MANCHESTER 1824

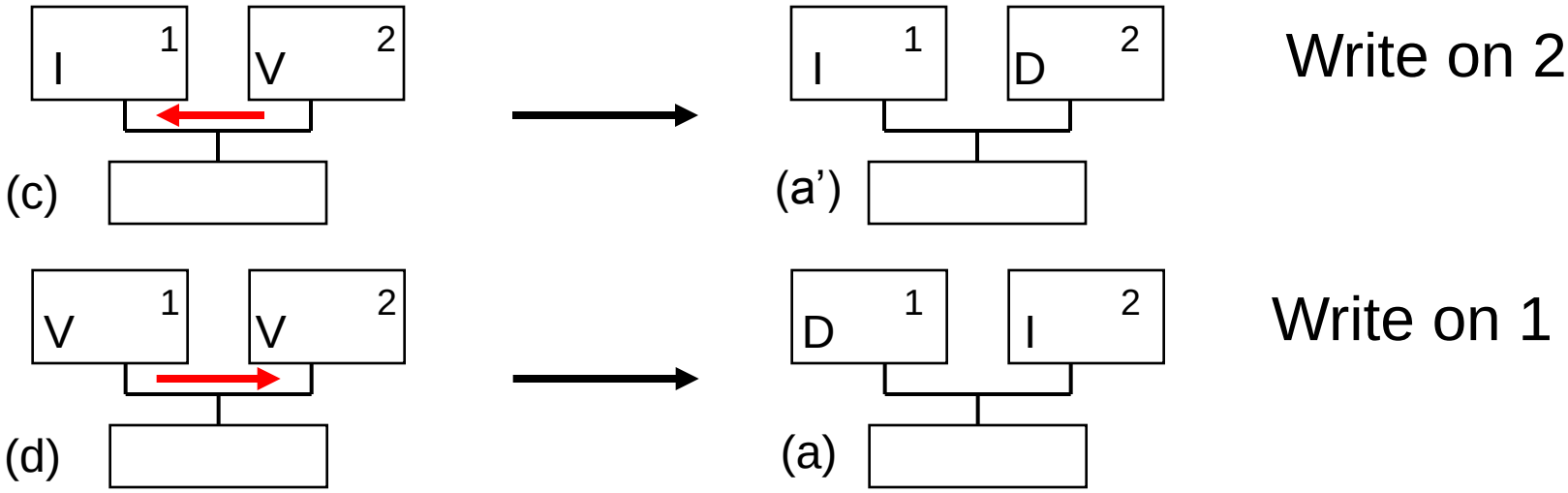
The University of Manchester

Chip Multiprocessors COMP35112

Lecture 5 - Other Cache Coherence Protocols

Graham Riley

Unnecessary Communication



- Note that in the (c) → (a') transition the invalidate broadcast is unnecessary
- However, in the (d) → (a) transition it is needed to invalidate the other V value in core 2

Optimising for non-shared values

- The bus is a critical shared resource – unnecessary use could impact performance
- Can we distinguish between the two cases
  - Cache holds **only** copy of a value (which is the same as that in memory i.e. not dirty)
  - Cache holds a copy of the value but there are also other copies in other caches (truly "shared")
- In the first case we do not need to send an invalidate on write whereas in the second an invalidate is needed

A Common Case

- The reason this is important is that the unshared case is very common
- In real problems, the majority of variables are unshared
  - e.g. a thread is likely to have local variables (on its stack) – these are almost certainly not shared (although it is possible, particularly in C)
- We therefore split the V state into two states
  - E – exclusive (unshared)
  - S – (truly) shared

MESI Protocol

- Two states are easy to determine
  - Read causes a fetch from memory – goes to state E
  - Read gets value from another cache – goes to state S
- The overall protocol can then be extended to include these states, resulting in less use of bus bandwidth
  - We will not cover the detail, but note that cache line eviction can cause a line in state S to be the only remaining copy! In practice this is too difficult to detect, so leave it in state S
- In practice MESI is more widely used than MSI as it is a simple extension and its effect on bus usage is significant

MOESI Protocol

- A further optimisation – split the M state into two:
  - M – modified, as before – the cache contains a copy which differs from that in memory but there are no other copies
  - O – owned – the cache contains a copy which differs from that in memory and there may be copies in other caches which are in state S (these have the same value as the owner)
- This allows the latest value to be shared without having to write it back to memory immediately
  - The semantics of state S is slightly different
  - Only when a cache line in state O or M gets evicted will any write back to memory be done

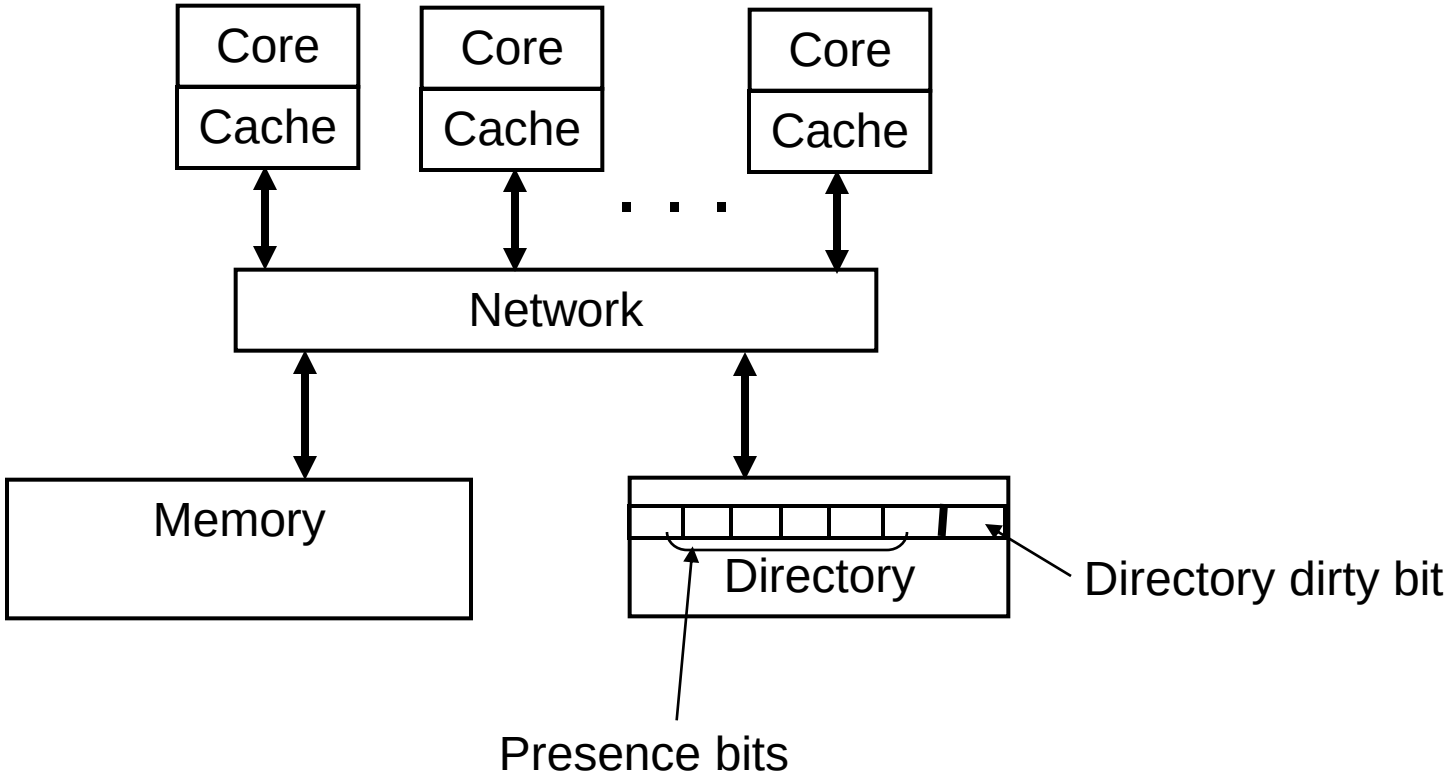
Directory Based Coherence

- Previous schemes have relied on a shared bus where all cores communicate ‘instantaneously’ over the bus
- Can we implement a coherence scheme with a network which is less directly connected? e.g. a grid or general packet switched network
- This is possible using a centralised **directory** which holds information about every value (in reality every cache line) in the memory (remember a cache line usually contains multiple words)
- Aim for a (simple) MSI-like protocol

Directory Structure

- Each directory entry contains the following
  - Information on which core has a copy – usually held as a bit map (i.e. 1 bit per core)
  - Whether-or-not the copy is dirty (if so, there is only one owner and thus one ‘true’ bit in the bit map)
- Each line in every cache also has a valid and a dirty bit
- A core wishing to make a memory access may need to query the directory about the state of the line to be accessed (see following slides)

General (Logical) Structure



Directory Protocol

- Read hit in local cache
  - No need for directory access – simply read local value
- Read miss in local cache
  - Access directory
    - Directory dirty bit = false
      - Read data from main memory into local cache
      - Set directory presence bit p[i] (core i is reading)
      - Set local valid bit (in core i)
    - Directory dirty bit = true
      - Cause the (one-and-only) owner core to update memory
      - Updated data into local cache
      - Clear directory dirty bit and set presence bit p[i]
      - Set local valid bit

Directory Protocol (cont.)

- Write miss in local cache
  - Write allocate in local cache
    - Set local dirty bit
  - Access directory
    - Directory dirty bit = false
      - Send invalidate to any cores x with p[x] set and then clear these bits
      - Set p[i] bit for writing core and set directory dirty bit
    - Directory dirty bit = true
      - Send message to owner core to update memory
      - Clear owner's p[x] bit and set p[i] bit
      - Leave directory dirty bit set

Directory Protocol (cont.)

- Write hit in local cache
  - If local dirty bit set – just update local cache
  - If local dirty bit not set – update local cache
    - Set local dirty bit
    - Access directory
      - Directory dirty bit = false
        - Send invalidate to any cores x with p[x] set and then clear these bits
        - Set p[i] bit for writing core and set directory dirty bit
      - Directory dirty bit = true … normally would:
        - Send message to owner core to update memory
        - Clear owner's p[x] bit and set p[i] bit for writing core
        - Leave directory dirty bit set
        - BUT local cache had a hit and line NOT dirty, so directory cannot be dirty (line must be "shared"), so this cannot happen

Analysis

- This is roughly equivalent to MSI bus based protocol
- Several protocol optimisations possible
- Central directory is a serious bottleneck
  - Distribute directory and cache it
    - i.e. each core has a directory responsible for part of the address space
  - Often coupled with a distributed memory structure where part of the memory is physically local to the processor (particularly in big multi-processor systems i.e. not single chip)

Drawbacks

- Without a common bus network many of the previous communications will take a significant number of CPU cycles
- In the presence of longer delays (possibly variable) such protocols usually need ‘handshakes’ (replies to messages) in order to work correctly
- Although real machines use(d) this (or variants) e.g. SGI Origin – up to 2048 cores, and the more recent Xeon Phi – 60 cores.
  - many doubt that it can be made to work efficiently for heavily shared memory applications

Summary

- Cache coherence implemented by bus snooping does not really scale to large numbers of cores
- Directory systems do not need a bus but the inherent delays and communication overheads are unlikely to lead to a solution for heavily used large-scale shared memory
- A major question is whether cache coherence is really necessary in a shared memory system – much of this is concerned with the parallel programming model used (we shall return to the topic of ‘memory consistency’ in a later lecture – but, before we do so, we need to look further at data sharing parallel programming)