1. A shared-bus SMP 4-processor system uses the write-through invalidate protocol for cache coherency. If each processor runs at 3.2 GHz with 2.2 CPI and has cache blocks of 16 bytes, find the percentage of instructions that must be stores to saturate a 2 GB/s bus.

Solution:

We know that our processor runs at 3.2 GHz, and that the CPI is 2.2. Thus, we can find the number of instructions per second as follows:

Processor Speed: 3.2* 10^9 Clocks/second, CPI = 2.2 Clock/Instruction

#instruction/Second = $\frac{\mathrm{Pr}\,ocessorSpeed}{CPI} = \frac{3.2*10^9}{2.2} = 1.455*10^9\ \frac{Instruction}{Second}\ IPS$

We now have 1.455 * 10^9 instructions/s per processor. Since we have a write-though invalidate protocol, each processor write will produce a bus-write. The cache block size is 16 bytes, so each write will use up 16 bytes of the bus bandwidth. Assuming each processor gets a fair share of the bus, we can deduce:

Bandwidth per processor = $\frac{BusBandwidth}{\#\,ofprocessor} = \frac{2GB/s}{4} = 0.5GB/s$

Since we have 0.5 GB/s of bandwidth per processor, we can find the percentage of instruction that must be stored as follow

$\frac{instruction}{s}$ * % of stores * bytes per store = Bandwidth per processor

% of stores = $\frac{0.5GB/s}{1.455*10^9\,inst/s*16bytes}*100 = 2.15\%$

Thus, at most 2.15% of all instructions in a processor can be store to get the bus saturated.

2. For the following systems, state whether or not the caches provide inclusion naturally. If not, state the problem or give an example that violates inclusion:

a. L1: 8-KB direct-mapped primary instruction cache, 32-byte line size 8-KB direct-mapped primary data cache, write through, 32-byte line size
L2: 4-MB four-way set-associative unified secondary cache, 32-byte line size

b. L1: 16KB direct-mapped unified primary cache, write-through, 32-byte line size
L2: 4-MB four-way set-associative unified secondary cache 64-byte line size

a. We must first determine the # of sets in each of our caches as follows:

Capacity = associatively * line size * #sets

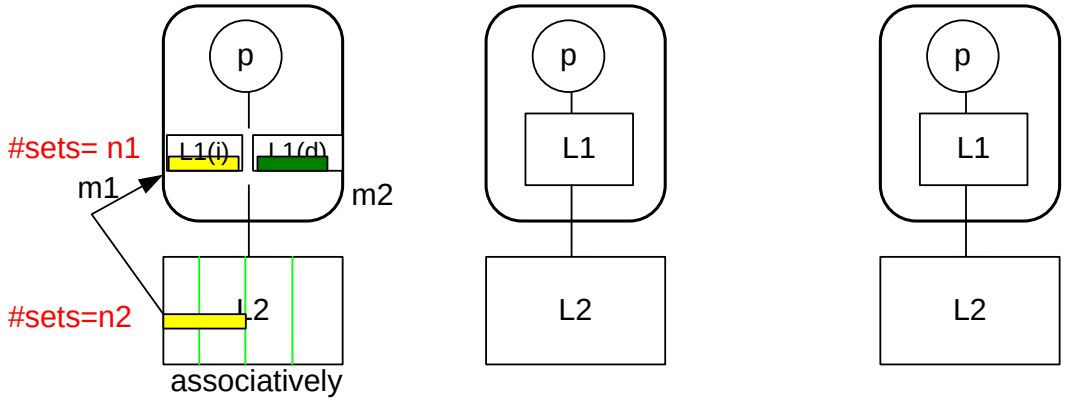
- L1 data and instruction cache

8 KB = 1 * 32 bytes/set * n
n1= 8 * 1024/32 = 256 sets

- L2 unified cache

4 MB = 4 lines/set * 32 bytes / line * n
n2 = 4 * (1024 * 1024) /32 = 131072 sets

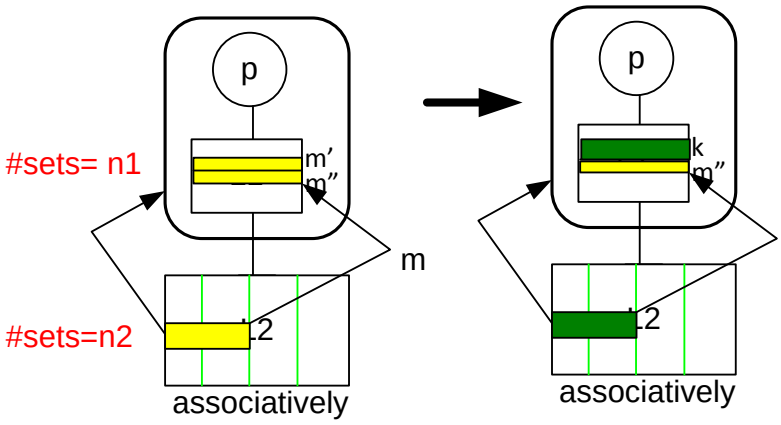
So we can see that n1 < n2 (sets in L1 and L2 respectively). The block sizes are equal, and L1 is direct mapped. So it appears that all our conditions are satisfied for natural inclusion, but we must not ignore the fact that our L1 cache is split into a data and instruction cache. As such, this factor will negate our inclusion. So inclusion is not naturally held.



Capacity = associatively * line size * #sets
N2>n1?
Block Size ?
L1 is direct map?

Take for example, the case when one instruction block m1 and one data block m2 are in conflict. Assume m1 is in memory. If a request for m2 occurs and it falls in the same set as m1 in the L2 cache, it is possible that m1 might be evicted to be replaced by m2. In the L1 cache on the other hand, this will not be an issue since the instruction and data caches are separate and will never be in conflict.

b. In this case, we simply take note of the fact that our block sizes are not equal in the L1 and L2 caches. So our rules for natural inclusion do not hold. The inclusion is not automatic.



Take for example a block of data m in L2. This block will map to m' and m'' in the L1 cache (since the block size is 32 bytes which is half of 64 bytes). Now, a cache miss may cause block m in L2 to be replaced, however only m' or m'' in L1 might need to be replaced. As such, either m' or m'' can remain in L1 even though it is not present in L2. More concretely, address 0 and 1 will map

to two sets in L1 and one set in L2. So it is possible for either 0 or 1 to be present in L1 although this set was replaced in L2.

3-Compared to a shared first-level cache, what are the advantages and disadvantages of having private first-level cache but a shared second-level cache?

Advantages of private L1 cache:

- information that is more relevant can be stored in the private first level cache
- The hit latency decreases as there is no interconnect between each processor and the
- L1 cache. The clock cycle can be increased.
- L1 cache can be faster (clock cycle increased) as they are smaller than a unified level.

A. we have to make sure that we do not violate the inclusion principle
B. additional circuitry
C. Larger latency for inter-processor communications (it correlates to the level of memory at which they meet, here L2 instead of L1).
D. we will have to deal with several misses in the first level cache before reaching the second level cache and this might slow down the CPU.

4- Assume that each processor has separate instruction and data caches and that there are no instruction misses. Further assume that, when active, the processor issues a data cache request every 3 clock cycles, the miss rate is 1%, and miss latency is 30 cycles. Assume that tag reads take 1 clock cycle but modifications to the tag take 2 clock cycles.

A. Quantify the performance lost to cache tag contention if a single-level data cache with only one set of cache tags is used. Assume that the bus transactions requiring snoop occur every 5 clock cycles and that 10% of these invalidate a block in the cache. Further assume that snoops are given preference over processor accesses to tags.
B. What is the performance lost to tag contention if separate sets of tags for processor and snooping are used?
C. In general, would you give priority in accessing tags to processor references or bus snoops?

a. In our single-level data cache, we know that a bus snoop occurs 1 out of every 5 clock cycles and 10% of these are invalidations. We can thus quantify the number of invalid bus snoops per clock cycle as follows:

$$Invalidations\ per\ cc = \frac{1\ snoop}{5\ cc} * 10\% \frac{invalidations}{snoop} = \mathbf{0.02}$$

Since we know that each tag read takes 1 clock cycle and each modification takes 2 clock cycles, we can determine the fraction of time the bus snoop takes in the cache .

$$\%\ of\ snoop\ time = \frac{1}{5} tag\ reads/cc * 1\ cc/tag\ read + \\ + 0.02\ invalidates/cc * 2\ cc/invalidate$$

$$\%\ of\ snoop\ time = \frac{1}{5} * 1 + 0.02 * 2 = 0.24 = \mathbf{24}\%$$

So the bus snoop must access the cache 24% of the time. We must now look at the data cache accesses. We are given the exact rate of cache accesses. Assuming the latency on cache hits is 1 clock cycle, we can determine the time spent by the processor in the data cache:

$$\%\ of\ processor\ time = \frac{1}{3}\ cache\ accesses/cc * \left[99\%\ hit\ rate * \frac{cc}{hit} + 1\%\ miss\ rate * \frac{cc}{miss}\right]$$

$$\%\ of\ processor\ time = \frac{1}{3} * [0.99 * 1 + 0.01 * 30] = 0.43 = \mathbf{43}\%$$

Assuming that our bus and processor cache access rates are independent, we can quantify the contention as the product of these two percentages:

$$\%\ contention\ time = snoop\ access\ time * \ processor\ access\ time$$

$$\%\ contention\ time = 24\% * \ 43\% = \mathbf{10.32}\%$$

So, we can conclude that 10.32% of our performance (or cache access time) is lost to contention.

Alternate solution

If we assume that each instruction in our processor takes 1 clock cycle, it will imply that each cache access will occur 1 in every 3 instructions. We can now calculate the CPI and use it to find the % of processor time as follows:

$$CPI = \frac{2}{3} ALUops * 1cc + \frac{1}{3} mem\ access * [0.99 * 1cc + 0.01 * 30cc]$$
$$CPI = 1.1\ cc/instruction$$

$$\#\ instructions\ per\ cc = \frac{1}{CPI}$$

$$\%\ processor\ time = \#\ instr\ per\ cc * \frac{1}{3} mem \frac{access}{cc}$$

$$\%\ processor\ time = \frac{1}{1.1} * \frac{1}{3} = \mathbf{30.3}\%$$

Now, we can calculate the contention time as:

$$\%\ contention\ time = snoop\ access\ time *\ processor\ access\ time$$
$$\%\ contention\ time = 24\% *\ 30.3\% = \mathbf{7.28}\%$$

b. If separate tags are used for the processor and the bus, then there will be no contention during tag reads. However, we will still have problems when invalidations (and so modifications) occur in the cache. On the bus, this can be quantified as follows:

$$\%\ of\ snoop\ time = \frac{1\ snoop}{5\ cc} * 10\% \frac{invalidations}{snoop} * 2 \frac{cc}{invalidation}$$
$$\%\ of\ snoop\ time = \frac{1}{5} * 0.1 * 2 = 0.04 = \mathbf{4}\%$$

The time spent accessing the processor cache tags is still the same as before: 43%
We can again quantify the contention as:

$$\%\ contention\ time = snoop\ access\ time *\ processor\ access\ time$$
$$\%\ contention\ time = 4\% *\ 43\% = \mathbf{1.72}\%$$

So, 1.72% of our performance is lost to contention. We can see that it is considerably reduced.
Alternate Solution

$$\%\ contention\ time = 4\% *\ 30.3\% = \mathbf{1.21}\%$$

c. The issue of preference is a rather tricky one. Although bus snoops are essential in ensuring correctness and ensuring cache coherence, it might be best to give preference to the processor.
If we give preference to the bus snoops, we will inevitably hurt the processor performance. At the same time, bus snoops will be delayed as a result of being second to processors. If cache coherence and fast snooping is more essential than processor performance, then we might want to consider giving preference to bus snoops.

5- Although the challenge supports the MESI protocol states, it does not support the cache-to-cache transfer feature of the original Illinois MESI protocol.

(a) Discuss the possible reasons for this choice.
(b) Extend the challenge implementation to support cache-to-cache transfer. Describe the extra signal needed on the bus, if any and keep in mind the issue of fairness.
(c) Although the challenge MESI protocol has four states the tags stored with the cache controller chip keep track of only three states (I, S, and E+M) Explain why this is still works correctly. Why they made this optimization?

Solution:

When the protocol is not in an M state and a miss happens, in the Illinois MESI protocol it was assumed that getting data from another cache was faster than from memory. If this revealed to be false with time, then the Challenge MESI protocol may have decided to not support cache-to- cache transfers so as not to intervene in another cache as that would be slower (it would cause more congestion) than going to main memory to get it. It could also be because supporting cache-to-cache transfers adds complexity (ex.: who supplies data when multiple caches have the data being requested by a controller) and a selection mechanism must be put in place.

b) To support cache-to-cache transfers, we need the ability for any cache controller to respond to a BusRd, instead of only the main memory.
The principle is then for the cache controllers to snoop on the bus for BusRd requests, and return data on the bus if the requested address is contained in their associated cache. Main memory will also fetch the data and respond, while this may take longer. In case a cache

controller responds before main memory does, main memory cancels its fetching operation. No extra wires on the bus are necessary, as there's already a snooping mechanism in place. The logic of the snoopers has to be changed so as to place data on the bus after a BusRd, instead of just asserting the shared line.