19 Multiprocessor Consistency and Coherence

18-548/15-548 Memory System Architecture

Philip Koopman

November 18, 1998

(Based on a lecture by LeMonté Green)

Required Reading: **[Cragon]:** **Chapter 4**

Recommended Reading:

[H&P]:	**Chapter 8**
[Adve 96]:	**"Shared Memory Consistency Models: A Tutorial"**
[Lenoski 90]:	**"The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor"**
[Schimmel]:	**pp. 59-68, 83-87, 99-104, Chapter 15**

Carnegie Mellon

Assignments

- **By next class read about Fault Tolerance:**
  - Cragon pp. 278-283
  - Siewiorek & Swarz handouts
  - Supplemental reading:
    - Hennessy & Patterson 6.5
    - Koopman & Siewiorek 5.7
    - IBM Tech. Note: Fault Tolerance and DRAMS
- **Homework 11 due *Monday*, November 30**
- **Test #3 Wednesday December 2**
  - In-class review Monday November 30
  - More like test #2 than test #1 (*i.e.*, system-level, multi-concept problems)

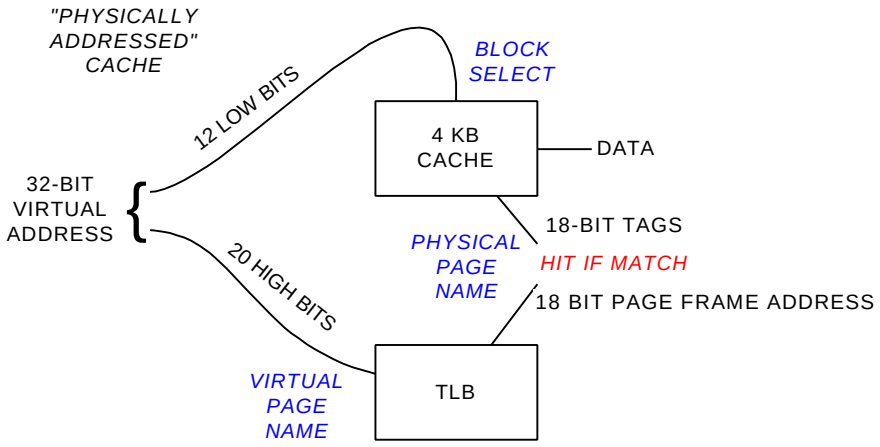
Preview

- **Virtual Caches**
  - Design issues and solutions of virtual caches
- **Multiprocessor Consistency**
  - *When* does a memory write show up at another CPU?
  - A programming model
- **Multiprocessor Coherence**
  - *How* are memory accesses coordinated among CPUs?
  - A mechanism
- **Performance & Software**
  - Shared resources and spin locks
  - Cache aligning data structures

VIRTUAL CACHES

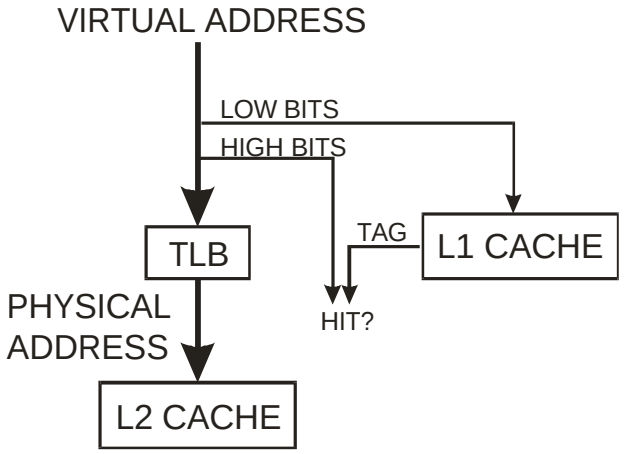
Refresher: Limit to Physical Caches

- **Remember the physically addressed cache?**
  - Number of untranslated address bits limited cache size
  - TLB in critical path for determining hit/miss, but could be done concurrently



Virtual Cache -- Unconstrained L1 Size

- **L1 cache addressed with virtual address alone**
- **TLB operates to convert to physical addresses for L2 cache and beyond**
  - L1 cache size is not constrained -- good idea for L1 I-cache especially
  - Address translation only required on L1 cache miss

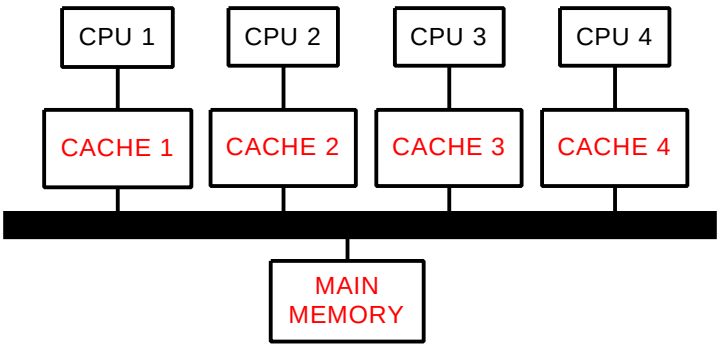


Problems	Solutions
◆ **Ambiguity** • One virtual address refers to 2 different physical locations • How? – memory manager performs remapping – context switching	◆ **OS flushes caches** • On context switch • On "unsafe" operations • But, creates a cold cache
◆ **Alias** • More than one virtual address is used to refer to the same physical memory location • How? – context switching – 2 processes using different addresses to refer to same shared memory location	◆ **Include process ID with tags** • Solves all but intra-process problems ◆ **Check physical address after the fact** • Take an exception if there's a problem

MULTIPROCESSOR CACHE CONSISTENCY & COHERENCE

Multiple Processors

- **Simple Multiprocessor has multiple CPUs on a single bus**
  - Global memory address space with multiple threads working on a single problem
  - Caches used not only to improve latency, but also filter bus traffic
- **Problems:**
  - Consistency -- when does another CPU see a memory update?
  - Coherence -- how do other CPUs see a memory update?



Consistency

- **Consistency addresses WHEN a processor sees an update to memory**
  - If two processors touch a memory location, what happens?
- **Depending on the consistency model, both of the below sequences might execute the conditional statement for zero variable value**
  - The outcome depends on consistency model
  - There is no single "correct" behavior for all machines

CPU 1 Executes:

```
P1:    A = 0;
       .....
       A = 1;
L1:    if (B == 0) ....
```

CPU 2 Executes:

```
P2:    B = 0;
       .....
       B = 1;
L2:    if (A == 0) ....
```

Consistency Models

- **Why not use a strong consistency model?**
  - How are concurrent loads and stores ordered by memory accesses by multiple CPUs
    - Simplest conceptual model is it looks like a multi-tasking single CPU
    - Attempting strong (uniprocessor-like) consistency can cause a global bottleneck -- costs performance
- **"Weak" consistency models are used to improve performance**
  - Permits out-of-order execution within individual CPUs
  - Relaxes latency issues with near-simultaneous accesses by different CPUs
  - Programmer MUST take into account the memory consistency model to create correct software

Sequential Consistency (Strong Ordering)

- **Requirements:**
  - All memory operations appear to execute one at a time
  - All memory operations from a single CPU appear to execute in-order
  - All memory operations from different processors are "cleanly" interleaved with each other (serialization)
    - Delay all memory accesses until invalidates are done.
- **Sequential consistency forces all reads and writes to shared data to be atomic**
  - Once begun, the memory operation can't be interrupted or interfered with
  - Resource is locked and unusable until operation is completed

Spin Locks Under Sequential Consistency

- **Sequential consistency is not a silver bullet…….**
  behavior STILL nondeterministic
  - Data races still can occur due to relative timing of the CPUs
  - Similar situation to single CPU with multiple threads
  - Solution: lock critical resources (shared data). Common to use spin locks of atomic read-modify-write operations (test and set).

```
int test_and_set(volatile int *addr)
{ /* sets address to 1, returns previous value */
  int old_value;
  old_value = swap_atomic(addr, 1);
  return(old_value);
}

void lock(volatile int *lock_status)
{ /* wait until lock is captured */
   while (test_and_set(lock_status) == 1);
}
```

Sequential Consistency Problems

- **Can't use important hardware optimizations**
  - Problem with anything that interferes with strict execution order
    - Write buffers, Write assembly caches, Non-blocking caches…
  - Not a problem with uniprocessors
- **May not be able to use important software optimizations**
  - If you want to be really strict about it, source code must execute as-is, so no:
    - Code motion, register allocation, eliminating common subexpressions…
  - Same problem exists with uniprocessor concurrency
- **Relaxed memory consistency models:**
  - Permit performance optimizations
  - BUT, require programmer to take responsibility for concurrency issues

Total Store Ordering

- **Relaxed Consistency**
  - Stores must complete in-order
  - But, stores need not complete before a read to a given location takes place
- **Allows reads to bypass pending writes.**
  - Store buffers allowed!
  - But, writes MUST exit the store buffer in FIFO order.
- **Problem: Other CPUs don't check the store buffer for data.**
  - So, a read from CPU #2 might not see that data has "already" been changed by CPU #1
  - Synchronization of some sort required before reading potentially shared data

Partial Store Ordering

- **Even more relaxed consistency**
  - Stores to any given memory location complete in-order
  - But, stores to different locations may complete out of order
  - And, stores need not complete before a read to a given location takes place
  - Like total store ordering, but ordering concept applied only on a per-location basis
- **Additional Problem: Spin locks may not work**
  - Modifying a shared variable involves:
    - Writing to the variable's memory location
    - Changing the spin lock value to "available"
    - But, what if the spin lock write completes before the variable write?
  - Solution: hardware must support some sort of barrier synchronization
    - All CPUs wait at barrier until global memory state is synchronized
    - Release spin lock only after barrier synch.

Weak Consistency

- **Really relaxed consistency**
  - Anything goes, except at barrier synchronization points
  - Global memory state must be completely settled at each synchronization
  - Memory state may correspond to *any* ordering of reads and writes between synchronization points
- **Permits fastest execution**
  - But, managing concurrency is entirely the programmer's responsibility

MULTIPROCESSOR CACHE COHERENCE

Cache Coherence

- **Coherence is the hardware protocol that ensures updates to memory locations are propagated**
  - Every write much eventually be accessible via a read (unless over-written first)
  - All reads/writes must support desired consistency model
- **Coherence issue for uniprocessors**
  - DMA changes memory while bypassing cache
- **Coherence for multiprocessors**
  - One CPU may change memory location already cached by another CPU
    - Intentional changes to shared data structures
    - Accidental changes to variables inhabiting the same cache block
  - Shared variables may be used for intentional communication
    - So, coherence protocol performance may matter a lot

Snooping vs. Directory-Based Coherence

- **Snooping Solution (Snoopy Bus):**
  - *(Solution useful for smaller systems, including uniprocessor DMA problem)*
  - Send all requests for data to all processors
    - Processors snoop to see if they have a copy and respond accordingly
    - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
    - But, scaling limited by cache miss & write traffic saturating the bus
  - Dominates for small scale machines (most of the market)
- **Directory-Based Schemes**
  - *(Scalable Multiprocessor solution)*
  - Keep track of what is being shared in a directory
  - Distributed memory => distributed directory (avoids bottlenecks)
  - Send point-to-point requests to processors

Basic Snoopy Protocols

- **Write Invalidate Protocol:**
  - Multiple readers, single writer
  - Write to shared data:
    - An invalidate is sent to all caches which snoop and *invalidate* any copies
  - Read Miss:
    - Write-through: memory is always up-to-date
    - Write-back: force other caches to update copy in main memory, then snoop that value
  - Can use a separate invalidate bus for write traffic

- **Write Broadcast Protocol:**
  - Write to shared data: broadcast on bus, processors snoop, and *update* copies
  - Read miss: memory is always up-to-date
  - Higher bandwidth (transmit data + address), but lower latency for readers
    - From a bandwidth point of view, looks like write-through cache

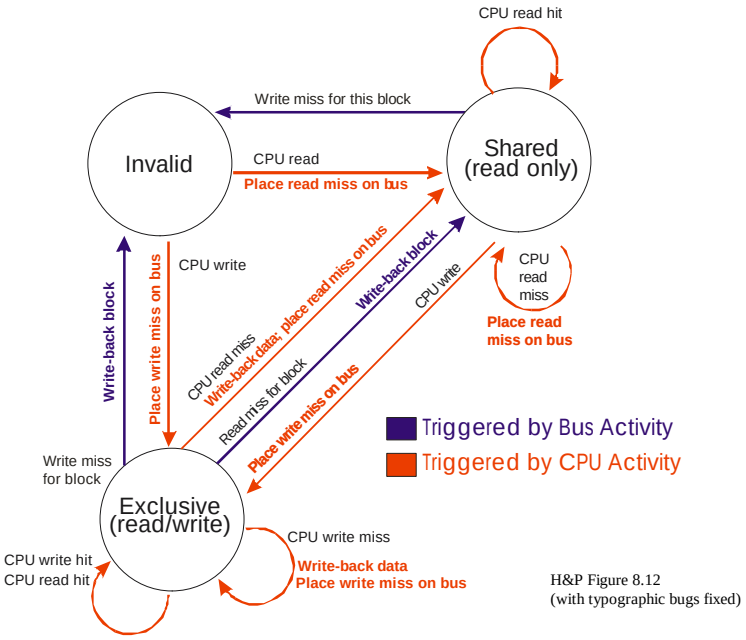
An Example Snoopy Protocol

- **Invalidation protocol, write-back cache**
- **Each block of memory is in one state:**
  - Clean in some subset caches and up-to-date in memory
  - OR Dirty in exactly one cache
  - OR Not in any caches

- **Each cache block is in one state:**
  - Shared: block can be read
  - OR Exclusive: cache has only copy, its writeable, and dirty
  - OR Invalid: block contains no data

- **Read misses: cause all caches to snoop**
- **Writes to clean line are treated as misses**

Snoopy Protocol Example



H&P Figure 8.12
(with typographic bugs fixed)

Snoopy Protocol Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Snoopy Protocol Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Snoopy Protocol Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Snoopy Protocol Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10		10
				Shar.	A1	10	RdDa	P2	A1	10		10
P2: Write 20 to A1												10
P2: Write 40 to A2												10
												10

Assumes A1 and A2 map to same cache block

Snoopy Protocol Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10		10
				Shar.	A1	10	RdDa	P2	A1	10		10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1			10
P2: Write 40 to A2												10
												10

Assumes A1 and A2 map to same cache block

Snoopy Protocol Example

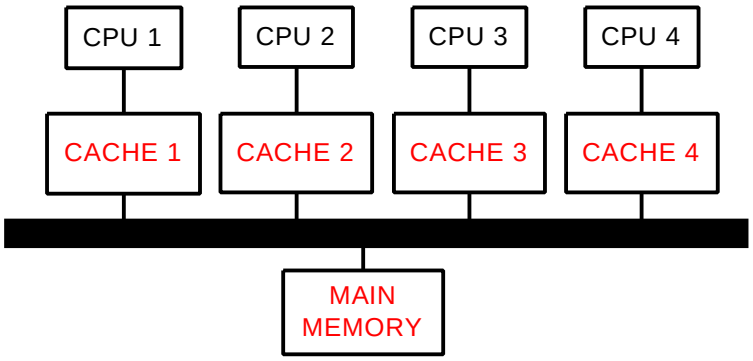
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10		10
				Shar.	A1	10	RdDa	P2	A1	10		10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1			10
P2: Write 40 to A2							WrMs	P2	A2			10
				Excl.	A2	40	WrBk	P2	A1	20		20

Assumes A1 and A2 map to same cache block

MULTIPROCESSOR MEMORY MODELS

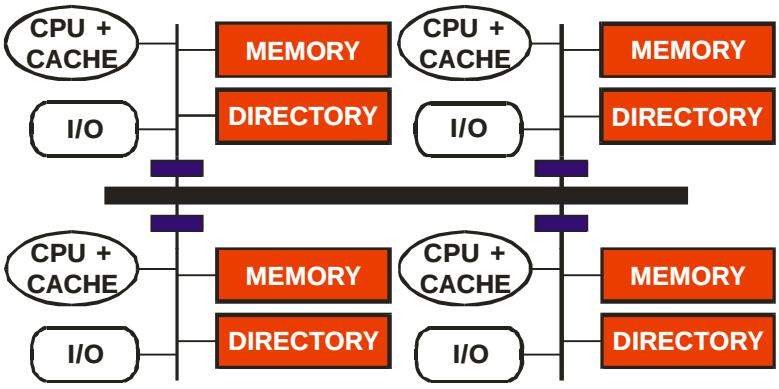
Multiprocessors -- UMA

- **UMA - Uniform Memory Access**
  - Several CPUs interconnect with shared memory/common bus
  - Caches used to filter bus traffic
  - Works well up to 8-16 nodes (*e.g.*, Encore Multimax)



Multiprocessors -- NUMA

- **CC-NUMA - Cache Coherent Non-Uniform Memory Access**
  - Numerous clusters with interconnect; global address space
  - Scales to many CPUs (*as long as application has locality*)
  - Becomes a "multicomputer" if each cluster has a separate address space instead of global memory addressing



Do Caches Work In Multiprocessors?

- **Basic cache functions are still a "win":**
  - Caches reduce average memory access time as long as there is locality
    - Memory can "self-organize" by migrating pages to cluster where data is being used
  - Caches filter memory requests
    - Significantly reduce bus traffic on single-bus model
- **But, there are new challenges:**
  - Software must account for consistency model on any multiprocessor
    - Tradeoff of software complexity vs. performance with relaxed consistency model
  - A new cache "C" is revealed -- Coherence misses
    - Two processes on two CPUs could cause data to migrate back and forth, causing cache misses because the data is being used frequently (rather than because it is used infrequently)

REVIEW

Review

- **Virtual Caches**
  - TLB access not required for L1 cache; relaxes address limit for L1
  - But, introduces potential problems with coherence
- **Multiprocessor Consistency**
  - Sequential consistency
  - Total Store Ordering
  - Partial Store Ordering
- **Multiprocessor Coherence**
  - Snooping vs. directory
  - Snoopy Cache protocol example
- **UMA/NUMA**