CS 61C Spring 2024 — Caches, AMAT — Discussion 11

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 We cannot use a 1KB cache in a 32-bit system because it's too small and cannot contain all possible addresses.

False. The purpose of the cache is not to hold every possible piece of memory at the same time, but rather to hold some parts of it only, so a 1KB cache is not "too small".

1.2 If a piece of data is both in the cache and in memory, reading it from cache is faster than reading from memory.

True. The cache is smaller and faster than memory.

1.3 Caches see an immediate improvement in memory access time at program execution.

False. A cache starts off 'cold', and required loading in values in blocks at first directly from memory, forcing compulsory misses. This can be somewhat alleviated by the use of a hardware prefetcher, that uses the current pattern of misses to predict and prefetch data that may be accessed later on.

1.4 Increasing cache size by adding more blocks always improves (increases) hit rate for all programs.

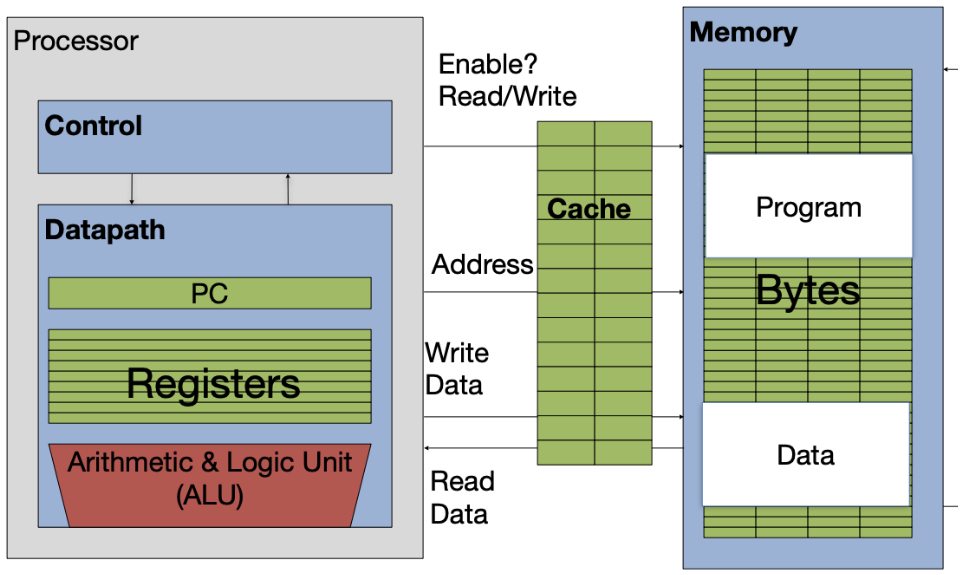
False. Whether this improves the hit rate for a given program depends on the characteristics of the program. As an example, it is possible for a program that only consists of a loop that runs through an array once to have each access be separated by more than one block (say, the block size is 8B, but 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.

1.5 Decreasing block size to increase the **number** of blocks held by the cache improves the program speed for all programs.

False. This question is similar to the one above, in that the answer to it depends on the program that is running. If we have a program with a for loop that loops through continuous memory (like an array), having a bigger blocks size and fewer blocks might be helpful, as the single blocks will holds more continuous data. For example, lets say cache A has 10 lines and a block size of 8 bytes, while cache B has 20 lines with a a block size of 4 bytes and the array we loop through has 80 characters. Cache A in this case will have 10 cache misses and 70 hits, while Cache B will have 20 misses and 60 hits.

2 Understanding T/I/O

We use caches to make our access to data faster. When working with main memory (RAM), the main problem faced is the fact that access to the main memory is very slow. In fact, modern processors take about 100 instructions cycles or more to access the main memory, meaning memory accesses become the bottleneck of our programs. Caches help fix this problem for us - they hold a portion of the data in main memory, that we might access again later on. They are closer to the processor in the memory hierarchy, and thus accessing a cache is much faster than accessing the main memory.



As seen above, the access to cache is the middle step between the CPU asking for a memory bit, and the actual main memory access - if the data is not found in the cache, only then is main memory accessed. This way unnecessary trips to main memory are avoided. One important detail is that caches are much smaller in size than main memory - this is why we have to be efficient in what we hold in caches. When we are saving data in caches, we need to be as efficient as possible. In order to do this, we make use of locality. We have two different kinds of locality to consider.

- **Temporal Locality:** If we have accessed a piece of information recently, it is possible that we will access it again. So, we hold this data in the cache.
- **Spatial Locality:** If we have accessed a memory location recently, it is probable that we will access the neighbouring addresses as well. So, we also keep the neighbouring addresses within the cache. An example is array accesses - if we access the 0th element of an array, it is probable we will also access the 1st one.

Note that caches hold the data in blocks that have a size equal to the block size of the cache.

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:

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

Given these definitions, the following is true:

$$\log_2(\text{memory size}) = \#\text{ memory address bits} = \#\text{ tag bits} + \#\text{ index bits} + \#\text{ offset bits}$$

Another useful equality to remember is:

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

One thing to consider when calculating index, offset, and tag bits is their order within an address:

Tag	Index	Offset

As seen above, the tag bits are to the left (most significant), the index bits are in the middle, and the offset bits are the to the right (least significant).

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 tag, index, and offset of the cache?

We can start by finding which bits correspond to the offset bits. 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. In this case, the offset is the 3 least significant bits. Denoting the most significant bit (MSB, on the left) as it 31 and the least significant bit (LSB, on the right) as bit 0, our offset bits are bits 0, 1, and 2.

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. 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.
From out T:I:O breakdown, we can see that the offset bits are the least significant bits and the next set of least significant bits is the index bits. We calculated that

there were 3 offset bits, so our index bits will start at bit 3 (remember the least significant bit is bit 0!). Since we have 2 index bits, this means that we can find the index bits at bits 3 and 4.

From our T:I:O breakdown, we can see that the tag bits are the most significant bits. Our tag is the remainder most-significant bits, so we can find our tag bits at bits 5-31.

2.2 Assume that we have the same cache scheme as the previous part (direct-mapped byte-addressed cache with capacity 32B and block size of 8B). 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`		
`0x000000CF`		
`0x000000F3`		

(Feel free to use the space below to draw your cache.)

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
0x000000CF     Tag 6, Index 1, Offset 7: R, Conflict
0x000000F3     Tag 7, Index 2, Offset 3: M, Compulsory
```

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

3 The 3 C's of Cache Misses

In order to evaluate cache performance and hit rate, especially with determining how effective our current cache structure is, it is useful to analyze the misses that do occur, and adjust accordingly. Below, we categorize cache misses into three types:

- **Compulsory**: A miss that must occur when you bring in a certain block for a first time, hence "compulsory". Compulsory misses occur when a program is first started, and the cache does not contain any of that program's data.

- **Conflict**: A miss that occurs if the block was fetched before and in our cache, but was evicted while the cache was not full. Increasing the associativity of the cache may help avoid conflict misses.
- **Capacity**: A miss that occurs if the block was fetched before and in our cache, but evicted while the cache was full. Capacity misses may be resolved by increasing the size of the cache.

3.1 True or False: Fully associative caches can never have conflict misses.

Fully associative caches are designed such that any block of data can be stored in any cache line. Doing so eliminates the possibility of conflict misses, which happens when multiple data blocks compete for the same cache line, seen in direct-mapped or set-associative caches.

3.2 Go back to question 2 and classify each miss (M) and replacement (R) as one of the 3 types of misses described above.

See solutions for Q3!

4 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
}
```

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

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

4.2 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 $= 20 - 4 - 10 = 6$

4.3 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%**.

4.4 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%**.

5 Cache Performance

Recall that AMAT stands for Average Memory Access Time. The main formula for it is:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

In a multi-level cache structure, we can separate miss rates into two types that we consider for each level.

- **Global:** Calculated as the number of accesses that missed at that level divided by the total number of accesses *to the cache system.*
- **Local:** Calculated as the number of accesses that missed at that level divided by the total number of accesses *to that cache level.*

5.1 In a 2-level cache system, after 100 total accesses to the cache system, we find that the L2\$ (L2 cache) ended up missing 20 times. What is the global miss rate of L2\$?

$\frac{20}{100} = 20\%$

5.2 Given the system from the previous subpart, if L1\$ had a local miss rate of 50%, what is the local miss rate of L2\$?

$\frac{20}{50\% * 100} = \frac{20}{50} = 40\%$. We know that L2\$ is accessed when L1\$ misses, so if L1\$ misses 50% of the time, that means we access L2\$ 50 times, of which we ended up having 20 misses in L2\$.

Suppose your system consists of:

1. An L1\$ that has a hit time of 2 cycles and has a local miss rate of 20%
2. An L2\$ that has a hit time of 15 cycles and has a global miss rate of 5%
3. Main memory where accesses take 100 cycles

5.3 What is the local miss rate of L2\$?

The number of accesses to the L2\$ is the number of misses in L1\$, so we divide the global miss rate of L2\$ with the miss rate of L1\$.

$\text{L2\$ Local miss rate} = \frac{\text{Misses In L2\$}}{\text{Accesses in L2\$}} = \frac{\text{Misses in L2\$}}{\text{Total Accesses}} / \frac{\text{Misses in L1\$}}{\text{Total Accesses}} =$

$\frac{\text{Global Miss Rate}}{\text{L1\$ Miss Rate}} = \frac{5\%}{20\%} = 0.25 = 25\%$

5.4 What is the AMAT of the system?

AMAT = 2 + 20% x (15 + 25% x 100) = 10 cycles, as the Miss Penalty of the L1\$ is the 'local' AMAT of the L2\$.

Using global rates of each level, alternatively, AMAT = 2 + 20% x 15 + 5% x 100 = 10 cycles (using global miss rates)

$\boxed{5.5}$ Suppose we want to reduce the AMAT of the system to 8 cycles or lower by adding in a L3$. If the L3$ has a local miss rate of 30%, what is the largest hit time that the L3$ can have?

Let H = hit time of the cache. Extending the AMAT equation so that the Miss Penalty of the L2$ is the 'local' AMAT of the L3$, we can write:
$2 + 20\% * (15 + 25\% * (H + 30\% * 100)) \leq 8$
Solving for H, we find that $H \leq 30$. So the largest hit time is 30 cycles.