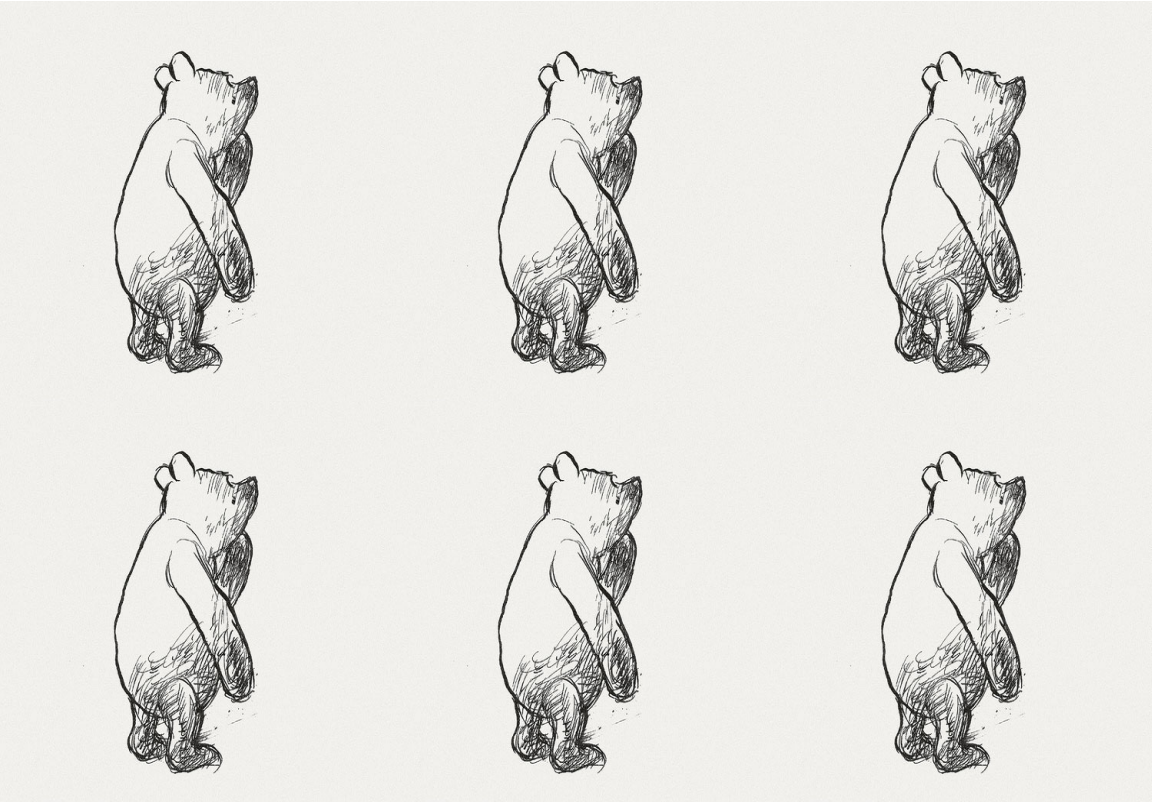
Chapter

Cache Coherence



Caches in multi-processor systems

Cache is for temporary storage and fast access of data and instructions.
For data the contents of cache and the contents of memory may not be identical if the data item has been updated.

What happens if we have more than 1 core
- the core can be common to all cores
- each core can have its own cache

Common cache
All cores have the same view of the data
but
access the cores become a problem …
- the cache must allow for multiple independent accesses, or cache access will become a bottle neck. (Expense and complication)
- the amount of cache for each core is effectively reduced (increasing cache sizes causes its own problems

Private data – access by one processor to improve performance as for single processor.

Shared data – replicated in multiple caches. Reduces access time, memory bandwidth **plus** contention.

BUT introduces the problem of *cache coherence*

Actually even defining what is meant by seeing the same data is not trivial.

How to ensure that all processors see the same data.

Two interlinked problems.

Coherence defines values returned by a read

Consistency determines when a written value will be returned by a read.

Consistency when – coherence how

4 Cache

Multi-core machine.

Each core has its own cache. (L1)

Shared cache -> multiple simultaneous read/writes.

For n cores you expect to need n times as much cache.

Bigger cache means slower performance

Simple cache local to each core provides best performance at least at level 1.

Level 2 cache is larger and is not accessed by every core for every instruction.

Level 2 tends to be shared

Cache Coherence

Data (instructions) read from memory causes no problems.

A value in memory duplicated in one or more caches.

Writing values to memory locations causes problems.

Single processor time taken

Multi-processor consistency.

Options

Write back: cache is updated. Memory is only updated when cache line is over written. Multiple writes to cache only require one write to memory.

Write through: write operations to main memory as well as cache. Cache line replacement does not provoke a write.

Maintaining coherence

Can use a directory protocol or a snooping protocol

A directory protocol is centralised, a snooping protocol is distributed.

Snooping
Each cache controller must have intelligence to be able to generate and respond to messages from other cache controllers. They collectively respond to a core's request for a data item with the owner of the item returning the item.
They depend on the performance of the interconnection network, which must deliver broadcasts in a consistent order

Directory protocol
Cache controller's unicast to the memory controller that is *home* to the block.
Memory controller maintains a directory for each block.

7 Cache write

Write back: **Write through**:

Line loaded into cache, which had been updated in another cache – only problem with *write back*

but any cache which already has data has a problem, no matter which option is used.

Software solution

Compiler identifies places where data may be unsafe for caching. OS prevents them from being cached.

Enforces repeated reads from memory, even when it would have been safe to use a cached value.

Hardware simple

Hardware solutions:

Directory Snooping

Snoop

To pry into the private affairs of others, especially by prowling about.

Directory based:

Centralised controller (part of main memory)

Directory in main memory

- keeps information about data in the local caches.
- issues commands to transfer data between caches and between memory and caches
- local actions with global impact, must be reported to the controller.

Overhead higher, but scaling better. Good for large scale systems with multiple buses.

Status of a block in central location – single point of failure

Snooping:

Cache with a copy of the block has a copy of the sharing status of the block – no central directory

Caches all available via a bus or switch.

Update actions on cache must be broadcast.

All controllers monitor (snoop) the bus or switch to see if they have a copy of the block.

Suitable for a single bus based system.

Local caches reduce bus based traffic, the broadcast/snoop has the potential to nullify this advantage.

Two approaches: write invalidate; write uppdate.

Directory has a global view of the system.

Individual cache controllers request access via the directory which issues commands to transfer data between caches and between caches and memory.

A core wants to write to a cache location.

It sends a request to the directory. The directory sends a message to all cores which have that cache line, warning them that the contents are no longer valid.

It receives acknowledgements to all messages, and grants the requestor exclusive access to the line.

A core that wants to read a value from a line for which another core has exclusive access sends a request to the directory.

The directory provides an up to date line to the requestor – cache to cache copy; write back to memory and then fill from either memory or cache.

Maintaining coherence

Directory protocol

Cache controller's unicast to the memory controller that is *home* to the block.
Memory controller maintains a directory for each block, with information such as current sharers of a block, or current owner of a block.

So it responds to the request for a data item either by satisfying the request or by forwarding it to the cache controller which owns the data.

Snooping is simpler, but it doesn't scale well.
Directory protocols tend to be slower when an extra message is generated to satisfy a request

Cache with a copy of the block needs to know if any other cache has a copy of the block.

If a core wants to update a value in a cache line. It broadcasts this fact.

Options are write update and write invalidate.

Write invalidate:

The cache line is then “owned” by the writing process. It can continue to make updates until another core needs the line. Another core which needs the line, must request it and the line reverts to shared. At this point the memory can either be updated and the cache filled from memory, or the requesting cache can be updated from the modified cache. May have a dedicated invalidate bus

We can use write through in which case the memory is already up to date

Write update:

the cache which wishes to write into a cache, broadcasts the new value and this is used by other controller to update the appropriate line in their cache and in memory.

Multi-processor

Dedicated cache

Need to ensure that all cores have the same view of the data

This is referred to as *cache coherence*

It means that when any core accesses any item of data, it must get the same value.

(Normal problems of parallel calculations which require access to the same items of data and these need to be addressed separately.)

Multi-processor

Consider the following situation.

Processor 1 requires a data item **A** and it is not in any cache.
1 fetches **A** from memory and uses it, but does not change it.

Processor 2 requires **A** and it is not in its cache. Processor 2 may get the item from processor 1's cache or from memory. It has to know processor 1's cache has the data item.

What happens if either 1 or 2 change **A.** Don't want to force an update to both caches, because we don't know that the other processor will ever want **A** again. If they do, then they have to get the value from the correct place**.**

How to ensure that the correct value is available to all processors, while still maintaining the performance advantage of the cache.

Multi-processor

How to communicate what is going on?

No central information.

View of the system different from every machine.

Behaviour is emergent

MESI

Illinois Protocol

Each cache line has a 2 bit tag encoding one of four states

Modified **E**xclusive **S**hared **I**nvalid

Exclusive
Present in the current cache and its value is the same as the value in memory – c*lean*

State when first in cache

Shared
The cache line may be stored in other caches and is clean

Modified
Present in the current cache and its value is the different from the value in memory – *dirty*. The value must be written back before the main memory can be accessed by another core

Invalid
Cache line is invalid (unused)

MESI read

The caches share a common bus to main memory. Aim is to minimise accesses to the main memory.

The first time that a value is requested by a core (1), then it is copied into the local cache and marked *exclusive*.
Subsequent requests from that core are satisfied by that cache.

Actions on read

A request from another core (2) is intercepted and satisfied from the existing cache. The cache line in both caches is marked as *shared*.

Repeated requests from cores 1 and 2 are satisfied from their local cores. A request from a third core can be satisfied from either cache. All three caches will marked as shared

MESI write

An write is made to a cache line which is in *exclusive* state.
The value is changed in cache and the tag is changed to *modified*.

Modified means that at some point the data in memory must be updated.

An write is made to a cache line which is in *shared* state.
The value is changed in cache and the tag is changed to *modified* and the state of any that line in any other cache is set to *invalid*.

A read from any cache line marked as invalid, must be satisfied by fetching the data from the *modified* cache.

For a cache to be set *invalid*. Either the cache which is moving to the *modified* state, must broadcast, **R**equest **F**or **O**wnership or the other caches must **snoop** on the write lines of all other caches and mark their own caches as invalid as appropriate

Cache full

As with single core caches it may be necessary to overwrite a line.
If the cache line is *shared* or *exclusive*, this may be done at any time

If the cache line is *modified*, the value in memory must be updated before the value is overwritten.
Writing the value to memory delays fetching the value from memory.
Provide a write buffer to store the modified value.
Write buffer must also snoop to pick up any requests to modified value before data is moved from write buffer to main memory

The caches must all snoop on the read/write requests to ensure that the tag bit stays up to date.

Exclusive listens for read and switches to *Shared*
Modified listens to read and arrange the most up to date value to be returned: either by returning the value; or by forcing the read to pause and writing the value back to main memory.

20 Operation

A single core can try to read or write a value into a cache line.

Four possible outcomes

Read Hit, Read Miss, Write Hit, Write Miss.

Read Hit: the data word is in the local cache. The word is transferred to the register.

Remains **M, E, S** as it was before.

21 Operation

A single core can try to read or write a value into a cache line.

Four possible outcomes

Read Hit, Read Miss, Write Hit, Write Miss.

Read Miss:

- No other cache contains the line. Reads the line from memory and sets that line from **I** to **E**
- If exactly one other cache has a clean copy (it will be **E**) it returns a message that it has a copy of this line. It moves from **E** to **S.** The requesting cache fetches the line from memory and marks it **S**.
- If more than one cache has a copy they all respond and the requesting cache fetches the line from memory and marks it **S**
- If one cache has the line in an **M** state. It blocks the read and returns the modified line to that cache, this modified line is also picked up my the main memory which updates the values. Both caches set the value to **S**

A single core can try to read or write a value into a cache line.

Four possible outcomes

Read Hit, Read Miss, Write Hit, Write Miss.

Write Miss:

- Signals on the bus a Read-With-Intention-To-Modify. When loaded it is marked **M**. If no other cache has a copy, then none responds. It one other has a clean copy in the **E** state it invalidates (**I**) it. If several others have the line in **S** they all mark it **I**.
- If another cache has the line in the **M** state, it responds warning the requestor that it has a modified copy and the requestor releases the bus. The core with the line in the **M** state writes it back to memory and then marks its copy **I**. The requestor then issues another RWITM which receives no response and loads from memory as before.

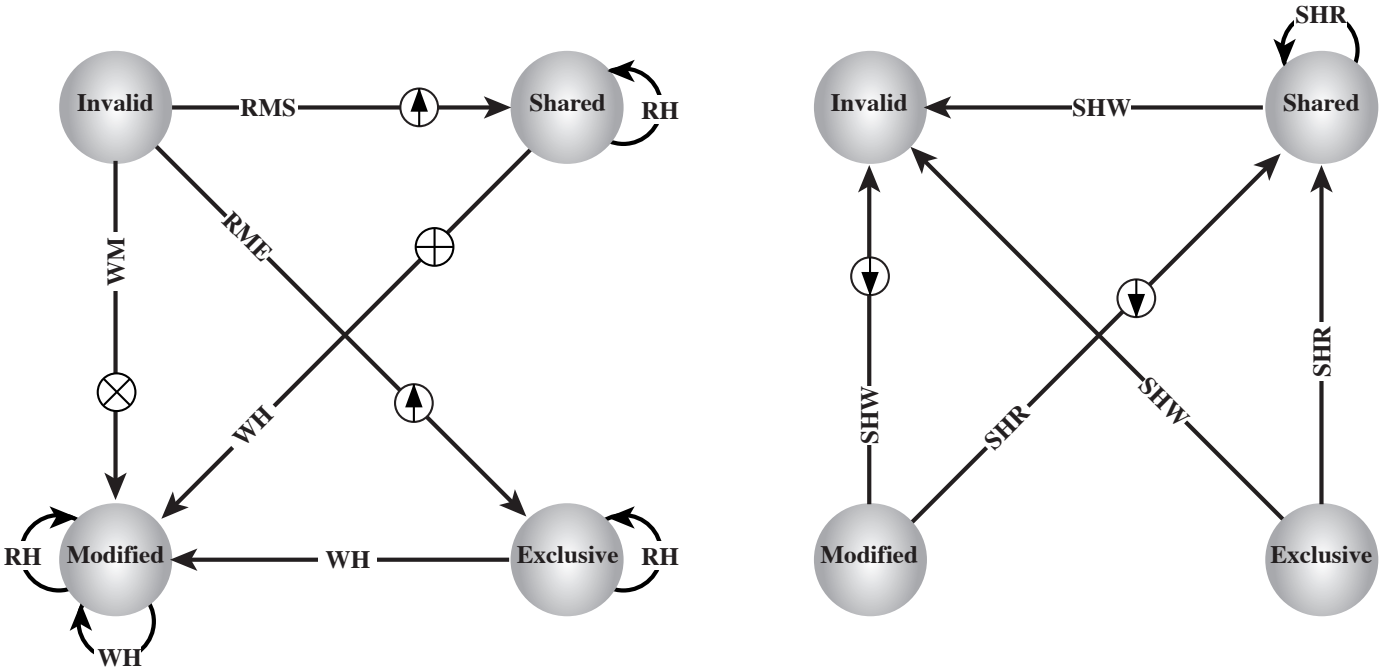
A single core can try to read or write a value into a cache line.

Four possible outcomes

Read Hit, Read Miss, Write Hit, Write Miss.

Write Hit:

- **M** it must already have exclusive access to this line, which it has already modified. It modifies it again.
- **E** with exclusive access it simply moves to the **M** state and updates the value.
- **S** signals on the bus and the other copies are marked as **I∫.** This one moves to **M**.



Above is a state diagram which describes the MESI protocol.

Each processor has its own state bits and will have a different view of the system

Atomic Instruction (sequence).

Retrieve and change

Used in software – construction of complex synchronisation schemes.

Simplest **atomic exchange**

Swap the values of a value in memory with a value in a register.

Set the value in register: Swap

value in memory is value that was in register

value in register is value that was in memory.

Lock	Memory	Value	Meaning
	A	0	Free or unlocked
	B	1	in use or locked.

Cannot be used for synchronisation

26 Locks (i)

Single ops

Want to acquire a lock

Store 1 in a register

Swap with a memory location. Atomic.

Memory now **must** contain 1.

No new process can acquire the lock

Check register

if 1. Was already in use. Try again

if 0. Not in use. Can use.

At end store 0.

Other methods are test and set.

set 1 if value is 0

Also fetch and increment.

0 free. >0 in use.

27 Locks (ii)

Multiple ops

Link register – special register.

Load linked – sets value in the link register to the memory address

Link register is reset on a context switch

Load link fails if the link register is set.

Store conditional works only if the link register has the store address.

28 Consistency

How consistent?

Strange question …. But not a true/false question.

What we naively expect.

Any write instantaneously updates the state of any copies in cache (and memory).

Insisting on this means pausing while time for actual updates occurs and hence slowing system.

Allow R and W to complete out of order and use sync to enforce ordering.

In fact the only important processes are writing and reading.

We can ask if the four possible orderings relaxed

Write must complete before Read

Write must complete before Write

Read must complete before Write

Read must complete before Read

Faster execution

More complex design.

29 Speculation

Use speculation to hide latency from cache coherence.

Similar to use in uni-processor.

Rapid review of the problems of multiprocessor architectures and description of the basic ideas needed to solve them.

Application of these ideas is far from simple - Hennessey has a good description of them and is an excellent source book.

Development here is consistent with Hennessey .