

Lecture 8

Snooping Cache Based Multiprocessors

Slides were used during lectures by David Patterson, Berkeley, spring 2006

Review

- 1 instruction operates on vectors of data
- Vector loads get data from memory into big register files, operate, and then vector store
- E.g., Indexed load, store for sparse matrix
- Easy to add vector to commodity instruction set
 - E.g., Morph SIMD into vector
- Vector is very efficient architecture for vectorizable codes, including multimedia and many scientific codes
- “End” of uniprocessors speedup \Rightarrow Multiprocessors
- Parallelism challenges: % parallelizable, long latency to remote memory
- Centralized vs. distributed memory
 - Small MP vs. lower latency, larger BW for larger MP
- Message Passing vs. Shared Address
 - Uniform access time vs. Non-uniform access time

Outline

- Review
- Coherence
- Write Consistency
- Snooping
- Building Blocks
- Snooping protocols and examples
- Coherence traffic and Performance on MP
- Conclusion

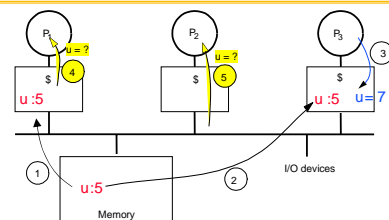
Challenges of Parallel Processing

1. Application parallelism \Rightarrow primarily via new algorithms that have better parallel performance
 2. Long remote latency impact \Rightarrow both by architect and by the programmer
- For example, reduce frequency of remote accesses either by
 - Caching shared data (HW)
 - Restructuring the data layout to make more accesses local (SW)
 - Today’s lecture on HW to help latency via caches

Symmetric Shared-Memory Architectures

- From multiple boards on a shared bus to multiple processors inside a single chip
- Caches both
 - Private data are used by a single processor
 - Shared data are used by multiple processors
- Caching shared data
 - \Rightarrow reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth
 - \Rightarrow cache coherence problem

Example Cache Coherence Problem



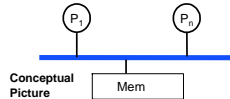
- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value
 - » Processes accessing main memory may see very stale value
- Unacceptable for programming, and its frequent!

Example

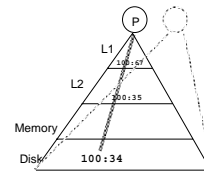
```

P1                               P2
-----                             -----
/* Assume initial value of A and flag is 0 */
A = 1;                               while (flag == 0); /* spin idly */
flag = 1;                             print A;
    
```

- Intuition not guaranteed by coherence
- Expect memory to respect order between accesses to *different* locations issued by a given process
 - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
 - pertains only to single location



Intuitive Memory Model



- Reading an address should return the **last value written to that address**
 - Easy in uniprocessors, except for I/O

- Too vague and simplistic; 2 issues
 1. **Coherence** defines **values** returned by a read
 2. **Consistency** determines **when** a written value will be returned by a read
- Coherence defines behavior to same location, Consistency defines behavior to other locations

Defining Coherent Memory System

1. **Preserve Program Order:** A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
2. **Coherent view of memory:** Read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses
3. **Write serialization:** 2 writes to same location by any 2 processors are seen in the same order by all processors
 - If not, a processor could keep value 1 since saw as last write
 - For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1

Write Consistency

For now assume

1. A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
 2. The processor does not change the order of any write with respect to any other memory access
- ⇒ if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

These restrictions allow the processor to reorder reads, but forces the processor to finish writes in program order

Basic Schemes for Enforcing Coherence

- Program on multiple processors will normally have copies of the same data in several caches
 - Unlike I/O, where its rare
- Rather than trying to avoid sharing in SW, SMPs use a HW protocol to maintain coherent caches
 - Migration and Replication key to performance of shared data
- **Migration** – data can be moved to a local cache and used there in a transparent fashion
 - Reduces both latency to access shared data that is allocated remotely and bandwidth demand on the shared memory
- **Replication** – for reading shared data simultaneously, since caches make a copy of data in local cache
 - Reduces both latency of access and contention for read shared data

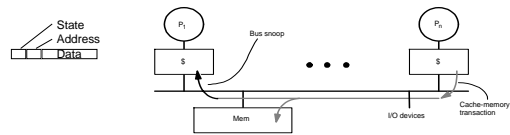
Outline

- Review
- Coherence
- Write Consistency
- Snooping
- Building Blocks
- Snooping protocols and examples
- Coherence traffic and Performance on MP
- Conclusion

Two Classes of Cache Coherence Protocols

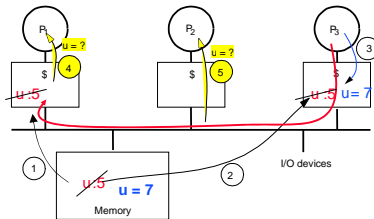
1. **Directory based** — Sharing status of a block of physical memory is kept in just one location, the **directory**
2. **Snooping** — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
 - All caches are accessible via some broadcast medium (a bus or switch)
 - All cache controllers monitor or **snoop** on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access

Snooping Cache-Coherence Protocols



- Cache Controller “snoops” all transactions on the shared medium (bus or switch)
 - **relevant transaction** if for a block it contains
 - take action to ensure coherence
 - » invalidate, update, or supply value
 - depends on state of the block and the protocol
- Either get exclusive access before write via write invalidate or update all copies on write

Example: Write-through Invalidate



- Must invalidate before step 3
- Write update uses more broadcast medium BW
 - ⇒ all recent MPUs use write invalidate

Architectural Building Blocks

- **Cache block state transition diagram**
 - FSM specifying how disposition of block changes
 - » invalid, valid, exclusive
- **Broadcast Medium Transactions (e.g., bus)**
 - Fundamental system design abstraction
 - Logically single set of wires connect several devices
 - Protocol: arbitration, command/addr, data
 - ⇒ Every device observes every transaction
- **Broadcast medium enforces serialization of read or write accesses ⇒ Write serialization**
 - 1st processor to get medium invalidates others copies
 - Implies cannot complete write until it obtains bus
 - All coherence schemes require serializing accesses to same cache block
- Also need to find up-to-date copy of cache block

Locate up-to-date copy of data

- **Write-through: get up-to-date copy from memory**
 - Write through simpler if enough memory BW
- **Write-back harder**
 - Most recent copy can be in a cache
- Can use same snooping mechanism
 1. Snoop every address placed on the bus
 2. If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
 - Complexity from retrieving cache block from cache, which can take longer than retrieving it from memory
- **Write-back needs lower memory bandwidth**
 - ⇒ Support larger numbers of faster processors
 - ⇒ Most multiprocessors use write-back

Cache Resources for WB Snooping

- Normal cache tags can be used for snooping
- Valid bit per block makes invalidation easy
- Read misses easy since rely on snooping
- **Writes ⇒ Need to know whether any other copies of the block are cached**
 - No other copies ⇒ No need to place write on bus for WB
 - Other copies ⇒ Need to place invalidate on bus

Cache Resources for WB Snooping

- To track whether a cache block is shared, add extra state bit associated with each cache block, like valid bit and dirty bit
 - Write to Shared block \Rightarrow Need to place invalidate on bus and mark cache block as private (if an option)
 - No further invalidations will be sent for that block
 - This processor called **owner** of cache block
 - Owner then changes state from shared to unshared (or exclusive)

Cache behavior in response to bus

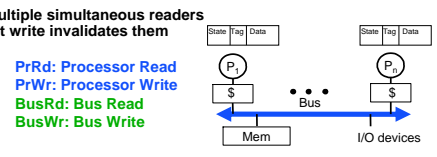
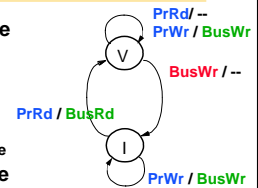
- Every bus transaction must check the cache-address tags
 - could potentially interfere with processor cache accesses
- A way to reduce interference is to duplicate tags
 - One set for caches access, one set for bus accesses
- Another way to reduce interference is to use L2 tags
 - Since L2 less heavily used than L1
 - \Rightarrow Every entry in L1 cache must be present in the L2 cache, called the **inclusion property**
 - If Snoop gets a hit in L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor

Example Protocol

- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node
- Logically, think of a separate controller associated with each cache block
 - That is, snooping operations or cache requests for different blocks can proceed independently
- In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
 - That is, one operation may be initiated before another is completed, even though only one cache access or one bus access is allowed at time

Write-through Invalidate Protocol

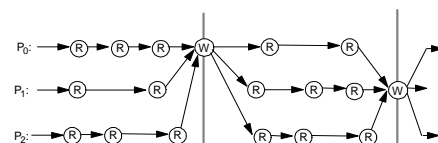
- 2 states per block in each cache
 - as in uniprocessor
 - state of a block is a p -vector of states
 - Hardware state bits associated with blocks that are in the cache
 - other blocks can be seen as being in invalid (not-present) state in that cache
- Writes invalidate all other cache copies
 - can have multiple simultaneous readers of block, but write invalidates them



Is 2-state Protocol Coherent?

- Processor only observes state of memory system by issuing memory operations
- Assume bus transactions and memory operations are atomic and a one-level cache
 - all phases of one bus transaction complete before next one starts
 - processor waits for memory operation to complete before issuing next
 - with one-level cache, assume invalidations applied during bus transaction
- All writes go to bus + atomicity
 - Writes **serialized** by order in which they appear on bus (bus order)
 - \Rightarrow invalidations applied to caches in bus order
- How to insert reads in this order?
 - Important since processors see writes through reads, so determines whether write serialization is satisfied
 - But read hits may happen independently and do not appear on bus or enter directly in bus order
- Let's understand other ordering issues

Ordering



- Writes establish a partial order
- Doesn't constrain ordering of reads, though shared-medium (bus) will order read misses too
 - any order among reads between writes is fine, as long as in program order

Example

step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Proc.	Addr	Value	Memory 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, initial cache state is invalid

Example

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

Assumes A1 and A2 map to same cache block

Example

step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Proc.	Addr	Value	Memory 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

Example

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

Assumes A1 and A2 map to same cache block

Example

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

Assumes A1 and A2 map to same cache block

Example

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

Assumes A1 and A2 map to same cache block, but A1 != A2

Implementation Complications

- **Write Races:**
 - Cannot update cache until bus is obtained
 - » Otherwise, another processor may get bus first, and then write the same cache block!
 - Two step process:
 - » Arbitrate for bus
 - » Place miss on bus and complete operation
 - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
 - Split transaction bus:
 - » Bus transaction is not atomic: can have multiple outstanding transactions for a block
 - » Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
 - » Must track and prevent multiple misses for one block
- **Must support interventions and invalidations**

Implementing Snooping Caches

- **Multiple processors must be on bus, access to both addresses and data**
- **Add a few new commands to perform coherency, in addition to read and write**
- **Processors continuously snoop on address bus**
 - If address matches tag, either invalidate or update
- **Since every bus transaction checks cache tags, could interfere with CPU just to check:**
 - solution 1: **duplicate set of tags for L1 caches** just to allow checks in parallel with CPU
 - solution 2: L2 cache already duplicate, **provided L2 obeys inclusion** with L1 cache
 - » block size, associativity of L2 affects L1

Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

- **Single memory accommodate all CPUs**
 - ⇒ **Multiple memory banks**
- **Bus-based multiprocessor, bus must support both coherence traffic & normal memory traffic**
 - ⇒ **Multiple buses or interconnection networks (cross bar or small point-to-point)**
- **Opteron**
 - Memory connected directly to each dual-core chip
 - Point-to-point connections for up to 4 chips
 - Remote memory and local memory latency are similar, allowing OS Opteron as UMA computer

Outline

- Review
- Coherence
- Write Consistency
- Snooping
- Building Blocks
- Snooping protocols and examples
- Coherence traffic and Performance on MP
- Conclusion

Performance of Symmetric Shared-Memory Multiprocessors

- Cache performance is combination of
1. **Uniprocessor cache miss traffic**
 2. **Traffic caused by communication**
 - Results in invalidations and subsequent cache misses
- 4th C: **coherence miss**
- Joins Compulsory, Capacity, Conflict

Coherency Misses

1. **True sharing misses** arise from the communication of data through the cache coherence mechanism
 - Invalidates due to 1st write to shared block
 - Reads by another CPU of modified block in different cache
 - Miss would still occur if block size were 1 word
2. **False sharing misses** when a block is invalidated because some word in the block, other than the one being read, is written into
 - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
 - Block is shared, but no word in block is actually shared
 - ⇒ miss would not occur if block size were 1 word

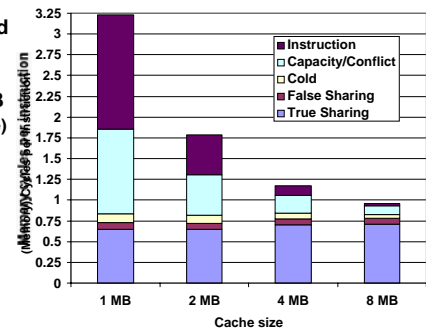
Example: True v. False Sharing v. Hit?

Assume x1 and x2 in same cache block.
P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

MP Performance 4 Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

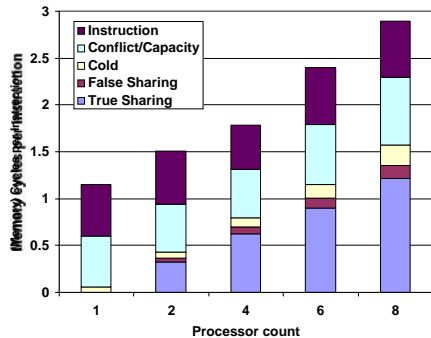
True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)



Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)

MP Performance 2MB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

True sharing, false sharing increase going from 1 to 8 CPUs



A Cache Coherent System Must:

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
 - (0) Determine when to invoke coherence protocol
 - (a) Find info about state of block in other caches to determine action
 - » whether need to communicate with other cached copies
 - (b) Locate the other copies
 - (c) Communicate with those copies (invalidate/update)
- (0) is done the same way on all systems
 - state of the line is maintained in the cache
 - protocol is invoked if an "access fault" occurs on the line
- Different approaches distinguished by (a) to (c)

Bus-based Coherence

- All of (a), (b), (c) done through broadcast on bus
 - faulting processor sends out a "search"
 - others respond to the search probe and take necessary action
- Could do it in scalable network too
 - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn't scale with p
 - on bus, bus bandwidth doesn't scale
 - on scalable network, every fault leads to at least p network transactions
- Scalable coherence:
 - can have same cache states and state transition diagram
 - different mechanisms to manage protocol

And in Conclusion ...

- Caches contain all information on state of cached memory blocks
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data ⇒ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
- MPs are highly effective for multiprogrammed workloads
- MPs proved effective for intensive commercial workloads, such as OLTP (assuming enough I/O to be CPU-limited), DSS applications (where query optimization is critical), and large-scale, web searching applications

Reading and Schedule

- This lecture:
 - 4.2 *Symmetric Shared-Memory Architectures*
 - 4.3 *Performance of Symmetric Shared-Memory Multiprocessors*
- This afternoon: *feedback on assignment 2a*
- Next week, Nov 14th:
 - 11.15-13.00h: *directory-based MP & rest of chapter 4*
 - 13.45-15.30h: *chapter 5 memory hierarchy design*