CSE 351 Section 7 – More Caches, Processes & Concurrency

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

Code Analysis

Consider the following code that accesses a two-dimensional array (of size 64×64 `int`s).
Assume we are using a direct-mapped, 1 KiB cache with 16 B block size, and that the cache starts cold.
Also assume that the loop variables `i` and `j` are stored in registers.

```
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 64; j++)
        array[i][j] = 0;          // assume &array = 0x600000
```

a) What is the miss rate of the execution of the entire loop?
Every block can hold 4 ints (16B/4B per int), so we will need to pull a new block from memory every 4 accesses of the array. This means this miss rate is $\frac{4\ bytes\ per\ int}{16\ bytes\ per\ block} = \frac{1\ block}{4\ ints} = 0.25 = 25\%$

b) What code modifications can change the miss rate? Brainstorm before trying to analyze.
Possible answers: switch the loops (i.e. make j the outer loop and i the inner loop), switch j and i in the array access, make the array a different type (e.g. char[ ][ ], long[ ][ ], etc.), make array an array of Linked Lists or a 2-level array, etc.

(NOTE: Answer to part **(c)** on next page)

c) What cache parameter changes (size, associativity, block size) can change the miss rate?
Let's consider each of the three parameters individually.

First, let's consider modifying the size of the cache. Will it change the miss rate?
No, it doesn't matter how big the cache is in this case (if the block size doesn't change). We will still be pulling the same amount of data each miss, and we will still have to go to memory every time we exhaust that data

Next, let's consider modifying the associativity of the cache. Will it change the miss rate?
No, this is helpful if we want to reduce conflict misses, but since the data we're accessing is all in contiguous memory (thanks arrays!), booting old data to replace it with new data isn't an issue.

Finally, let's consider modifying the block size of the cache. Will it change the miss rate?
Yes, bigger blocks mean we pull bigger chunks of contiguous elements in the array every time we have a miss. Bigger chunks at a time means fewer misses down the line. Likewise, smaller blocks increase the frequency with which we need to go to memory (think back to the calculations we did in part **(a)** to see why this is the case)

So, in conclusion, changing block size can change the miss rate. Changing size or associativity will NOT change the miss rate.

NOTE: Remember that the results we got were for this specific example. There are some code examples in which changing the size or associativity of the cache will change the miss rate.

Practice Cache Exam Problem (11 pts)

We have a 64 KiB address space. The cache is a 1 KiB, direct-mapped cache using 256-byte blocks with write-back and write-allocate policies.

a) Calculate the TIO address breakdown for:

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

Tag	Index	Offset
16 − 2 − 8 = **6**	$\frac{Cache}{Block} = \frac{2^{10}}{2^8} = 2^2 \rightarrow$ **2**	$2^8 = 256 \rightarrow$ **8**

b) During some part of a running program, the cache'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

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 0 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.

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

32

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 $= \frac{256}{8} = 32$.

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

[✓ Increase Block Size]   Increase Cache Size   Add an L2 Cache   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) What are the three kinds of cache misses? When do they occur? Circle the kind of miss that happens in part (c).

[✓ Compulsory]: occurs the first time a block is accessed—no way to avoid this (a.k.a. cold miss)	Conflict: occurs when multiple blocks map to the same slot in the cache—could be avoided if the cache had a greater associativity, or perhaps if using different access pattern	Capacity: occurs when the set of active cache blocks (“working set”) evict each other because there’s not enough space in the cache—even if it were fully-associative, they wouldn’t all fit

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?

After access (1), values from address 0 to address [block size - 1] will be put in the cache. This is because caches load a full block from memory at a time and 0 will always be aligned to the beginning of a block. Thus, if access (2) to address 8 is a miss, it means that the block size must be $\leq$ 8.

2) Assuming that the block size is 8 bytes, can this cache be… (Hint: draw the cache and simulate it)

a. Direct-mapped?

Index	Address (*not* tag)
0	~~0x0~~ 0x10
1	0x8

Does this cache work for the access results?

Yes, Yes, Yes, Yes (evict 0), No (8 would still be in cache)

b. 2-way set associative?

Index	Address (*not* tag)
0	0x0
0	~~0x8~~ 0x10

Does this cache work for the access results?

Yes, Yes, Yes, Yes (evict 8 b/c it's the least recently used), Yes (8 is no longer in cache)

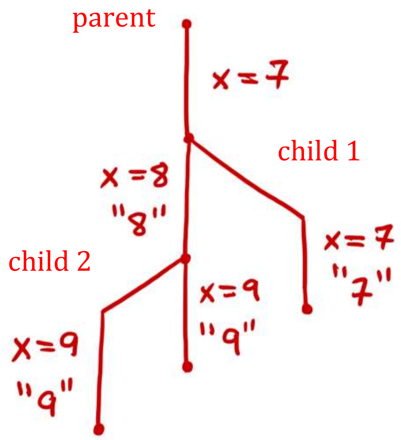
c. 4-way set associative?

No, because the block size is 8, multiplied by 4 lines per set, and that's 32B, which is already bigger than the entire cache.

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.)

Note: `fork()` returns 0 to the child, and the child's process ID (PID) to the parent.

From our first fork, we know child 1 will print "7", but since this print statement is not dependent on any other code (besides the initial `fork()`), it could be printed at any time.

We also know the parent will have to print "8" before the second call to `fork()`, meaning that the "8" is printed before the "9"s. Since the parent and child 2 both print out "9", even if the ordering of their prints changes, the output will not.

Possible orderings:

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