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.

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.

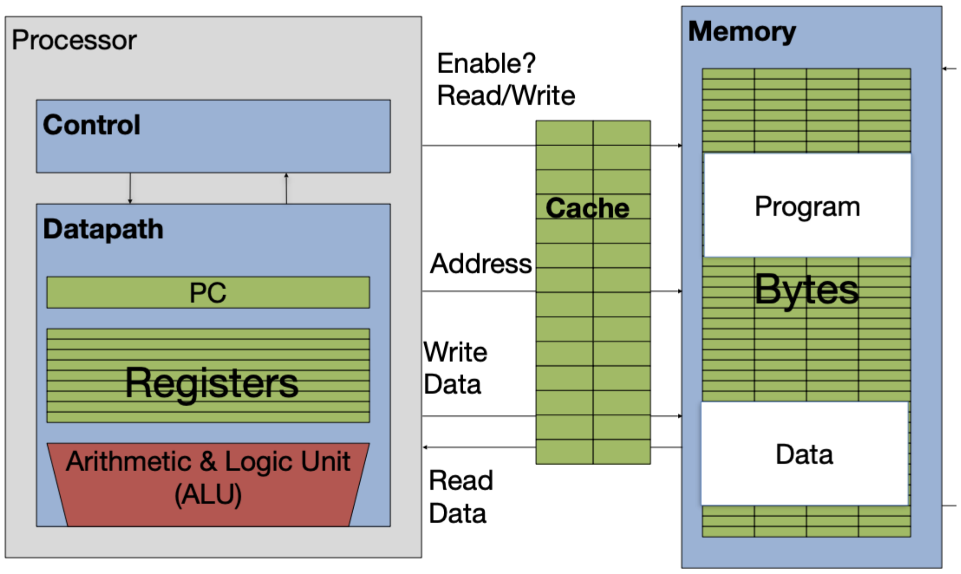
1.3 Caches see an immediate improvement in memory access time at program execution.

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

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

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?

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.)

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 the first time, hence "compulsory". Compulsory misses are cache attempts that would never be a hit regardless of the cache design.
- **Capacity**: A cache attempt that would be a hit if the cache capacity is increased.
- **Conflict**: A cache attempt that would be a hit if the cache associativity is increased. A conflict miss would not have happened if the cache is fully associative under at least one "consistent eviction policy". A "consistent eviction policy" is an eviction policy that keeps all the data in the lower-associativity cache.

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

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.

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?

4.2 What is the T:I:O breakdown?

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

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

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$?

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$?

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$?

5.4 What is the AMAT of the system?

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?