

Homework 4: CS 211 Fall 2008 Solutions

Ques. 1:

- (a) A 64KB, direct mapped cache has 16 byte blocks. If addresses are 32 bits, how many bits are used the tag, index, and offset in this cache?
 - Tag: 16 bits. Index: 12 bits. Offset: 4 bits.
 - In a 64KB direct mapped cache with 16 byte blocks there are 4096 cache lines (blocks), therefore requiring 12 bits for the index (i.e., to select the block). Since the blocks are 16 bytes each, 4 bits are required for the offset. The remaining 16 bits are used for the tag. Note that the tag bits (for direct mapped) can also be derived as total address bits (32) minus total bits to address the cache size (i.e., to address 64KB we need 16 bits).
- (b) How would the address be divided if the cache were 4-way set associative instead?
 - Tag: 18 bits. Index: 10 bits. Offset: 4 bits. If the cache were 4-way associative there would be 4 blocks in each set and 1024 sets or lines. Therefore we require 10 bits for the index (to specify the set to where the block is mapped). The offset would remain at 4 bits, and the tag would now use 18 bits.
- (c) How many bits is the index for a fully associative cache. Explain your answer.
 - A fully associative cache has no bits in the index, since any address can be stored anywhere in the cache.

Ques.2: An 8 byte, 2-way set associative (using LRU replacement) with 2 byte blocks receives requests for the following addresses (represented in binary):

0110, 0000, 0010, 0001, 0011, 0100, 1001, 0000, 1010, 1111, 0111

For each access, determine the address in the cache (after the access), whether each access hits or misses, and the categorization of each miss under the “3 C” model. Fill in the worksheet in the format shown below with your answer to this question (Note that the first access is done for you). You should fill in the cache lines with the tags that reside there, with the most recently used tag first.

Note: In some cases there may be multiple types of miss; I indicated the primary cause of the miss.

Address	Line 0	Line 1	Hit or Miss Type
0110	(empty)	01/(empty)	Compulsory Miss
0000	00/(empty)	01/(empty)	Compulsory miss
0010	00/(empty)	00/01	Compulsory miss
0001	00/(empty)	00/01	Hit
0011	00/(empty)	00/01	Hit
0100	01/00	00/01	Compulsory miss
1001	10/01	00/01	Compulsory miss
0000	00/10	00/01	Conflict miss
1010	00/10	10/00	Compulsory miss
1111	00/10	11/10	Compulsory miss
0111	00/10	01/11	Capacity miss

Ques.3: Looking at the surface of the three C’s cache miss model, a fully associative cache *should* have fewer non-compulsory misses (capacity plus conflict) than an equal size direct

mapped cache because conflict misses are in addition to capacity misses and occur only in set associative or direct mapped caches.

Capacity misses are defined as those misses in a fully associative cache that occur when a block is retrieved any time(s) after its initial compulsory miss. The genesis of the opportunity for a small direct mapped cache to outperform an equally sized fully associative cache can be found in the question hiding within this definition of capacity misses and left begging for an answer. If fully associative cache capacity misses are caused by blocks being discarded before their final use, why are these blocks discarded and must otherwise equivalent set associative or direct mapped caches discard the same blocks at the same times during program execution?

Blocks are discarded, or replaced, based on the decision of the replacement policy. This replacement decision is very important to cache performance. If the block chosen for replacement is not referenced in the future by the program, then no capacity miss or conflict miss can occur in the future. This is the ideal case. If the block chosen for replacement will be used again in the very near future or is used frequently, as compared to other candidate blocks for replacement, then the replacement choice is a poor one and cache performance will be generally worse than ideal. Because fully associative, set associative, and direct mapped caches have different block *placement* constraints, the block *re-placement* policy for one cache type cannot consider the same blocks for replacement as are considered by the same policy on another organizational type. To see this more clearly, consider an example.

Let a program loop access three distinct addresses, A, B, and C, and then repeat the sequence from A. The reference stream for this program at this point would look like this: ABCABCABCA... . To simplify the discussion we assume that the direct mapped and fully associative caches each can hold two blocks and that addresses A and C are from different cache block frames in memory but map to the same location in the direct mapped cache, while address B maps to the other location in the cache. If the replacement policy for the fully associative cache is LRU, then every reference generated by the loop is a miss. If the replacement policy for the direct mapped cache is LRU (a degenerate form to be sure, because with only one block in each set whatever blocks are in a direct mapped cache are all always “least recently used”), accesses to A or C will always miss, but we will always hit on B (ignoring its compulsory miss).

The replacement policy of a fully associative cache can cast its eye on all the blocks in the cache, and in our example, makes the worst possible choice for replacement from all the blocks every time. For the direct mapped cache this choice is also always the worst possible, but is limited to two of the three blocks by cache structure. The result is that the direct mapped cache performs better.

Ques.4: Appendix C discussed a number of cache performance equations, and you will find that there a number of ways to derive cache performance metrics. Assume you have a processor with an ideal CPI without memory stalls for each instruction type as follows: ALU=1, Load/Store=1.5, Branch=1.5, Jump=1. Consider an application which has an instruction mix of 40% ALU and logical operations, 30% load and store, 20% branch and 10% jump instructions.

(a) Assume a 4-way set associative 1-level separate data and instruction cache with a miss rate of 20% (0.20) for data accesses and miss rate of 10% (0.10) for instructions, and a miss penalty of 50 cycles for both instruction and data caches (and assume a cache hit takes 1 cycle). What is the effective CPU time (or effective CPI with memory stalls) and the average memory access time for this application with this Cache organization?

First compute ideal CPI without memory stalls:
 $CPI_{ideal} = (0.4*1) + (0.3*1.5) + (0.2*1.5) + (0.1*1) = 1.25.$

The effective CPI, and therefore execution time, includes the memory stalls. This is given by $CPI_{actual} = CPI_{ideal} + \text{Memory Stalls per instruction}.$

The Memory stalls per instruction, Stalls/Inst is given by

$$\text{Stalls/Inst} = (\text{stalls-data}/\text{data}) + (\text{stalls-inst}/\text{inst})$$

$$\text{Stalls-data (stalls due to data)} = (\text{data miss rate} * \text{data miss-penalty}) * \text{data accesses}/\text{inst}$$

$$\text{Stalls-inst (stalls due to inst)} = (\text{inst. Miss rate} * \text{miss penalty}) * \text{inst.accesses}/\text{inst}$$

The data accesses take place during the 30% Load/Store operations. The instruction accesses take place once for each instruction – i.e., 100% of the program.

$$\text{Therefore, stalls/inst} = ((0.2*50)*0.3) + (0.1*50)*1 = 8$$

Now compute the Average Memory Access time (AMAT). This is given by:

$$\text{AMAT} = \text{percentage of data accesses (hit time + data miss rate* miss penalty)} + \text{percentage of inst. Accesses (hit time + inst.miss rate*miss penalty)}.$$

The percentage data accesses is 0.3/1.3 (i.e., total of 1.3 accesses to memory during entire program execution, of which 0.3 are for data), and for inst it is 1/1.3

$$\text{Therefore AMAT} = (0.3/1.3)(1 + 0.2*50) + (1/1.3)(1 + 0.1*50).$$

Note that another way to derive stalls per instruction is to multiply the average number of memory accesses per instruction by the AMAT minus hit time. In this example, the average number of memory accesses per instruction is 1.3 (i.e., 1 for instruction and 0.3 for data due to Load/Store). Therefore the average stall cycles per instruction can be derived as:

$$1.3*(\text{AMAT}-1) = 1.3*[(0.3/1.3)(1 + 0.2*50) + (1/1.3)(1 + 0.1*50) - 1] = 8 \text{ which is identical to what we got earlier.}$$

(b) Now consider a 2 level 4-way unified cache with a level 1 (L1) miss rate of 20% (0.20) and a level 2 (L2) local miss rate of 30% (0.30). Assume hit time in L1 is 1 cycle, assume miss penalty is 10 cycles if you miss in L1 and hit in L2 (i.e., hit time in L2 is 10 cycles), and assume miss penalty is 50 cycles if you miss in L2 (i.e., miss penalty in L2 is 50 cycles). Derive the equation for the effective CPU time (or effective CPI) and the average memory access time for the same instruction mix as part (a) for this cache organization.

First note that the miss rate for L2 is given as the local miss rate. Average memory accesses per instruction = 1.3 as noted earlier (0.3 for data and 1 for inst).

$$AMAT = (\text{hit time } L1) + (\text{miss rate } L1) * (\text{Hit time } L2 + (\text{Local miss rate } L2 * \text{Miss Penalty}))$$

The global miss rate for L2 is not the same as L1 – but you can derive the global miss rate from the local miss rate of L1 and L2. Note that the local and global miss rate of L1 are the same.

$$AMAT = (1 + (0.2) * (10 + (0.3 * 50))) = 6.$$

Effective CPI = Ideal CPI + average memory stalls per instruction.

Average memory stalls per instruction = misses per instruction-L1 * Hit-time L2 + Misses-per-instruction L2*miss-penalty

This is equivalent to: Average mem.stalls per instruction = (mem.accesses/inst)*(miss rate L1 * Hit time L2 + Miss rate L2 * miss penalty).

However, note that the Miss rate L2 in the above equation refers to the global miss rate of L2.

Alternately, we can use our earlier observation that average mem.stalls per instruction can be derived as Mem.accesses per instruction * (AMAT -1). This gives us

$$\text{Avg.Mem.stalls/inst} = 1.3 * (\text{AMAT} - 1) = 1.3 * (6 - 1) = 6.5 \text{ cycles.}$$

(c) Which of the two designs (between part a and part b) gives a better performance ? Explain your answer.

The second design, the multi-level cache, gave us lower AMAT and avg. mem. Stalls/inst and therefore it is a better design.