

Memory Design

Course Name: EG-212, Computer Architecture- Processor Design

Student Names: Sathak Raj, Archit jaju, Harjot singh

Student ID's: IMT2023569, IMT2023128, IMT2023064

6 Sept 2024

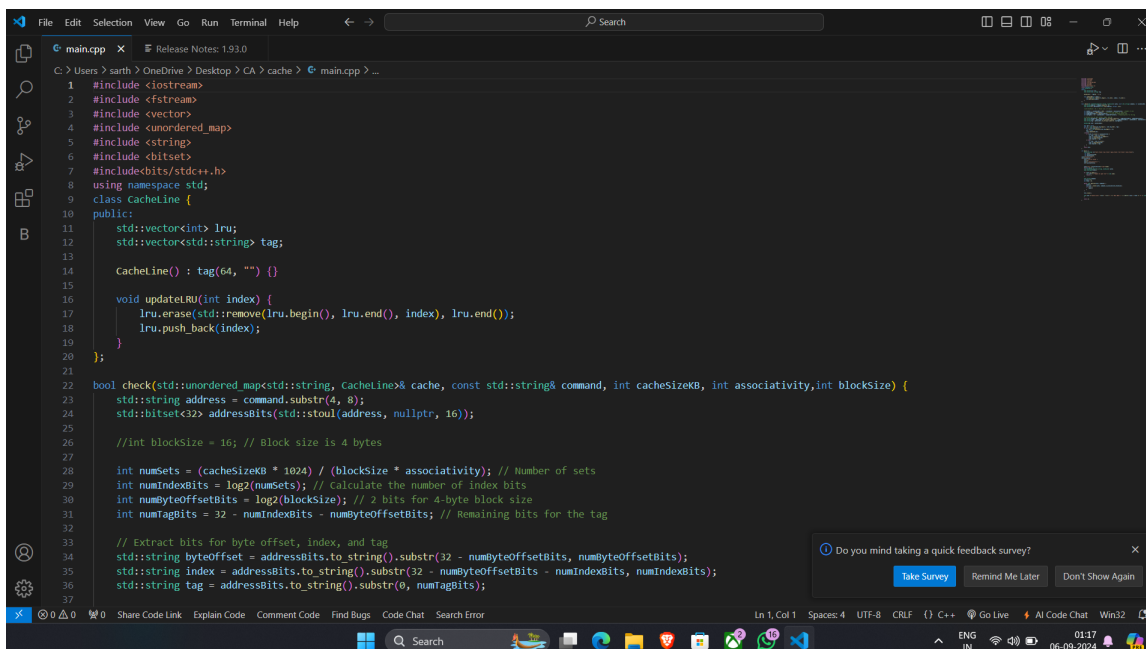
Abstract

This project is a detailed simulator for fully associative caches, designed to mimic how these caches work in real-world scenarios. It handles essential tasks like calculating tag bits, byte offsets, and indexes to accurately determine whether data access results in a cache hit or miss. To manage the cache efficiently, the simulator uses the LRU (Least Recently Used) algorithm, which helps decide which cache blocks to replace based on usage history. This ensures that the cache memory is used as effectively as possible.

1 Question A

The code simulates a fully associative cache with a Least Recently Used (LRU) replacement policy. It processes memory addresses from a trace file, checking for cache hits and misses. Memory addresses are divided into tag, index, and byteOffset. Hits update the LRU order, while misses replace the least recently used tag. The miss rate is calculated and printed at the end.

1.1 Code



```
1 #include <iostream>
2 #include <fstream>
3 #include <vector>
4 #include <unordered_map>
5 #include <string>
6 #include <bitset>
7 #include <bits/stdc++.h>
8 using namespace std;
9 class CacheLine {
10 public:
11     std::vector<int> lru;
12     std::vector<std::string> tag;
13
14     CacheLine() : tag(64, "") {}
15
16     void updateLRU(int index) {
17         lru.erase(std::remove(lru.begin(), lru.end(), index), lru.end());
18         lru.push_back(index);
19     }
20 };
21
22 bool check(std::unordered_map<std::string, CacheLine>& cache, const std::string& command, int cacheSizeKB, int associativity, int blockSize) {
23     std::string address = command.substr(4, 8);
24     std::bitset<32> addressBits(std::stoul(address, nullptr, 16));
25
26     //int blockSize = 16; // Block size is 4 bytes
27
28     int numSets = (cacheSizeKB * 1024) / (blockSize * associativity); // Number of sets
29     int numIndexBits = log2(numSets); // calculate the number of index bits
30     int numByteOffsetBits = log2(blockSize); // 2 bits for 4-byte block size
31     int numTagBits = 32 - numIndexBits - numByteOffsetBits; // Remaining bits for the tag
32
33     // Extract bits for byte offset, index, and tag
34     std::string byteOffset = addressBits.to_string().substr(32 - numByteOffsetBits, numByteOffsetBits);
35     std::string index = addressBits.to_string().substr(32 - numByteOffsetBits - numIndexBits, numIndexBits);
36     std::string tag = addressBits.to_string().substr(0, numTagBits);
37 }
```

```

37
38 CacheLine& line = cache[index];
39
40 bool hit = false;
41 auto it = std::find(line.tag.begin(), line.tag.end(), tag);
42 if (it != line.tag.end()) {
43     int ind = std::distance(line.tag.begin(), it);
44     hit = true;
45     line.updateLRU(ind);
46 } else {
47     if (line.lru.size() == associativity) {
48         int ind = line.lru.front();
49         line.lru.erase(line.lru.begin());
50         line.lru.push_back(ind);
51         line.tag[ind] = tag;
52     } else {
53         int ind = line.lru.size();
54         line.lru.push_back(ind);
55         line.tag[ind] = tag;
56     }
57 }
58 return hit;
59
60
61 int main() {
62     vector<string> v={"twolf.trace","gcc.trace","gzip.trace","mcf.trace","swim.trace"};
63     int sz=0;
64     int associativity=0;
65     int blockSize=0;
66     cout<<"block size-> ";
67     cin>>blockSize;
68     cout<<"size of cache ";
69     cin>>sz;
70     cout<<" associativity-> ";
71     cin>>associativity;
72
73

```

```

73
74 cout<<"for "<<associativity<<"-way"<<endl;
75 for(auto &s:v) {
76     std::unordered_map<std::string, CacheLine> cache;
77     std::ifstream file(s);
78
79     if (!file.is_open()) {
80         std::cerr << "Unable to open file" << std::endl;
81         return 1;
82     }
83
84     std::string command;
85     int miss = 0;
86     int total = 0;
87
88     while (std::getline(file, command)) {
89         total++;
90         bool hit = check(cache, command,sz,associativity,blockSize);
91         if (hit) {
92             miss++;
93         }
94     }
95     file.close();
96
97     std::cout <<"cache size-> "<<sz<<" "<<miss<<" "<<"Miss Rate = " << (100.0 * miss) / total << "%" << std::endl;
98 }
99
100 return 0;
101
102
103

```

This code is the base code.

The base code works for question (a).

For (b), varying the memory is done by changing the number of bits allotted to index and tag.

Each extra bit for index doubles the memory size. For (c), varying the block size means adding more offset bits and reducing index bits.

If block size is being reduced, the existing byte offset will be removed and index bits will be increased.

For (d), varying the number of ways involves increasing the size of the list with the tag values and varying index bits, offset bits and tag bits.

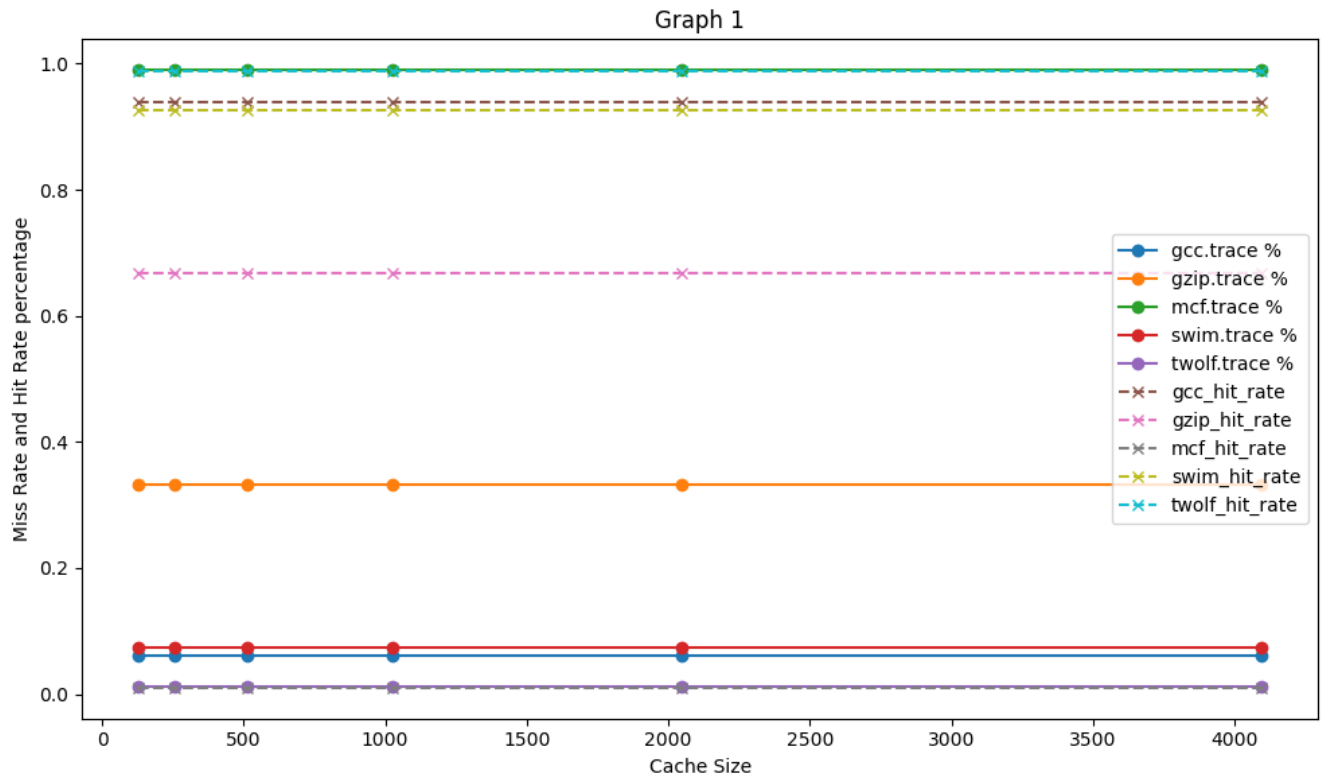
2 Question B

The base code is used and the memory size is changed by varying the bits allotted to the index tag.

2.1 Result

Cache Size	gcc.trace	gzip.trace	mcf.trace	swim.trace	twolf.trace	gcc.trace hitrate
128	0.0619838	0.332945	0.989676	0.0738012	0.0123875	0.9380162
256	0.0619838	0.332945	0.989676	0.0738012	0.0123875	0.9380162
512	0.0619838	0.332945	0.989676	0.0738012	0.0123875	0.9380162
1024	0.0619838	0.332945	0.989676	0.0738012	0.0123875	0.9380162
2048	0.0619838	0.332945	0.989676	0.0738012	0.0123875	0.9380162
4096	0.0619838	0.332945	0.989676	0.0738012	0.0123875	0.9380162

2.2 Graphs



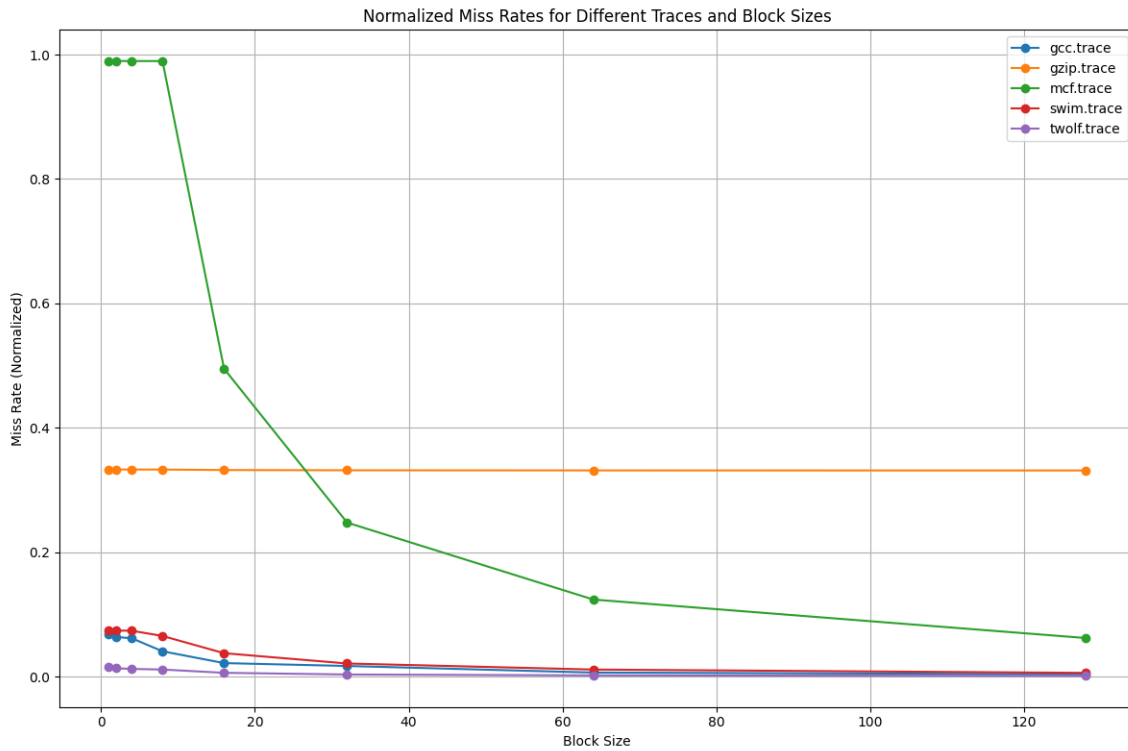
3 Question C

The base code is used with varying the block size means adding more offset bits and reducing index bits.

3.1 Result

Block Size	gcc.trace	gzip.trace	mcf.trace	swim.trace	twolf.trace
1	6.80108	33.2961	98.9754	7.45565	1.52312
2	6.37523	33.2959	98.9713	7.4065	1.3392
4	6.16445	33.2945	98.9676	7.37748	1.23855
8	4.07343	33.2928	98.9617	6.53577	1.14017
16	2.17498	33.2144	49.497	3.76757	0.612024
32	1.71107	33.1747	24.7622	2.10955	0.340083
64	0.654084	33.1539	12.392	1.13888	0.197588
128	0.379109	33.1435	6.2045	0.602257	0.119091

3.2 Graphs



3.3 Results

4 Question D

The base code is used with varying the number of ways involves increasing the size of the list with the tag values and varying index bits, offset bits and tag bits.

4.1 Results

Associativity	gcc.trace	gzip.trace	mcf.trace	swim.trace	twolf.trace
1-way	0.0616949	0.332945	0.989685	0.0737946	0.0125367
2-way	0.0616949	0.332945	0.989685	0.0737946	0.0125367
4-way	0.0616949	0.332945	0.989685	0.0737946	0.0125367
8-way	0.0616949	0.332945	0.989685	0.0737946	0.0125367
16-way	0.0616949	0.332945	0.989685	0.0737946	0.0125367
32-way	0.0616949	0.332945	0.989685	0.0737946	0.0125367
64-way	0.0616949	0.332945	0.989685	0.0737946	0.0125367

4.2 Graphs



5 Conclusion

5.1 Varying Cache Size from 128KB to 4096KB and Observing Hit/Miss Rates:

As you increase the cache size, the miss rate typically decreases because larger caches can hold more data, reducing the chance of a miss. The graph of miss rate vs. cache size should show a downward trend in miss rates as cache size increases, but the rate of improvement will diminish after a certain point due to the principle of diminishing returns. Different traces behave differently due to variations in memory access patterns. Traces with more temporal or spatial locality will see greater improvements with larger caches, while traces with random access patterns might not benefit as much.

5.2 Varying Block Size from 1 Byte to 128 Bytes:

Increasing block size reduces the number of cache lines because the total cache size remains fixed. Miss rate vs. block size graph: Initially, increasing the block size might improve performance by exploiting spatial locality, but after a certain point, larger blocks may increase the miss rate due to cache pollution, where irrelevant

data fills the cache. Different traces exhibit different behaviors depending on their memory access patterns. Some traces with strong spatial locality will benefit from larger blocks, while others with less predictable access patterns may see an increase in miss rate as block size grows.

5.3 Varying Associativity from 1-Way to 64-Way:

Hit rate vs. associativity: Increasing associativity generally improves the hit rate because it reduces conflict misses (i.e., when multiple blocks compete for the same set). However, beyond a certain level of associativity, the improvement diminishes as conflict misses become rare. Some traces behave differently because traces with frequent conflict misses will benefit more from increased associativity. Others with fewer conflict misses may not see much improvement as associativity increases.

5.4 What Happened:

Cache Size Variation: Larger cache sizes reduce the miss rate, but the benefit decreases after a certain point as the cache starts to hold all the relevant data. Block Size Variation: Larger block sizes initially help by leveraging spatial locality, but if too large, they can increase misses by bringing in unnecessary data. Associativity Variation: Higher associativity reduces conflict misses, but beyond a certain level, further improvement becomes minimal.