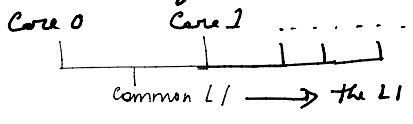
Cache coherence

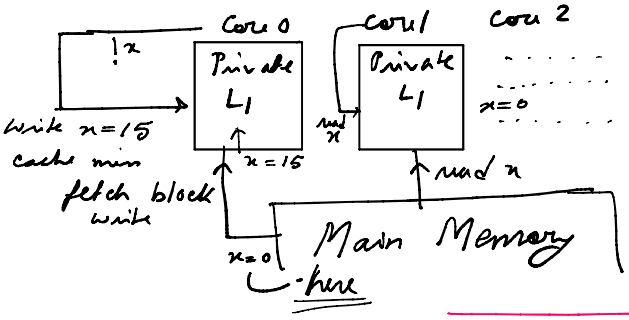
20 April 2020 21:03

Consider a shared mem arch

Shared memory should offer a consistent view to all cores



→ the L1 will be too big & hence too slow.



Core 0 can keep on writing to x other values in future in private L_1, Core 1 keeps on reading from its own private L_1 as $x=0$

INCOHERENT

Purpose of Shared mem is lost.

What is Coherence

Requirements →

1) Any read r by core C_i from some addr x must return the most recent write w to addr x by core C_i if no other core has written to addr x in between w and r.

→ Uni-core view

2) If core C_1 writes to X & C_2 reads after sufficient time & there are no other writes to X in between by any core, then C_2's read returns the value from C_1's write.

3) Writes to the same location have to be serialized, i.e., any two writes to X must occur in the same order on all the cores. Every core sees the same order of write values for a variable.

How to Achieve Coherence in presence of Caches

1. No cache → bad performance
2. Common L_1 → "
3. Private write-through cache — then the shared memory is coherent, but once a read is done, it is not updated following writes by others. (Incoherent)
4. Need to force cache to see writes by other caches.

Any one of these can ensure property ②:

- Broadcast writes to update other caches.
  - → Write update coherence.
- Writes prevent hits to other caches.
  - → Write invalidate coherence

property ①:

- Write broadcast on shared bus, the order in which they appear on bus is the unique order seen by all cores (shared bus can be bottleneck) } snooping
- Each block is assigned an 'ordering point' → (directory)

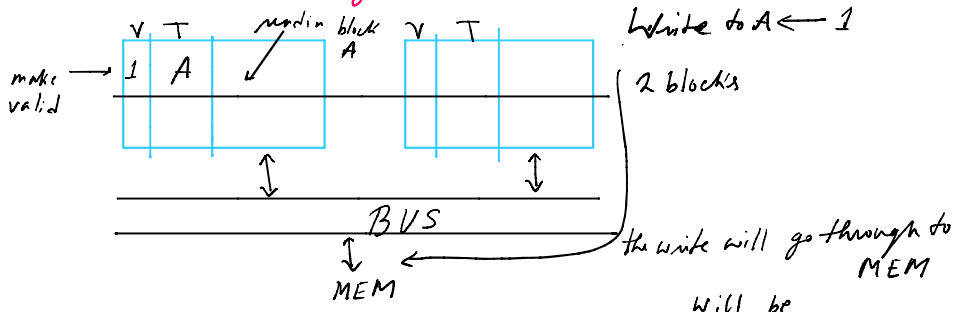
Different ordering points can be used for diff blocks. For each block, all accesses are ordered by the same entity. Diff blocks have diff entity → no contention.

(Directory based coherence)

Diff blocks have diff arbitr → no contention.
(Directory based coherence)

2X2 → 4 possible coherence schemes.

Write Update Snooping coherence



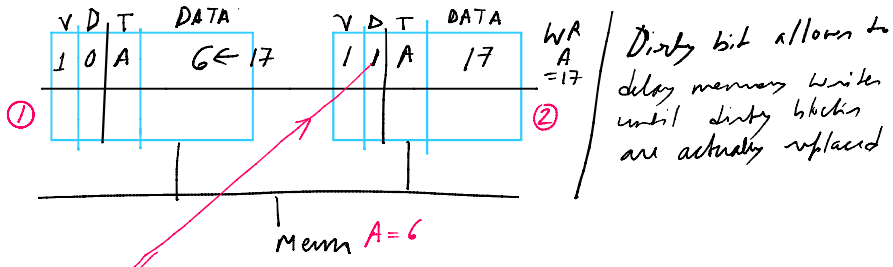
will be now monitored by the other cache

Write update in Cache 1. — checks it also has block A ← snooping

What if both processors try write to
- same location exactly at same time
  - bus arbitration logic will decide the order.

Optimization for write updates

in W update scheme, every write to a location by every core, need updating in memory, which is slow.
Mem — can't handle so much write traffic.



Dirty bit allows to delay memory write until dirty blocks are actually replaced

The mem not yet updated
If cache 1 has a new read for block A
(after replacing previous instance)
cache 1 makes a read req,
cache 2 snoops req from Bus, the dirty bit tells it that it only has the updated value & memory does not have updated value.
Cache 2 will give the value.

If now cache 2 updates A=20, then cache 1 snoops the value by cache 2's broadcast.
Cache 2 still has dirty bit set as mem still not updated. Only if cache 2 has the block replaced, then memory will get updated.
So, as long as one cache has dirty bit set, it

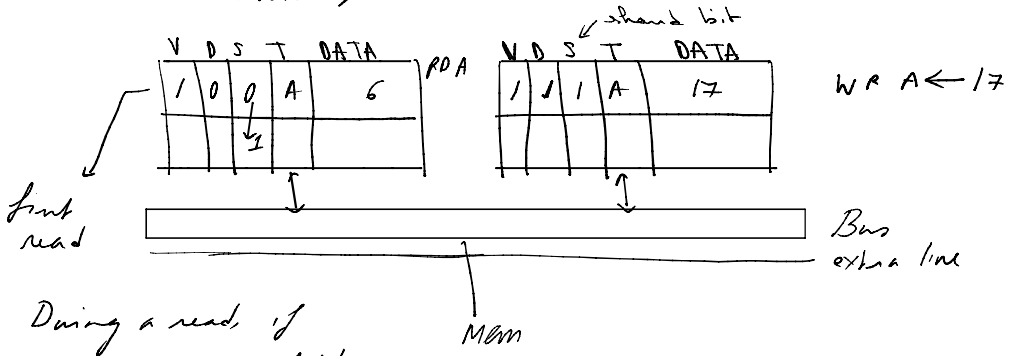
then memory will get updated.
So, as long as one cache has dirty bit set, it has most recent value.
Now if cache 1 writes A = 30, then it is snooped by cache 2 & the dirty bit of cache 1 gets set with that of cache 2 getting unset.

Thus many memory updates are avoided by designating a cache dynamically as having the recent value.
Only when the cache with 'dirty' set (i.e., the last writer) gets block replaced, then Mem write will happen.

We also read from memory only when no cache has dirty bit set.

Write Update Opt for bus writes

Memory is saved by bus is still bottleneck.



During a read, if any cache sees that it already has the same block, it pulls the shared line up;
When A is written by cpu 2, cpu 1 cache snoops the value, updates its shared bit & also pulls up the shared line.
Then cache 2 also will set S = 1 knowing that the block is shared.
Now if cache 2 updates A, it has S = 1, so it needs to broadcast.
Say cache 1 reads B = 2 from mem, V = 1, D = 0, S = 0;
Now if it updates B = 5, it checks S = 0, hence no broadcast required, but sets D = 1 as no more consistent with memory.

Write Invalidate Snooping coherence

Consider last scheme setting
Let cpu 1 read A, V = 1, T = A, S = D = 0 ← changes valid bit in cache 1 = 0.
. . . . 2 writes A = 3, will be snooped by cache 1
V = 1, T = A, S = 0, D = 1

Now A read in cache 1 will have a miss, it sends a request to the bus & it is provided by cache 2.

Now, in cache 1 ⟶ V = 1, T = A, S = 1, D = 0
cache 2 ⟶ V = 1, T = A, S = 1, D = 1

So in this type protocol,
if somebody writes then there is a (low latency) miss by others in next read.

So in this type protocol,
if somebody writes then there is a (low latency) miss by others in next read.

Advantage → (many unnecessary writes avoided).
Ex: say core 2 writes A, then core 1 no more has valid A but S = 1.
Now, core 2 can make many updates (S = 0, D = 1) on its local copy w/o any broadcast.
Only when core 1 reads, it will have a miss & then get last update by core 2.
So the data is periodically processed by diff cores & any core makes many consecutive updates, this is really a good idea.

Application Activity	Update	Invalidate
Burst of writes to one addr	Each WR is broadcast ✗	first WR invalidates, others on hits ✓
WR diff words in same block	update sent for each word. ✗	" ✓
One core has data (WR) another core consumes (RD)	Producer sends update Consumer hits ✓	Producer invalidates Consumer Misses ✗

Modern CPUs —— invalidation technique preferred.

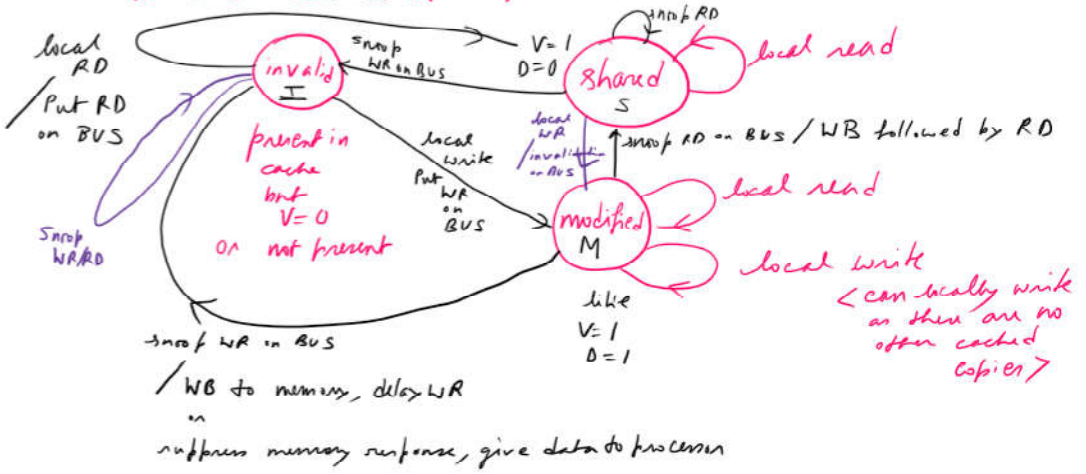
Primary reason → OS migrates threads, data remains in source core.

Update
Keeps updating old core cache until they can get replaced due to code running in old core

Invalidate
First write invalidates old copy, no more traffic

MSI Coherence (invalidation based)

A block can be in 3 possible cache states.



Cache — Cache Transfer

- Core 1 (C1) has block B in M state
- C2 puts RD request on BUS
- C1 provides data →

Solution 1 C1 cancels C2 req, C1 WB to mem ←

- C2 puts RD request on BUS
- C1 provides data →

Solution 1 — C1 cancels C2 req, C1 WB to mem, C2 retries & gets data from mem
2 times mem latency

Solution 2
C1 subverts mem & responds directly to C2 (intervention BUS signal)
Mem gets data from C1 in parallel.
↳ HW complex but popular

Memory has to get the fresh data from C1 since after transfer both C1 & C2 share the data (S state). Both think the block is not dirty. So in future there is no body clearly responsible to WB the data.

Avoiding Memory write in cache—Cache transfer

MSI: We are writing to memory everytime we do a cache to cache transfer.

C1 has a block B in M state
C2 wants to read B, C1 has it → C1 : S, C2 : S ← Mem write
Next C2 writes to B, C1 : I, C2 : M
Next C1 Read, C2 responds with data → C1 : S, C2 : S

Next C3 Reads ← Mem provides data ← Mem read
as C1/C2 both in S state with D = 0

both have data but none provides <no clean rule like who will provide>

Next C4 read ← Mem, C1 : S
C2 : S
C3 : S

Many unnecessary Mem reads (costly)

We should have had a non-modified (non-M) version of the same block in just one of the caches & we could have made it responsible for distributing data to other cache & finally WB to Mem.

Introduce 'O' (OWN) state

MOSI protocol

like MSI except
M ⟹ snoop a RD ⟹ O (not S)

.. 11

like MSI except

M ⟹ snoop a RD ⟹ O (not S)
no Mem access. (data given directly)

O is like S except

O ⟹ snoop a RD ⟹ give data
WB to mem if block replaced

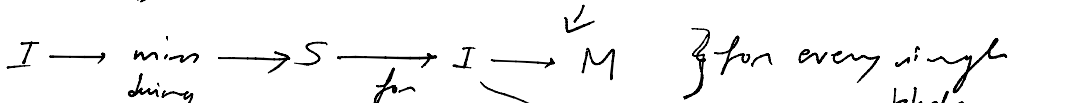
M: exclusive RD/WR access, Dirty
S: shared read access, clean (mem has up to date copy)
O: Shared read access, Dirty (only one cache in O state)

MOSI inefficiency

Thread-private data (data accessible to only one thread)
e.g. stack of each thread

(or 4 separate program threads running in 4 cores)

Data read then write behavior



uniprocessor → V:0 ⟹ miss during read ⟹ V:1 ⟹ hit write ⟹ D:1

The E (Exclusive) State

M: Exclusive Access (RD/WR), Dirty
S: Shared Access (RD), clean
O: Shared Access (RO), Dirty
E: Exclusive Access (RO/WR), clean (and not update mem)

	MSI	MOSI	MESI	MOESI
RD A	I → S (miss)	I → S (M)	I → E (M)	I → E (M)
WR A	S → M (Invalidation sent)	S → M (Invalidation sent)	E → M	E → M
	2 bus accesses	2 bus accesses	invalidation req (off chip bus access not required)	

detect during read that we are the only one having the block