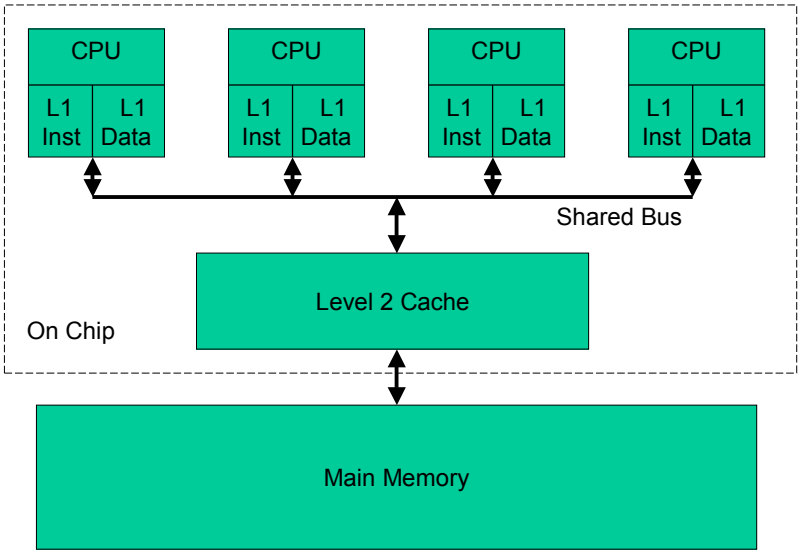
Shared Memory Coherence

Ian Watson & Mikel Lujan
Advanced Processor Technologies Group

Multi-Core Programs

- Will meet details of programming later in the course
- However, it is clear that multiple cores will each be running their own pieces of code
- This can either be
  - Processes – operating system level processes e.g. separate applications – in many cases do not share any data – separate virtual memory spaces
  - Threads – parallel parts of the same application sharing the same memory – this is where the problems lie – assume we are talking about threads

Typical Multi-Core Structure



Memory Coherence

- What is the coherence problem?
  - Core writes to location in its L1 cache
  - Other L1 caches may hold shared copies - these will immediately be out of date
- Core may either
  - Write through to L2 cache and/or memory
  - Copy back only when cache line is rejected
- In either case we have a problem
- Because each core may have its own copy, it is not sufficient just to update L2 and/or memory

Bus Snooping

- Scheme where every core knows who has a copy of its cached data is far too complex.
- So each core (cache system) 'snoops' (i.e. watches continually) for activity concerned with data addresses which it has cached.
- This assumes a bus structure which is 'global', i.e all communication can be seen by all
- There are 'directory based' coherence schemes for non-global comms. structures will not consider at present

Snooping Protocols

- Write Invalidate
  - CPU wanting to write to an address, grabs a bus cycle and sends a 'write invalidate' message which contains the address
  - All snooping caches invalidate their copy of appropriate cache line
  - CPU writes to its cached copy (assume for now that it also writes through to memory)
  - Any shared read in other CPUs will now miss in cache and re-fetch new data.

Snooping Protocols

- Write Update
  - CPU wanting to write grabs bus cycle and broadcasts address & new data as it updates its own copy
  - All snooping caches update their copy
- Note that in both schemes, problem of simultaneous writes is taken care of by bus arbitration - only one CPU can use the bus at any one time.

Update or Invalidate?

- Update looks the simplest, most obvious and fastest, but:-
  - Multiple writes to same word (no intervening read) need only one invalidate message but would require an update for each
  - Writes to same block in (usual) multi-word cache block require only one invalidate but would require multiple updates.

Update or Invalidate?

- Due to both spatial and temporal locality, previous cases occur often.
- Bus bandwidth is a precious commodity in shared memory multi-processors
- Experience has shown that invalidate protocols use significantly less bandwidth.
- Will consider implementation details only of invalidate.

Implementation Issues

- In both schemes, knowing if a cached value is not shared (copy in another cache) can avoid sending any messages.
- Invalidate description assumed that a cache value update was written through to memory. If we used a 'copy back' scheme (usual for high performance) other processors could re-fetch incorrect old value on a cache miss.
- We need a protocol to handle all this.

MESI Protocol (1)

- A practical multiprocessor invalidate protocol which attempts to minimize bus usage.
- Allows usage of a 'copy back' scheme - i.e. L2/main memory not updated until 'dirty' cache line is displaced
- Extension of usual cache tags, i.e. invalid tag and 'dirty' tag in normal copy back cache.

- To make description simpler, we will ignore L2 cache and treat L2/main memory as a single main memory unit

MESI Protocol (2)

Any cache line can be in one of 4 states (2 bits)

- **Modified** - cache line has been modified, is different from main memory - is the only cached copy. (multiprocessor 'dirty')
- **Exclusive** - cache line is the same as main memory and is the only cached copy
- **Shared** - Same as main memory but copies may exist in other caches.
- **Invalid** - Line data is not valid (as in simple cache)

MESI Protocol (3)

- Cache line changes state as a function of memory access events.
- Event may be either
  - Due to local processor activity (i.e. cache access)
  - Due to bus activity - as a result of snooping
- Each cache line has its own state affected only if address matches

MESI Protocol (4)

- Operation can be described informally by looking at action in local processor
  - Read Hit
  - Read Miss
  - Write Hit
  - Write Miss
- More formally by state transition diagram (later)

MESI Local Read Hit

- Line must be in one of MES
- This must be correct local value (if M it must have been modified locally)
- Simply return value
- No state change

MESI Local Read Miss (1)

- CPU makes read request to main memory
- One cache has E copy
  - Snooping cache puts copy value on the bus
  - Memory access is abandoned
  - Local processor caches value
  - Both lines set to S
- No other copy in caches
  - CPU waits for memory response
  - Value stored to local cache, marked E

MESI Local Read Miss (2)

- Several caches have S copy
  - One cache puts copy value on the bus (arbitrated)
  - Memory access is abandoned
  - Local processor caches value
  - Local copy set to S
  - Other copies remain S

MESI Local Read Miss (3)

- One cache has M copy
  - Snooping cache puts copy value on the bus
  - Memory access is abandoned
  - Local processor caches value
  - Local copy tagged S
  - **Source (M) value copied back to memory**
  - Source value M -> S

MESI Local Write Hit (1)

Line must be one of MES

- M
  - line is exclusive and already 'dirty'
  - Update local cache value
  - no state change
- E
  - Update local cache value
  - State E -> M
- S
  - Processor broadcasts an invalidate on bus
  - Snooping processors with S copy change S->I
  - Local cache value is updated
  - Local state change S->M

MESI Local Write Miss (1)

Detailed action depends on copies in other processors

- No other copies
  - Value read from memory to local cache (?)
  - Value updated
  - Local copy state set to M

MESI Local Write Miss (2)

- Other copies, either one in state E or more in state S
  - Value read from memory to local cache - bus transaction marked RWITM (read with intent to modify)
  - Snooping processors see this and set their copy state to I
  - Local copy updated & state set to M

MESI Local Write Miss (3)

Another copy in state M

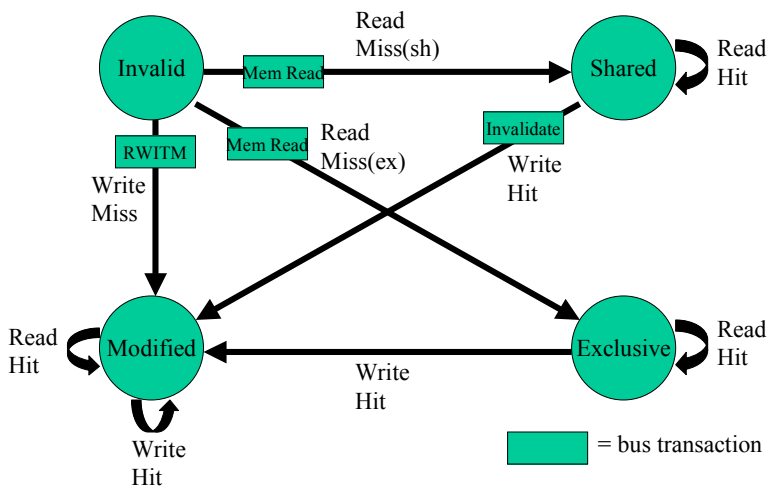
- Processor issues bus transaction marked RWITM
- Snooping processor sees this
  - Blocks RWITM request
  - Takes control of bus
  - Writes back its copy to memory
  - Sets its copy state to I

MESI Local Write Miss (4)

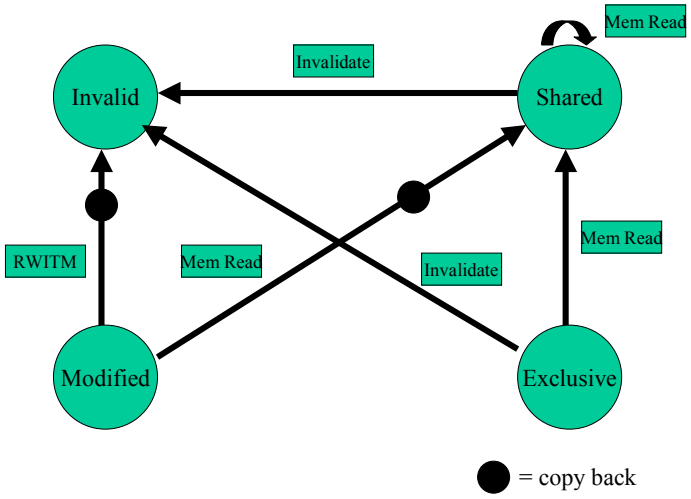
Another copy in state M (continued)

- Original local processor re-issues RWITM request
- Is now simple no-copy case
  - Value read from memory to local cache
  - Local copy value updated
  - Local copy state set to M

MESI - local cache view



MESI - snooping cache view



Comments on MESI Protocol

- Relies on global view of all memory activity – usually implies a global bus
- Bus is a limited shared resource
- As number of cores increases
  - Demands on bus bandwidth increase – more total memory activity
  - Bus gets slower due to increased capacitive load
- General consensus is that bus-based systems cannot be extended beyond a small number (8 or 16?) cores