CS61C
Fall 2026

# Precheck: Caches

Discussion 10

## 1 Pre-Check: Caches T/F

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

1.2 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".

1.3 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.

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

False. A cache starts off 'cold' and requires 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, which uses the current pattern of misses to predict and prefetch data that may be accessed later on.

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

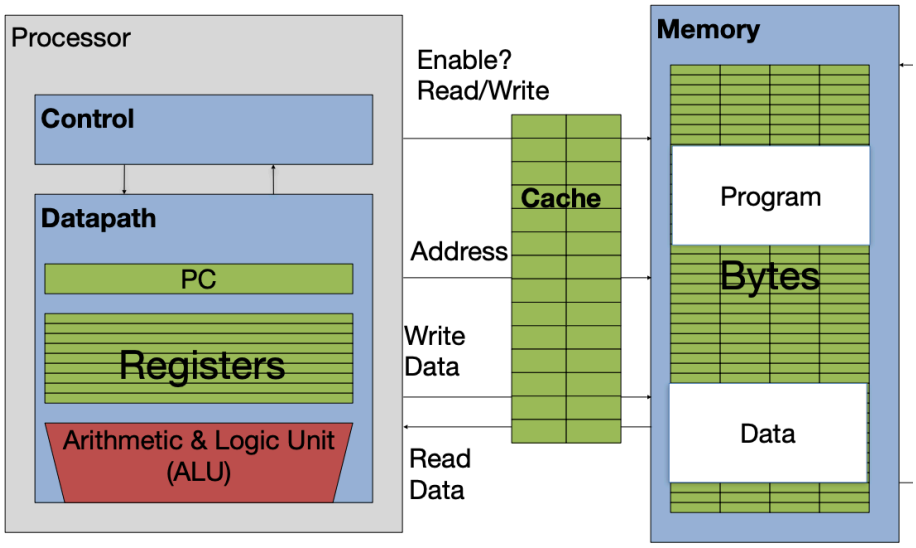
False. Whether this improves the hit rate for a given program depends on the characteristics of the program. For example, a program that loops through an array once may have each access be separated by more than one block (e.g., if the block size is 8B but we access every fourth element of an integer array, our accesses are separated by 16B). This results in compulsory misses, which cannot be reduced just by adding more blocks to the cache.

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

False. Similar to the previous question, the impact depends on the program. If a program iterates through contiguous memory (like an array), having larger block sizes with fewer blocks may be beneficial as each block contains more contiguous data. For instance, if Cache A has 10 blocks and a block size of 8 bytes while Cache B has 20 block and a block size of 4 bytes, and we loop through an array of 80 characters, Cache A will experience 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 instruction 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 neighboring addresses as well. So, we also keep the neighboring 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:

- **Tag:** Used to distinguish different blocks that use the same index.

  Number of Tag Bits = (# bits in memory address) - Index Bits - Offset Bits

- **Index**: The set that this piece of memory will be placed in.

$$\text{Number of Index Bits} = \log_2(\#\text{ of Indices})$$

- **Offset:** The location of the byte in the block.

$$\text{Number of Offset Bits} = \log_2(\text{Block Size})$$

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 to the right (least significant).

# 3 Cache Misses

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

1. **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

2. **Noncompulsory**: A cache miss that occurs after the data has already been brought into the cache and then evicted afterwards. If the miss could have been alleviated via increasing the cache size or associativity, then the miss is considered noncompulsory.

# 4 Cache Associativity

Direct-Mapped caches–where each block of memory maps to specifically one slot in our cache–is good for fast searching, simple hardware, and quick replacement, but not so good for spatial locality!

This is where we bring associativity into the matter. Associativity is the number of slots a memory block can map to in our cache. Thus, a Fully-Associative cache has the most associativity, meaning one memory block can map to any cache block. Our Direct-Mapped cache, on the other hand, has the least (being only 1-way set associative) because one memory block can only map to a single cache block.

For an N-way set associative cache, the following relationships are true:

$$\text{Number of Blocks} = N \times \text{Number of Sets}$$

$$\text{Index bits} = \log_2(\text{Number of Sets})$$

Ex: for a 2-way set associative cache with 4 index bits, there will be $2^4 = 16$ sets for $2 \times 16 = 32$ blocks in the cache. A single address will map to one of the 16 sets and will be placed in one of two blocks.

# 5 Replacement Policies

For direct-mapped caches, each block of memory maps to one specific block in our cache. On a cache miss, if there is data present in that cache block, then we must evict the block to make room for our new data.

For non-direct-mapped caches, we can choose one of multiple cache blocks to place our new data. When our cache is full, we will have to decide which block to evict to make space for the new data. **Block Replacement / Eviction policies** decide which block should be evicted. Common ones we may see in this class:

- **Least Recently Used (LRU)**
  - Replace the entry that has not been used for the longest time
  - Pro: Temporal Locality
  - Con: complicated hardware to keep track of access history
  - **Implementation**: bit counters for each cache block (see lecture slides for example)
- **Most Recently Used (MRU)**
  - Replace the entry that has the newest previous access
  - Pro: may support a workload that has less temporal locality
  - **Implementation**: MRU bits to keep track of most recent access
- **First-in, First-out (FIFO)**
  - Replace the oldest block in the set (queue)
  - Pro: reasonable approximation to LRU
  - **Implementation**: FIFO queue or similar approximation
- **Last-in, First-out (LIFO)**
  - Replace the newest block in the set (stack)
  - Pro: reasonable approximation to MRU

  - **Implementation**: LIFO stack or similar approximation
- **Random**
  - Pro: easy to implement and can work surprisingly well when given workload with low temporal locality

# 6 Write Policies

Store instructions write to memory which change the data. With a cache, we need to ensure that our main memory will eventually be in sync with our cache if we are changing the values. There exist two common **write policies** with different tradeoffs:

- **Write-through**: write to the cache and memory at the same time such that the data in cache and main memory will always be in sync.
  - Simple to implement but…
  - More writes to memory ⇒ longer AMAT
- **Write-back**: only write data to the cache and keep track of "dirty" blocks by setting a dirty bit to 1. When dirty block gets evicted, write changes back to memory.
  - More difficult to implement but…
  - Fewer writes to main memory ⇒ shorter AMAT

What happens when we have multiple caches simultaneously reading and writing to/from main memory? Take CS152 to learn about cache coherency and consistency!

# 7 AMAT (Average Memory Access Time)

Recall that AMAT stands for Average Memory Access Time. This is a way to measure the performance of a cache system. The formula for AMAT is:

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

In a multi-level memory hierarchies (e.g. multi-level caches), 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 memory system**.
- **Local:** Calculated as the number of accesses that missed at that level divided by the total number of accesses **to that memory level**.