Cache Memory Simulation

What is a cache and why do we need them?

In a computer, the CPU processes instructions and data. To run efficiently, the CPU needs to be able to access instructions and data quickly and these are generally stored on the Hard Drive (HDD), in main memory (RAM), and in Cache Memory. An HDD is slow, cheap, bulk storage. RAM is orders of magnitude faster than an HDD but also more expensive, and a Cache Memory is orders of magnitude faster than RAM, but is also the most expensive kind of storage. Cache Memory is therefore a compromise between speed and cost.

Imagine that in a library, all of the book shelves are main memory, but there are also shelves in a study room where books that have just been used are kept. The shelves in the study room are like the CPU Cache where they can be accessed very quickly but have much more limited space than the entire library. There are also many different methods you could use to determine which books remain on the study room shelves and when books should be replaced by others from the main library collection. The same thing happens with Cache Memory and each cache implementation has advantages and disadvantages. Three possible cache implementations are Direct Mapped, Fully Associative, and Set Associative.

With books, it is easy to find what you are looking for based on the book title, but what about when accessing data and instructions? Data and Instructions are stored in Memory on rows. Each row has a unique address used to access it. When the data is copied from Main Memory into Cache Memory, parts of the address from Main Memory are used to determine where in the cache that information should be stored and allow the data to be accessed again. In this way, using our library analogy, rows would correspond to books, while addresses would correspond to book titles.

Direct Mapped Cache

How does a Direct Mapped Cache Work?

A Direct Mapped Cache provides a specific cache location for any address in main memory. This makes it a very simple, very fast caching algorithm. Since each location in main memory maps to only a single cache location, whenever a collision occurs, the data already stored in the cache is simply replaced.

Think about a direct mapped cache like a bookshelf where books are arranged in alphabetical order, but only 26 slots exist, so if a books title starts with the letter ‘A’, there is only one spot for that book to go on the shelf. If another book starting with ‘A’ needs to go on the shelf, the first book must be replaced. Additionally, if you know the starting letter of a book, it’s very easy to see if the book you want is on the bookshelf, because you know exactly where on the bookshelf it lies.

Advantages & Disadvantages

The greatest advantage of a direct mapped cache is its simplicity. Since it rather closely emulates a hash table, it’s very easy to simulate with software. But, a direct mapped cache is the least flexible of the three types of caches simulated. If you think of this like a hash table, you can see that it is using an extremely simple hash function. This means there is a high likelihood of collisions. So, depending on which addresses from memory are being requested, there could be a large number of replacements occurring, resulting in significantly slower cache performance

The Simulation

This was the easiest of the cache types to simulate. Since each row from memory can only be placed in a single cache row, it was a relatively trivial thing to retrieve that cache row to check the tag and offset for cache hits and misses. Also this cache implementation didn’t require building or defining any type of replacement policy for the cache.

Since this was the first simulation that was completed, the interface did pose some challenges. Several versions and programming languages were attempted before finally settling on C# Windows Presentation Foundation as the final language and framework.

Fully Associative Cache

How does a Fully Associative Cache Work?

In a Fully Associative Cache, any location from main memory can be stored in any location within the cache. When the cache is full, a replacement policy is used to determine which row to replace.

Some replacement policies that could be used are random, least recently used, least frequently used, first in first out, but there are many others. The efficiency of a fully associative cache depends largely on the replacement policy that is chosen.

Using the bookshelf analogy then, a fully associative cache could allow any book with any starting letter in any spot on the bookshelf. However once the bookshelf was full, in order to put a new book on the shelf, you would have to follow the rules of your replacement policy to determine which book should be replaced.

Advantages & Disadvantages

The greatest advantage this cache scheme has over the Direct Mapped cache scheme is that every row in the cache will be immediately be replaced without having to implement a second search for the least recently used location.

Potential disadvantages occur depending on the replacement policy chosen. Numerous policies exist such as Least Frequently Used (LFU), Least Recently Used (LRU), and even Most Frequently Used (MFU). Additionally, especially with large cache sizes, it becomes cost prohibitive to implement this type of cache using certain replacement policies.

The Simulation

The biggest difference between the simulation for the Direct Mapped Cache and this Cache scheme was the manner in which cache hits and misses are found, and determining which rows to replace once the cache was filled.

In the simulation a Least Recently Used replacement policy was used as it is one of the most trivial to implement in software. To determine if a request resulted in a cache hit or a cache miss, the entire cache had to be scanned serially. In the scanning loop, the least recently accessed element was also stored so that if a miss occurred, the proper row could be immediately replaced without having to scan the entire cache for a second search for the least recently used location.

Set Associative Cache

How does a Set Associative Cache Work?

More generally referred to as an N-Way Set Associative cache, this type of cache is like a hybrid between the Direct Mapped Cache, and the Fully Associative cache. In implementation it has proven to be an efficient and cost effective compromise between the two previous cache implementations. Like the direct mapped cache, it uses an index to determine which locations map to main memory. However, at each index location a set of N rows from memory can be stored.

To check the cache for a hit, first the index is used to find the proper set just like in a Direct Mapped Cache. Next, the set is searched and the data returned or, in the event of a cache miss, a replacement policy is used to determine which row of the set to replace just like a Fully Associative cache.

Back to the bookshelf analogy, think of there being one shelf for books starting with any given letter and each shelf was able to hold exactly 16 books. So you could store 16 books starting with the letter A and any 16 books starting with the letter B etc. But when you need to store a 17th book starting with the letter A, one of the 16 books starting with A must be replaced.

Advantages & Disadvantages

The Set Associative Cache is a very functional hybrid of the Direct Mapped and Fully Associative cache schemes. It allows the simplicity of the direct mapped cache with each address in memory mapping to a set of rows in the cache, but has some of the flexibility of the Fully Associative cache when it comes to handling collisions allowing it to have a higher level of cache utilization.

Once again many of the disadvantages of this scheme come from the replacement policy chosen within the sets. Generally however a Least Recently Used or Pseudo Least Recently used replacement policy is used.

The Simulation

This was the most complex cache scheme to simulate both programmatically, and in regard to the User Interface. First the appropriate set was found based on the index, then that set had to be scanned to determine if the request was a cache hit or a cache miss.