Report

Approach:

The underlying structure for cache was an array of length equal to number of sets determined by associativity, and a doubly linked list for implementing cache lines where each block has a maximum length determined by number of blocks per cache line. Note that to scale complexity, I didn't make use of ANY standard library except for math.log2 for calculating logs. I implemented a doubly linked list standard myself with alterations specific to cache in this assignment.

I implemented the code in Python, but unfortunately, I saved an incorrect version (this is what happens when you don't use git) and have been unable to recover the correct implementation. Since I ran out of time in the end, I wasn't able to generate the correct results due to some bugs in the process of optimizing my LRU block. However, most of it works. This is because previously, I was performing an LRU update in O(n) worse-case scenario, but I found a way of making it O(1) by implementing a hash-table of doubly linked list. However, that didn't quite work as expected so I tried reverting back to my original code solution from memory (terrible idea in hindsight) and was unable to get it to work again.

The logic works as follows:

While reading data from memory, if you have a cache miss, simply add the new block to specific index (if not Fully Associative) of the cache. Also, increment the total bytes transferred from memory to cache by the value of block size. If you have a cache miss, you need to check for 2 conditions:

1. Cache Miss and available space in cache line.
    a. In this case, we simply append to the respective cache set as indicated by index and set the new node as a head to the doubly linked list.
2. Cache Miss and no available space in cache line.
    a. This is where we evict the LRU block from our cache by removing the tail node. After removing the tail node, we simply set the new node as the head pointer as shown in 1.a.

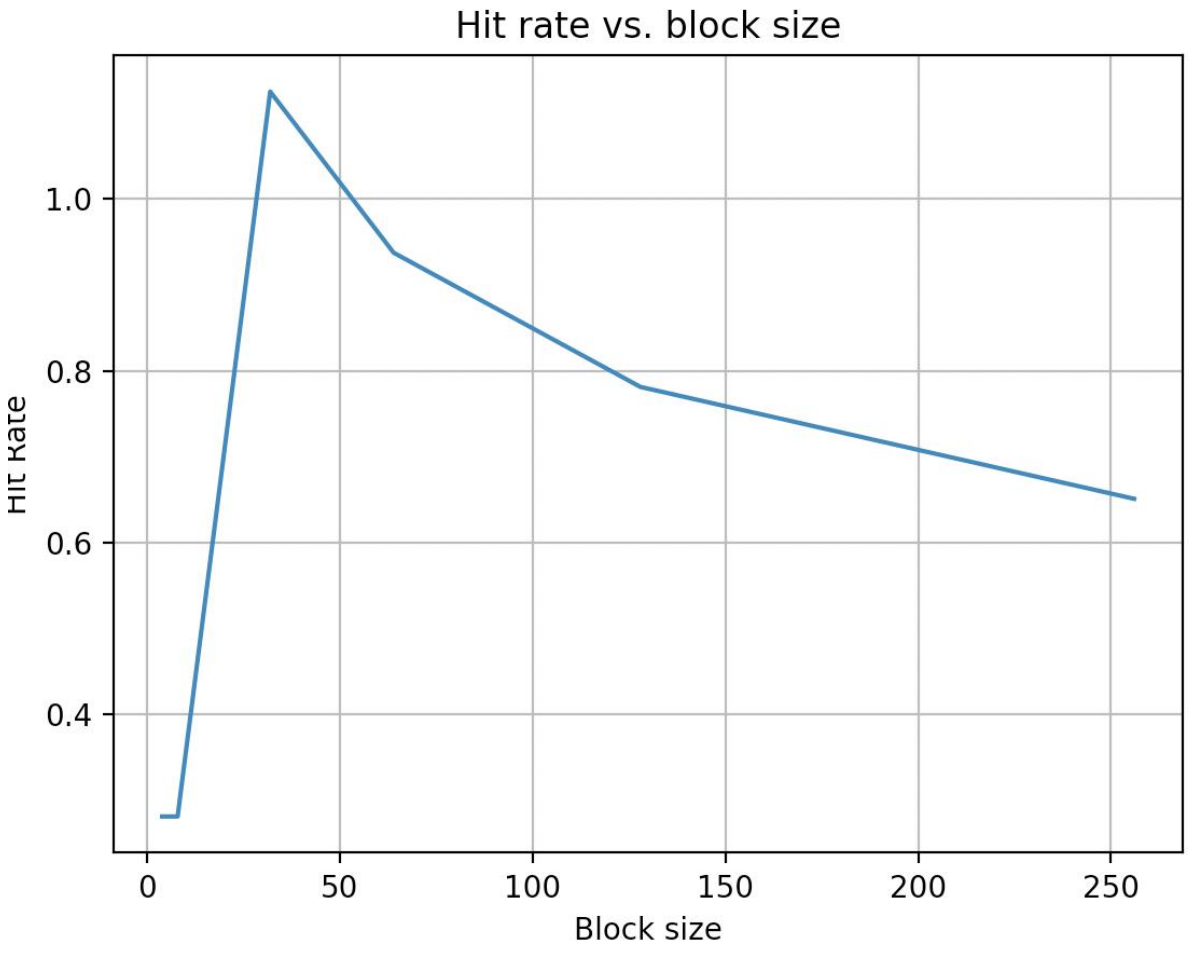
While storing, we need to consider 2 different processes:

1. Write Back: In this, if we get a hit, we simply set the dirty bit of the block from doubly linked list and move it to the set it as the tail for the list. Note that we don't move anything from cache to memory in this stage because the correct block is already in the cache. In the case of miss however, we need to load the correct block from memory. If this set is already full, we first need to evict the LRU block by simply removing the node from the doubly linked list and resetting the associative pointers. If this LRU block is dirty, we write from cache to memory, but on the other hand where this block is clean, we don't increment the transferred bytes counter (note that byte counter is incremented by block size and not word size). It is only when we have some space in the cache set that we set the dirty bit to be true for the entire block and not each word. Now, this is a bit different

from what I did, because I set the dirty bit to be true for each word. This is where using a hash table could've been super helpful because I could have a value representing dirty bit and perform O(1) operation instead of checking for any dirty bit status of N nodes for a specific cache line.

2. Write Through: In write through we write at a word level rather than a block level. If we get a hit on a write through, we increment by the block size. After this, we simply update the LRU as done previously. If we miss initially we load a new block in from memory you we also increment both counters.

Graphs:



Configuration: 1024, block sizes: [4, 8, 16, 32, 64, 128, 256], DM, WT

- Shows that between block size 8 through 32, hit rate increases and then goes down.