Cache Simulator

ECE/CS 472

(*Final Project*)

Tu Lam

June 8th, 2021

Abstract:

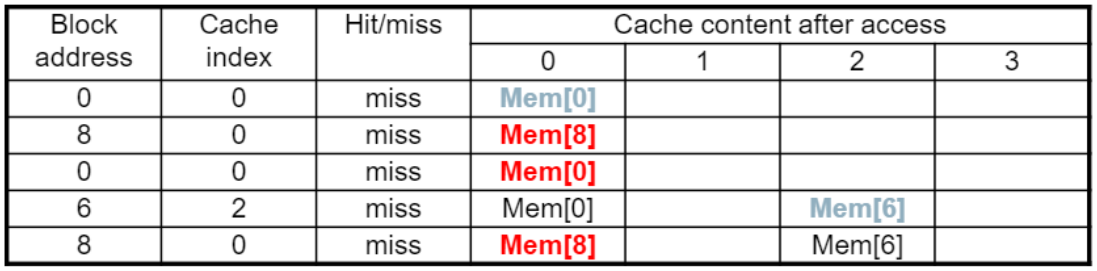
As the time of ECE/CS 472 coming to an end, we were present with the task of implementing a cache simulator, which we learn throughout the term, to prove how things are being write on the case of being “hit” and “miss” and demonstrate the understanding of the cache itself. In this report, we will discuss the background of the cache in the computer architecture, and then we will be showing the progress of how we build our cache simulator and show our understanding how cache work.

Introduction:

Cache, something that can be heard easily as any user with a computer can testify this. As browsing through the web on a web browser, many people stumble on page and thing are not being store and the computer would suggest something like why not clear your browser cache. This is where cache comes into mind for most people on the computer.

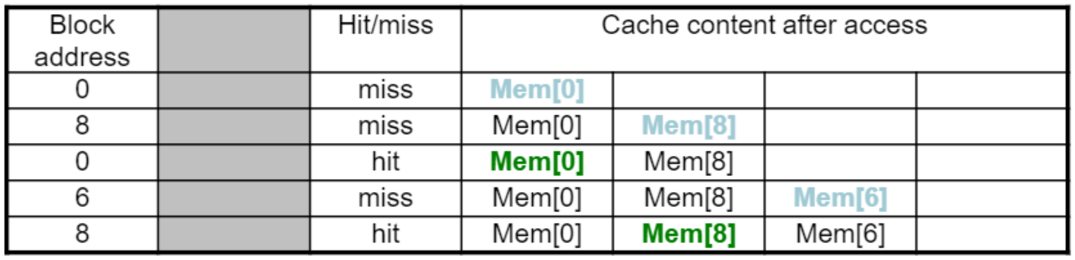
Cache the general layout act as both the hardware and software component that store data on a mapping for future request use. This way, when the component needed to be access again, the cache uses it algorithm to find the request that the user want faster to see if it is in the cache based on the concept of “hit” and “miss”. When the item matches with something in the cache, this represents a “hit” where the data in the cache is present and pulling that information out is faster. When it is a “miss” meaning that the data is not present in the data of the cache, and it will take longer to get to the information that the user want. Another analogy of this cache system is to think of it as a library. The library holds a vast large collection of books, if the user requests a book and the book is found in the library, that represent the same thing as a “hit”. While if the book is not there, then the library does not have it and have to hunt down where to get the book for the user. This shows how a “miss” being handle. Through analogy, this makes it easier how cache work behind the scenes.

That was just the backbone of the cache, but there is more thing then just the “hit” and “miss” in the cache. Other things that the cache needed to operate such as write policy, index of the table, replacement policy, cycle need to write/read from memory and from a block of cache, block size, associativity level, and more. Beside that, cache work in a way like a table where it will store data based on the associativity level. Below are a couple figures showing different type of mapping of cache based on the associativity.



Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

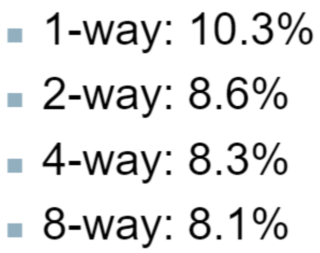
Figure No.1 – Direct Mapped Cache



Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

Figure No.2 – Fully Associative Cache

Through these associativity level cache table, this is how data are store and use to read and write to so later in the future, the request can be fetch faster based on the "hit" or "miss" implementation. As the level of associativity increase, the miss rate would also decrease as shown in the figure below. As that's the case, we will be discussion the implementation of the L1 cache simulator where we will handle the "hit" and "miss", the eviction, and how data are store and write.



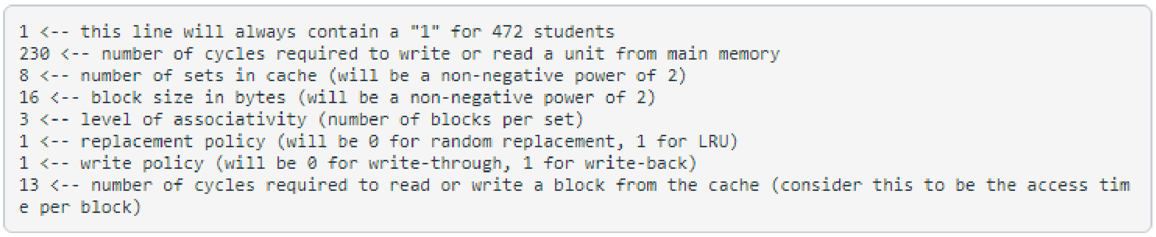
- 1-way: 10.3%
- 2-way: 8.6%
- 4-way: 8.3%
- 8-way: 8.1%

Figure No.3 – Percentage of Miss Rate as Associativity Level Increase

Implementation:

The whole implantation of the cache simulator will be done in the coding language of C. In the beginning, I created a code where it will take in two command line arguments as the user will provide a "config" and a "trace" file to be parse into data and store for later use to determine the type of cache table it will create for the simulator.

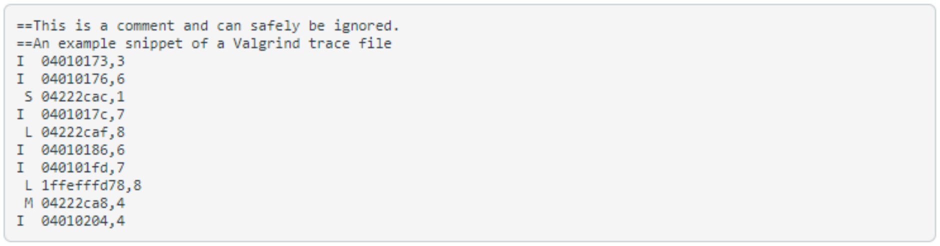
In the "config" file, this file contains the cache level (L1), the number of cycle use to read/write from main memory, number of set in cache, block size, associativity, replacement policy, write policy, and number of cycle use to read/write from a block of cache. Below shows how the layout will look like.



```
1 <-- this line will always contain a "1" for 472 students
230 <-- number of cycles required to write or read a unit from main memory
8 <-- number of sets in cache (will be a non-negative power of 2)
16 <-- block size in bytes (will be a non-negative power of 2)
3 <-- level of associativity (number of blocks per set)
1 <-- replacement policy (will be 0 for random replacement, 1 for LRU)
1 <-- write policy (will be 0 for write-through, 1 for write-back)
13 <-- number of cycles required to read or write a block from the cache (consider this to be the access tim
e per block)
```

Figure No.4 – Example of the Config File

On the other hand, the "trace" file will contain the instruction needed to determine what happen in the cache once obtain the data from the "config" file. Below will be a screenshot determine how the file will look.

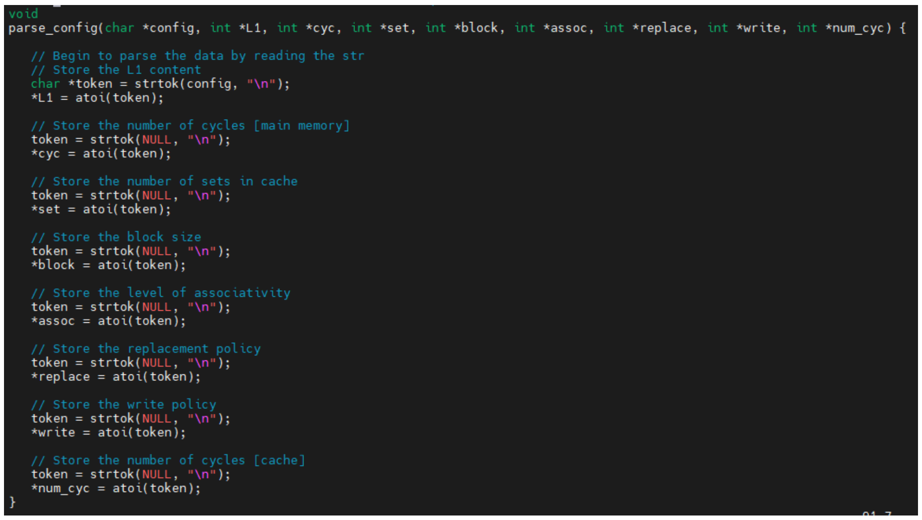


```
==This is a comment and can safely be ignored.
==An example snippet of a Valgrind trace file
I  04010173,3
I  04010176,6
 S 04222cac,1
I  0401017c,7
 L 04222caf,8
I  04010186,6
I  040101fd,7
 L 1ffefffd78,8
 M 04222ca8,4
I  04010204,4
```

Figure No.5 – Example of the Trace File

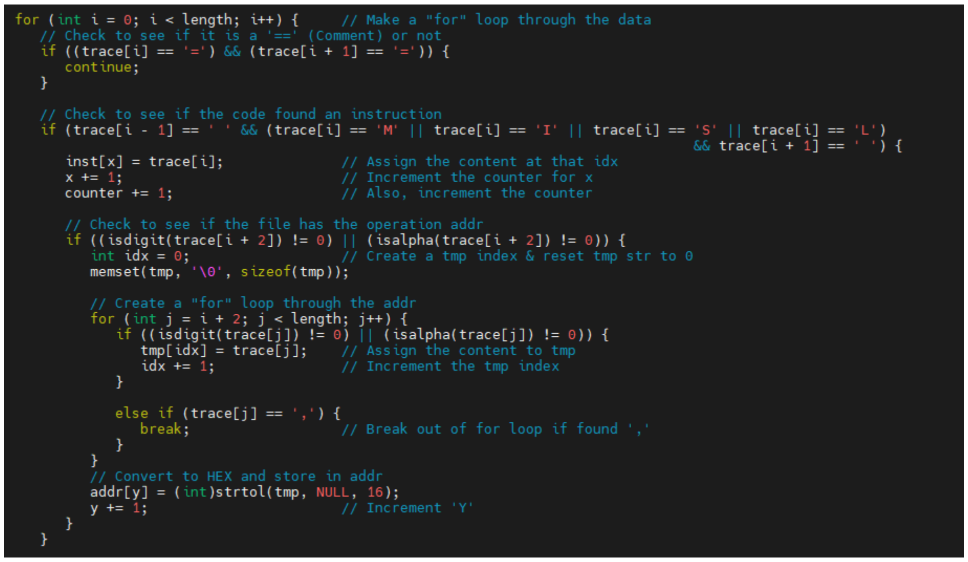
In the implementation, I initialize the data in the "config" and "trace" files through taking in the file and then give them each function and parse each data to their corresponding variable by passing it in by reference. The parsing is done through using the strtok() function for the function as that can easily be divided up for each component to their own variable.

Moving to the "trace" file. This file was harder to tackle on as there are more token to worry about to parse the data. This is in the range of the comment section '==', the space ' ', and the position to know where everything is to fetch from a file was challenging. I manage to get each data to their spot and parse it correctly. Firstly, in the function there is a "for" loop that loop through the entire data in the file and mark each index as it goes. If the instruction character was found, then it would store it into individual array. This is repeat again for the operation address and the size of operation. The entire of parsing of each individual data was done through the data structure of arrays, and below are few screenshot of the parsing code.



```
void
parse_config(char *config, int *L1, int *cyc, int *set, int *block, int *assoc, int *replace, int *write, int *num_cyc) {

   // Begin to parse the data by reading the str
   // Store the L1 content
   char *token = strtok(config, "\n");
   *L1 = atoi(token);

   // Store the number of cycles [main memory]
   token = strtok(NULL, "\n");
   *cyc = atoi(token);

   // Store the number of sets in cache
   token = strtok(NULL, "\n");
   *set = atoi(token);

   // Store the block size
   token = strtok(NULL, "\n");
   *block = atoi(token);

   // Store the level of associativity
   token = strtok(NULL, "\n");
   *assoc = atoi(token);

   // Store the replacement policy
   token = strtok(NULL, "\n");
   *replace = atoi(token);

   // Store the write policy
   token = strtok(NULL, "\n");
   *write = atoi(token);

   // Store the number of cycles [cache]
   token = strtok(NULL, "\n");
   *num_cyc = atoi(token);
}
```

Figure No.6 – The function of parse_config() for the parsing the config file



```c
for (int i = 0; i < length; i++) {     // Make a "for" loop through the data
   // Check to see if it is a '==' (Comment) or not
   if ((trace[i] == '=') && (trace[i + 1] == '=')) {
      continue;
   }

   // Check to see if the code found an instruction
   if (trace[i - 1] == ' ' && (trace[i] == 'M' || trace[i] == 'I' || trace[i] == 'S' || trace[i] == 'L')
                                                                                 && trace[i + 1] == ' ') {
      inst[x] = trace[i];                // Assign the content at that idx
      x += 1;                            // Increment the counter for x
      counter += 1;                      // Also, increment the counter

      // Check to see if the file has the operation addr
      if ((isdigit(trace[i + 2]) != 0) || (isalpha(trace[i + 2]) != 0)) {
         int idx = 0;                    // Create a tmp index & reset tmp str to 0
         memset(tmp, '\0', sizeof(tmp));

         // Create a "for" loop through the addr
         for (int j = i + 2; j < length; j++) {
            if ((isdigit(trace[j]) != 0) || (isalpha(trace[j]) != 0)) {
               tmp[idx] = trace[j];      // Assign the content to tmp
               idx += 1;                 // Increment the tmp index
            }

            else if (trace[j] == ',') {
               break;                    // Break out of for loop if found ','
            }
         }
         // Convert to HEX and store in addr
         addr[y] = (int)strtol(tmp, NULL, 16);
         y += 1;                         // Increment 'Y'
      }
   }
```

Figure No.7 – A snippet of finding the Instruction and Operation Address in parse_trace()

For going forward with the implementation of the cache simulator, I was not able to finish the program but have it logically layout on how I would implement but have a hard time displaying that in the coding itself, but I created a way in the code to simulate of how the number of cycle handle and how thing is being evicted or "hit" or "miss". In general, I could not implement the real cache simulator, but display it through logically how it work in the code and send it out to a file.

In the end, to put the data together into the file that the "trace" is in, I was able to come up with a function that will take the name of the "trace" file and concatenated together with the prefix ".out" to build the new output trace file. From there, I would then open the file and append data onto it from how the project wanted to be display on there.

Conclusion:

Overall, the cache simulator gives us a glimpse on how computer in the background is doing all these data fetching request that we are giving to them to see if they can pull the website up faster if it in the cache. In the long run, we also got a chance to look at CPU work on how much cycle it takes to determine if the data is going to take less cycle or more cycle to get the data if the data is not presented in the cache.

Throughout coding this project, there were some good and bad moment that the project gave me when it comes to the implementation. The good thing was that we only have to deal with L1 cache to worry about, but there also the way of how to implement the simulator into coding and not just grasping the concept itself. This way, it helps me improve my coding but also let me think deeply about the algorithm on how the cache would work. When it comes to the challenges that I face in the project, there is a few things to point out. First, is parsing the trace file to match every data and store it correctly. This took me a while to do as at first, I thought I got it to work, but I have not considered that the "trace" file will have comment in the file, which throw off my algorithm that I build and have to redesign it. The next is figuring out a way to keep track the index of each set if the file have more than 1 mapping which mean the level of associativity increase from N = 1 to N = 64 (max). This was challenging as to see if the data was store correctly and at the index to fetch the item out or put in correctly.

If I were given a chance to done something differently in this project, I would take more time to start it early to take time implementing as it is not an easy project to do. Also, I would like to implement a struct in my structure to hold the data that I need for the cache instead of depending on the array. The array was easy as first, but it was hard to keep track of the data as there were too many arrays to handle to get the correct data that I wanted. Overall, the project was fun, and it was a great exposure that every students need when it comes to understand the computer architecture and how data work on that scale.

Questions:

1. *What data structures did you use to implement your design?*

Answer: The type of data structures that I use during the implementation of my design is the array. Through this, I was able to keep track and be able to locate each instruction onto to the program and to see how the instruction work and store data correctly into their spot on the cache.

2. *What were the primary challenges that you encountered while working on the project?*

Answer: The primary challenges that I encounter while doing the project was definitely the parsing of data. The file of “trace” was definitely difficult as there so many regex I needed to take into consideration to get the correct data I want into the array. Another primary challenge is getting it onto the implementation. I have such a hard time on coding the segment and instead I implement my logic of how I understand the way it works onto the code and hope to display that I have my knowledge of it but couldn’t finish in time. Another hardship is keeping track of N = 1 to N = 64 associativity level that it could have on the code and keeping track of it is another difficult part about this project when it comes to implementing it.

3. *Is there anything you would implement differently if you were to re-implement this project?*

Answer: If I were to implement this project differently, I would have done a struct that hold in the linked list of the information. Through it, I can easily find and access data on a list and point to the data I want. From there, the linked list can help out a lot of associativity level when it comes to tracking it and knowing which data it will take.

4. *How do you track the number of clock cycles needed to execute memory access instructions?*

Answer: The number of clock cycles is being tracked by looking at how each instruction are performed. In the case of implementation of L1 only, there is only one cache to look at, so we don't have to worry about L2, L3, or more. In the instruction, every time we store, load, or modify, it takes about the number of cycle to access L1 + the number of cycle to access RAM. This case is involving the case of "miss" for all cases and a "hit" for the 'S' and 'M' instruction. While if "hit" occur, then it is only going to take only the cycle to access the L1 component. Other cases of eviction, it will take double the amount of L1 + RAM to perform that operation. Overall, with these clock cycles, we can get a glimpse of how data are access to load or store in the cache simulator.

Rule of Thumbs:

Through the completion of the project, I was not able to do it and couldn't fully demonstrate the rule of thumbs between the direct-map and the 2-way set associative level in the cache simulator. Throughout this, I will provide an explanation why these two are fit in the rule of thumbs when using the 2-way associative and the direct map.

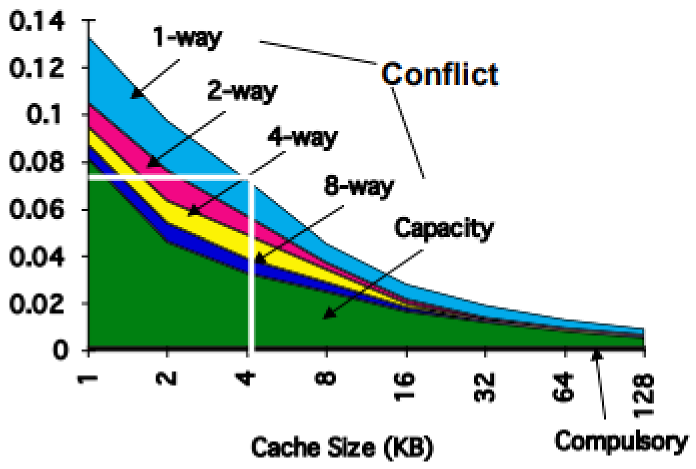
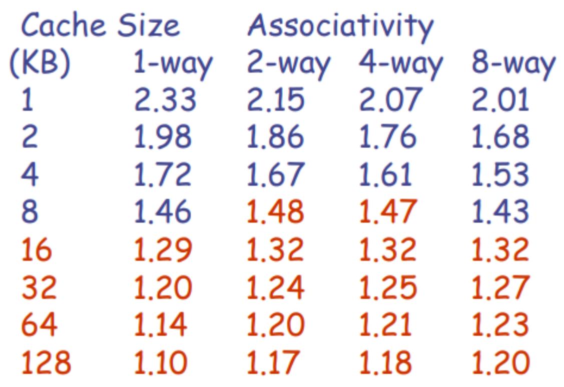


Figure No.8 – A graph of the N-way associative on their hit rate

This graph, shown above, display the hit rate of each N-way associativity level as it increases from direct-map to N-way. As you increase from 1 to a 2-way associative, the hit rate would still be the same or close to each other as the amount of space of item being miss is not that significant if we just added in an extra space in a 2-way associative. This show more prove that they fit into the rule of thumbs when examining the case of increasing block size. It said that when increase the block size, the miss rate goes down, but in the case of direct and 2-N, there not much significant in their number of miss rate. Below is another picture showing the increase of block size vs. miss rate.



Cache Size	Associativity			
(KB)	1-way	2-way	4-way	8-way
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	1.48	1.47	1.43
16	1.29	1.32	1.32	1.32
32	1.20	1.24	1.25	1.27
64	1.14	1.20	1.21	1.23
128	1.10	1.17	1.18	1.20

Figure No.9 – Block Size vs. Miss Rate for N-way

Through this we can see the number does decrease for miss rate but not that big of a gap throughout the increase of block size for cache. That’s why the rule of thumbs apply to 2-N and direct mapping as the miss rate for both cases is hand-in-hand close to each other to be determined as rule of thumbs that they are the same.