CSE 351 Lecture 16 – Memory & Caches I

IEC Prefixes

With the rapid growth of computing, we often need to specify very large powers of 2. The standard prefixes such as kilo-, mega-, and giga- unfortunately mean different things in different contexts. In the SI system, they mean powers of $10^3 = 1000$. When talking about computer-related quantities, they often instead refer to powers of $2^{10} = 1024$. To avoid this confusion, the ***IEC prefixes*** unambiguously refer to powers of 1024.

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

The names come from shortened versions of the original SI prefixes and "bi" is short for "binary," but pronounced "bee." Because the binary prefixes are powers of 2^{10}, we can convert as follows:

2^{XY} "things" = [Y = 0 → 1, Y = 1 → 2, Y = 2 → 4, Y = 3 → 8, Y = 4 → 16, Y = 5 → 32, Y = 6 → 64, Y = 7 → 128, Y = 8 → 256, Y = 9 → 512] + [X = 0 →, X = 1 → Kibi-, X = 2 → Mebi-, X = 3 → Gibi-, X = 4 → Tebi-, X = 5 → Pebi-, X = 6 → Exbi-, X = 7 → Zebi-, X = 8 → Yobi-] + "things"

Examples: 2^{32} bits = 4 Kibi-bits

To hold 13.2 TiB of memory, you would need a 44-bit address space ($2^{44} = 16$ TiB).

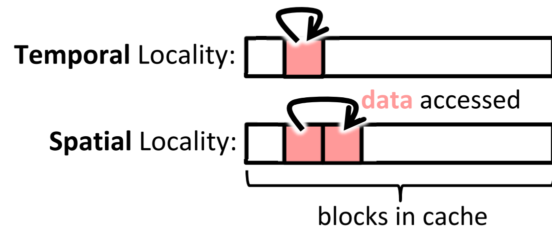
Caches and Cache Mechanics

We know that accessing memory is very slow compared to accessing registers. To help mitigate this issue, we introduce an intermediate level of ***caches*** (abbreviated as '\$'), which consist of memory with short access time used to store frequently or recently used data (including instructions). Caches hold a small subset of the data in memory, but are much faster to access (though slower than accessing registers). Data is transferred between caches and memory in ***blocks***, which are machine-specific fixed units of transfer much larger than a word.

When accessing memory, the CPU will always check the caches first. When the data we are looking for are found there, it is called a ***cache hit*** and the data are returned quickly. When the data we are looking for are not found in the caches, it is called a ***cache miss*** and we must fetch the data from memory and copy it into the cache. Depending on the current state of the cache, we will invoke the cache's *placement* and/or *replacement policies* to determine where the block will go in the cache.

Principle of Locality

Caches can greatly improve the performance of memory accesses due to the ***principle of locality***, which states that programs tend to use data at addresses equal to or near those that have been used recently. Separating these two related scenarios further, ***temporal locality*** states that recently referenced items are likely to be reference again in the near future while ***spatial locality*** states that items with nearby addresses tend to be referenced close together in time.



We take advantage of *temporal locality* by copying data into the cache on a cache miss, so that future accesses to that data will result in cache hits. For example, a local variable stored on the stack often gets accessed multiple times in quick succession during the execution of that function. We take advantage of *spatial locality* by bringing not just the referenced data, but an entire *block*'s worth of data into the cache on a cache miss. For example, when traversing an array, a block of memory typically will contain many elements of the array so that after one cache miss, the next array accesses will be cache hits even though it is our first time accessing those elements.

Cache Performance Metrics

More than just saying that caching improves our program's performance, we can use performance metrics to see its effects! On a hit, we check the cache and then return the data to the CPU, on a miss, we *additionally* have to fetch the block of data from memory. Based on these mechanics, we define the following performance parameters:

- ***Hit Time*** (HT): How long a cache hit takes – the time to check the cache and return data to the CPU. This parameter is based on the cache hardware.
- ***Miss Penalty*** (MP): How long it takes to fetch a block of data from memory. This parameter is based on the cache and memory hardware.
- ***Hit Rate*** (HR)/***Miss Rate*** (MR): The fraction of your code's memory accesses that result in a cache hit/miss. HR + MR = 100% by definition. These parameters are based on *your code*.

The performance metric we will use is called ***Average Memory Access Time*** (AMAT), which is defined as:

$$\text{Average Memory Access Time} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

Multilevel Caching

We won't cover any details, but modern systems typically use multiple levels of cache, with each successive level sitting "lower" (*i.e.* closer to memory) and being larger and slower to access. Everything discussed here applies at *each* level of cache. Block size may differ between different levels. A memory access can miss in one level and then hit in the next, causing the block to be copied into the higher level. AMAT, however, is compute for your overall system caching, taking all levels of cache into account.