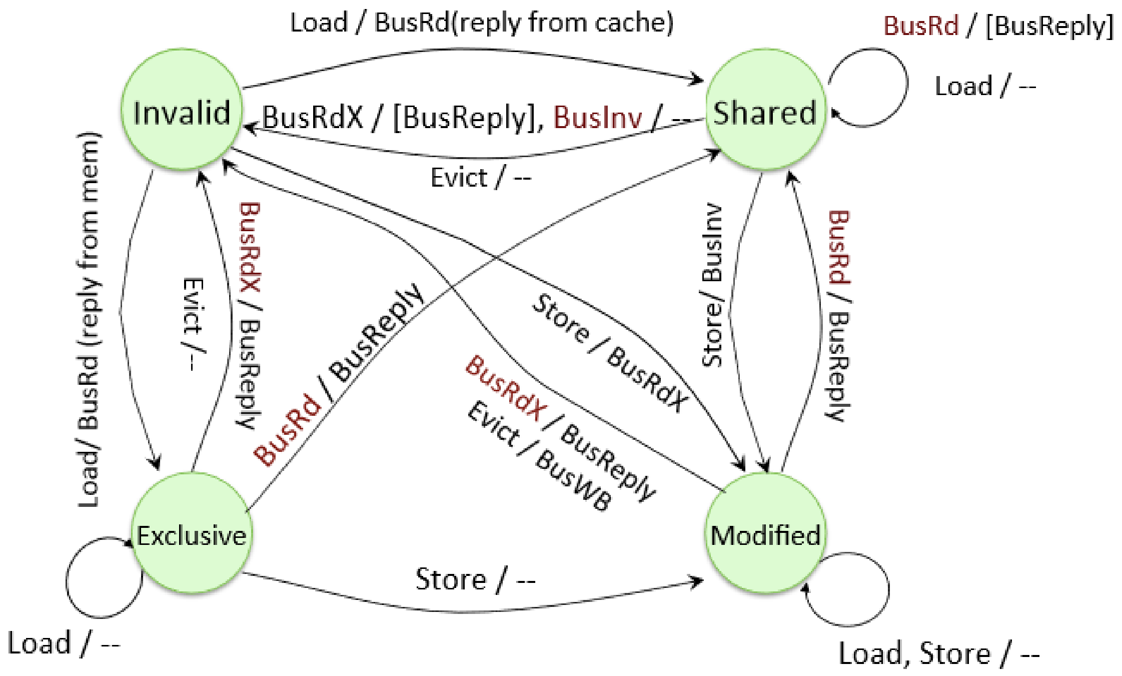
1 bit of storage overhead per cache frame

5.2 MSI Protocol



5.3 MESI Protocol

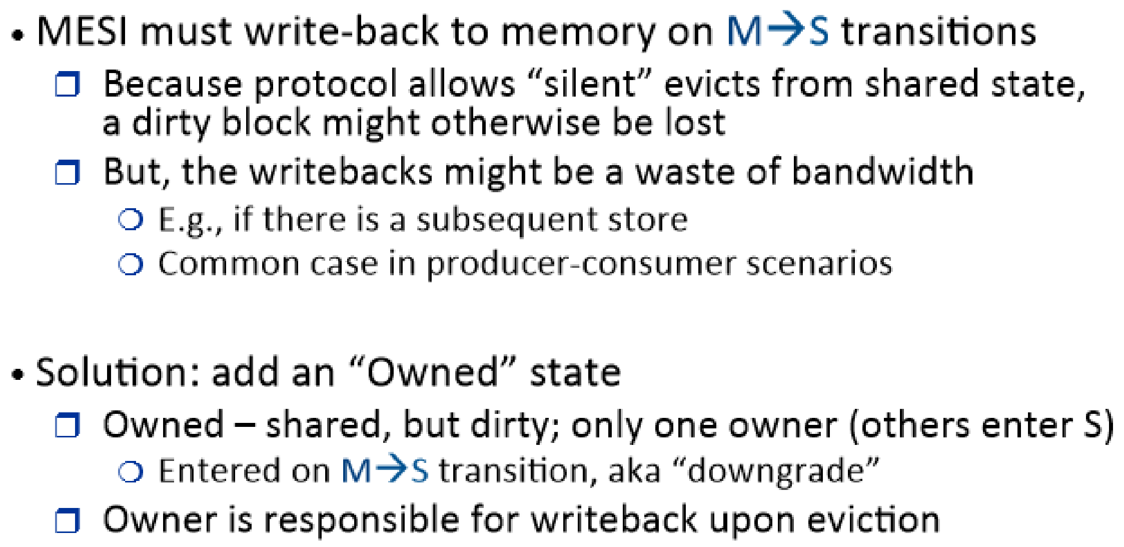
- MSI suffers from frequent read-upgrade sequences
  - leads to two bus transactions
- Solution: Add exclusive state
  - Exclusive - only one copy; writable; clean
  - Stores make it transition to modified to indicate data is dirty
  - We'll be able to read with "write" permissions so the line is clean and we haven't written to it yet



To sum:

- Reduce unnecessary writebacks
  - Faster S->M transition if there are no other readers
    - In MSI, S->M requires sending invalidates
  - If we're the exclusive reader E, can directly change to M state on a store
- Reduce bus traffic (as above)

5.4 MOESI Protocol



- MESI must write-back to memory on M→S transitions
  - Because protocol allows "silent" evicts from shared state, a dirty block might otherwise be lost
  - But, the writebacks might be a waste of bandwidth
    - E.g., if there is a subsequent store
    - Common case in producer-consumer scenarios
- Solution: add an "Owned" state
  - Owned – shared, but dirty; only one owner (others enter S)
    - Entered on M→S transition, aka "downgrade"
  - Owner is responsible for writeback upon eviction

- M->S requires writing the modified value back to memory
  - Writebacks to mem may be unnecessary
- With an O state, cache can supply modified data to other caches, avoiding memory writeback
  - On eviction, Owner is responsible for writeback
- Going to O vs M
  - We go to O when another processor wants to get data (down-grade from M)

[Sweazey & Smith ISCA86]

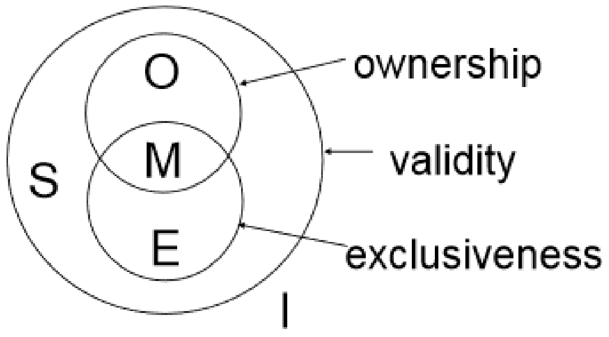
M - Modified (dirty)

O - Owned (dirty but shared) WHY?

E - Exclusive (clean unshared) only copy, not dirty

S - Shared

I - Invalid



Variants

- ❐ MSI
- ❐ MESI
- ❐ MOSI
- ❐ MOESI

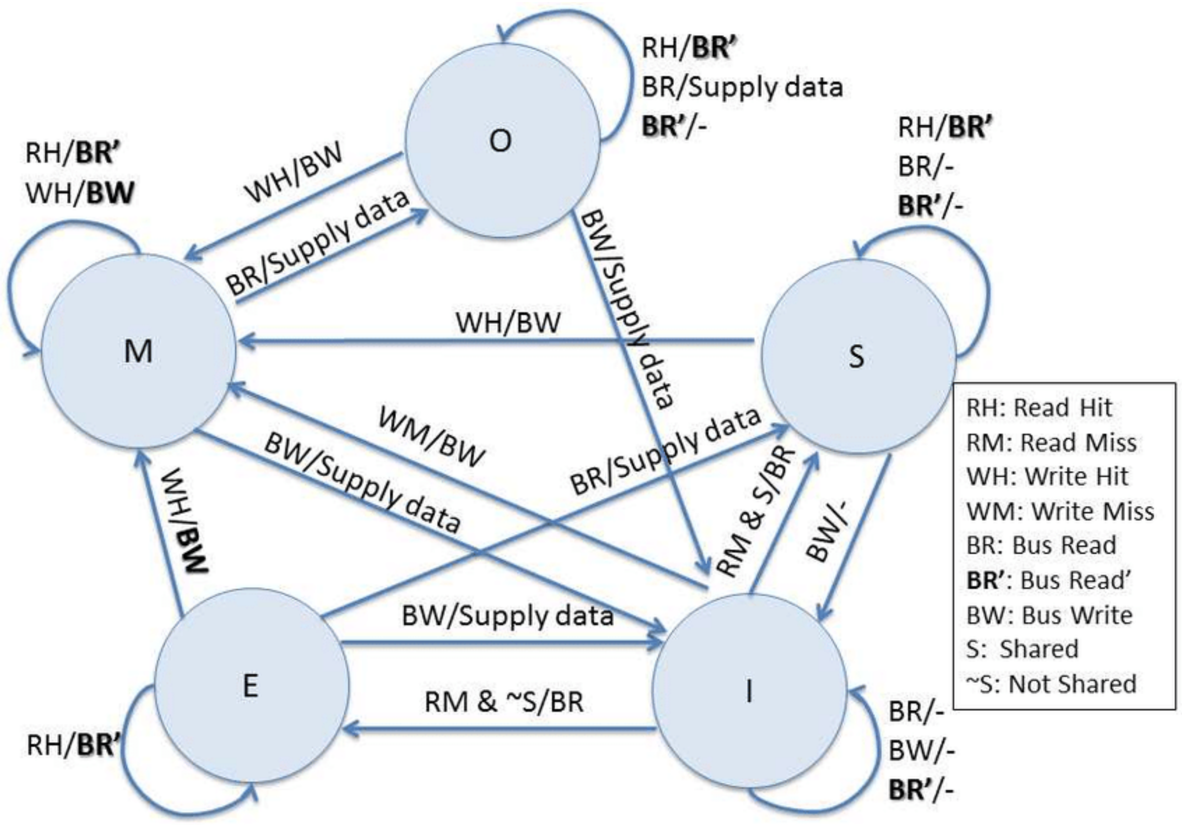


Figure 5.1: Source

5.5 MOSI

- O(wned): The block is valid, owned, and potentially dirty, but not exclusive. The cache has a read-only copy of the block and must respond to requests for the block. Other caches may have a read-only copy of the block, but they are not owners. The copy of the block in the LLC/memory is potentially stale.
- E(xclusive): The block is valid, exclusive, and clean. The cache has a read-only copy of the block. No other caches have a valid copy of the block, and the copy of the block in the LLC/memory is up-to-date. In this primer, we consider the block to be owned when it is in

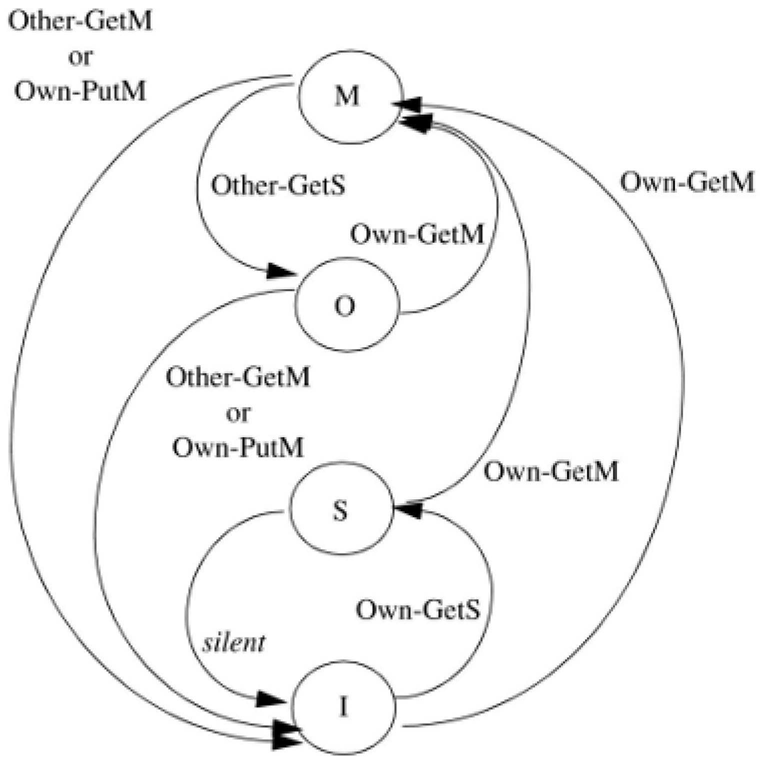


FIGURE 7.6: MOSI: Transitions between stable states at cache controller

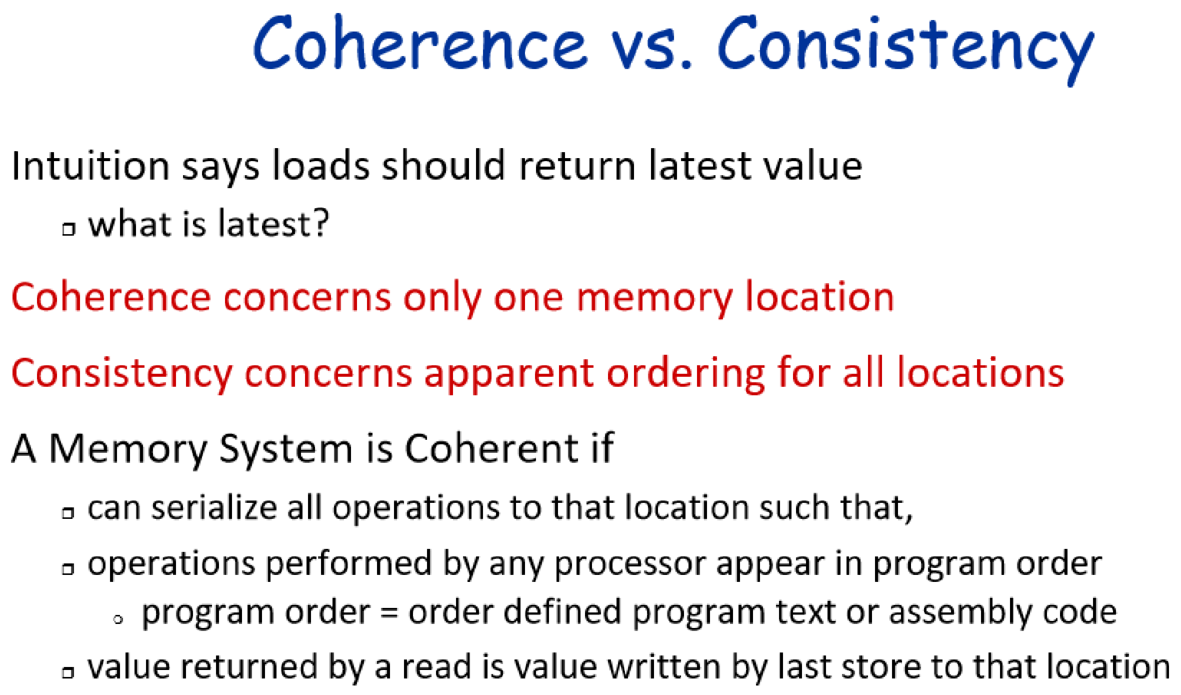
5.6 DEC Firefly Protocol

5.7 Implementing Snoopy Coherent SMPs

5.8 Directory-Based Coherenece

- Why? Bus-based coherence doesn't scale well

6 Memory Consistency



Coherence vs. Consistency

Intuition says loads should return latest value

- what is latest?

Coherence concerns only one memory location

Consistency concerns apparent ordering for all locations

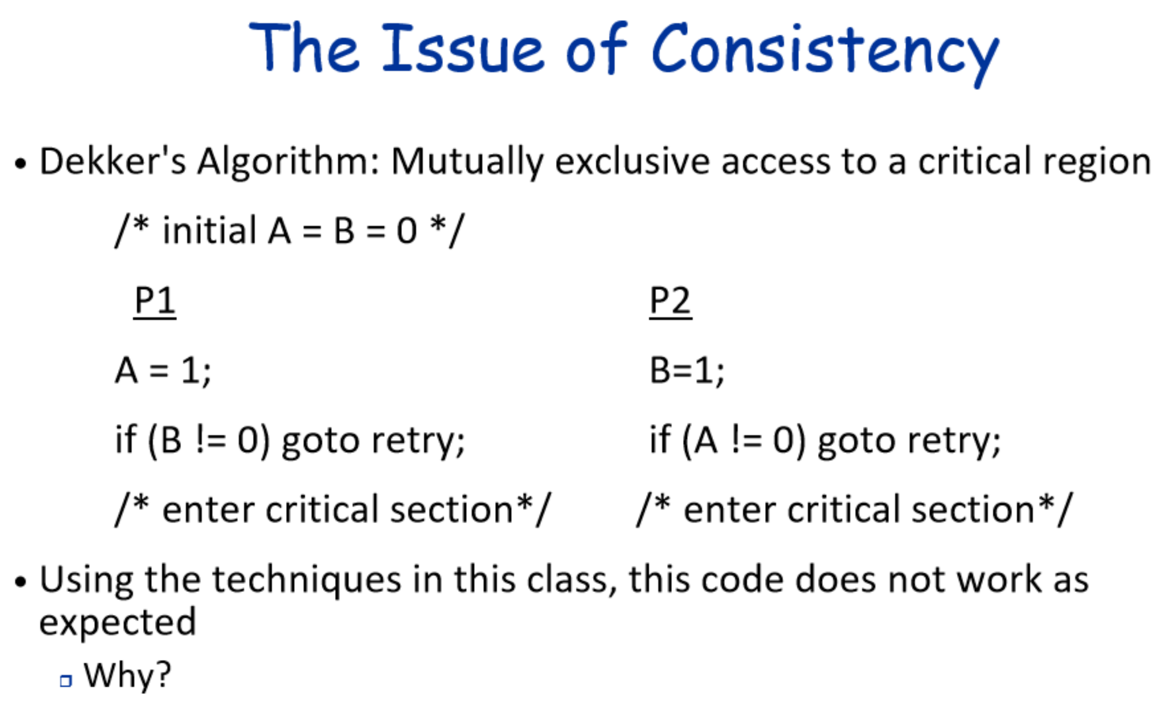
A Memory System is Coherent if

- can serialize all operations to that location such that,
- operations performed by any processor appear in program order
  - program order = order defined program text or assembly code
- value returned by a read is value written by last store to that location

While cache coherence ensures that multiple copies of a single memory location are consistent across caches, memory consistency defines how updates to different memory locations appear across processors.

Why Does Memory Consistency Matter?

- In a single-core processor, instructions execute sequentially, so memory operations follow a strict order. But in multi-core/multiprocessor systems, things get more complex:
- Multiple processors execute memory operations in parallel.
- Caches and store buffers can hold writes before committing to memory.
- Memory itself is a shared resource with limited bandwidth, and requests from different processors may arrive at the same time.
- Since memory can only commit one operation at a time per memory location, different processors may see memory updates in different orders.



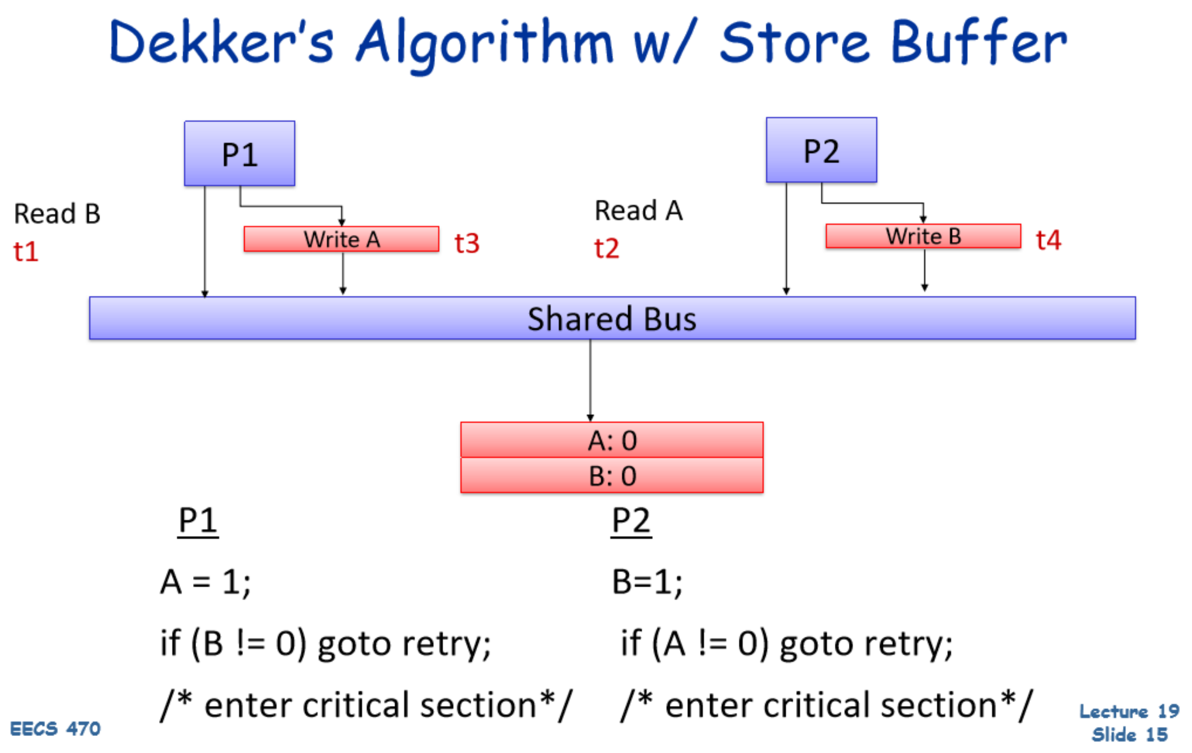
The Issue of Consistency

- Dekker's Algorithm: Mutually exclusive access to a critical region

```
/* initial A = B = 0 */

 P1                              P2

A = 1;                           B=1;

if (B != 0) goto retry;          if (A != 0) goto retry;

/* enter critical section*/     /* enter critical section*/
```

- Using the techniques in this class, this code does not work as expected
    - Why?

- "goto retry" means starting back at initial A = B = 0
- When we issue stores, memory may not be able to accept them right away
    - So this may allow loads to bypass stores that aren't completed yet
    - Since the store's aren't going to the same address as the loads, the values won't be forwarded
    - So we could be doing loads before the store actually updates memory



Dekker's Algorithm w/ Store Buffer

P1	P2
A = 1;	B=1;
if (B != 0) goto retry;	if (A != 0) goto retry;
/* enter critical section*/	/* enter critical section*/

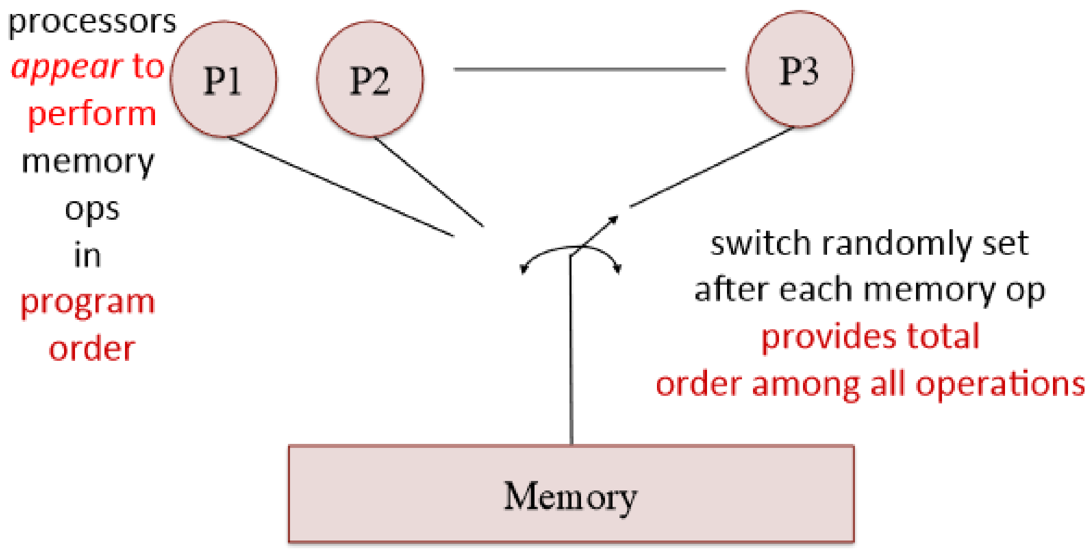
EECS 470

Lecture 19
Slide 15

6.1 Consistency Models

Memory consistency model specifies the order in which memory accesses may be performed by one thread, and the order they become visible to other threads in the program.

6.2 Sequential Consistency


processors *appear* to perform memory ops in program order
P1
P2
P3
switch randomly set after each memory op provides total order among all operations
Memory

A system is sequentially consistent if the result of any execution is the same as if all operations were executed in some sequential order, and the operations of each processor appear in that order.

Key Properties:

- Global Total Order: All memory operations appear to execute in a single, global sequence
- Program order preserved: each processor must execute its own operations in program order
- Interleaving across processors: The global order is an interleaving of memory operations from all processors, ensuring no reordering of loads and stores across different processors

Problems:

- Difficult to implement efficiently in HW
- Unnecessarily restrictive
  - Most parallel programs won't notice out-of-order accesses
- Conflicts with latency hiding techniques

6.3 Relaxed Consistency

Allow:

- Reordering of memory operations (under controlled conditions).
- Store buffering and speculative execution (to optimize performance).
- Explicit synchronization primitives (e.g., memory fences, atomic operations) to enforce ordering only where needed.

6.3.1 Total Store Order (TSO)

Loads can be reordered after stores, but stores appear in order to all processors

6.3.2 Processor Consistency

Writes from a single processor appear in order, but different processors may see updates in different orders.

6.3.3 Partial Store Order

- Stores can be reordered with other stores, but
- Loads still execute in order relative to each other (other loads)

6.3.4 Relaxed Memory Order

- Loads to be reordered with loads.
- Stores to be reordered with stores.
- Loads and stores to be reordered with each other.

6.3.5 Weak Ordering

Memory accesses can be reordered arbitrarily, but a synchronization operation (e.g., fence/barrier) must be issued to enforce ordering.

6.3.6 Release Consistency

- Acquire operation (before a critical section) ensures visibility of all previous writes.
- Release operation (after a critical section) ensures that writes are visible before allowing others to enter.

6.4 Examples

6.4.1 Basic

```
// Processor P1
X = 1;   // (A)
Y = 1;   // (B)

// Processor P2
r1 = Y;  // (C)
r2 = X;  // (D)
```

Since both processors are executing independently, different memory models allow different interleavings of these operations:

- Sequential Consistency (SC) ensures a single global order.
  - If P2 sees r1 = 1 (because Y=1 was written at (B)), then it must see r2 = 1 (because X=1 at (A) must have happened first).
- Relaxed Consistency (e.g., RMO, PSO) allows reordering.
  - r1 = 1 but r2 = 0 is possible if the store to Y (B) became visible before the store to X (A).

7 Interconnection Networks

8 Summary

In summary...

References