

# Cache Simulation having frequency and recency lists (Assignment-12)

Sarthak Behera - 2018CS10384

September 2020

## Introduction

A cache is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere. A cache **hit** occurs when the requested data can be found in a cache, while a cache **miss** occurs when it cannot. Cache **hits** are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store; thus, the more requests that can be served from the cache, the faster the system performs.

To be cost-effective and to enable efficient use of data, **caches must be relatively small**<sup>1</sup>. Nevertheless, caches have proven themselves in many areas of computing, because typical computer applications access data with a high degree of locality of reference. Such access patterns exhibit temporal locality, where data is requested that has been recently requested already, and spatial locality, where data is requested that is stored physically close to data that has already been requested.

## Background

Cache algorithms (also frequently called cache replacement algorithms or cache replacement policies) are optimizing instructions, or algorithms, that a computer program or a hardware-maintained structure can utilize in order to manage a cache of information stored on the computer. Caching improves performance by keeping recent or often-used data items in memory locations that are faster or computationally cheaper to access than normal memory stores. When the cache is full, the algorithm must choose which items to discard to make room for the new ones.

One of the earlier used cache replacement policy was **LRU**(Least Recently Used) eviction policy wherein we maintain a recency list and the elements are stored in the order of as to when they were last accessed. Here temporal locality is fully utilized

---

<sup>1</sup>To effectively reduce the number of multiplexors that are used to access a cache block

so when the cache is full and then the element to be evicted is the least recently accessed cache block to make space for the incoming cache block. This policy has some disadvantages namely :-

- On each cache **hit**, the accessed block has to be moved to the **MRU**(Most Recently Used) position which may not be acceptable in a high performance computing systems as moving the cache block may take up more than a cycle of clock and thus comes with a large overhead.
- While **LRU** fully utilizes recency, frequency is given no importance in this principle. Frequently accessed elements may even be evicted if they are not accessed for some time.
- A scan or a search may evict all the pre-existing cache blocks thus we tend to have higher percentages of **misses** after a scan or a search.

## Replacement policy used

The various disadvantages of **LRU** led me to search for a better algorithm to implement for cache simulation and which was similar to what the assignment expected us to implement and the two most promising algorithms were **ARC**(Adaptive Replacement Cache) and **CAR**(CLOCK with Adaptive Replacement). Both of these algorithms are very similar as **CAR** is inspired from **ARC** and both basically maintain both recency and frequency lists which in terms of what the assignment expects us to implement are referred to as low and high priority groups. I have implemented **CAR** as the co-author of the original **CAR** paper is currently a professor at IIT-Delhi - Sorav Bansal and it seemed to have lesser overhead than **ARC**.

## Replacement policy explained

The policy basically maintains two lists - a recency list(**T1**)(low priority) and a frequency list(**T2**)(high priority) to keep the cache blocks along with ghost lists B1 and B2 which are maintained to decide the balance of size between **T1** and **T2**. The ghost lists are aimed at maintaining a determined size for **T1** which is decided at runtime thus making the replacement policy adaptive. A cache miss maybe a hit in the ghost list which then changes the needed size of **T1**, i.e., if we get a cache miss but a hit in **B1** means that our **T1** size is smaller so we increase needed size of **T1** and if we get a hit in **B2** then the size of **T1** is larger than needed so we decrease the needed size of **T1**.

I have made changes to the algorithms to accommodate associativity in the cache lines so as to support all kinds of associativity from direct-mapped to fully-associative. We follow the **CLOCK** algorithm in case of eviction when the cache is full and **LRU** when ghost lists are full. **CLOCK** is implemented as a modification of requirement of the assignment wherein we were required to implement an algorithm which demotes high priority groups after a certain time 'T'. This actually has a large memory

overhead as we need to store the last accessed time as well. In comparison, **CLOCK** only has to store a page-reference bit which can be set on cache hits.

The various advantages of the above cache replacement policy are :-

- Frequently accessed cache blocks(high priority) cannot be evicted that easily so a scan or a search fails to evict them thus the algorithm is scan-resistant.
- Instead of storing time of last access, **CLOCK** algorithm is used which doesn't have the extra memory overhead and is proven to be a approximation to **LRU**.
- Ghost lists help is adaptively change the size of high and low priority groups thus making the implementation useful in a wide range of programs.

## Implementation

The implementation of the above replacement policy was done through the classes **CacheDir**, **CacheAssoc** and **CacheBlock**. These classes contained the necessary information to compute the state of the cache after execution of a set of instructions. The **CacheDir** class contained all the **CacheAssocs** that the configuration needed. The **CacheAssoc** then contained the **CacheBlocks** that belonged to a particular associative set. The **CacheBlock** contains the data in the block, the index of the block and various other boolean information like valid bit and dirty bit. The **CacheDir** also contains information about the target size of T1, current instruction, associativity, etc. The implementation follows the implementation details laid out in the **CAR** paper. In case of writing to memory, the dirty bit is set so that the information can be written at the end of program execution(or if the cache block is to be evicted then written to memory). In the end, the hit rate is displayed along with the present cache blocks in each of the lists in **T1,T2,B1,B2**.

## Testing

Testing was done through a variety of test-cases which tested for the validity as well as correctness of insertion of cache block into **T1** or **T2**. Testing was also done to check the adaptive capacity of the replacement policy.