CSci 2021, Fall 2018 Written Exercise Set 4 Solutions

Problem 1: (This problem is closely related to O'Hallaron 6.26)

The bits must add up to $m = t + s + b$, and the cache size must be $S = C * B * E$. Cache 1 is fully associative, hence $S = 1$ and $s = 0$.

Cache	*m*	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1.	32	1024	16	1	64	22	6	4
2.	32	4096	512	8	1	23	0	9
3.	32	2048	8	2	128	22	7	3
4.	64	8192	256	1	32	51	5	8

Problem 2: (This problem is closely related to O'Hallaron 6.29)

There are 2 set index and 2 byte offset bits each. The remaining bits are for the tag.

T	T	T	T	T	T	T	T	I	I	O	O
11	10	9	8	7	6	5	4	3	2	1	0

Using the above format we can extract the bits identifying the tag, set index, and line offset. With these we can identify the cache line and inspec the tags and valid bits to determine if there was a hit.

ADDR	T	I	O	Effect
0xFD2	FD	0	2	Hit, read byte 2 0x13
0x3F7	3F	1	3	Miss, tag found but invalid
0x2FC	2F	3	0	Hit, read byte 0 0x5C
0x63A	63	2	2	Miss, tag found but invalid

Problem 3: (This problem is closely related to O'Hallaron 6.37)

First see that $C = 16KB = 16 * 1024 = 2^{14}$ and $S = C/(E * B) = 2^{14}/(2 * 2^5) = 2^8$. Thus the 2-way cache has 8 s bits and 5 b bits.

`sumA` is straightforward, every cache fetch will load $B/$`sizeof(double)` $= 32/8 = 4$ columns into the cache block. We read each address in row-major order and don't re-read any addresses. Then for every read that misses, the next three reads will hit. **miss rate = 1/4**.

`sumB` is complicated by a stride-2 and column-major order access pattern. For an iteration of j, four columns will be cached but we only read two of them ((i, j) and $(i, j + 1)$). The critical question is: will the other two columns ($(j + 2)$ and $(j + 3)$) be read before that cache line gets evicted? The following table shows the memory access pattern. The address is computed in the usual way: $ADDR = (128 * 8) * row + 8 * col =$ `0x400` $* row +$ `0x8` $* col$.

row	col	ADDR	t	s	b	m	
0	0	0x8000000	0x4000	0x00	0x0	*	(Set 0 Line 1 Filled)
0	1	0x8000008	0x4000	0x00	0x8		
1	0	0x8000400	0x4000	0x20	0x0	*	
1	1	0x8000408	0x4000	0x20	0x8		
2	0	0x8000800	0x4000	0x40	0x0	*	
2	1	0x8000808	0x4000	0x40	0x0		
⋮			⋮				
8	0	0x8002000	0x4001	0x00	0x0	*	(Set 0 Line 2 Filled)
8	1	0x8002008	0x4001	0x00	0x0		
⋮			⋮				
16	0	0x8004008	0x4002	0x00	0x0	*	(Set 0 Line 1 evicted)
16	1	0x8004008	0x4002	0x00	0x0		
⋮			⋮				

We see that when we increment the row, the element address maps to a set `0x20` past the last row. But, we only have $2^8 - 1 =$ `0xFF` sets. When our next element address increments the set number past `0xFF`, the tag will increment and the set will reset to 0. You can see that this will happen every 8 rows. At this point the tag will be different than the current resident of the cache set and the second line will be filled. However, when this wrapping happens a second time, the ways for the set will be full, and a block must be evicted. Assuming an LRU strategy, the first line will be evicted. Thus at iteration $j+2$, the $(i, j+2)$ read will be a cache miss. After the fetch the $(i, j+3)$ read will hit. So far we've reasoned for an (i, j) pair, half of the reads will be misses. **miss rate = 1/2**

`sumC` also uses a stride-2 pattern and reads in row-major order. Unlike `sumB`, the line fetched for an (i, j) pair will still be intact by the time the loop gets around to reading the last two columns. This is evident from the memory access pattern table:

row	col	ADDR	t	s	b	m	
0	0	0x8000000	0x4000	0x00	0x0	*	(Set0 Line 1 Filled)
1	0	0x8000400	0x4000	0x20	0x0	*	(Set1 Line 1 Filled)
0	1	0x8000008	0x4000	0x00	0x8		
1	1	0x8000408	0x4000	0x20	0x8		
0	2	0x8000010	0x4000	0x00	0x8		
1	2	0x8000410	0x4000	0x20	0x8		
0	3	0x8000018	0x4000	0x00	0x8		
1	3	0x8000418	0x4000	0x20	0x8		
0	4	0x8000020	0x4000	0x01	0x8	*	(Set0 Line 2 Filled)
1	4	0x8000420	0x4000	0x21	0x8	*	(Set1 Line 2 Filled)
⋮			⋮				

Thus we reason again that for every missed read we have three 3 hit reads. **miss rate = 1/4**

Problem 4:(This problem is closely related to O'Hallaron 6.38)

The direct-map cache has 4 s bits and 3 b bits. The 4-way cache has 3 s bits and 2 b bits. The reasoning is simplified by the fact that if the first write to a pixel is a cache miss, then the subsequent writes to the same pixel will be cache hits. This gives a worst case $1/4$ write miss rate.

A. Every time we miss on a pixel write we fetch both the pixel and its successor because the block size 8 bytes holds two pixels. Since we write the pixels in the order of allocation, we miss one write every other pixel. Since there are four writes per pixel, we have one cache miss per eight writes. **miss rate = 1/8**

B. Since each 4-way cacheline only holds one pixel, and we never write the same pixel twice, every new pixel write must be a miss. Since there are four writes per pixel, we get one miss per four writes. **miss rate = 1/4**

C. This code demonstrates the effect of thrashing. Every other pixel address we visit maps to the same cache line but has a different tag. Thus we get a miss on every pixel. **miss rate = 1/4**

D. For any i, the pixels loaded by $j = \{0, 1, 2, 3\}$ will each result in a miss. But, since we revisit these same memory addresses when $j = \{4, ..., 15\}$ we get cache hits for all writes. Thus for each row i there will be $16 * 4 = 64$ writes and only 4 of them will be misses. **miss rate = 1/16** The table illustrates the memory access pattern.

			Direct-Map				4-Way			
$j\%4$	i	ADDR	t	s	b	m	t	s	b	m
0	0	0x00	0	0	0	*	0	0	0	*
1	0	0x40	0	4	0	*	0	2	0	*
2	0	0x80	1	0	0	*	0	4	0	*
3	0	0xc0	1	4	0	*	0	8	0	*
0	0	0x00	0	0	0	*	0	0	0	
1	0	0x40	0	4	0	*	0	2	0	
2	0	0x80	1	0	0	*	0	4	0	
3	0	0xc0	1	4	0	*	0	8	0	
⋮			⋮				⋮			