Computer Organization LAB6 Report

109550042 林律穎 109550168 林慧旻

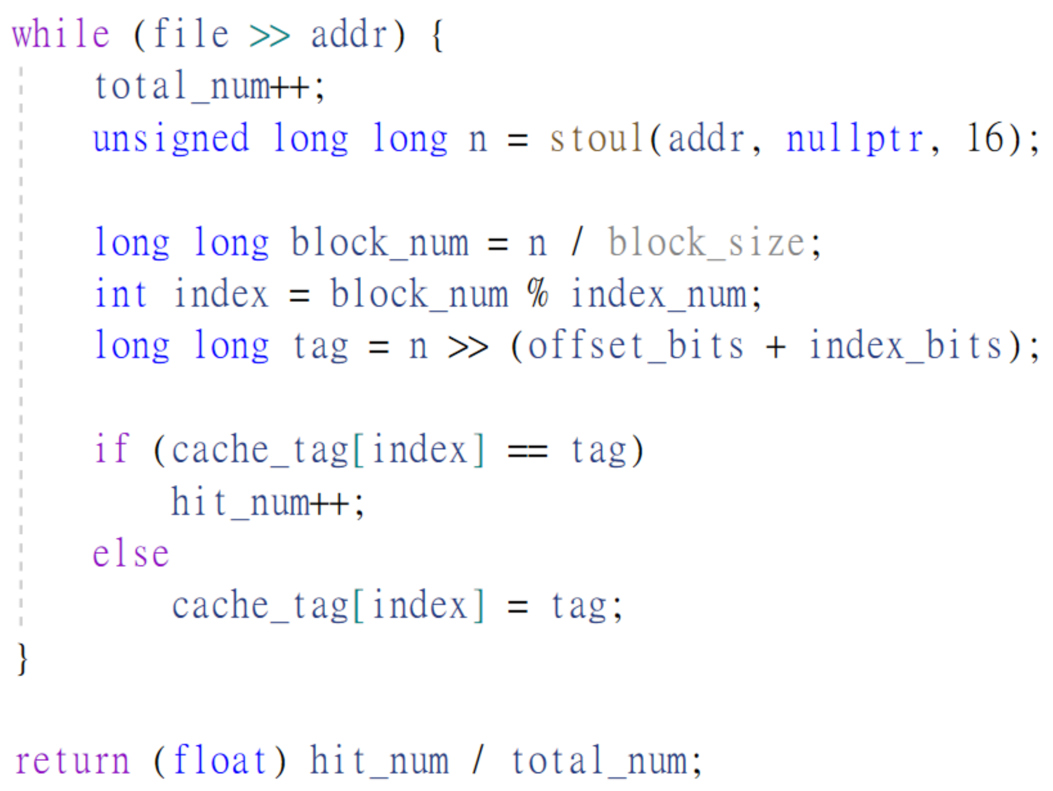
1. direct_mapped_cache.h, set_associative_cache.h

The two files are used to declare the functions.

2. direct_mapped_cache.ccp

```
ifstream file;
file.open(filename);

int index_num = cache_size / block_size;
int offset_bits = log2(block_size);
int index_bits = log2(index_num);

vector<long long> cache_tag(index_num, -1);

string addr;
```

In this file, firstly, open the input file and calculate the number of index, number of offset bits, and number of index bits. Then, create a vector for keeping the tags of cache.



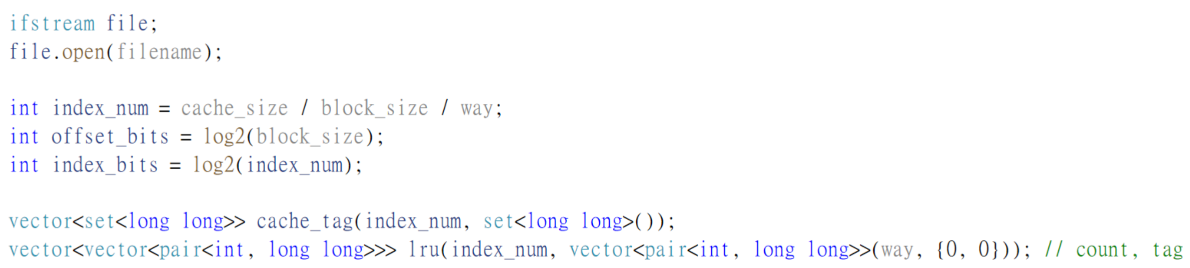
```
while (file >> addr) {
    total_num++;
    unsigned long long n = stoul(addr, nullptr, 16);

    long long block_num = n / block_size;
    int index = block_num % index_num;
    long long tag = n >> (offset_bits + index_bits);

    if (cache_tag[index] == tag)
        hit_num++;
    else
        cache_tag[index] = tag;
}

return (float) hit_num / total_num;
```

In the following loop, update the addresses of the file in each loop, and translate the input string address to a long long number, named "n". Compute the block number with n and number of index. Compute the index for with block number and number of index. Compute tag with "n" and offset bits and index bits.

Finally, go check if cache has the same value of tag by the index. If so, add I to hit number; else, replace the old tag in cache with the newly coming tag.

After dealing with all input data, calculate the hit rate.

3. set_associative_cache.



```
ifstream file;
file.open(filename);

int index_num = cache_size / block_size / way;
int offset_bits = log2(block_size);
int index_bits = log2(index_num);

vector<set<long long>> cache_tag(index_num, set<long long>());
vector<vector<pair<int, long long>>> lru(index_num, vector<pair<int, long long>>(way, {0, 0})); // count, tag
```

This part is the same as that in the direct_mapped_cache.ccp. The only difference is that there are two vector to keep cache tag and lru values. Remember to divide number of ways here.

```
while (file >> addr) {
    total_num++;
    unsigned long long n = stoul(addr, nullptr, 16);

    long long block_num = n / block_size;
    int index = block_num % index_num;
    long long tag = n >> (offset_bits + index_bits);
```

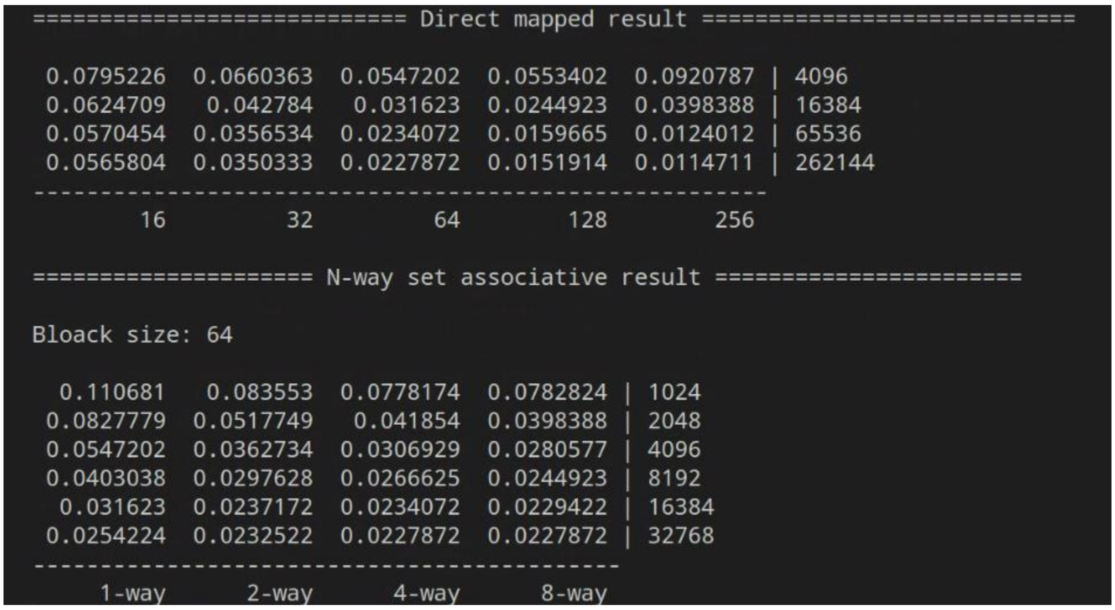
The same as that in the direct_mapped_cache.ccp. Draw the input data in each while loop and compute multiple values for latter usage.

```
    if (cache_tag[index].count(tag)) {
        hit_num++;
        for(int i = 0; i < way; i++)
            if(lru[index][i].second == tag)
                lru[index][i].first = total_num;
    }
    else {
        if(lru[index][0].first != 0){
            cache_tag[index].erase(lru[index][0].second);
            lru[index][0] = {0, 0};
        }
        cache_tag[index].insert(tag);
        lru[index][0] = {total_num, tag};
    }
    sort(lru[index].begin(), lru[index].end());

}

return (float) hit_num / total_num;
```

In here, it first needs to know if it hits in the cache. Is so, first add one to hit number . Second, let the lru[index][i].first where lru[index][i].second equals to tag keep the value of current total_num, which can be sorted later; else, go see if the set in that index is full or not. If it is full, then lru[index][0].first won’t be zero, so remove the tag with smallest second index, since that means it has been less used lately. Remember to set lru[index][0] = {0,0} so the space can be released for the next usage. If the set is not full, just insert the tag into the set. In each loop, sort the vector of lru.

Finally, calculate the hit rate.

4. Result



```
=========================== Direct mapped result ===========================

 0.0795226  0.0660363  0.0547202  0.0553402  0.0920787 | 4096
 0.0624709   0.042784   0.031623  0.0244923  0.0398388 | 16384
 0.0570454  0.0356534  0.0234072  0.0159665  0.0124012 | 65536
 0.0565804  0.0350333  0.0227872  0.0151914  0.0114711 | 262144
-----------------------------------------------------
        16         32         64        128        256

==================== N-way set associative result ======================

Bloack size: 64

  0.110681   0.083553  0.0778174  0.0782824 | 1024
 0.0827779  0.0517749   0.041854  0.0398388 | 2048
 0.0547202  0.0362734  0.0306929  0.0280577 | 4096
 0.0403038  0.0297628  0.0266625  0.0244923 | 8192
  0.031623  0.0237172  0.0234072  0.0229422 | 16384
 0.0254224  0.0232522  0.0227872  0.0227872 | 32768
-------------------------------------------
     1-way      2-way      4-way      8-way
```

5. Problem & Solution

The only problem we had in this homework is that don’t forget to divide the number of ways in the computation of index_num in set_associative_cache.