Lab06 : Cache Simulator

team 29, teammate ：109550053 楊立嘉，109550137 徐敏芝

direct_mapped

Implementation

```cpp
#include "direct_mapped_cache.h"
#include <math.h>
#include <fstream>
#include "string"
using namespace std;

struct cache_content {  // per row
    bool valid;
    unsigned int tag;
    // unsigned int data[16]; //不需要，因為只是要算 hit 數
};
double log2(double n) {
    return log(n) / log(2);
}
float direct_mapped(string filename, int block_size, int cache_size) {
    int total_num = 0;
    int hit_num = 0;

    //part1 : 先算那些+創建快取
    int offset_size = (int)round(log2(block_size));
    int index_size = (int)round(log2(cache_size / block_size));
    int row_num = cache_size / block_size;
    cache_content *cache = new cache_content[row_num];
    for (int j = 0; j < row_num; j++)
        cache[j].valid = false;

    //part2 : 一個一個位址讀進來丟進快取
    unsigned int address, tag, index;
    FILE *fp = fopen("testcase.txt", "r");
    while (fscanf(fp, "%x", &address) != EOF) {
        index = (address >> offset_size) & (row_num - 1);
        tag = address >> (index_size + offset_size);
        if (cache[index].valid && cache[index].tag == tag) {
            cache[index].valid = true;
            hit_num++;
            total_num++;
        } else {
            cache[index].valid = true;
            cache[index].tag = tag;
            total_num++;
        }
    }
    fclose(fp);
    delete[] cache;
    return (float)hit_num / total_num;
}
```

- Note that because I can not put variable in parameter of `fopen`, so I put `“testcase.txt”` directly, but not `filename`.
- First part : Using given block size and cache size to compute #of bit of index and offset, and create corresponding cache.

- Second part : Read in txt file one number by one number. For every number, calculate its index and test if it is in the cache. I represent index and tag in decimal, but calculate them in binary as following:
  - take address = 3214384416, 8 bit offset, 10 bits index, 1024 rows in cache for example, in binary, 3214384416 is 10111111100101 1110011101 00100000 (14bits/10bits/8bits)
  - `index = (address >> offset_size) & (row_num - 1);`
    `(address >> offset_size)` means remove offset from address (in binary get 10111111100101 1110011101) and `row_num -1` means the biggest number of index (in binary get 00000000000000 1111111111). So doing bitwise and on them will get index of address.
  - `tag = address >> (index_size + offset_size);`
    Just remove offset and index , then we can get tag.

result

cache size/block size	16	32	64	128	256
4k	0.0795226	0.660363	0.0547202	0.0553402	0.0920787
16k	0.0624709	0.042784	0.031623	0.0244923	0.0398388
32k	0.0570454	0.0356524	0.0234072	0.0159665	0.0124012
256k	0.0565804	0.0350333	0.0227872	0.0151914	0.0114711

- when block size increasing, miss rate decrease because of spatial locality. But once fix the cache size, if we increase block size unrestrained, miss rate will increase again.

set associative

Implementation

When reading the address, separate the result into 3 condition

(1) hit – find the same `tag`

add 1 to `hit_num` and `total_num`

update the `index_of_block` of the block (I didn't do this at first, so I got higher miss rate on 2/4/8 associative)

(2) miss but there is spare space in this set – if not hit and current block is valid

add 1 to `total_num`

store `tag` and `index_of_block`

(3) miss and need LRU

add 1 to `total_num`

find lru (which is INT_MAX at first)

– for each block in this set

– get their `index_of_block` , if it is bigger than lru, update lru

for the Least Recently Used block, update its `tag` and `index_of_block`

result

cache size/associativity	1-way	2-way	4-way	8-way
1k	0.110681	0.083553	0.0778174	0.0782824
2k	0.0827779	0.0517749	0.041854	0.0398388
4k	0.0547202	0.0362734	0.0306929	0.0280577
8k	0.0403038	0.0297628	0.0266625	0.0244923
16k	0.031623	0.0237172	0.0234072	0.0229422
32k	0.0254224	0.0232522	0.0227872	0.0227872

miss rate decrease when associativity increase

miss rate decrease when cache size decrease