1 2008 Paper 7 Question 5

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2008p7q5.pdf

(a) Why are multi-level caches often used in preference to a single larger cache. How might the parameters of an L1 and L2 data cache typically differ?

The access time of a larger cache grows roughly proportional to $\sqrt{n}$. Given that smaller caches can already have incredibly high hit rates (order of 99% for set-associative L1 caches), it does not make sense to double the access time to increase this from 99% to 99.9%. Even if we were to do this, it would make sense to add another higher-level cache to reduce the miss-penalty to RAM. This would be similar to increasing the size of the L1 cache and introducing another L2 – which reverts to a multi-level cache.

- L1 cache is smaller than L2 cache
- L1 cache has a lower latency than L2 cache
- L1 cache has a lower miss penalty than L2
- L1 cache is physically closer to the core than L2 cache
- L1 cache is private while L2 cache is often shared
- There are often separate L1 data and instruction caches while there is almost always a combined data-instruction L2 cache.
- L2 cache has a lower compulsory miss rate than L1 cache
- L2 cache is usually set-associative with a higher number of sets than L1 cache.

(b) A processor's multi-level cache hierarchy consists of L1 and L2 caches with the following characteristics: L1 miss rate is 2%, L1 hit time is 2 cycles, local L2 miss rate is 20%, L2 hit time is 10 cycles, L2 miss penalty is 200 cycles. It is suggested that reducing the size of the L1 cache will improve overall performance by reducing the L1 cache's hit time to a single cycle. The reduction in L1 cache size will increase the L1's miss rate to 3%. Will average memory access time actually be improved? Clearly state any formulae you use and show your calculations.

With p_{L1}, p_{L2} as the L1 and L2 hit rates; t_{L1}, t_{L2} as the L1 and L2 hit times and t_{RAM} as the L3 miss penalty.

The expected memory access time $\mathbb{E}(t)$ is given by:

$$\mathbb{E}(t) = t_{L1} + (1 - p_{L1})\,(t_{L2} + (1 - p_{L2})\,t_{RAM})$$

Under the original memory hierarchy, this is equal to:

$$\begin{aligned}\mathbb{E}(t) &= t_{L1} + (1 - p_{L1})\,(t_{L2} + (1 - p_{L2})\,t_{RAM}) \\ &= 2 + 0.02 \cdot (10 + 0.2 \cdot 200) \\ &= 2 + 0.02 \cdot (10 + 40) \\ &= 2 + 0.02 \cdot 50 \\ &= 2 + 1 \\ &= 3\end{aligned}$$

Under the proposed memory hierarchy, this is equal to:

$$\begin{aligned}\mathbb{E}(t) &= t_{L1} + (1 - p_{L1})\,(t_{L2} + (1 - p_{L2})\,t_{RAM}) \\ &= 1 + 0.03 \cdot (10 + 0.2 \cdot 200) \\ &= 1 + 0.03 \cdot 50 \\ &= 1 + 1.5 \\ &= 2.5\end{aligned}$$

Therefore the expected memory access time on the proposed memory hierarchy is improved.

2 2014 Paper 7 Question 5

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2014p7q5.pdf

(a) A 4KB, blocking, private L1 cache with 16B lines sees the follwoing sequence of accesses from its core:

0x00001000	Load
0x00001010	Store
0x00002000	Load
0x00001010	Load
0x00003000	Load
0x00001010	Store
0x00001010	Store
0x00002000	Load
0x00001000	Load
0x00002000	Load

Assuming a write-allocate cache that is empty at first and implements the least-recently-used (LRU) replacement algorithm, what is the hit rate if the cache is:

(i) direct-mapped?

The cache is sized such that the rightmost hex character is the position in the cacheline, and 2–3 rightmost hex characters are which cacheline the address maps to. Therefore the addresses 0x00001000, 0x00002000 and 0x00003000 map to the same cacheline, while the address 0x00001010 is cached properly.

Using a direct-mapped cache, there are 3 hits from 0x00001010 and 1 hit on 0x00002000 at the end. Since there were 10 total memory addresses, the hit rate is therefore 0.4.

(ii) fully-associative?

Under a fully associative cache, we can choose which cacheline to evict. Since initially the cache is empty and is larger than 4 cachelines, the cache will not fill up and so each time we load in a new cacheline we can evict an invalid entry. So in this situation, a fully associative cache will only have compulsory misses.

We use 4 different addresses and therefore have 4 compulsory misses out of 10 accesses. So the hit rate is $\frac{10-4}{10} = 0.6$.

(iii) 2-way set-associative?

In a 2-way set associative cache, 0x00001010 maps to a unique cacheline, but 0x00001000, 0x00002000, 0x00003000 map to the same cacheline. Since the cache is 2-way set associative, we can only store 2 of them. Following the LRU algorithm on the cacheline 00, we will miss 0x00001000 and cache it, then miss 0x00002000 and cache it, then miss 0x00003000 so evict 0x00001000 to cache 0x00003000. We will then hit 0x00002000, evict 0x00003000 to cache 0x00001000 and then hit 0x00002000.

So we get 3 cache hits on 0x00001010 and hit 0x00002000 twice. Therefore we will have a cache hit rate of $\frac{5}{10} = 0.5$.

(d) Assume that this core and cache are part of a chip-multiprocessor, with the cache connected to a shared L2 via a bus that maintains coherence through a snooping MESI protocol. What sequence of steps would be taken if another core wanted to load from 0x00001010 after the given sequence had finished?

After the given sequence had finished, 0x00001010 is held in cache and has been written to and is therefore in the *M* state. This means the core which performed the instruction sequence has the only copy of 0x00001010. So in order to get it, the new core will request it on the data bus. The original core will be snooping on the data bus and look up 0x00001010 in its own cache. This will return a result. Therefore the new core will push it out to the shared L2 cache and mark the entry as shared in both own cache and mark the entry in the L2 cache as being shared (assuming the other core issued a read request – if the request was a write then it would write is as modified in the L2 cache and invalid in it's own cache). The new core can then read the value from L2 cache.

This ensures that no two cores can have the cacheline in the modified state at once while enabling both to share it (have it in read-only).

3 2010 Paper 7 Question 7

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2010p7q7.pdf

(b) In what situation might a shared second-level cache offer a performance advantage over a memory hierarchy for a chip multiprocessor with private L2 caches?

If the cores were sharing many variables between them then having a shared L2 cache would mean sharing data would only require a L2 write rather than a write to main memory – this would be significantly faster.

Additionally, if the cores were frequently performing the same tasks then the second cache could still have good spatial and temporal locality for both cores.

However, if the cores were performing different tasks then each core would be trashing the other cores L2 cache.

Building on this, if the cores frequently had poor spatial and temporal locality then having a shared L2 cache may provide performance benefits – if they read from RAM then they have to check that the other core doesn't have the value they are looking for which requires looking in the other cores L2 cache – this is awkward to implement. With a shared L2 cache we don't have to do that – under an inclusive policy we can only check our own L2 and under an exclusive or non-inclusive-non-exclusive policy we check the other cores L1 cache – which is faster than checking their L2.

Having a shared L2 cache means that L2 access times are higher as the L2 cache has to be physically further from each core. This would decrease performance.

(d) How does adopting an inclusion policy simplify the implementatino of a cache coherence mechanism in a chip multiprocessor wiht private L1 and L2 caches.

Consider using the MSI protocol.

On a system with an inclusive cache policy, we don't have to check the other cores cache to ensure we have the only copy. We can check the copy in memory – if it is in the *M* state then we know that the other core has modified it and can request it from the other core. If it is in the *S* state then the requesting core knows it can load it into cache without conflicting with the other core (and potentially move the cacheline into *M*).

Having an inclusive cache policy removes the need for snoopy buses which greatly simplifies the system and scales better for larger numbers of cores.

Although, caches still need a way of signalling to other cores that they wish to access a cacheline that is in the modified state; but the other core will only have to address this if they have the cacheline – they won't have to look up their L2 cache every time the other core attempts to read from memory.