ECE2500 Project3 Report

Description

For this project, we need to simulate the interaction between Cache and Memory. In the simulation, we will use various Cache size and Block size to test. We need to see the difference between Caches with various Cache size and block size. The replacement strategy is LRU (Least Recently Used). We may assume the Cache is empty at the beginning of every test.

Test Case:

Cache Size (in bytes):

1. 1K
2. 4K
3. 64K
4. 128K

Block Size (in bytes):

1. 8
2. 16
3. 32
4. 128

Cache Placement Type:

1. Direct Mapped
2. 2-way set associative
3. 4-way set associative
4. Fully associative

Write Policy:

1. Write back
2. Write through

Implementation

To implement this project, I used a vector, “cache” (the name in my c++ project) as my data structure to simulate a cache. The vector “cache” is used to store multiple objects called “TagStorage”. Inside each “TagStorage”, there is an index number and another vector “tagstore” containing another object called “TagElement”. The index number is the index of the block. The vector “tagstore” can be treated as sets. If we need a two-way associate cache, there will be two objects “TagElement” inside the vector “tagstore”. If we need a four-way associate cache, there will be four objects “TagElenemt”. Direct map and fully associate will be treated as one-way associate cache and eight-way associate cache. Each object “TagElement”, contains three integers: “tag”, “priority”, “dirty bit”. “tag” is the tag stored in the block. “priority” is used for LRU to keep tracking the least recently used tag and most recently used tag (also, other “TagElement” will have priority number but least recently used tag is the one we need to focus). Every time I need to replace a tag, I will sort the vector “tagstore” based on the priority

of the elements inside. 0 means the most recently used tag. The bigger the number, the less the priority. After sorting, the bigger number, which means the least recently used tag, will be place in the first place of the vector. I can simply replace the one in the first place with my input tag then. "dirty bit" is used for writing back. When I need to replace a tag with dirty bit of 1, I need to write the block that needs to be replaced to the memory before replacement.

The purpose of "Read" function is to count how many bytes the cache read from memory and to place the block into the cache. For any given index and tag, the "Read" function will first check if the block exist in the cache. If the block doesn't not exist, it will go to the memory and fetch certain block to put into the cache and increase the total bytes transfer from memory to cache. To put the block into the cache, we need to check if the cache in certain index is full. If is it full, we need to use LRU as our principle to replace the block. During the replacement, if we see the dirty bit of certain block is 1, we need to increase the total byte transferred from cache to memory and set it back to 0.If it is not full, just simple push back the block.

The "WriteThrough" function is very similar to the "Read" function except that it will increase the total bytes transfer from cache to memory every time we call this function.

The "WriteBack" function is also very similar to the functions above. If the block already existed in the cache, we set the dirty bit of that block to 1 and do nothing else. If the block doesn't exist, we follow what we did in "Read" function except that we set that dirty bit of that block to 1 to remind us to write it back the memory next time when we need to replace the block.

The "CheckWriteBack" function is just to check if there are some block with dirty bit of 1 didn't write back to the memory. If there exist some blocks need to be wrote back. We need to increase the total bytes transferred from cache to memory accordingly.

The rest function is just for read the data in private.The main function will read in and output the data, and the Cache class will handle the calculation.

Trace file plot

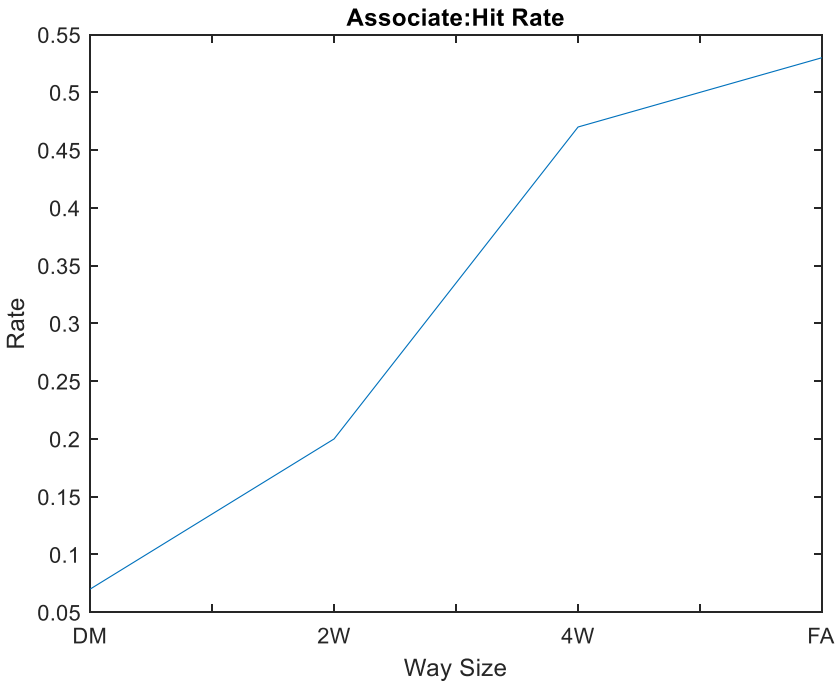
For associate.trace, the hit rate will increase as the ways/sets increase. More ways/ sets mean more space to store blocks in the same index. Normally it will keep increase and reach a max value. But here, I choose a case that the hit rate will keep increasing so we can see the difference.

For blocksize.trace, the hit rate will increase as the block size increase. The increment of the block size means that it has more possibility to include more data inside one block. So it is easier to hit. The cache size here I chose is 1024. No matter what the ways/sets are, the hit rate increase as the block size increase in the case I created.

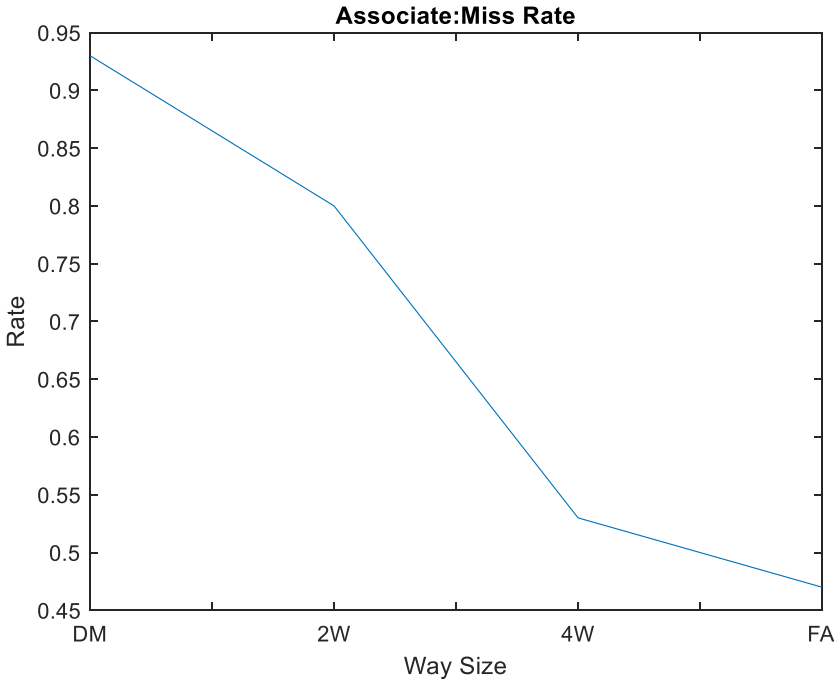
Associate.trace:

Cache Size:131072 Block Size:128

Hit Rate:



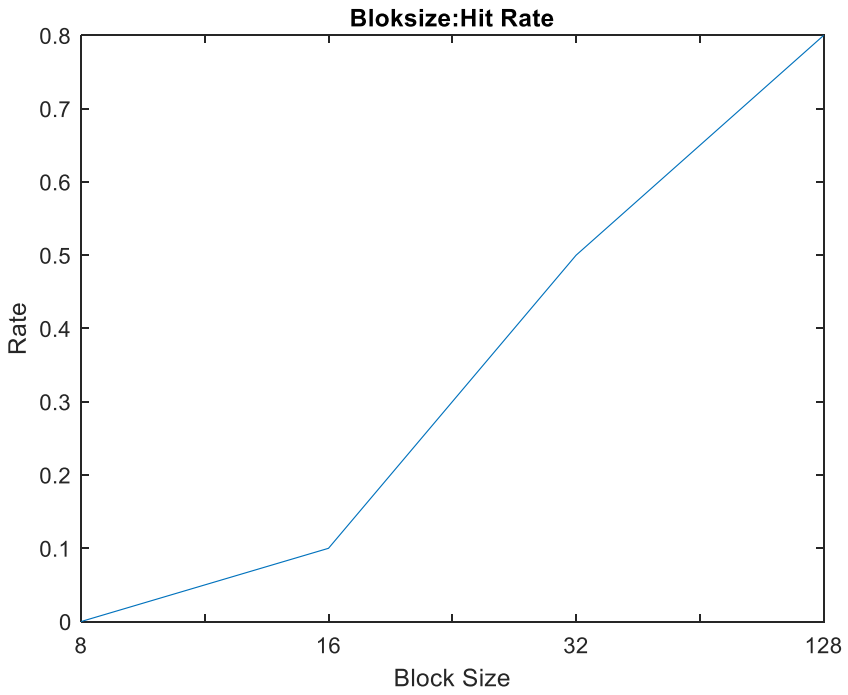
Miss Rate:



Blocksize.trace:

Cache Size:1024

Hit Rate:



Miss Rate: