Cache Coherence

Introduction

1. Coherence ensures that when one core writes to its cache, other cores are able to see when they read the value out of their own cache

Cache Coherence Problem

1. Programmer is expecting shared memory
   - When core A writes x = 15, core B reads X and sees 15
2. Hardware: Each core has (and needs) its own cache
   - One large L1 cache -> too slow, not enough throughput
   - Private (per-core) L1 caches
     - This can result in inconsistencies between memory and caches, which is called "incoherent"
     - A's and B's caches need to be kept coherent
3. Cache coherence: Make the cache/memory system behave as a single memory

Cache Incoherence Quiz

1. Consider a cache with the following properties:
   - Memory location A initially contains 0
   - Three cores, each with a write-back L1 cache
   - No coherence support
2. Each core loads the value in A, increments it, and writes the new value back to A
   - This happens in succession, no overlap
3. What are the possible values in A at the end of execution?
   - 1, 2, or 3
   - Depends on how many cores have updated their values in memory

Coherence Definition

1. Read (R) from address X on core C1 returns the value written by the most recent write (W) to X on C1 if no other core has written to X between W and R
2. If C1 writes to X and C2 reads after a sufficient amount of time, and there are no other writes in-between, C2's read returns the value from C1's write
3. Writes to the same location are serialized: any two writes to X must be seen to occur in the same order on all cores
   - This is not guaranteed by condition 2 alone

Coherence Definition Quiz

1. Two cores in a coherent system:

```c
// Core 1
A = 1;
while(A == 1);
A = 1;
printf("Done 1!");

// Core 2
A = 0;
while(A == 0);
A = 0;
printf("Done 2!");
```

2. Which of the following results are possible?
    - Done 1!Done 2! (possible)
    - Done 2!Done 1! (possible)
    - Done 1!
    - Done 2!
    - Nothing printed
3. Bonus: All are possible in an incoherent system
    - Coherence is allowed, but not guaranteed, in an incoherent system

How to Get Coherence

1. No caches
    - Bad performance
2. All cores share the same L1 cache
    - Bad performance
3. Private write-through caches
    - Doesn't solve the problem of one core reading an old value (incoherent)
4. Force read in one cache to see write made in another
    - Broadcast writes to update other caches
        - Called write-update coherence
    - Writes prevent hits to other copies
        - Called write-invalidate coherence
    - Writes broadcast on shared bus
        - Called snooping
    - Each block is assigned an "ordering point"
        - Called directory

Write Update Snooping Coherence

1. When cache 2 writes 1 to location A and cache 1 already has it in its cache, cache 1 listens ("snoops") the write from the shared bus
2. All caches see writes in the same order as they share the same bus
    - Processors must arbitrate for the bus since there is only one set of wires
        - This guarantees that simultaneous writes don't interfere

Write Update Coherence Quiz

1. There are four cores that access location 0x700 in the following way:
    - Core 0: RD 0x700 (gets 6)
    - Core 1: RD 0x700
    - Core 2: WR 0x700 (writes 17)
    - Core 3: RD 0x700
2. What are the valid and data bits for each core at the end of execution?
    - Core 0: Valid 1, Data 17
    - Core 1: Valid 1, Data 17
    - Core 2: Valid 1, Data 17
    - Core 3: Valid 1, Data 17

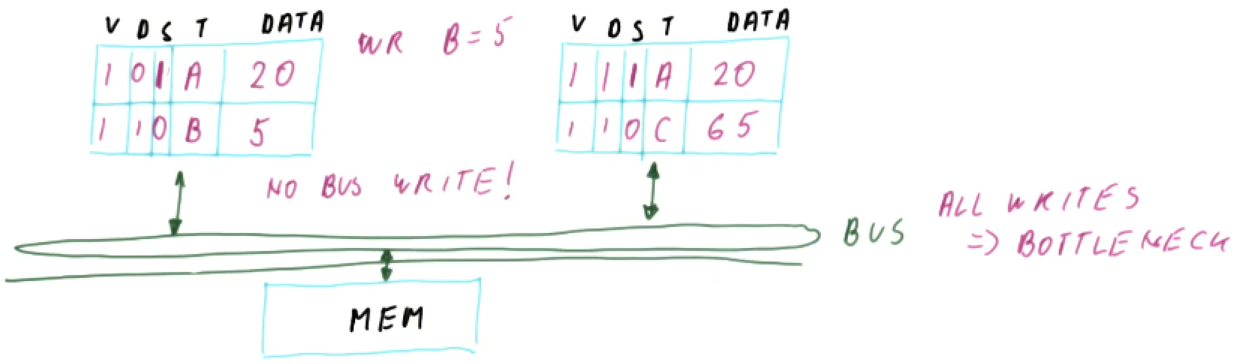
Write Update Optimization

1. Memory is slow, so it becomes a bottleneck in a write-through cache
    - Add a dirty bit to the cache
        - This means the memory isn't up to date
2. If a different core tries to read a value written to by another core, the core that did the write must snoop the read and respond

    - Cores can handle coherence without memory at all
    - Memory is only involved when a value is replaced from a cache
3. Dirty bit benefits
    - Write to memory only when block is replaced
    - Read from memory only if no D=1 copy

Write Update Optimization 2 Bus Writes

1. Even if not all writes need to go to memory, they all must go to the shared bus, which becomes the new bottleneck for the system
2. Add another bit called a "shared bit" to determine if a block is shared and needs to be broadcast or not
    - Add a separate line to the bus to communicate if a block is already in another core's cache
        - If a block is only used by one core, its writes don't go to the bus
3. In the example below, writes to B and C will not be brodcast while writes to A will be broadcast



Reducing Bus Writes

Write Update Optimization Quiz

1. Two cores, write-update coherence
2. Core 0 writes to A and then core 1 reads A 1000 times
3. Then, core 0 replaces A, then core 1 replaces A

	No Optimization	Dirty-bit Only	D and S Bits
How many bus uses?	1001	1002	1002
How many mem writes?	1000	1	1

Write Invalidate Snooping Coherence

1. When a write is broadcast, we don't need to broadcast the value
    - When a core sees a broadcast for a tag in its cache, it sets it valid bit to 0 so it knows to read the value if it uses it again
2. When a core needs a value with a valid bit of 0, it will send the request on the bus and the core with the value will broadcast the value
    - Both cores will set the shared bit
3. Disadvantage: Miss on all readers when a cache writes
4. Advantage: Fewer broadcasts (only the first, after that it's equivalent to working locally)
5. Benefits from the same optimizations as above

Write Update vs Write Invalidate Quiz 1

1. Core 0 writes to A and then core 1 reads A 1000 times
2. How many bus uses do we get with…
   - Write-update (S, D optimizations): 1001
   - Write-invalidate (S, D optimizations): 2000
     - Every access will have to communicate

Write Update vs Write Invalidate Quiz 2

1. Core 0 reads then writes A 500 times, then core 1 reads then writes A 500 times
2. How many bus uses do we get with…
   - Write-update (S, D optimizations): 502
     - Once the data becomes shared, all writes go to the bus
   - Write-invalidate (S, D optimizations): 3

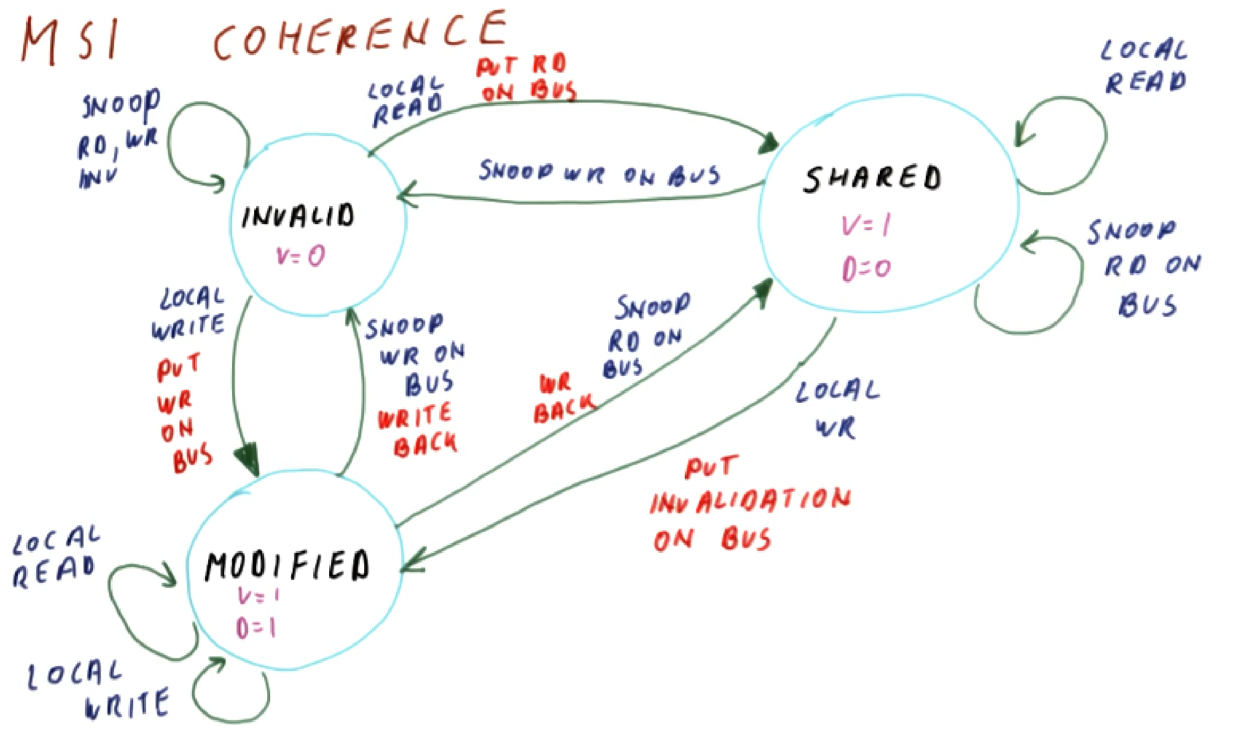
Update vs Invalidate Coherence

Application Does	Update	Invalidate
Burst of writes to addr	Each write sends update	First invalidate -> hits
Diff words, same block	Update for each word	First invalidate -> hits
Producer/consumer	P updates, C hits	P invalidates, C misses, …
Thread changes core	Updates old cache	First invalidate -> no traffic

1. The first two are good for invalidate, the last is good for update
2. All modern processors use invalidate protocols
   - On average, they are slightly better than update protocols
3. Threads often change cores in modern operating systems, so an update protocol's performance in this case drives processors to use invalidate

MSI Coherence

1. Invalid state: Present in cache but valid bit is not set, or not in cache
   - V = 0
2. Shared state: Local reads require no action, but writes require broadcast
   - V = 1, D = 0
3. Modified state: Can perform local reads and writes freely
   - We're sure there are no other cores sharing, so we can work locally
   - V = 1, D = 1
   - Only one cache can be in M state for a block at a time



MSI State Machine

Cache to Cache Transfers

1. C1 has block B in Modified state
2. C2 puts read request on bus
3. C1 has to provide data (even memory doesn't have up to date copy)
4. Two possible solutions:
    - Abort/Retry:
        - C1 cancels C2's request ("ABORT" bus signal)
        - C1 does normal write-back to memory
        - C2 retries, gets data from memory
        - C1 read miss has 2x memory latency
    - Intervention:
        - C1 tells memory it will supply the data ("INTERVENTION" bus signal)
        - C1 responds with data
        - Memory picks up data and stores in memory block (once C1 responds with data, both will go to shared state and think the block is clean)
        - Hardware is more complex, but this is typically used in modern processors (performance doesn't suffer)

MSI Quiz 1

1. Two cores with private caches using MSI coherence
2. Initially, block X is in memory

C1	C2	State of X in C1 cache	State of X in C2 cache
Read X	-	S	I
-	Read X	S	S
Write X	-	M	I

MSI Quiz 2

1. Two cores with private caches using MSI coherence
2. Initially, block X is in memory

C1	C2	State of X in C1 cache	State of X in C2 cache
Read X	-	S	I
-	Write X	I	M
Write X	-	M	I

Avoiding Memory Writes on Cache to Cache Transfers

1. C1 has block in M state
2. C2 wants to read, C1 responds with data
    - C1: S, C2: S
3. C2 writes, C1: I, C2: M
    - Repeats
    - Data is moving between caches, but memory is also written each time
4. C1 read, C2 responds with data
    - C1: S, C2: S
5. C3 read, memory provides data
    - We can avoid this memory read by having another cache respond
6. C4 read, . . .
    - As long as a cache holds the most recent block, we want to avoid going to memory
7. Need to make a non-modified version responsible for
    - Giving data to other caches
    - Eventually writing block to memory
8. New state: O (owned)
    - Only one cache owns the block if it's being shared
    - Owner responds to read requests and writes to memory if replaced

MOSI Coherence

1. Like MSI, except. . .
    - M -> Snoop A read -> O (not S)
        - Memory does not get accessed
    - O state like S, except
        - Snoop A read -> Provide data
        - Write-back to memory if replaced
2. M: Exclusive read/write access, dirty
3. S: Shared read access, clean
4. O: Shared read access, dirty
    - Only one cache with block in O state

M(O)SI Inefficiency

1. Thread-private data: Data only accessed by a single thread

        - All data in single-threaded programs
        - Stack in multi-threaded programs
2. Data read, then write
    - Both MSI and MOSI: I -> miss -> S -> invalidate -> M
        - Must do this for every block of data
        - Additional bus request
        - Don't want to pay the penalty for sharing if we aren't sharing
    - Uniprocessor: V = 0 -> miss -> V = 1 -> hit -> D = 1
3. Introduce new "E" state (exclusive)

The E State

1. M: Exclusive access (R/W), dirty
2. S: Shared access (R), clean
3. O: Shared access (R), dirty
4. E: Exclusive access (R/W), clean

	MSI	MOSI	MESI	MOESI
RD A	I -> S (miss)	I -> S (miss)	I -> E (miss)	I -> E (miss)
WR A	S -> M (inv)	S -> M (inv)	E -> M (hit)	E -> M (hit)

MOESI Quiz

1. 3 cores, MOESI coherence, X starts out only in memory

	C0	C1	C2
C0 Read X	E	I	I
C1 Read X	S	S	I
C2 Read X	S	S	S
C1 Write X	I	M	I

MESI - MOSI - MOESI Quiz

1. 3 cores, A in I state in all three caches
2. Consider the following sequence of accesses
    - C1: RD A (C1: S or E)
    - C1: WR A (C1: M)
    - C2: RD A (C1: O or S, C2: S)
    - C2: WR A (C1: I, C2: M)
    - C3: RD A (C1: I, C2: O or S, C3: S)
    - C1: RD A (C1: S, C2: O or S, C3: S)
    - C2: RD A (C1: S, C2: O or S, C3: S)
3. How many…

	MESI	MOSI	MOESI
Memory reads	2	1	1
Bus requests	5	6	5

Directory Based Coherence

1. Snooping: Broadcast requests so others see them and to establish ordering
   - Single bus on which all requests from all cores travel
   - Bus becomes a bottleneck
     - Snooping does not work well with > 8-16 cores
2. Non-broadcast network
   - How do we observe requests we need to see?
   - How do we order requests to the same block?

Directory

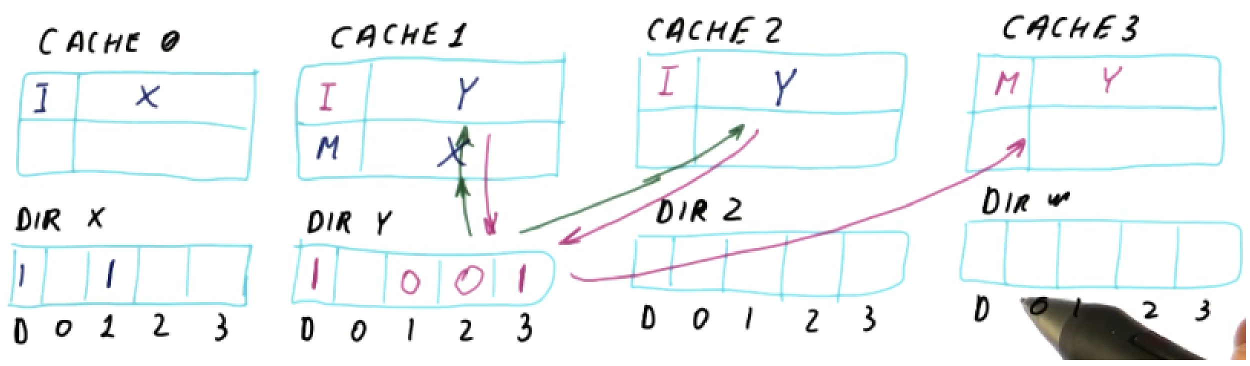
1. Distributed structure across cores
   - Not all requests go to the same part of the directory
   - Each "slice" serves a set of blocks
     - One entry for each block it serves
     - Entry tracks which caches have block (in non-I state)
     - Order of accesses determined by "home" slice
     - Can operate independently because it serves a disjoint set of blocks
   - Caches still have same states
     - Requests no longer go on bus; directory figures out what to do

Directory Entry

1. 1 dirty bit
2. 1 bit per cache: Present in that cache
3. Allows for concurrent accesses to different blocks
   - Accesses to the same block are serialized
   - Saves bandwidth by using point-to-point network links
4. If a cache is present in the directory for a read and another cache writes to the same location, the reader is evicted from the directory

Directory Example

1. Consider the following sequence of accesses:
   - C0: WR X, C1: RD Y
   - C1: WR X, C2: RD Y (C2 reads Y from C1)
   - C3: WR Y



Directory Entry

Directory MOESI Quiz

1. 4 cores numbered 0-3, directory for A in slice 0
2. Consider the following sequence of accesses:
   - C0: RD A (request sent to directory, directory sends response)
   - C0: WR A (request sent to directory, directory forwards message, directory gets response, directory sends response)
   - C1: RD A (request sent to directory, directory sends response)
     - C0: O, C1: S
   - C2: RD A (request sent to directory, directory sends response)
     - C2: S
   - C3: RD A (request sent to directory, directory sends response)
     - C3: S
   - C0: WR A (request sent to directory, directory forwards 3 messages, directory gets 3 responses, directory sends response)
     - C0: O, C1: I, C2: I, C3: I
3. How many…
   - Requests sent to the directory: 5
   - Requests forwarded by directory: 4
   - Responses received by directory: 4
   - Responses sent by directory: 5

Cache Misses with Coherence

1. 3 C's: Compulsory, conflict, capacity
   - Coherence adds another "C" (coherence)
     - We read something, somebody writes it, we want to read it again -> this is only a miss due to coherence
2. Two types of coherence misses:
   - True sharing: Different cores access same data
   - False sharing: Different cores access different data but in same block
     - Maintain coherence at the granularity of blocks

False Sharing Quiz

1. The cache is initially in the following state:

A	B	C	D
X	Y	Z	W

2. Which of these sets of accesses have false sharing misses?
   - Set 1: (no, only compulsory misses)
     - C0: RD X
     - C1: WR X
     - C2: RD A
     - C3: WR B
   - Set 2: (no, only compulsory misses)
     - C0: RD X
     - C1: WR X
     - C2: RD Z
     - C3: WR Z
   - Set 3: (no, sharing between C0 and C1 is true sharing)
     - C0: RD X
     - C1: WR X

    - C2: WR W
    - C0: RD X

Conclusion

1. Each core needs its own cache, but these must be kept coherent to present the illusion of shared memory