4190.308

Computer Architecture

Spring 2019

Cache Lab Report

05/22 ~ 06/05

Name:	Jinje Han
Student ID:	2018-18786

Introduction

In this lab, students are expected to implement cache memory in C code. Since one main objective of the lab is understanding how cache memory works, students can implement simplified version of the memory. There are three replacement policies and two write allocate policies, but interactions between cache and memory are omitted. Therefore, the cache doesn’t need to include real blocks of data, just supposing the presence of them. Another main objective is seeking the relationship between cache’s properties and its performance. The cache is implemented as the user can set properties of it, so students are expected to do some experiments using it.

Important Concepts

The structure of cache, and how to deal with data using certain address is essential in this lab. Cache is divided into sets, and sets are divided into blocks. When an address is input to a cache to read a data, it is generally divided into three parts – tag, index, offset. Cache finds appropriate set with the index, and find if there is a line that contains the same tag in the address. If there is, it means that the desired data is in the block in the line, which can be found by the offset. If there’s not, the cache updates itself by getting data from memory, and returns data. If the set is full, cache determines which line will be evicted based on replacement policy. On the other hand, when an address is input to memory to write a data, whether cache data will be updated depends on writing policy. If the policy is write-allocate, the data is written in cache for now, and written on memory later. If it’s not, the data is written into memory directly, not touching the cache.

Implementations

There are two source code files that that constitute the imaginary cache memory. In <cache.h>, structure of components in a cache and headers of functions that are for using the cache are defined. Cache is composed of three components-main cache interface, set, and lines.

A line structure is the smallest unit of a cache. It contains a valid bit, which indicates if this line has valid information, and tag bits, which is used to find a certain block. In reality, there must be a block for saving data (which is the reason why use cache), but in this lab it doesn’t have to. In addition, in order to implement LRU(least-recently used) replacement policy, there has to be time value to record the last time I use the line.

In a set structure, there is a pointer that points to an array of lines. However, not all of the lines are always valid at every time. There can be lines that are not valid yet, and whether the lines are all full is important on the way read and write data. Therefore, the number of valid lines is stored. Further, if the replacement policy is round-robin, the index of the line that needs to be replaced next should be saved.

In main cache interface (which is named to distinguish from the whole cache), there is a pointer that points to an array of sets. The main data of cache are recorded in the sets, so data that are read and written are processed through this pointer. The write policy and data replacement policy are also in the interface. In order to easily access to the data that is looking for, the number of sets and associativity, and the number of index bits and offset bits in an address are recorded. Beside these, to measure the performance of the cache, number of cache access, cache hit, cache miss, and data replacement are recorded.

In <cache.c>, the main logics of functions using the cache whose headers are in <cache.h>. The first function is creating a cache when properties are given. In this function, first of all, it has to check the properties are valid. Capacity and block size must be powers of 2, and the former has to be bigger than the latter. Checking it, the number of sets, and index and offset bits are calculated sequentially. Second, the memory is allocated to cache and its components and their fields are initialized. Third, the values of cache are set. Values include the ones calculated above, and writing and replacing policy. There is also a function that deletes the cache, which deallocates the memory for it. Line, set, main cache interface have to be deallocated in order.

The second function is accessing the cache with address. The target cache, address, and behaviour type (which determines whether to read the data or write data on the address) are inputs. The information of number of bits of each section are gotten from the cache interface. Using it, the set index and line tag can be calculated, and whether it is cache hit or cache miss has to be determined. If it is cache hit, the number of cache hits in cache interface is increased, and if the replacement policy if LRU, the access time of the cache hit line has to be updated. If it is cache miss, many particulars has to be determined. First one is whether to write the data or not. If the instruction is writing data and the write policy is no write-allocate, nothing actually has to be done because there is no code about memory in this project. If it is decided to write data, whether the target set is full has to be determined. If the set is not full, the function has to find empty line to write the data in accordance with replacement policy. Round-robin policy selects victim line in order. Random policy selects victim line randomly, and if the line is not empty, function founds the next closest empty line and select it as victim. LRU policy just find empty line by linear search. The empty line is indicated by the valid bit value 0. If the set if full, the replacement method is little different from above. Round-robin is same, but random policy just select one victim line without additional operation. LRU policy finds the line that most time has passed after access, using the time field of the Line structure. After the victim line is decided, the tag and valid bit modified properly, and statistics including number of access, cache hit, cache miss, and data replacement is updated.

Experiment

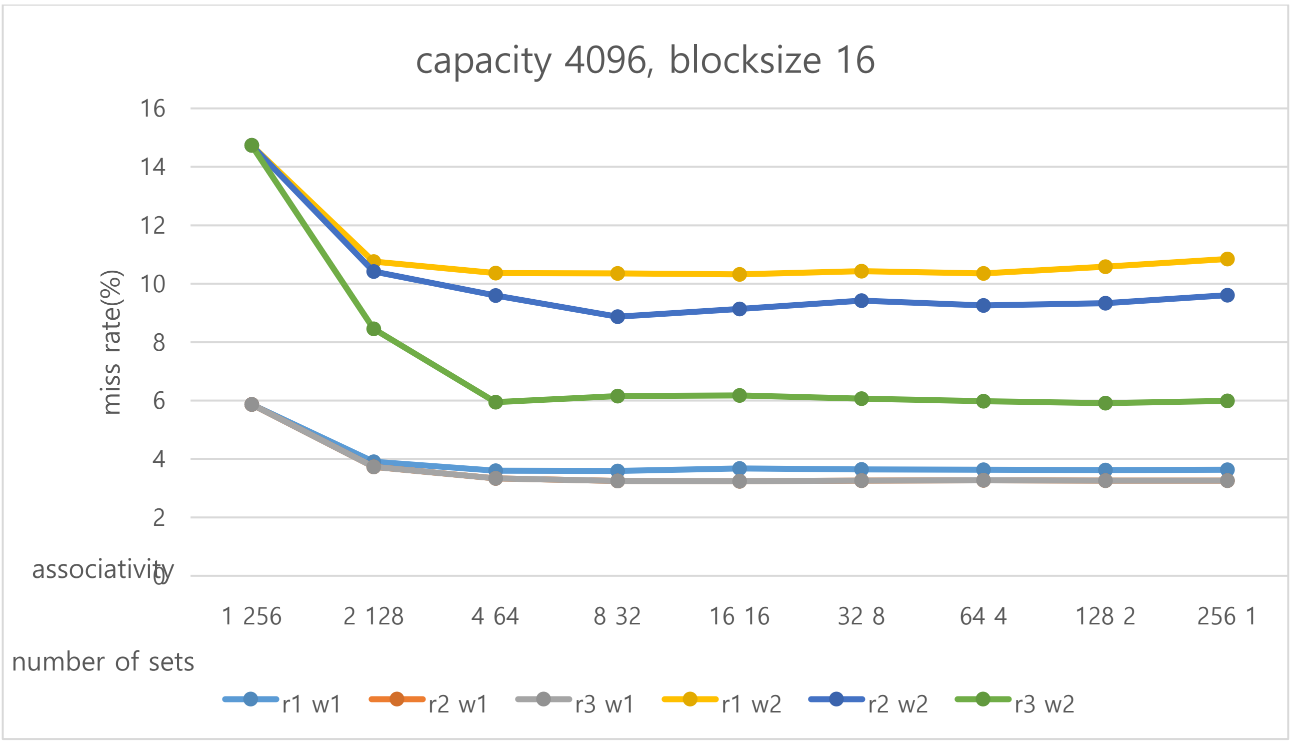
The following experiments are measurements of miss rate when properties of cache is modulated. The size of capacity is fixed on all experiments, then there are three variables left – block size, associativity, and number of sets. Three experiments are processed in fixing one of the three variables and changing the other two.

In each situations, experiments are done size times following combinations of write and replacement policy. Abbreviations in the result is as follows.

r1	replacement policy : round-robin
r2	replacement policy : random
r3	replacement policy : LRU(least-recently used)
w1	writing policy : write-allocate
w2	writing policy : no write-allocate

① fixed block size

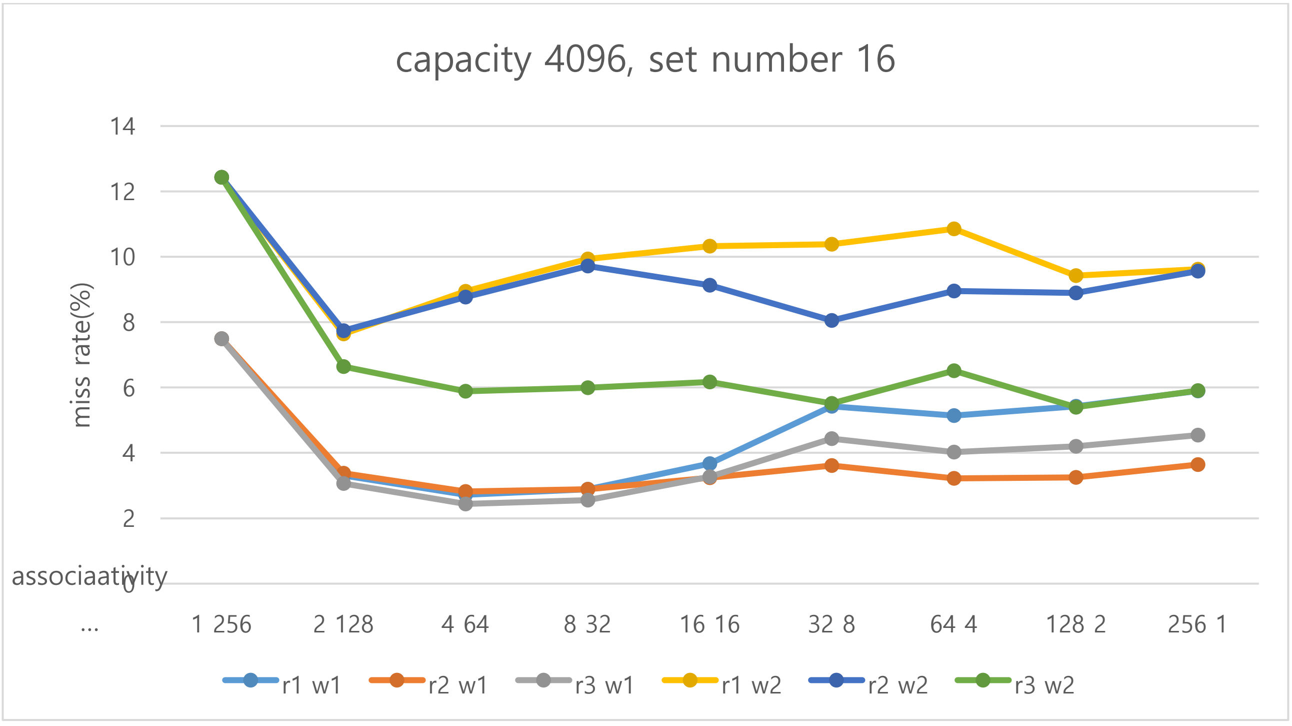
capacity = 4096 (=2^{12}), block size = 16 (=2^4)



The miss rate was almost same when it is write-allocate policy. However, it increased and varied a lot in no write-allocate policy. Therefore, we can find that it is more desirable to write data on cache because it's likely to be used later. Also, we can see the miss rate is dramatically increasing when the number of sets approaches to one in all policy conditions. It's assumed because the number of capacity misses is overall equal, but conflict miss increases because the number of set decreases.

② fixed number of sets

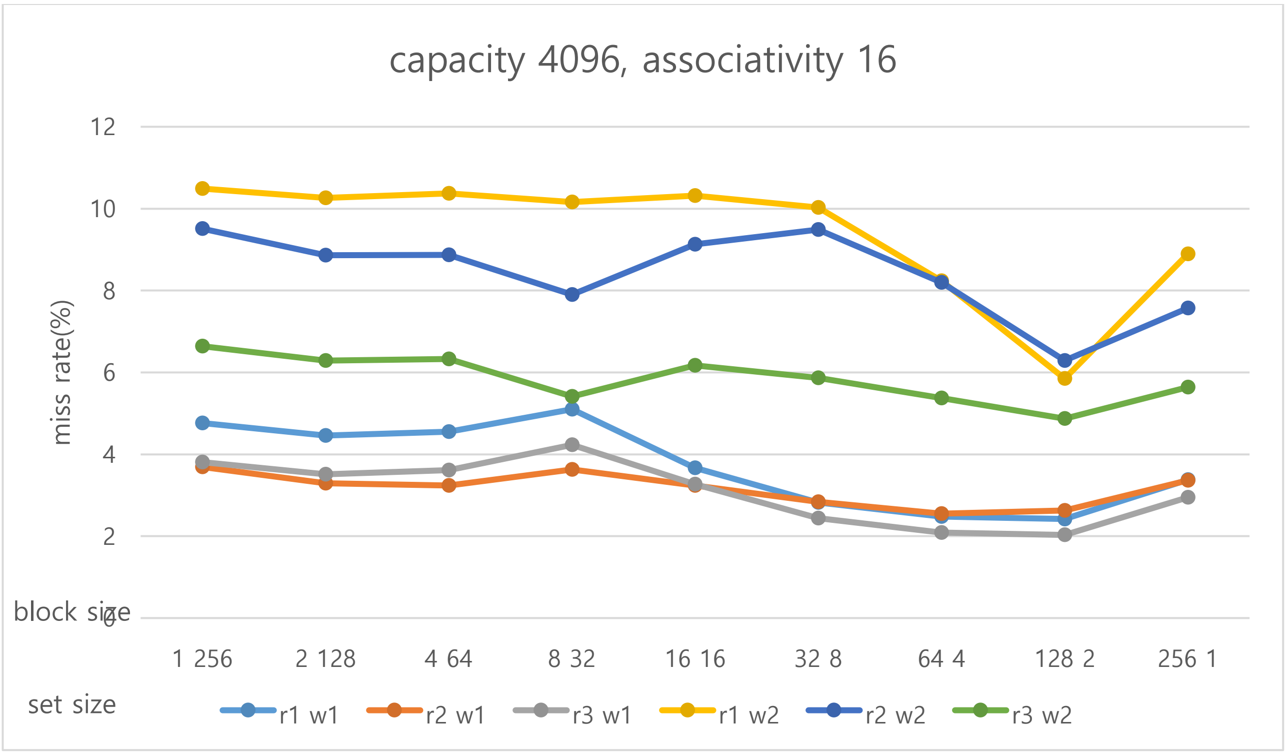
capacity = 4096 (=2^{12}), number of sets = 16 (=2^4)



As in the experiment above, overall miss rate was bigger in no write-allocate policy. The performance of cache was worst when the block size was smallest, and the other were pretty same. It can be inferred that as the number of tag bits gets bigger, it's much harder to find the right tag in a set.

③ fixed associativity

capacity = 4096 (=2^{12}), associativity = 16 (=2^4)



Situation when associativity is fixed showed best performance when both variables are not on both extremes. Associated with experiment 2, it concluded that the performance of cache is best on moderately small block size.

Conclusion

I think omitting memory and concentrating on miss rate and cache performance was good idea, since accessing to memory can be too complicated so that emphasis on understanding cache memory can be weakened. Thanks to that, I could efficiently learn the structure of cache memory and how it works. Implementing cache was generally not very complicated.