CS 61C Spring 2022

Caches

Discussion 9

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 For the same cache size and block size, a 4-way set associative cache will have fewer index bits than a direct-mapped cache.

True. A direct mapped cache needs to index every line of the cache, whereas a 4-way set associative cache needs to index every set of 4 lines. The 4-way set associative cache will have 2 fewer index bits than the direct-mapped cache.

1.2 Any cache miss that occurs when the cache is full is a capacity miss.

False. When the cache is full, you can still get compulsory misses (when a block of data is put in the cache for the first time) and conflict misses (if a fully associative cache with LRU replacement wouldn't have missed).

1.3 Increasing cache size by adding more blocks always improves (increases) hit rate.

False. Whether this improves the hit rate for a given program depends on the characteristics of the program. As an example, it is possible for a program that only consists of a loop that runs through an array once to have each access be separated by more than one block (say, the block size is 8B, but we have an integer array and accessing every fourth element, so our access are separated by 16B). This makes every miss a compulsory miss, and there is no way for us to reduce the number of compulsory misses just by adding more blocks to our cache.

2 Understanding T/I/O

When working with caches, we have to be able to break down the memory addresses we work with to understand where they fit into our caches. There are three fields:

Tag - Used to distinguish different blocks that use the same index. Number of bits: (# of bits in memory address) - Index Bits - Offset Bits
Index - The set that this piece of memory will be placed in. Number of bits: $\log_2$(# of indices)
Offset - The location of the byte in the block. Number of bits: $\log_2$(size of block)

Given these definitions, the following is true:

$\log_2$(memory size) = address bit-width = # tag bits + # index bits + # offset bits

Another useful equality to remember is:

$$\text{cache size} = \text{block size} * \text{num blocks}$$

2.1 Assume we have a direct-mapped byte-addressed cache with capacity 32B and block size of 8B. Of the 32 bits in each address, which bits do we use to find the index of the cache to use?

We can determine the number of index bits we need from the number of sets our cache has. Since our cache is direct-mapped, the number of sets is the same as the number of blocks, so we just need to figure out how many blocks our cache has. Using the equality from above, we see that num blocks = cache size/block size, so our cache has $32/8 = 4$ blocks. We need $\log_2(4) = 2$ bits to differentiate the 4 blocks, so we have 2 index bits.
In order to determine where exactly the index bits are, we need to calculate the number of offset bits and tag bits we have. The number of offset bits is just dependent on the block size, so since our blocks are size 8B, we need $\log_2(8) = 3$ bits to differentiate the 8 bytes in the block, so we have 3 offset bits.
our offset bits take up the least significant bits, with the index bits being the set of next most significant bits. Denoting the most significant bit (MSB, on the left) as 31 and the least significant bit (LSB, on the right) as 0, having 3 offset bits means our index bits start at bit 3, and thus we use bits 3 and 4 as the index bits.

2.2 Which bits are our tag bits? What about our offset?

The offset (in this case) is the 3 least significant bits, so reusing the convention from the previous quesiton, the offset bits are bits 0, 1, and 2. Our tag is the remaining high-order bits, so our tag bits are bits 5-31.

2.3 Classify each of the following byte memory accesses as a cache hit (H), cache miss (M), or cache miss with replacement (R). Tip: Drawing out the cache can help you see the replacements more clearly.

Address	T/I/O	Hit, Miss, Replace
0x00000004		
0x00000005		
0x00000068		
0x000000C8		
0x00000068		
0x000000DD		
0x00000045		
0x00000004		
0x000000C8		

Ignore miss types (compulsory/conflict/capacity) until Q4.

```
0x00000004     Tag 0, Index 0, Offset 4: M, Compulsory
0x00000005     Tag 0, Index 0, Offset 5: H
0x00000068     Tag 3, Index 1, Offset 0: M, Compulsory
0x000000C8     Tag 6, Index 1, Offset 0: R, Compulsory
0x00000068     Tag 3, Index 1, Offset 0: R, Conflict
0x000000DD     Tag 6, Index 3, Offset 5: M, Compulsory
0x00000045     Tag 2, Index 0, Offset 5: R, Compulsory
0x00000004     Tag 0, Index 0, Offset 4: R, Capacity
0x000000C8     Tag 6, Index 1, Offset 0: R, Capacity
```

Note that the M and R distinction here is for student understanding, and that the cache doesn't behave differently for these cases.

3 Cache Associativity

In the previous problem, we had a Direct-Mapped cache, in which blocks map to specifically one slot in our cache. This is good for quick replacement and finding out block, but not good for efficiency of space!

This is where we bring associativity into the matter. We define associativity as the number of slots a block can potentially map to in our cache. Thus, a Fully-Associative cache has the most associativity, meaning every block can go anywhere in the cache.

For an N-way associative cache, the following is true:

$$\text{N} * \#\text{ sets} = \#\text{ blocks}$$

3.1 Here's some practice involving a 2-way set associative cache. This time we have an 8-bit address space, 8 B blocks, and a cache size of 32 B. Classify each of the following accesses as a cache hit (H), cache miss (M) or cache miss with replacement (R). For any misses, list out which type of miss it is. Assume that we have an LRU replacement policy (in general, this is not the case).

Address	T/I/O	Hit, Miss, Replace
`0b0000 0100`		
`0b0000 0101`		
`0b0110 1000`		
`0b1100 1000`		
`0b0110 1000`		
`0b1101 1101`		
`0b0100 0101`		
`0b0000 0100`		
`0b1100 1000`		

Since our cache is 2-way set associative, there are 2 blocks in a set. Our cache dimensions are identical to the previous example, so we have 4 blocks. Thus, there are $4/2 = 2$ sets in our cache. We need $\log_2(2) = 1$ bit to differentiate the 2 sets, so we have 1 index bit. Our block size is identical to the previous question, so we know that we have 3 offset bits, and that the rest of our bits are our tag bits. Again, bits 0, 1, and 2 are our offset bits, but now the only index bit is bit 3, with bits 4-31 being the tag bits.

```
0b0000 0100     Tag 0000, Index 0, Offset 100 - M, Compulsory
0b0000 0101     Tag 0000, Index 0, Offset 101 - H
0b0110 1000     Tag 0110, Index 1, Offset 000 - M, Compulsory
0b1100 1000     Tag 1100, Index 1, Offset 000 - M, Compulsory
0b0110 1000     Tag 0110, Index 1, Offset 000 - H
0b1101 1101     Tag 1101, Index 1, Offset 101 - R, Compulsory
0b0100 0101     Tag 0100, Index 0, Offset 101 - M, Compulsory
0b0000 0100     Tag 0000, Index 0, Offset 100 - H
0b1100 1000     Tag 1100, Index 1, Offset 000 - R, Capacity
```

3.2 What is the hit rate of our above accesses?

$\frac{3 \text{ hits}}{9 \text{ accesses}} = \frac{1}{3}$ hit rate

4 The 3 C's of Cache Misses

4.1 Go back to questions 2 and 3 and classify each M and R as one of the 3 types of misses described below:

1. Compulsory: First time you ask the cache for a certain block. A miss that must occur when you first bring in a block. Reduce compulsory misses by having longer cache lines (bigger blocks), which bring in the surrounding addresses along with our requested data. Can also pre-fetch blocks beforehand using a hardware prefetcher (a special circuit that tries to guess the next few blocks that you will want).
2. Conflict: Occurs if, hypothetically, you went through the ENTIRE string of accesses with a fully associative cache (with an LRU replacement policy) and wouldn't have missed for that specific access. Increasing the associativity or improving the replacement policy would remove the miss.
3. Capacity: Capacity misses are independent of the associativity of your cache. If you hypothetically ran the ENTIRE string of memory accesses with a fully associative cache (with an LRU replacement policy) of the same size as your cache, and it was a miss for that specific access, then this miss is a capacity miss. The only way to remove the miss is to increase the cache capacity.

Note: The test you can use to see if a miss is a conflict miss is the same as the test you can use to see if a miss is a capacity miss.

Note: There are many different ways of fixing misses. The name of the miss doesn't necessarily tell us the best way to reduce the number of misses.

See solutions for Q2 and Q3.

5 Code Analysis

Given the follow chunk of code, analyze the hit rate given that we have a byte-addressed computer with a total memory of **1 MiB**. It also features a **16 KiB** Direct-Mapped cache with **1 KiB** blocks. Assume that your cache begins cold.

```
#define NUM_INTS 8192   // 2^13
int A[NUM_INTS];        // A lives at 0x10000
int i, total = 0;
for (i = 0; i < NUM_INTS; i += 128) {
    A[i] = i;           // Line 1
}
for (i = 0; i < NUM_INTS; i += 128) {
    total += A[i];      // Line 2
}
```

5.1 How many bits make up a memory address on this computer?

We take $\log_2(1 \text{ MiB}) = \log_2(2^{20}) = 20$.

5.2 What is the T:I:O breakdown?

Offset $= \log_2(1 \text{ KiB} = \log_2(2^{10}) = 10$
Index $= \log_2(\frac{16 \text{ KiB}}{1 \text{ KiB}}) = \log_2(16) = 4$
Tag $= 20 - 4 - 10 = 6$

5.3 Calculate the cache hit rate for the line marked Line 1:

The integer accesses are $4 * 128 = 512$ bytes apart, which means there are 2 accesses per block. The first accesses in each block is a compulsory cache miss, but the second is a hit because `A[i]` and A[i+128] are in the same cache block. Thus, we end up with a hit rate of **50%**.

5.4 Calculate the cache hit rate for the line marked Line 2:

The size of A is $8192 * 4 = 2^{15}$ bytes. This is exactly twice the size of our cache. At the end of Line 1, we have the second half of A inside our cache, but Line 2 starts with the first half of A. Thus, we cannot reuse any of the cache data brought in from Line 1 and must start from the beginning. Thus our hit rate is the same as Line 1 since we access memory in the same exact way as Line 1. We don't have to consider cache hits for total, as the compiler will most likely store it in a register. Thus, we end up with a hit rate of **50%**.