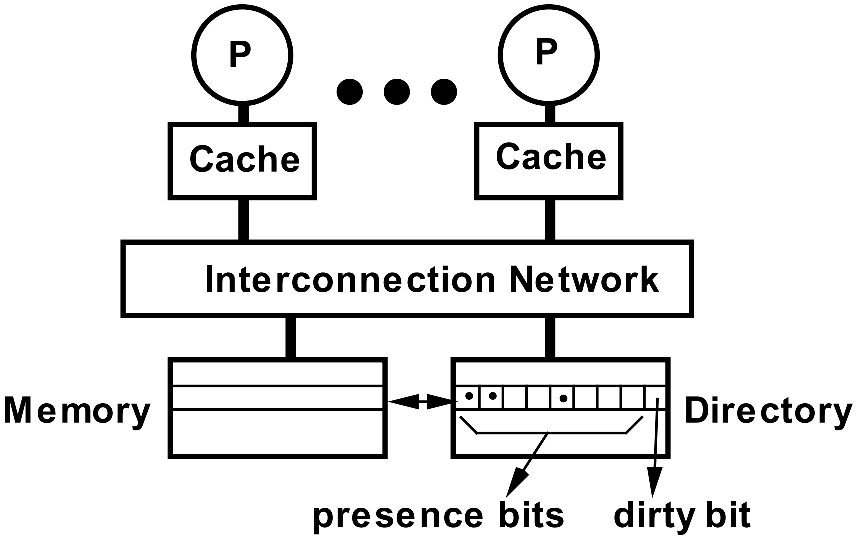
Basic DSM Cache Coherence

[§10.3] Let us start off by considering a full bit-vector approach.



For each block of memory, assuming there are *k* processors, it maintains at the home node of the block …

- *k* presence bits *p*[..]
- 1 dirty bit *D*

Cache state is represented the same way as in bus-based designs (MSI, MESI, etc.).

- On a read by processor *i*, the home node reacts this way:
  - **If** (*D* == 0) { supply data; *p*[*i*] = 1; }
  - **else** { send intervention to owner; update home; *D* = 0; *p*[*i*] = 1; supply data to *i*;}
- On a write by processor *i*; tell how the home reacts:
- On a write by processor *i*; tell how the home reacts:
  - **if** (*D* == 0) { ______________________________ ; *D* = 1; *p*[*i*]=1; supply data to node *i*; }
  - **else** { ____________________________________; *p*[*owner*] = 0; *p*[*i*] = 1 ; supply data to *i*;}

On the replacement of a dirty block by node *i*, the data is written back to memory and the directory is updated to turn off the dirty bit and *p*[*i*].

On the replacement of a shared block, the directory may or may not be updated.

How does a directory help? It keeps track of which nodes have copies of a block, eliminating the need for _____________ .

Would directories be valuable if most data were shared by most of the nodes in the system?

Fortunately, the number of valid copies of data on most writes is small.

The attached animation uses the MESI protocol, with 3 block states in main memory:

- EM (exclusive or modified)
- S (shared)
- U (unowned)

✓ Network transactions for coherence

  - Read: read request
  - ReadX: read exclusive (or write) request
  - Upgr: upgrade request
  - ReplyD: home replies with data to requestor
  - Reply: home replies to requestor with IDs of sharers
  - Inv: home asks sharer to invalidate
  - WB+Inv: home asks owner to flush and invalidate
  - WB+Int: home asks owner to flush and change to S
  - Flush: owner flushes data to home + requestor
  - InvAck: sharer/owner acks an invalidation msg
  - Flush+InvAck: Flush, piggybacking an InvAck message

- ✓ Notation
    - Transaction (Source → Destination)
    - H = Home node

The following example is used in the animation:

Proc action	P1 state	P2 state	P3 state	Dir state @home	Network messages	# of hops
R1	E	–	–	EM, 100	Read (P1 → H), ReplyD (H → P1)	2
W1	M	–	–	EM, 100	—	0
R3	S	–	S	S, 101	Read (P3 → H), WB+Int (H → P1), Flush (P1 → H, P3)	3
W3	I	–	M	EM, 001	Upgr (P3 → H), Reply (H → P3) // Inv (H → P1), InvAck(P1 → P3)	3
R1	S	–	S	S, 101	Read (P1 → H), WB+Int (H → P3), Flush (P3 → H, P1)	3
R3	S	–	S	S, 101	—	0
R2	S	S	S	S, 111	Read (P2 → H), ReplyD (H → P2)	2

Scaling with number of processors

In order for directory schemes to be practical, they must scale gracefully.

- Scaling of memory and directory bandwidth
    - Centralized directory is bandwidth bottleneck, just like centralized memory.

    - How can we maintain directory information in a distributed way?

- Scaling of performance characteristics
    - traffic: # of network transactions each time protocol is invoked
    - latency: # of network transactions in critical path each time
- Scaling of directory storage requirements
    - Number of presence bits needed grows as the number of processors.

      E.g., 64-byte block size and 1024 processors. How many bits in block, vs. # of bits in directory?

Directory organization affects all of these issues.

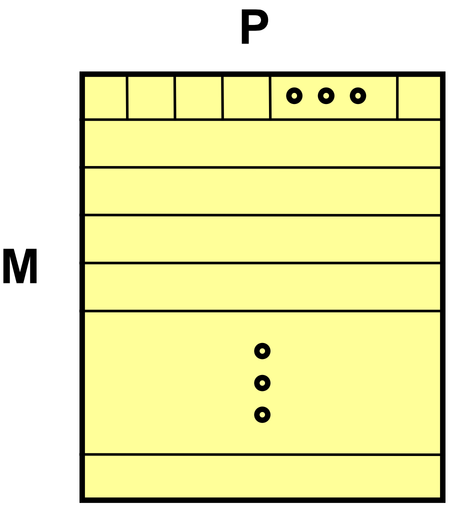
Organizing a memory-based directory scheme

All info about copies is colocated with the block itself at the home

This works just like a centralized scheme, except that it is distributed.

Scaling of performance characteristics

- Traffic on a write is proportional to number of sharers.
- Latency? Can issue invalidations in parallel.



Scaling of storage overhead? Assume representation is a full bit-vector.

Optimizations for full bit-vector schemes

- Increase ____(1)___ size (reduces storage overhead proportionally).
- Use multicore nodes (one bit per multicore node, not per processor)

- still scales as *pm*, but only a problem for very large machines
    - 256 procs, 4 per chip, 128B line: ____(2)___% o’head

► Reducing “width”: addressing the *p* term

- Observation: most blocks are cached by only few nodes
- Instead of keeping a bit per node, make entry contain a few ____(3)___.

  If *p* = 1024, 10-bit _______ ⇒ can use 100 ______ and still save space.
- Sharing patterns indicate a few pointers should suffice (five or so).
- We also need an overflow strategy for when there are more sharers than pointers.

► Reducing “height”: addressing the *m* term.

- Observation: number of memory blocks >> number of cache lines.
- Thus, most blocks will not be cached at any particular time; therefore, most directory entries are useless at any given time
    - organize directory as a cache, rather than having one entry per memory block (key is ___(4)___, value is ___(5)__)

Organizing a cache-based directory scheme.

In a cache-based scheme, the home node only holds a pointer to the rest of the directory information.

The copies are linked together via a distributed list that weaves through caches.

Each cache tag has a pointer that points to the next cache with a copy.

- On a read, a processor adds itself to the head of the list (communication needed).
- On a write, it makes itself the head node on the list, then propagates a chain of invalidations down the list.

Each invalidation must be acknowledged.

- On a write-back, the node must delete itself from the list (and therefore communicate with the nodes before and after it).

Disadvantages: All operations require communicating with at least three nodes (node that is being operated on, previous node, and next node).

Write latency is proportional to number of sharers.

Synchronization is needed

Advantages: Directory overhead is small.

Work of performing invalidations can be distributed among sharers.

The IEEE Scalable Coherent Interface has formalized protocols for handling cache-based directory schemes.

The SSCI protocol

- SCI (Scalable Coherence Interface) protocol
    - IEEE standard, ratified in 1993
    - 7 state bits, 29 stable states + many pending states
- For illustration we will use Simple SCI (SSCI)
    - Retains similarity with full-bit vector protocol:
        - MESI states in the cache
        - U, S, EM states in the memory directory
        - Replaces the presence bits with a pointer
    - Similar features to SCI
        - Overall protocol operation
        - Doubly linked list
    - Many possible race conditions, which are mostly ignored in the illustration

- Additional coherence network transactions (in addition to those used in full bit-vector approach):
    - WB+Int+UpdPtr
    - UpdPtr: update next/prev/head pointers

Here is the example used in the animation.

Proc action	P1 state	P2 state	P3 state	Dir state @home	Network message	# of hops
R1	E,0,0	–	–	EM, 1	Read (P1→ H), ReplyD (H→P1)	2
W1	M,0,0	–	–	EM, 1	—	0
R3	S,3,0	–	S,0,1	S, 3	Read (P3 → H), Reply (H → P3), WB+Int+UpdPtr (P3→P1), Flush (P1 → H, P3)	4
W3	I,3,0	–	M,0,0	EM, 3	Upgr (P3 → H) // Inv (P3 → P1) InvAck(P1 → P3)	2
R1	S,0,3	–	S,1,0	S, 1	Read (P1 → H), Reply (H → P1), WB+Int+UpdPtr (P1→P3), Flush (P3 → H, P1)	4
R3	S,0,3	–	S,1,0	S, 1	—	0
R2	S,2,3	S,0,1	S,1,0	S, 2	Read (P2 → H), ReplyD/ID (H → P2), UpdPtr (P2 → P1)	3