

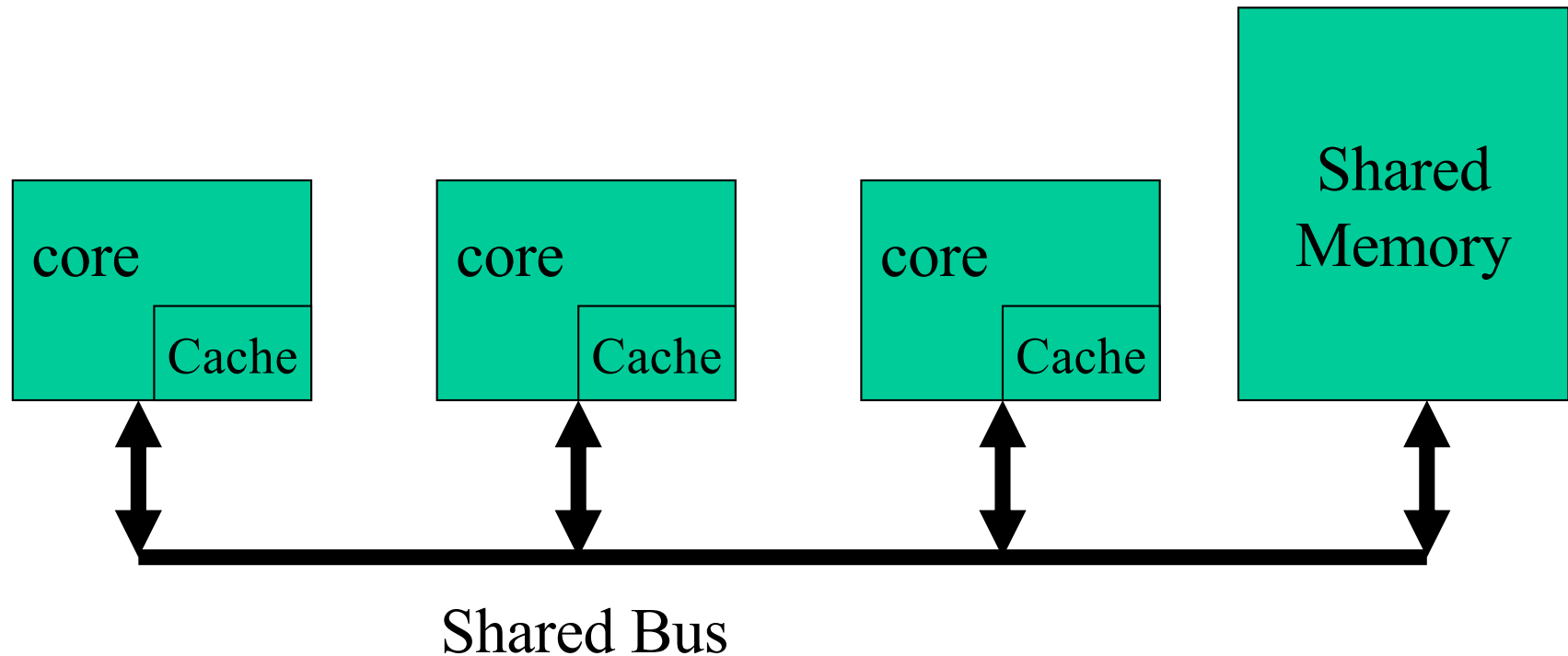
Cache coherence in shared-memory architectures

Overview

- Sequential programs
 - Caches are “transparent” to program: result does not depend on whether there are caches or not
 - Given memory location may be cached in several levels of caches but loads and stores to that location always behave semantically as though there is only one copy of location in memory
- Shared-memory programs
 - Require *cache coherence* to ensure the same behavior

Bus-based Shared Memory Organization

Basic picture is simple :-



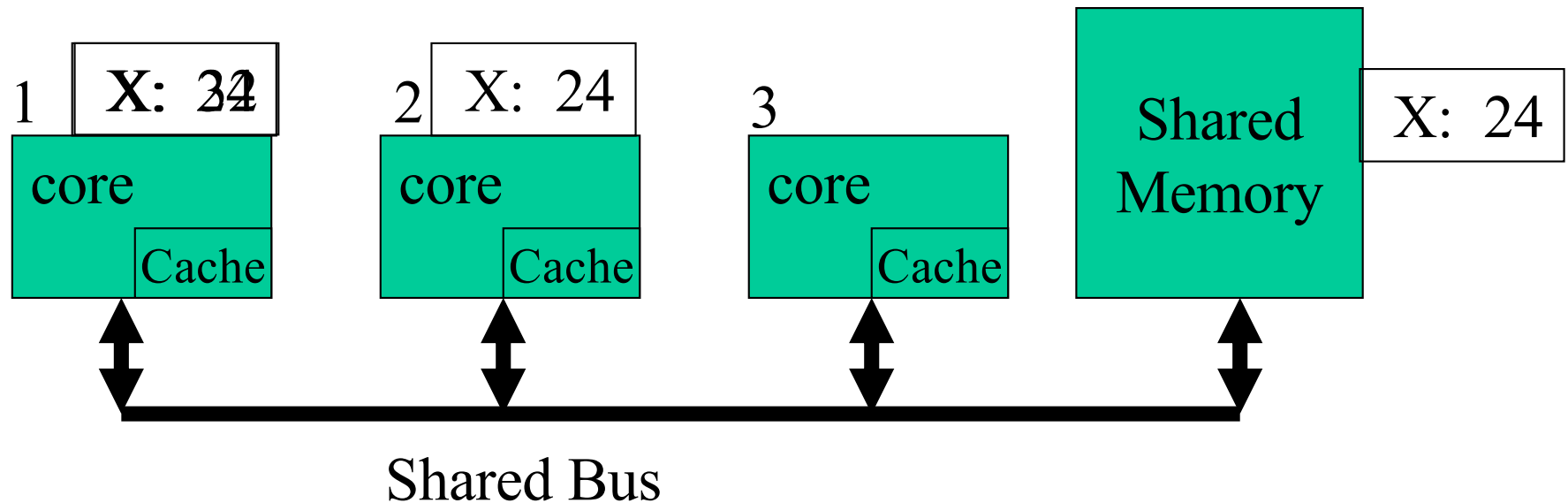
Organization

- Bus is usually simple physical connection (wires)
- Bus bandwidth limits number of cores
- Could be multiple memory elements
- For now, assume that each core has only a single level of cache

Problem of Memory Coherence

- Assume just single level caches and main memory
- Core writes to location in its cache
- Other caches may hold shared copies - these will be out of date
- Updating main memory alone is not enough

Example



Core 1 reads X: obtains 24 from memory and caches it

Core 2 reads X: obtains 24 from memory and caches it

Core 1 writes 32 to X: its locally cached copy is updated

Core 3 reads X: what value should it get?

Memory and core 2 think it is 24

Core 1 thinks it is 32

Notice that having write-through caches is not good enough⁷

Bus Snooping

- Cache on each core ‘snoops’ (i.e. watches continually) for write activity concerned with lines which it has cached.
- This assumes a bus structure which is ‘global’, i.e all communication can be seen by all.
- More scalable solution: ‘directory based’ coherence schemes

Snooping Protocols

- Write Invalidate

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

Snooping Protocols

- Write Update (aka write broadcast)
 - core wanting to write grabs bus cycle and broadcasts 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 core 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-cores
- 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 'write-back' scheme other cores might re-fetch old value from memory on a cache miss ☹
- We need a protocol to handle all this.

MESI Protocol (1)

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

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. (multicore '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 core activity (i.e. cache access)
 - Due to bus activity - as a result of snooping
- Cache line has its state affected only if address matches

MESI Protocol (4)

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

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)

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

MESI Local Read Miss (2)

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

MESI Local Read Miss (3)

- One cache has M copy
 - core makes bus request to memory
 - Snooping cache puts copy value on the bus
 - Memory access is abandoned
 - Local core 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

MESI Local Write Hit (2)

- S
 - core broadcasts an invalidate on bus
 - Snooping cores 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 cores

- No other copies
 - Value read from memory to local cache (?)
 - Line is 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 cores 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

- core issues bus transaction marked RWITM
- Snooping core 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)

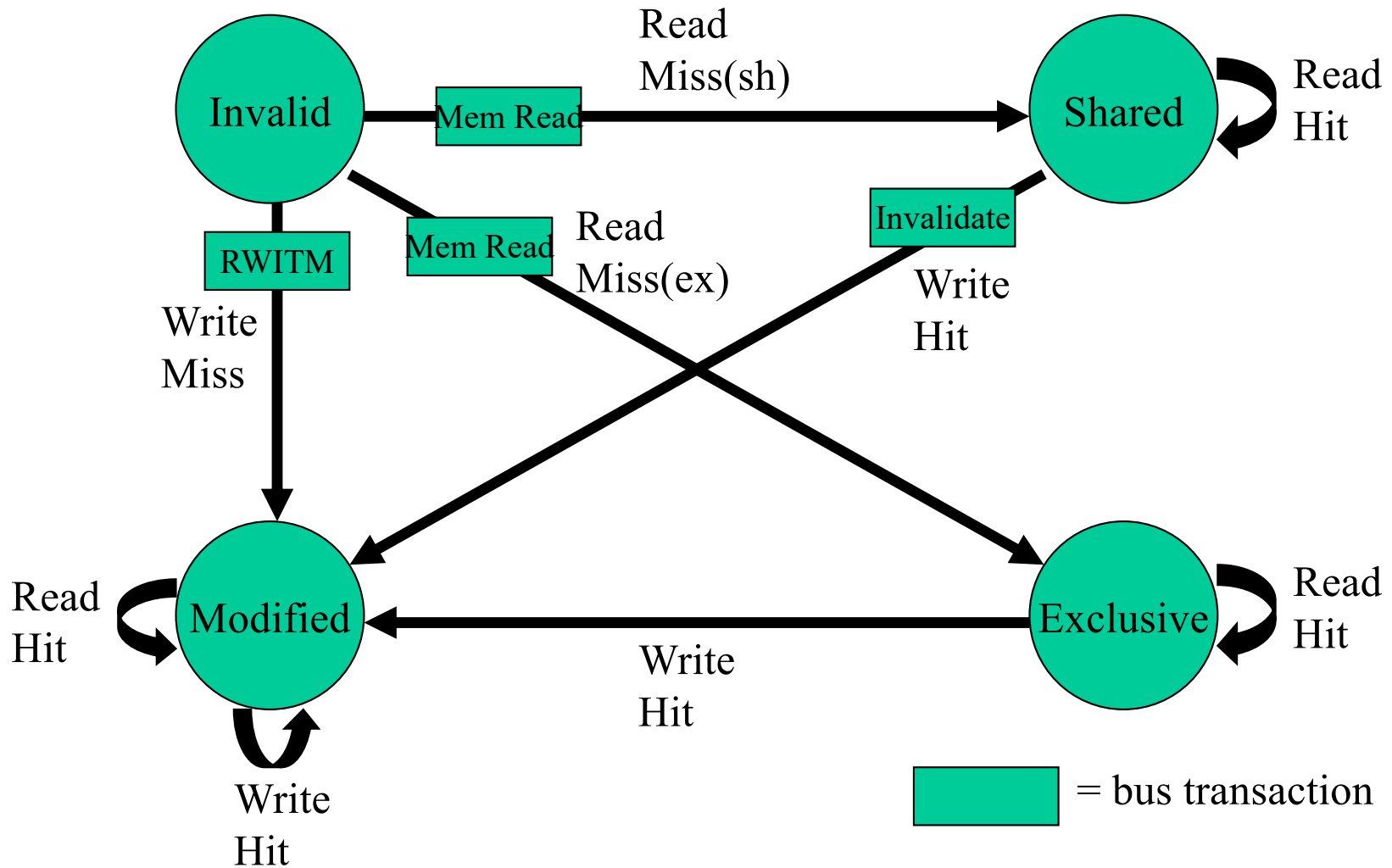
Another copy in state M (continued)

- Original local core 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

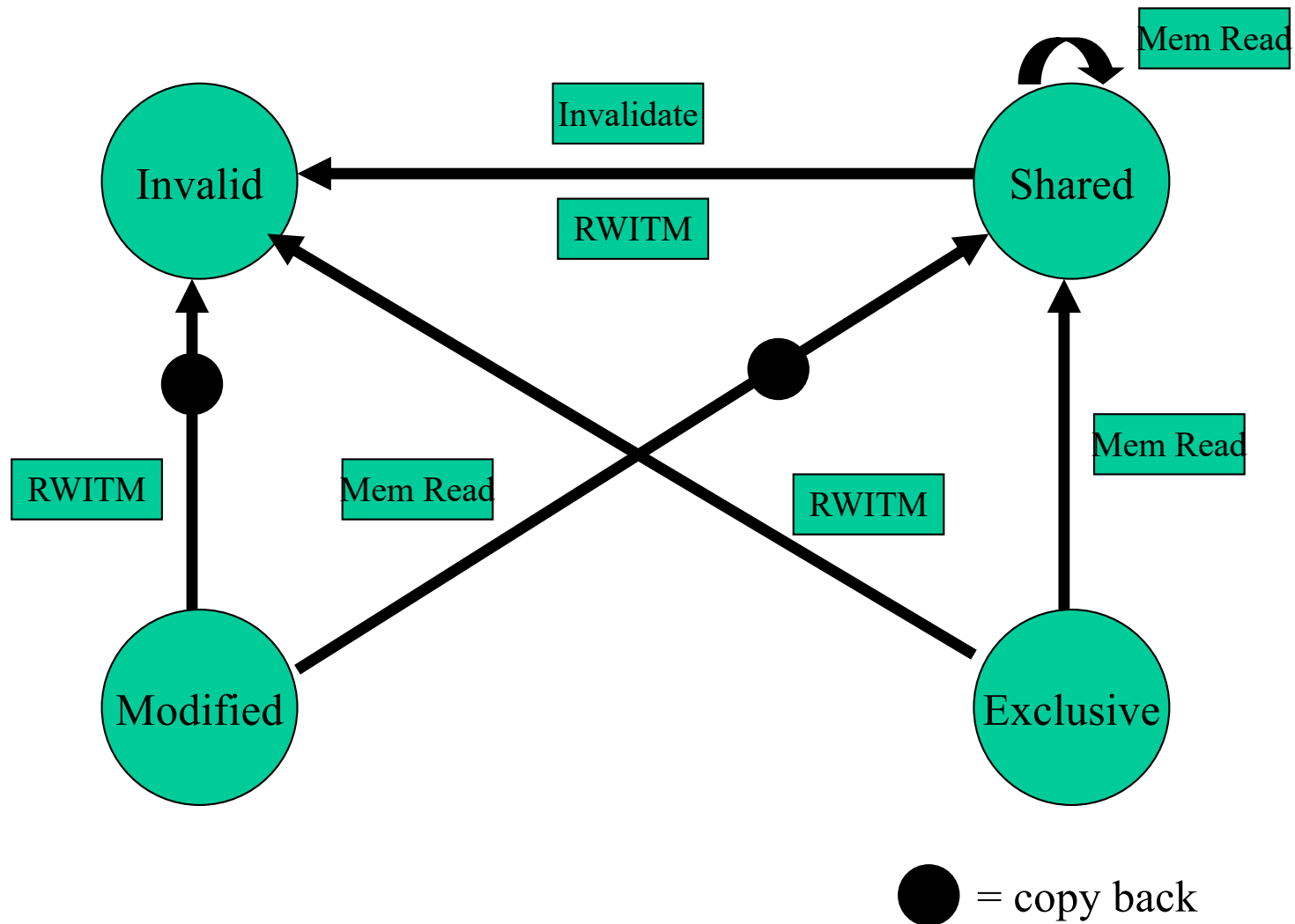
Putting it all together

- All of this information can be described compactly using a state transition diagram
- Diagram shows what happens to a cache line in a core as a result of
 - memory accesses made by that core (read hit/miss, write hit/miss)
 - memory accesses made by other cores that result in bus transactions observed by this snoop cache (Mem read, RWITM, Invalidate)

MESI – locally initiated accesses



MESI – remotely initiated accesses



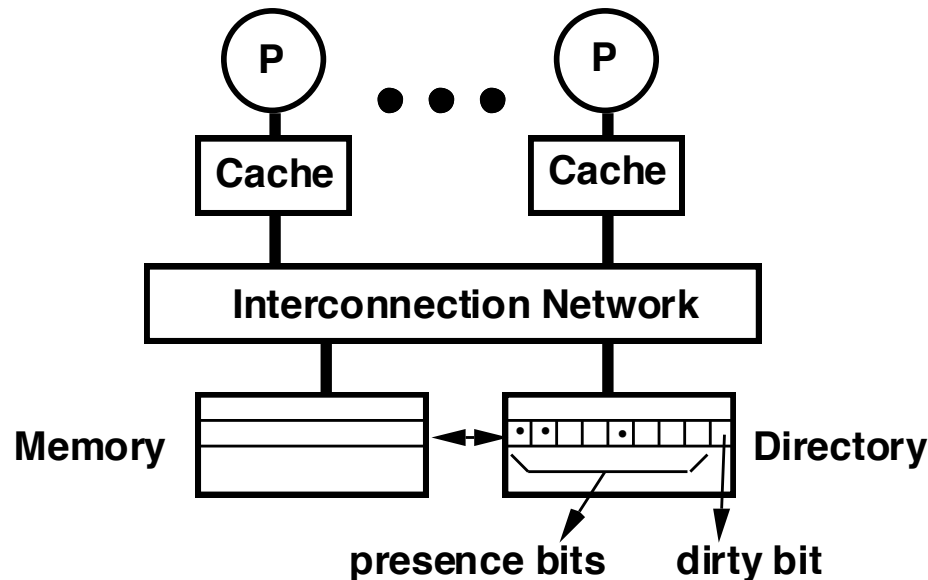
MESI notes

- There are minor variations (particularly to do with write miss)
- Normal ‘write back’ when cache line is evicted is done if line state is M
- Multi-level caches
 - If caches are inclusive, only the lowest level cache needs to snoop on the bus

Directory Schemes

- Snoopy schemes do not scale because they rely on cheap broadcast (bus)
- Directory-based schemes allow scaling.
 - avoid broadcasts by keeping track of all PEs caching a memory block, and then using point-to-point messages to maintain coherence
 - they allow the flexibility to use any scalable point-to-point network

Basic Scheme (Censier & Feautrier)



- Assume "k" cores.
- With each cache-block in memory: k presence-bits, and 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

– Read from main memory by PE-i:

- If dirty-bit is OFF then { read from main memory; turn $p[i]$ ON; }
- if dirty-bit is ON then { recall line from dirty PE (cache state to shared); update memory; turn dirty-bit OFF; turn $p[i]$ ON; supply recalled data to PE-i; }

– Write to main memory:

- If dirty-bit OFF then { send invalidations to all PEs caching that block; turn dirty-bit ON; turn $P[i]$ ON; ... }
- ...

Implications for software

- Cache misses in sequential programs: 3C's
 - Cold
 - Capacity
 - Conflict
- Cache misses in shared-memory programs: 4C's
 - Coherence misses: cache line can get evicted because of invalidation from another core

Sources of invalidation misses

- True-sharing
 - A variable or array element is read and written by two or more cores repeatedly
- False-sharing
 - Two or more cores read and write distinct variables or array elements that happen to be in the same cache line
- Sharing results in “ping-ponging” of cache lines between cores
 - Reduces performance
 - To improve performance, try to reduce sharing of cache lines between cores

Implementing atomic instructions

- Cache coherence mechanism can be used to implement atomic instructions on data that fits on a cache line
- Key idea:
 - Bring line into cache and “pin” it so it cannot be stolen by other cores
 - If another core tries to access line, that instruction is aborted and must re-execute
 - Perform operation on line
 - Unpin line
- Examples:
 - swap(reg,addr)
 - swap value in memory at address addr with value in register reg
 - Implementation:
 - Bring line into cache and pin it
 - Swap values in register and cache line
 - Unpin cache line
 - atomic add(reg,addr)
 - Similar to swap except that core performs an add operation and writes value to cache line
 - Other atomic operations are provided in ISAs

Implementing atomic instructions

- What if atomic operation you want is not implemented by hardware?
- Two cases:
 - Data fits in cache line but operation is not implemented in ISA
 - compare-and-swap (addr, reg1, reg2) //CAS
 - check if (value in addr) = (value in reg1)
 - if so, swap values in reg2 and addr and return SUCCESS else return FAIL.
 - Using CAS

```
top: v = M[addr];  
    w = f(v); //f is complex operation not supported in hardware  
    if (CAS (addr,v,w) fails) go to top;
```
 - Data does not fit in cache line
 - Usually handled in software using locks (see later)

Summary

- Caches are transparent to sequential programs
 - Do not affect semantics
 - May improve performance
- Caches in shared-memory machines
 - Need cache coherence
 - Write-invalidate vs. write broadcast
 - MESI protocol is pretty much standard
- Software implications
 - New source of misses: coherence/invalidation misses
 - 4 C's of cache misses: cold, capacity, conflict, coherence