Cache Coherence

(Architectural Supports for Efficient Shared Memory)

Mainak Chaudhuri
mainakc@cse.iitk.ac.in
Agenda

• Setting
  - Software: shared address space
  - Hardware: shared memory multiprocessors
• Cache coherence
• Invariants and implementation
• Cache coherence protocols: MSI, MESI
• Formal definitions
Shared address space

- Part of the address space is shared between multiple threads or processes
  - Achieved by declaring shared variables as global for sharing among threads within a process (POSIX thread model)
  - Achieved by allocating memory through specialized system calls for sharing among different processes (UNIX shmget/shmat)
  - Variables in the shared address space can be read from and written to by multiple different threads/processes (load/store ins)
  - A mode of communication between threads/processes for exchanging results
Shared memory multiprocessors

• Platform to schedule in parallel multiple threads or processes that share memory

• Many different designs exist
  - We will assume that each processor has a hierarchy of caches (possibly shared)
  - Shared cache: popular in chip-multiprocessors (CMPs)
  - Private cache: found in some CMPs, symmetric multi-processors (SMPs), and multi-node servers
  - Distributed shared memory: popular in medium to large-scale multi-node servers
Shared cache

Single chip

P0
CACHE

P1
CACHE

P2
CACHE

P3
CACHE

INTERCONNECT

SHARED CACHE

DRAM
Private cache

May or may not be a single chip

PO
CACHE

P1
CACHE

P2
CACHE

P3
CACHE

INTERCONNECT

DRAM
Distributed shared memory
Distributed shared memory

Pass explicit messages between nodes for data exchange
(Will not be discussed in this session)
Cache coherence

• Processors employ private caching of data to improve performance
  - Private copies of shared data must be “coherent”
    • Roughly speaking, all copies must have the same value (enough if this holds eventually)
  - For sequential programs, a memory location must return the latest value written to it
  - For parallel programs, we expect the same provided “latest” is well-defined
    • For now, latest value of a location is the latest value “committed” by any thread/process
    • A cache coherence protocol is a set of actions that ensure that a load to address A returns the “last committed” value to A
Cache coherence: Example

- Assume 3Ps with write-through caches
  - P0: reads x from memory, puts it in its cache, and gets the value 5
  - P1: reads x from memory, puts it in its cache, and gets the value 5
  - P1: writes x=7, updates its cached value and memory value
  - P0: reads x from its cache and gets the value 5
  - P2: reads x from memory, puts it in its cache, and gets the value 7 (now the system is completely incoherent)
  - P2: writes x=10, updates its cached value and memory value
Cache coherence: Example

- Consider the same example for writeback caches
  - P0 has a cached value 5, P1 has 7, P2 has 10, memory has 5 (since caches are not write through)
  - The state of the line in P1 and P2 is M while the line in P0 is clean
  - Eviction of the line from P1 and P2 will issue writebacks while eviction of the line from P0 will not issue a writeback (clean lines do not need writeback)
  - Suppose P2 evicts the line first, and then P1
  - Final memory value is 7: we lost the store x=10 from P2
What went wrong?

• For write through cache
  - The memory value may be correct if the writes are correctly ordered
  - But the system allowed a store to proceed when there is already a cached copy
  - Lesson learned: must invalidate all cached copies before allowing a store to proceed

• Writeback cache
  - Problem is even more complicated: stores are no longer visible to memory immediately
  - Writeback order is important
  - Lesson learned: do not allow more than one copy of a cache line in M state
Implementations

• **Must invalidate all cached copies before allowing a store to proceed**
  - Need to know where the cached copies are
  - **Solution 1:** Just tell everyone that you are going to do a store
    • Leads to broadcast snoopy protocols
    • Popular with small-scale machines
    • Typically, the interconnect is a shared bus; AMD Opteron implements it on a distributed network (the Hammer protocol)
  - **Solution 2:** Keep track of the sharers and invalidate them when needed
    • Where and how is this information stored?
    • Leads to directory-based scalable protocols"
Implementations

• Directory-based protocols
  - Maintain one directory entry per memory block
  - Each directory entry contains a sharer bitvector and state bits
Distributed shared memory

Where and how would you place the directory? New concept: home node
Where and how would you place the directory?

Sparse directory

Single chip
Implementations

• Do not allow more than one copy of a cache line in M state
  - Need some form of access control mechanism
  - Before a processor does a store it must take “permission” from the current “owner” (if any)
  - Need to know who the current owner is
    • Either a processor or main memory
  - Solution1 and Solution2 apply here also
    • Tell everybody or tell the owner
Implementations

• Latest value must be propagated to the requester
  - Notion of “latest” is very fuzzy
  - Once we know the owner, this is easy
  - Solution1 and Solution2 apply here also
    • Tell everybody or tell the owner
Implementations

• Invariant: if a cache block is not in M state in any processor, memory must provide the block to the requester
  - Memory must be updated when a block transitions from M state to S state
  - Note that a transition from M to I always updates memory in systems with writeback caches (these are normal writeback operations)

• Most of the implementations of a coherence protocol deals with uncommon cases and races
Invalidation vs. Update

- Two main classes of protocols:
  - Dictates what action should be taken on a store
  - Invalidation-based protocols invalidate sharers when a store miss appears
  - Update-based protocols update the sharer caches with new value on a store
  - Advantage of update-based protocols: sharers continue to hit in the cache while in invalidation-based protocols sharers will miss next time they try to access the line
  - Advantage of invalidation-based protocols: only store misses go on bus and subsequent stores to the same line are cache hits
Sharing patterns

• Producer-consumer
  - One thread produces a value and other threads consume it
  - Example (flag is zero initially):
    • T1: x=y; flag=1;
    • T2-Tk: while(!flag); use x;

• Migratory
  - Each thread reads and writes to a shared variable in sequence
  - Example (1<=i<=k) flag is one initially:
    • Ti: while(flag != i); x = f(x); flag++;
  - Migratory hand-off?
Migratory hand-off

- Needs a memory writeback on every hand-off
  - r0, w0, r1, w1, r2, w2, r3, w3, r4, w4, ...
  - How to avoid these unnecessary writebacks?
  - Saves memory bandwidth
  - Solution: add an owner state (different from M) in caches
  - Only owner can write a line back on eviction
  - Ownership shifts along the migratory chain
States of a cache line

- Invalid (I), Shared (S), Modified or dirty (M), Clean exclusive (E), Owned (O)
  - Every processor does not support all five states
  - E state is equivalent to M in the sense that the line has permission to write, but in E state the line is not yet modified and the copy in memory is the same as in cache; if someone else requests the line the memory will provide the line after querying the E state holder
  - O state: memory is not responsible for servicing requests to the line; the owner must supply the line (just as in M state); no write permission
Stores

• Look at stores a little more closely
  - There are three situations at the time a store issues: the line is not in the cache, the line is in the cache in S state, the line is in the cache in one of M, E and O states
  - If the line is in I state, the store generates a read-exclusive request on the bus and gets the line in M state
  - If the line is in S or O state, that means the processor only has read permission for that line; the store generates an upgrade request on the bus and the upgrade acknowledgment gives it the write permission (this is a data-less transaction)
MSI protocol

- Forms the foundation of invalidation-based writeback protocols
  - Assumes only three supported cache line states: I, S, and M
  - There may be multiple processors caching a line in S state
  - There must be exactly one processor caching a line in M state and it is the owner of the line
  - If none of the caches have the line, memory must have the most up-to-date copy of the line
State transition

I

PrRd/BusRd

PrRd/BusRd/

{BusRdX, BusUpgr}/

CacheEvict/

S

PrWr/BusRdX

PrRd/BusUpgr

PrRd/-

BusRd/Flush

M

BusRdX/Flush

CacheEvict/BusWB

PrRd/-

PrWr/-
**MSI example**

- Take the following example
  - P0 reads x, P1 reads x, P1 writes x, P0 reads x, P2 reads x, P3 writes x
  - Assume the state of the cache line containing the address of x is I in all processors
  P0 generates BusRd, memory provides line, P0 puts line in S state
  P1 generates BusRd, memory provides line, P1 puts line in S state
  P1 generates BusUpgr, P0 snoops and invalidates line, memory does not respond, P1 sets state of line to M
  P0 generates BusRd, P1 flushes line and goes to S state, P0 puts line in S state, memory writes back
  P2 generates BusRd, memory provides line, P2 puts line in S state
  P3 generates BusRdX, P0, P1, P2 snoop and invalidate, memory provides line, P3 puts line in cache in M state
MESI protocol

• The most popular invalidation-based protocol e.g., appears in Intel Xeon MP

• Why need E state?
  - The MSI protocol requires two transactions to go from I to M even if there is no intervening requests for the line: BusRd followed by BusUpgr
  - Save one transaction by having memory controller respond to the first BusRd with E state if there is no other sharer in the system
  - Needs a dedicated control wire that gets asserted by a sharer (wired OR)
  - Processor can write to a line in E state silently
**MESI example**

- Take the following example
  - P0 reads x, P0 writes x, P1 reads x, P1 writes x, ...
  - P0 generates BusRd, memory provides line, P0 puts line in cache in E state
  - P0 does write silently, goes to M state
  - P1 generates BusRd, P0 provides line, P1 puts line in cache in S state, P0 transitions to S state
  - Rest is identical to MSI
- Consider this example: P0 reads x, P1 reads x, ...
  - P0 generates BusRd, memory provides line, P0 puts line in cache in E state
  - P1 generates BusRd, memory provides line, P1 puts line in cache in S state, P0 transitions to S state (no cache-to-cache sharing)
  - Rest is same as MSI
Definitions

• **Memory operation**: a read (load), a write (store), or a read-modify-write
  - Assumed to take place atomically

• A memory operation is said to **issue** when it leaves the issue queue and looks up the cache

• A memory operation is said to **perform** with respect to a processor when a processor can tell that from other issued memory operations
Definitions

• A read is said to perform with respect to a processor when subsequent writes issued by that processor cannot affect the returned read value.

• A write is said to perform with respect to a processor when a subsequent read from that processor to the same address returns the new value.

• A memory operation is said to complete when it has performed with respect to all processors in the system.
Ordering memory op

- Assume that there is a single shared memory and no caches
  - Memory operations complete in shared memory when they access the corresponding memory locations
  - Operations from the same processor complete in program order: this imposes a partial order among the memory operations
  - Operations from different processors are interleaved in such a way that the program order is maintained for each processor: memory imposes some total order (many are possible)
Example

P0: x=8; u=y; v=9;
P1: r=5; y=4; t=v;
Total order: x=8; u=y; r=5; y=4; t=v; v=9;
Another legal total order:
x=8; r=5; y=4; u=y; v=9; t=v;

• “Last” means the most recent in some legal total order

• A system is coherent if
  - Reads get the last written value in the total order
  - All processors see writes to a location in the same order
Cache coherence

• Formal definition
  - A memory system is coherent if the values returned by reads to a memory location during an execution of a program are such that all operations to that location can form a hypothetical total order that is consistent with the serial order and has the following two properties:

  1. Operations issued by any particular processor perform according to the issue order

  2. The value returned by a read is the value written to that location by the last write in the total order
Cache coherence

• Two necessary features that follow from above:
  A. Write propagation: writes must eventually become visible to all processors
  B. Write serialization: Every processor should see the writes to a location in the same order (if I see w1 before w2, you should not see w2 before w1)