MANCHESTER 1824

The University of Manchester

Chip Multiprocessors COMP35112

Lecture 4 - Shared Memory Multiprocessors

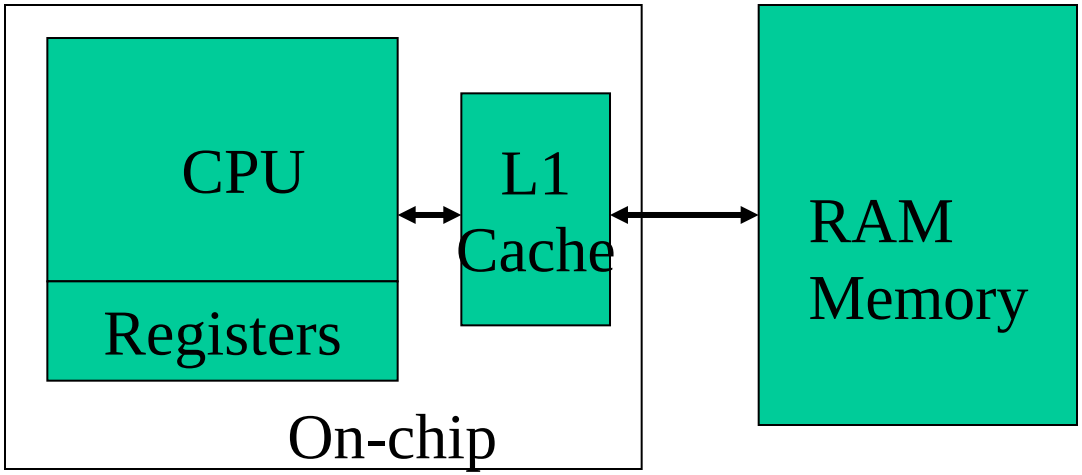
Graham Riley

Multiprocessor Structure

- The majority of general purpose multiprocessors are shared memory
  - Mainly because they are easier to program
  - But shared memory hardware is usually more complex
- Bus-based shared memory systems do not really scale beyond a certain number (~32?) of cores
  - Intel/AMD, and others, have developed solutions such as QPI (Intel, ~2009), Ultrapath Interconnect (Intel, 2017) and Coherent Hyper-transport (AMD)
- We need to understand the problems
  - With a focus on bus-based systems initially

Caches

- Recall that a high performance uniprocessor has the following structure



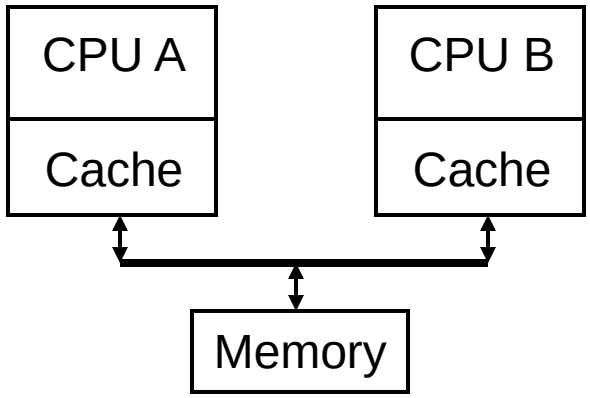
- Cache is fast small local memory that holds recently used data and instructions (note: there may be multiple levels)

Why Caches?

- Main memory is far too slow (50-100×) to keep up with modern processor speed
- Small on-chip memory can be very fast
- If program 'working set' can be kept in caches most of the time, CPU can run at full speed
- But:
  - New data/instructions needed by the program have to be fetched from memory (on each cache miss)
  - Newly written data in cache must eventually be written back to main memory

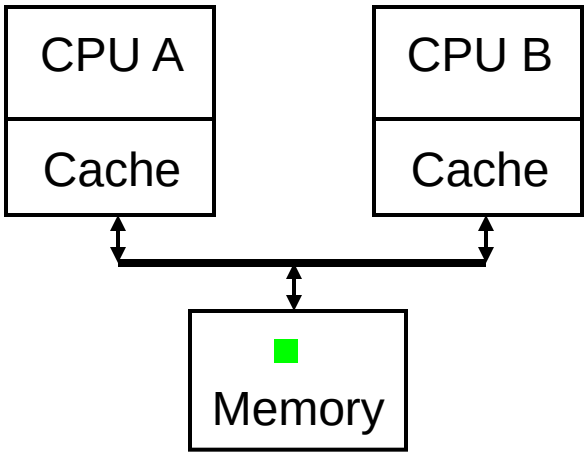
The Cache Coherence Problem

- With just one CPU there is no problem, data just written to cache can be read correctly whether or not it has been written to memory
- But when we have multiple processors



- They may share variables, i.e. one can write a value that the other needs to read

The Cache Coherence Problem



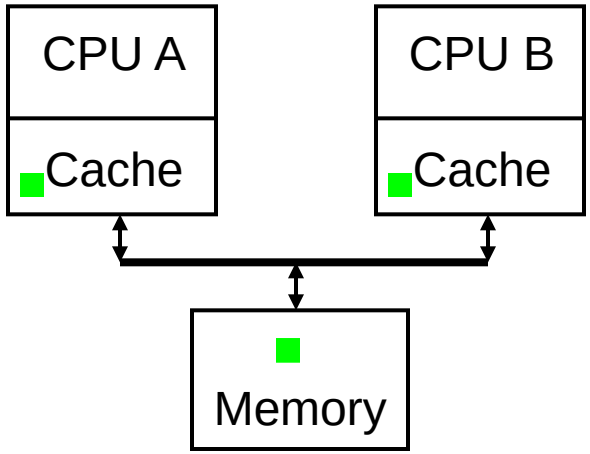
- CPU A reads a value x from memory
- CPU A writes to x (the copy in its cache)
- CPU B reads x but gets the old value
- We (probably) expect to get the latest value!

The Cache Coherence Problem

- Apparently obvious solution would be to ensure that every write is updated in memory (a ‘write through’ policy)
- However, this would mean that every such write/read across processors would involve two main memory accesses – i.e. a long time
- As this is the way in which threads communicate in a shared memory system, such a delay is undesirable

The Cache Coherence Problem

- In any case, on its own, it doesn't solve the problem
- If CPU B had already cached the value, it will use the value it already has



The Cache Coherence Problem

- How can we overcome these problems?
  - Can we overcome the delay problem by communicating cache-to-cache rather than via memory?
  - When a new value is written in one cache, all other values somehow need to be either updated or invalidated
- Other problems
  - What if two processors try to write to the same location? – two separate cache copies
- Gets complex, we need to develop a model

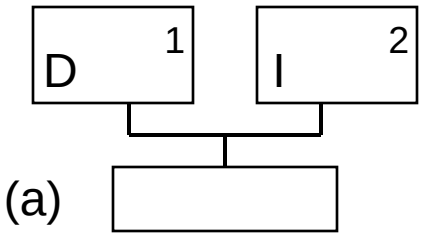
Cache States

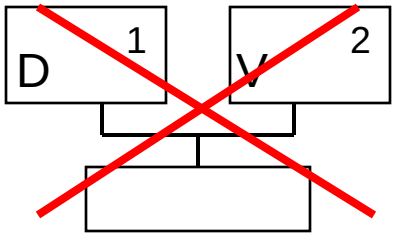
- Recall a single cache – it has 2 control bits for each line of data indicating the state:
  - **Invalid –** there may be an address match on this line but the data is not valid, i.e. we must go to memory and fetch it
  - **Dirty –** the cache line is valid and has been written to but the latest values have not been updated in memory
- There is clearly an implicit third state, **Valid**, which is not invalid and not dirty – i.e. a valid cache entry exists and the line has the same values as main memory

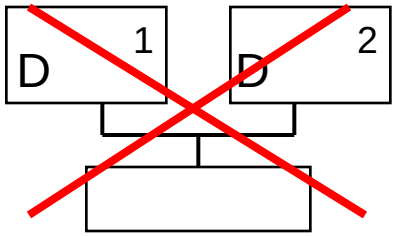
Coping With Multiple Cores

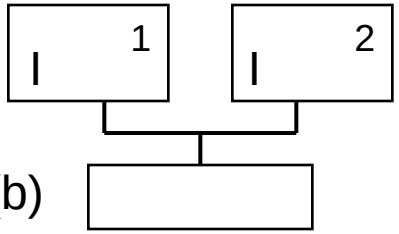
- The first step we need is the introduction of the concept of **‘bus snooping’** via a bus attached to every cache
- This means that we introduce hardware, attached to each core’s cache, which observes all transactions on the bus and is able to modify the cache independently of the core
- This hardware is able to take action on seeing pertinent transfers between another core’s cache and memory and, because it is attached to the bus, is also able to send messages to other caches

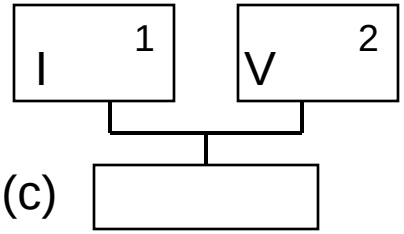
Possible States – 2 Cores

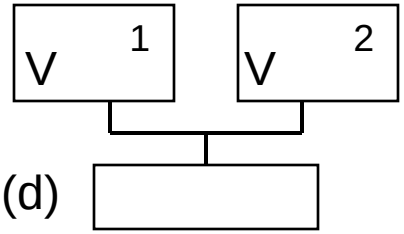












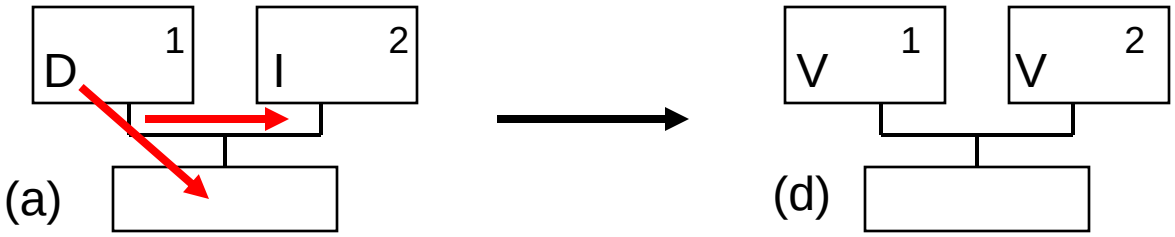
Label possible states

- All possible states assuming symmetry
- But DV is disallowed, as V core (2) would read wrong value
- Also DD not allowed, as line would have two values

State Transitions

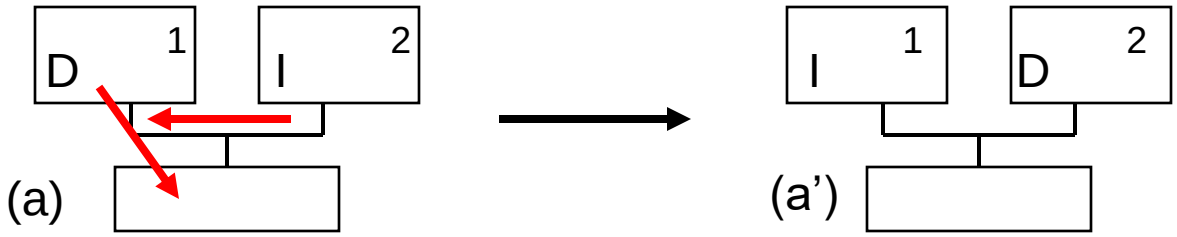
- (a), (b), (c), (d) (with alternates (a') and (c') where not symmetrical) are the legal states in which a pair of caches can exist to ensure correct operation
- We now study how reads and writes in the various cores need to affect these states
- This will consist of three aspects:
  - Accesses which need to be made to main memory
  - Messages which need to be sent between caches
  - Changes between the (legal) states which result

State Transitions from State (a)



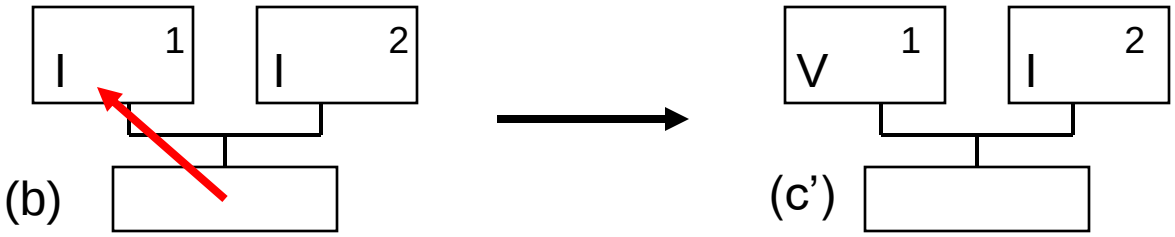
- Read on core 1 – no change
- Write on core 1 – no change
- Read on core 2
  - 2 requests value from 1 and goes to V state
  - But DV state not allowed – 1 writes its dirty value to memory and goes to V state
  - Overall change to state (d)

State Transitions from State (a)



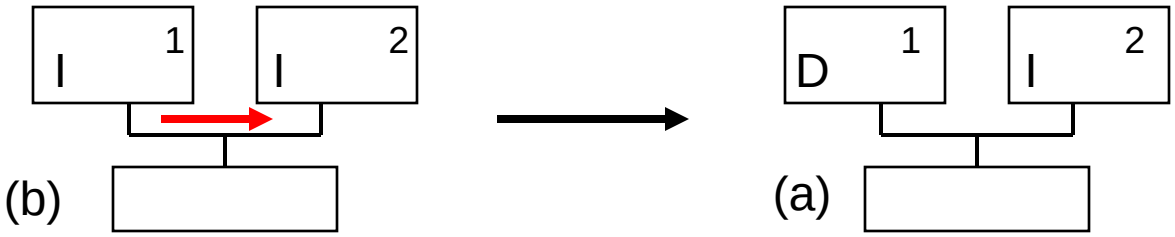
- Write on core 2
  - 2 does a write allocate in cache – needs to go to D
  - But DD state not allowed – 2 sends a message to 1 to invalidate its entry – 1 is dirty so must write its value to memory – 1 goes to I
  - Overall change to state (a') – inverse of (a)

State Transitions from State (b)



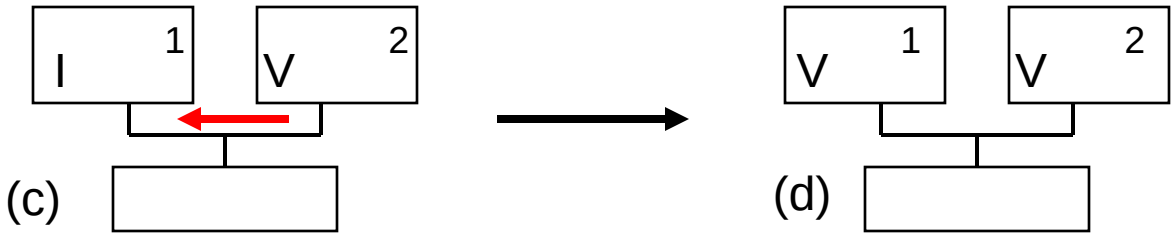
- Read on core 1
  - Cache miss – fetches from memory
  - Goes to state (c') – inverse of (c)
- Read on core 2 is similar by symmetry – goes to state (c)

State Transitions from State (b)



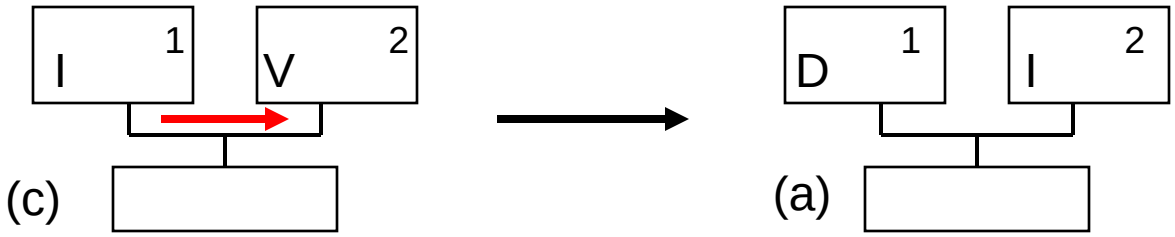
- Write on core 1
  - 1 does a write allocate in cache – goes to D
  - But 1 doesn't know 2's state so it must broadcast an invalidate message – 2 remains I
  - State goes to (a)
- Write on core 2 is similar by symmetry – goes to state (a')

State Transitions from State (c)



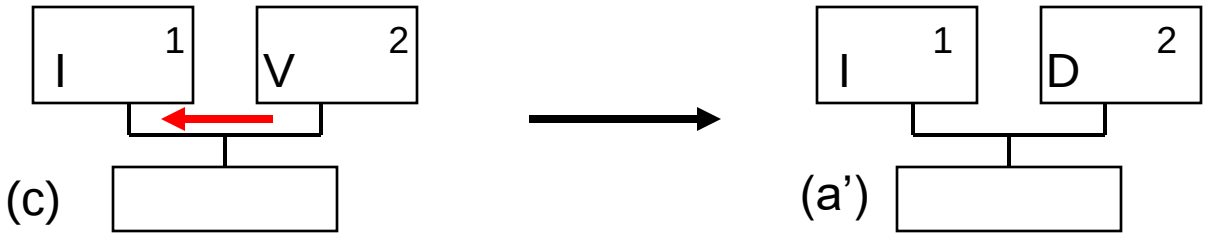
- Read on core 1
  - 1 requests value from 2 and goes to V
  - Overall state goes to (d)
- Read on core 2 is simple and state remains (c)

State Transitions from State (c)



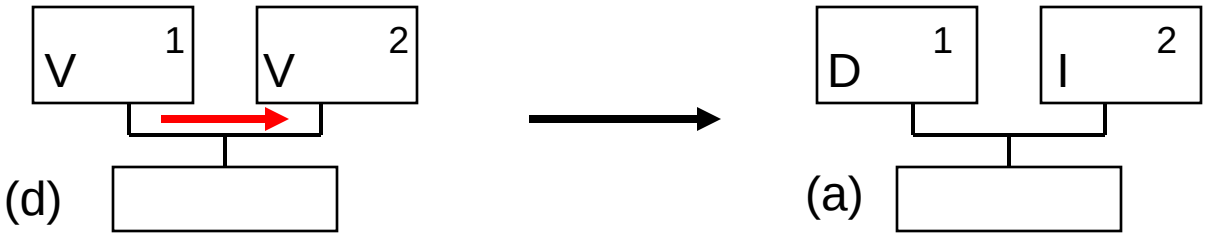
- Write on core 1
  - 1 does a write allocate in cache – goes to D
  - But DV state not allowed – 1 sends an invalidate message to 2 – no need for 2 to writeback to memory – 2 goes to I
  - Overall state goes to (a)

State Transitions from State (c)



- Write on core 2
  - 2 overwrites value – goes to D
  - But 2 doesn't know 1's state so it must broadcast an invalidate – 1 remains I
  - Overall state goes to (a') – inverse of (a)

State Transitions from State (d)



- Read on core 1 or core 2 – no change
- Write on core 1
  - 1 overwrites value – goes to D
  - But DV state not allowed – 1 sends an invalidate message to 2 – no need for 2 to writeback to memory – 2 goes to I
  - Overall state goes to (a)
- Write on core 2 – symmetry – state goes to (a')

Messages Between Cores

- There are two types of messages between cores
  - Read requests for a cache line
  - Invalidates of other cache's line
- We can easily extend beyond 2 cores
  - Any core with a valid value can respond to a read request (bus arbitration means one will 'win')
  - Invalidates can equally apply to many caches as long as the written line ends up in a Dirty state and corresponding lines in all other caches are Invalid – but this implies that the 'invalidate' message must be broadcast to all cores

Major Implication

- Cache coherence normally requires that all cores must always see exactly the same state of a location in memory
- If one core writes and sends an invalidate, no other core must be able to perform a read or write to that location as though they haven't seen the invalidate
- This essentially requires that they all see the invalidate at the same time, i.e. in the same bus cycle

Major Implication (cont.)

- The more cores connected to the bus, the more difficult this will become
- As we connect more cores
  - The bus will get physically longer, so the signals will take longer to propagate
  - The bus will have more capacitance which will inevitably slow it down
- If the bus cycle is slowed this will impact performance
- **The coherence protocol is a major limitation to the number of cores that can be supported**

The MSI Protocol

- We have derived the protocol from a consideration of uniprocessor cache 'valid' & 'dirty' bits, and this led to the **I**, **V** and **D** state terminology
- The result is nowadays called the **MSI protocol**
  - **M**odified == **D**irty
  - **S**hared == **V**alid
  - **I**nvalid
- In the above, the term 'Shared' is slightly confusing – it is defined as 'one or more copies each holding the same value as main memory' – we shall discuss further in the next lecture