CSE 351 Section 8 – More Caches, Processes & Concurrency

Hi there! Welcome back to section, we're happy that you're here ☺

Practice Cache Exam Problem (11 pts)

We have a 64 KiB address space and two different caches. Both are 1 KiB, direct-mapped caches with random replacement and write-back policies. **Cache X** uses 64 B blocks and **Cache Y** uses 256 B blocks.

a) Calculate the TIO address breakdown for **Cache X**:

$2^{16} = 64$ KiB, so we have 16 bit addresses.

Tag	Index	Offset
16 − 4 − 6 = **6**	$\frac{2^{10}}{2^6} = 2^4$ → **4**	$2^6 = 64$ → **6**

b) During some part of a running program, **Cache Y**'s management bits are as shown below. Four options for the next two memory accesses are given (R = read, W = write). Circle the option that results in data from the cache being *written to memory*.

Line	Valid	Dirty	Tag
00	0	0	1000 01
01	1	1	0101 01
10	1	0	1110 00
11	0	0	0000 11

Cache Y:

- Tag: 6 bits
- Index: $\frac{2^{10}}{2^8} = 2^2$ → 2 bits
- Offset: $256 = 2^8$ → 8 bits

Note that, since the last 8 bits form the offset, we can ignore the last two hex digits for this problem.

(1) `R 0x4C00, W 0x5C00`

R 0b0100 1100… , W 0b0101 1100…

The read evicts line 0, but the dirty bit was not set so nothing is written (also, line 0 was initially invalid). The write overwrites line 0 again but since the cache is write-back nothing is written to memory.

(2) `W 0x5500, W 0x7A00`

W 0b01010101… , W 0b0111 1010…

The first write doesn't evict anything because the tags match. The second write evicts the old data but the dirty bit was not set so the old data doesn't need to be written back to memory.

(3) `W 0x2300, R 0x0F00`

W 0b0010 0011… , R 0000 1111…

The write evicts line 3 which was invalid and also not dirty, so nothing is written. The read, however, also maps to line 3 so it must write the *value changed in the write* back to memory before it can update the cache.

(4) `R 0x3000, R 0x3000`

R 0b0011 0000… , R 0011 0000…

Line 3 is initially not dirty (and invalid) so nothing is written back to memory from either of these reads (which both read from the same line).

c) The code snippet below loops through a character array. Give the value of LEAP that results in a Hit Rate of 15/16 for **Cache Y**.

```
#define ARRAY_SIZE 8192
char string[ARRAY_SIZE];                // &string = 0x8000
for(i = 0; i < ARRAY_SIZE; i += LEAP) {
    string[i] |= 0x20;          // to lower
}
```

Note that |= is a read *and* a write (i.e., two accesses). To obtain a 15/16 hit rate, we want to perform $\frac{256}{16} = 16$ accesses per block (the first access will be a miss, subsequent accesses will be hits). However, since each loop iteration performs two accesses, we want to loop 8 times per block. Therefore $LEAP = \frac{256}{8} = 32$.

32

d) For the loop shown in part (c), let LEAP = 64. Circle ONE of the following changes that increases the hit rate of **Cache X**:

[✓ Increase Block Size] Increase Cache Size Add a L2$ Increase LEAP

- Larger block size mean that we can fit more bytes in a block, so more information will be pulled in on each miss. Therefore, hit rate will increase.
- Increasing cache size will not change hit rate since we are accessing data contiguously.
- Adding a L2 cache will not change the hit rate (it will just decrease the miss penalty).
- Increasing LEAP will *increase* the miss rate since data accessed will be further apart in memory.

e) For the following cache access parameters, calculate the AMAT. Please simplify and include units.

L1$ Hit Time	L1$ **Miss** Rate	MEM Hit Time
2 ns	40%	400 ns

AMAT = (hit time) + (miss rate)(miss time)
You *always* pay for hit time. You *also* pay for miss time during a cache miss.

162 ns

$2 + (0.4)(400) = 162$ ns

Benedict Cumbercache:

Given the following sequence of access results (addresses are given in decimal) on a cold/empty cache of size 16 bytes, what can we *deduce* about its properties? Assume an LRU replacement policy.

```
      (1)        (2)        (3)        (4)         (5)
    (0, Miss),(8, Miss),(0, Hit),(16, Miss),(8, Miss)
```

1) What can we say about the block size?

The block size must be ≤ 8 because access (2) to address 8 is a miss after access (1) to address 0 is a hit.

2) What is this cache's associativity?

Associativity $E \leq 2$ because access (4) caused address 8 from access (2) to be evicted. If it were > 2, then we would be able to fit addresses 0, 8, and 16 all at once even if they mapped to the same set.

Now, consider the case where $E = 1$. Then there are four possibilities for block size $K$and number of sets S:

Block Size (K)	Offset Bits (k)	Number of Sets (S)	Set Bits (s)	$s + k$
1	0	16	4	4
2	1	8	3	4
4	2	4	2	4
8	3	2	1	4

Note that in each of these cases, $s + k = 4$, so the 4th bit from the right (i.e. the 4th least significant bit) *will always be a set bit*. The address 8 (0b01000) and 16 (0b10000) differ on the 4th bit from the right, so in any direct-mapped cache, they cannot map to the same set! However, access (4) to address 16 evicted access (2) to address 8 in our example (since access (3) was a hit, it didn't evict anything). This means that it is impossible for the cache to be direct mapped, so it must be 2-way set associative.

3) How many sets could this cache have?

To find set size, we need to know block size. We know that the cache is 2-way set associative and has a total size $C = 16\text{B}$. Since $16 = 2 * S * K$, and we know from question 1 that $K \in \{1, 2, 4, 8\}$, we know that $S \in \{1, 2, 4, 8\}$. So we might have:

Block Size (K)	Offset Bits (k)	Number of Sets (S)	Set Bits (s)	$s + k$
1	0	8	3	3
2	1	4	2	3
4	2	2	1	3
8	3	1	0	3

Note that, in any case, $s + k = 3$, i.e., we have 3 bits for the set and offset. The last three bits of every address in our access pattern are the same, so they will all map to the same set regardless of which configuration we have. This means that any one of these configurations is equally possible, so all we can say is that $S \in \{1B, 2B, 4B, 8B\}$.

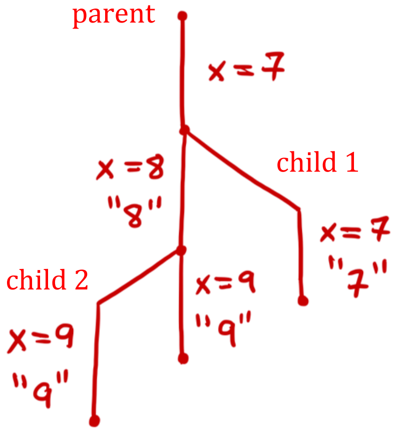
4) How many bits will the tag use given an n-bit address?

We determined in the previous problem that the set and offset bits will always take up 3 bits together, regardless of the number of sets. So, with n-bit addresses, we will have $n - 3$ bit sized tags.

Fork and Concurrency:

Consider this code using Linux's `fork`:

```
int x = 7;
if( fork() ) {
    x++;
    printf(" %d ", x);
    fork();
    x++;
    printf(" %d ", x);
} else {
    printf(" %d ", x);
}
```



What are *all* the different possible outputs (i.e. order of things printed) for this code?
(Hint: there are four of them.)

`fork()` returns 0 to the child, and the child's process ID (PID) to the parent. Notice that first call to `fork()` is the only time it is called conditionally. So the time at which child 1 prints "7" is unknown. However, the parent will print "8" before the second call to `fork()`, meaning that the "8" is printed before the "9"s, then the parent and child 2 will both print out "9". (The ordering of the 9s may change, but that doesn't matter because they are both 9).

Possible orderings:

- 7 8 9 9
- 8 7 9 9
- 8 9 7 9
- 8 9 9 7