

Cache Simulator

Quinn Campfield

CS-472 Final Project, Spring 2021

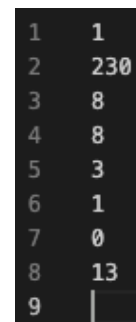
campfieq@oregonstate.edu , 933040836

Introduction:

The task for this assignment was to implement a Cache Simulator. The program runs with two arguments supplied by a user, one being a configuration file and the next being a trace file with a sequence of memory operations. Only a L1 cache simulator was implemented in this project, as per the project requirements. The cache will be built to handle store, load, and modify operations and will skip over comment lined and instruction operations as we can assume a different cache will handle those.

The configuration file will be structured the same way for all versions. First being the number of caches, for my project this will always be 1. Then the number of cycles required to read or write a unit from main memory. The third line is the number of sets in the cache, this will be a positive number and a power of 2. Line 4 will be the block size in bytes, also non-negative power of 2. Line 5 is the level of associativity, which is the number of blocks per set. Line 6 will be the replacement policy, 0 for random and 1 for LRU. Line 7 will be the write policy, 0 for write-through and 1 for write-back. Finally line 8 will be the number of cycles required to read or write a block from cache. This is an example of what a configuration file would look like.

| | |
|-----|--|
| 1 | 1 cache |
| 230 | Number of cycles to read/write from memory |
| 8 | Number of sets in cache |
| 8 | Block size in bytes |
| 3 | Level of associativity |
| 1 | Replacement policy (1=LRU) |
| 0 | Write policy (0=write-through) |
| 13 | Cycles to read/write a block from cache |



```
1 1
2 230
3 8
4 8
5 3
6 1
7 0
8 13
9 |
```

Figure 1: A table of each line in a configuration file and its meaning. To the right is a screenshot of the file that is used as an example.

The trace file will have a lot more variability but will also still be very structured. Each line in the trace file is an operation line or a comment. All comments will start with an equals sign and you skip these lines. All operation lines in the trace file will follow the same pattern of "operation address,size". This allows easy parsing to get the variables from this file. So the types of instructions you can expect are Instruction load, Store operation, Load operation, and Modify operation. All conveniently signaled by the first letter of their names I, S, L, and M respectively. For the purpose of the project, we are assuming a different cache is handling Instruction Load operations. A Store operation means data needs to be written from the CPU into the cache or/and main memory depending on the write policy. A Load operation means data is loaded from the cache into the CPU. Modify operation means there is a special case of a data load followed immediately by a data store. The address will be the 64-bit hexadecimal

number representing the address of the first byte that is being requested. The size of the memory operation is a decimal value indicating number of bytes. This would be an example of what a trace file would look like.

| | |
|----------------|----------------------------------|
| ==comment line | Just a comment |
| L 100,6 | Load, address 100, size 6 |
| S 180,5 | Store, address 180, size 5 |
| M 180,8 | Modify, address 180, size 8 |
| I 80,6 | Instruction, address 100, size 6 |

Figure 2: These are example of lines to expect in a trace file. Shown are comment, load, store, modify, and instruction from top to bottom.

Cache Structure:

For this assignment the cache is structured through using a vector to hold values. The vector cache is then stored with a high value for the amount of blocks that we were given in the config file. If the value at an index is still equal to this max then we know it hasn't been hit yet. The vector is of type unsigned long int which is 4 bytes per index.

```
34 // create your cache structure
35 // ...
36 int numBlocks = this->ci.numberSets * this->ci.associativity;
37 this->max = 999999999;
38 static vector<unsigned long int> cache;
39 this->cache = &cache;
40 for (int i = 0; i < numBlocks; i++){
41     cache.push_back(this->max);
42 }
43
```

Figure 3: The code for generation of the cache vector.

We also have a vector to use for LRU filled with values for the number of sets given. This is for the LRU offset that we will only use if the replacement policy is 1 in the config file.

```
44 static vector<std::queue<unsigned int> > lruOffset;
45 for (unsigned int i = 0; i < this->ci.numberSets; i++){
46     lruOffset.push_back(queue<unsigned int>());
47     lruOffset[i].push(0);
48 }
49 this->lruoffset = &lruOffset;
```

Figure 4: The code for generation of the LRUoffset vector.

Clock Cycles:

Clock cycles are given in the config file on line 2 and line 8. From line 2 we are given the number of cycles it takes to read or write from main memory. Line 8 is the number of cycles to read or write a block from cache. We can then use these two numbers depending on what operation we are running and depending on if we get a hit, miss, or eviction we add a different amount of cycles. It also depends on the operation we are running because we will have read or write set or not set. If we are doing a store operation we will set write to true and then we can also check for the write policy. Depending on both of those we will add a certain amount of cycles to the global counter and print to that line the number of cycles. For a load operation we don't need to check the write policy, it just varies depending on a hit, miss, or eviction. A modify operation will run both a store and load individually and that is why we get two lines in the output file for each modify operation called.

Challenges:

I think using the started code was extremely use full for this project, but it also meant I had to fully understand the process that was setup to occur for each event. Thankfully the code was well commented but I still found myself doing long traces when something wouldn't output how I wanted to. However, once I started understanding the code more thoroughly it felt much easier. It was also hard to find some issues in my code where running one trace file wouldn't generate the issue but another would and when testing I would stick to one trace for a while and not realize I was moving on without fixing an issue.

Things to Implement Differently:

I think the structure of my cache could have been more thought out. I was trying to make a Cache struct to use to hold information about each cache but resorted to using a vector. I think an obvious thing to implement differently is add more caches, it wasn't required for myself, but I think that would improve the performance that we get as of now. I also think there is an easier way to calculate the cycles than what I did because I have a lot of if else statements embedded in each other and overlapping amount of cycles being added that could be simplified.

Part 2: There is a general rule of thumb that a direct-mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2.

| Direct-mapped cache, size N | 2-way set associative cache, size N/2 |
|--|---|
| <pre>resources > ≡ testconfig1 1 1 2 230 3 8 4 8 5 1 6 1 7 0 8 13</pre> <p>Config file with Direct mapping due to the 1 on line 5. Each block is 8 bytes, with 8 sets in cache.</p> <pre>resources > ≡ simpletracefile.out 1 L 0,8 244 L1 miss 2 L 80,8 487 L1 miss eviction 3 L 100,6 487 L1 miss eviction 4 L 0,8 487 L1 miss eviction 5 S 180,5 487 L1 miss eviction 6 L 100,2 487 L1 miss eviction 7 L 80,8 487 L1 miss eviction 8 M 180,6 487 L1 miss eviction 9 M 180,6 243 L1 hit 10 L 100,8 487 L1 miss eviction 11 S 100,5 243 L1 hit 12 S 180,8 487 L1 miss eviction 13 L 80,8 487 L1 miss eviction 14 M 0,8 487 L1 miss eviction 15 M 0,8 243 L1 hit 16 M 0,4 13 L1 hit 17 M 0,4 243 L1 hit 18 L 280,4 487 L1 miss eviction 19 M 80,8 487 L1 miss eviction 20 M 80,8 243 L1 hit 21 L1 Cache: Hits: 6 Misses: 14 Evictions: 13 22 Cycles: 7803 Reads: 13 Writes: 7 23</pre> <p>Output from Trace file with direct mapping.</p> | <pre>resources > ≡ testconfig2 1 1 2 230 3 4 4 8 5 2 6 1 7 0 8 13</pre> <p>Config File 2-way associative due to line 5 being 2. Each block is 8 bytes, with 4 sets in cache.</p> <pre>resources > ≡ simpletracefile2.out 1 L 0,8 245 L1 miss 2 L 80,8 245 L1 miss 3 L 100,6 488 L1 miss eviction 4 L 0,8 488 L1 miss eviction 5 S 180,5 488 L1 miss eviction 6 L 100,2 488 L1 miss eviction 7 L 80,8 488 L1 miss eviction 8 M 180,6 488 L1 miss eviction 9 M 180,6 243 L1 hit 10 L 100,8 488 L1 miss eviction 11 S 100,5 243 L1 hit 12 S 180,8 488 L1 miss eviction 13 L 80,8 13 L1 hit 14 M 0,8 488 L1 miss eviction 15 M 0,8 243 L1 hit 16 M 0,4 13 L1 hit 17 M 0,4 243 L1 hit 18 L 280,4 488 L1 miss eviction 19 M 80,8 488 L1 miss eviction 20 M 80,8 243 L1 hit 21 L1 Cache: Hits: 7 Misses: 13 Evictions: 11 22 Cycles: 7099 Reads: 13 Writes: 7 23</pre> <p>Output from trace file with 2-way associative.</p> |

```
resources > ≡ sample1_trace.out
1 L 0,4 244 L1 miss
2 L 80,8 487 L1 miss eviction
3 L 10,4 244 L1 miss
4 L 102,6 487 L1 miss eviction
5 L 2,6 487 L1 miss eviction
6 S 180,5 487 L1 miss eviction
7 L 100,2 487 L1 miss eviction
8 L 4,20 487 L1 miss eviction
9 L 100,2 487 L1 miss eviction
10 L 80,8 487 L1 miss eviction
11 M 180,6 487 L1 miss eviction
12 M 180,6 243 L1 hit
13 L 100,8 487 L1 miss eviction
14 S 100,5 243 L1 hit
15 S 180,8 487 L1 miss eviction
16 L 80,8 487 L1 miss eviction
17 M 0,8 487 L1 miss eviction
18 M 0,8 243 L1 hit
19 M 0,4 13 L1 hit
20 M 0,4 243 L1 hit
21 L 280,4 487 L1 miss eviction
22 M 80,8 487 L1 miss eviction
23 M 80,8 243 L1 hit
24 L1 Cache: Hits: 6 Misses: 17 Evictions: 15
25 Cycles: 9021 Reads: 16 Writes: 7
26
```

Output from Trace file with direct mapping.

```
resources > ≡ sample2_trace.out
1 L 0,4 244 L1 miss
2 L 80,8 487 L1 miss eviction
3 L 10,4 244 L1 miss
4 L 102,6 487 L1 miss eviction
5 L 2,6 487 L1 miss eviction
6 S 180,5 487 L1 miss eviction
7 L 100,2 487 L1 miss eviction
8 L 4,20 487 L1 miss eviction
9 L 100,2 487 L1 miss eviction
10 L 80,8 487 L1 miss eviction
11 M 180,6 487 L1 miss eviction
12 M 180,6 243 L1 hit
13 L 100,8 487 L1 miss eviction
14 S 100,5 243 L1 hit
15 S 180,8 487 L1 miss eviction
16 L 80,8 487 L1 miss eviction
17 M 0,8 487 L1 miss eviction
18 M 0,8 243 L1 hit
19 M 0,4 13 L1 hit
20 M 0,4 243 L1 hit
21 L 280,4 487 L1 miss eviction
22 M 80,8 487 L1 miss eviction
23 M 80,8 243 L1 hit
24 L1 Cache: Hits: 6 Misses: 17 Evictions: 15
25 Cycles: 9021 Reads: 16 Writes: 7
26
```

Output from Trace file with direct mapping.

```
resources > ≡ sample3_trace.out
1 L 0,4 244 L1 miss
2 L 1f,1 244 L1 miss
3 L 20,1 244 L1 miss
4 L1 Cache: Hits: 0 Misses: 3 Evictions: 0
5 Cycles: 732 Reads: 3 Writes: 0
6
```

Output from Trace file with direct mapping.

```
resources > ≡ sample1_trace2.out
1 L 0,4 245 L1 miss
2 L 80,8 245 L1 miss
3 L 10,4 245 L1 miss
4 L 102,6 488 L1 miss eviction
5 L 2,6 488 L1 miss eviction
6 S 180,5 488 L1 miss eviction
7 L 100,2 488 L1 miss eviction
8 L 4,20 488 L1 miss eviction
9 L 100,2 13 L1 hit
10 L 80,8 488 L1 miss eviction
11 M 180,6 488 L1 miss eviction
12 M 180,6 243 L1 hit
13 L 100,8 13 L1 hit
14 S 100,5 243 L1 hit
15 S 180,8 243 L1 hit
16 L 80,8 488 L1 miss eviction
17 M 0,8 488 L1 miss eviction
18 M 0,8 243 L1 hit
19 M 0,4 13 L1 hit
20 M 0,4 243 L1 hit
21 L 280,4 488 L1 miss eviction
22 M 80,8 488 L1 miss eviction
23 M 80,8 243 L1 hit
24 L1 Cache: Hits: 9 Misses: 14 Evictions: 11
25 Cycles: 7600 Reads: 16 Writes: 7
26
```

Output from trace file with 2-way associative.

```
resources > ≡ sample2_trace2.out
1 L 0,4 245 L1 miss
2 L 80,8 245 L1 miss
3 L 10,4 245 L1 miss
4 L 102,6 488 L1 miss eviction
5 L 2,6 488 L1 miss eviction
6 S 180,5 488 L1 miss eviction
7 L 100,2 488 L1 miss eviction
8 L 4,20 488 L1 miss eviction
9 L 100,2 13 L1 hit
10 L 80,8 488 L1 miss eviction
11 M 180,6 488 L1 miss eviction
12 M 180,6 243 L1 hit
13 L 100,8 13 L1 hit
14 S 100,5 243 L1 hit
15 S 180,8 243 L1 hit
16 L 80,8 488 L1 miss eviction
17 M 0,8 488 L1 miss eviction
18 M 0,8 243 L1 hit
19 M 0,4 13 L1 hit
20 M 0,4 243 L1 hit
21 L 280,4 488 L1 miss eviction
22 M 80,8 488 L1 miss eviction
23 M 80,8 243 L1 hit
24 L1 Cache: Hits: 9 Misses: 14 Evictions: 11
25 Cycles: 7600 Reads: 16 Writes: 7
26
```

Output from trace file with 2-way associative.

```
resources > ≡ sample3_trace2.out
1 L 0,4 245 L1 miss
2 L 1f,1 245 L1 miss
3 L 20,1 245 L1 miss
4 L1 Cache: Hits: 0 Misses: 3 Evictions: 0
5 Cycles: 735 Reads: 3 Writes: 0
6
```

Output from trace file with 2-way associative.

Figure5: Table of tests for Direct-mapped on the left and 2-way set associative on the left.

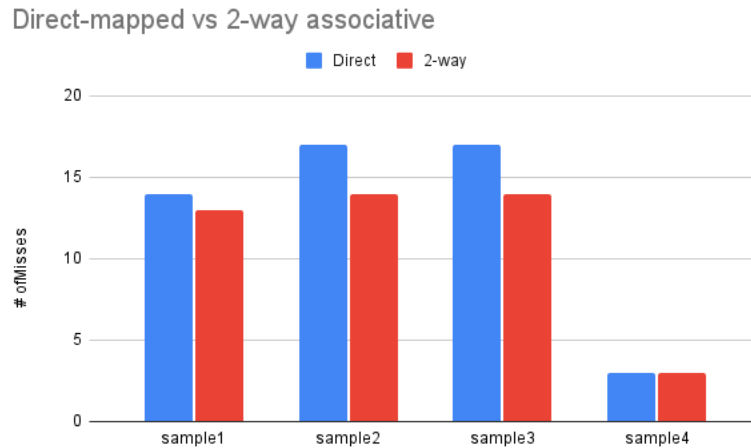


Figure 6: A graph comparing the number of misses for direct-mapped and 2-way associative for each test run.

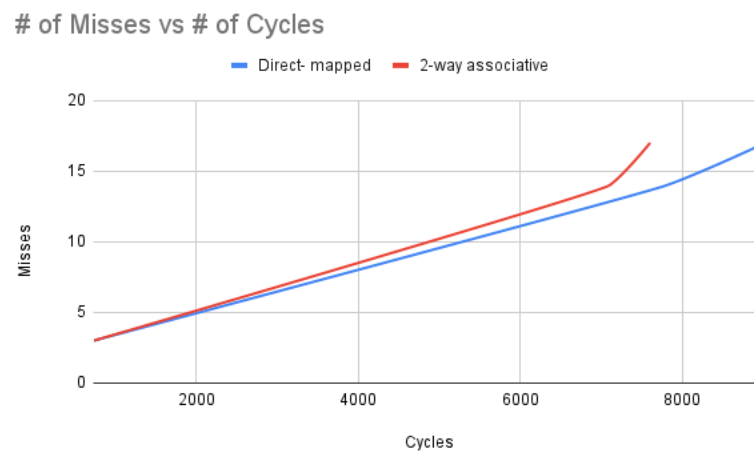


Figure 7: A graph comparing the number of misses for to the number of cycles for direct-mapped and 2-way associative.

In these figures we can see that the number of misses for both direct-mapped and 2-way associative increases as total cycles increase. We can also see that direct-mapped and 2-way associative have a different number of misses than each other for each test. However for the number of cycles and the amount of misses the two are still very similar. On the bigger tests Direct Mapped with a miss rate of 0.00188 or 0.188% and 2-way associative with a miss rate of 0.00224 or 0.224% are extremely close being less than 0.05% apart makes the difference insignificant. Based on my tests I would have to argue that a direct-mapped cache of size N and a 2-way set associative cache of size $N/2$ do in fact have the same miss rate.