**Report-Lab06**

**Name:杜峯　　ID:109550096**

**Name:林念慈　　ID:109550130**

## 1. Direct-mapped cache

```
8 > inline int simple_hash(char in){···
58
59 > int string_to_int(string in){···
```

At first, I have defined two function. The first one is to map the char into integer. For example, '1' to 1 and 'f' to 15. The second one is to transform the string which is composed by '0' and '1' into integer. These two functions will be used in later.

```
75      cache_size/=block_size;
76      int table_size=cache_size;
77      int block_num=-1;
78      int cache_num=-1;
79      while(block_size!=0){
80          block_size/=2;
81          block_num++;
82      }
83      while(cache_size!=0){
84          cache_size/=2;
85          cache_num++;
86      }
87      int tag_num=(32-cache_num-block_num);
88
89      int *table=new int[table_size];
90
91      ifstream file_input;
92      file_input.open(filename);
93
94      string input;
95      bitset<4> in;
96      bitset<32> address;
```

In the function that we need to do, I first calculated the number of total blocks in line 75, and stored this value as "table_size". Then I calculated the number of bits of the index and the offset which are stored as "block_num" and "cache_num". In the end, I calculated the number of bits of tag.

As we knew the size of blocks, there was a table whose size was "table_size". It was used to store the cache.

At line 91 and 92, it just the file I/O.

Then, there were three variables, "input\in\address", whose types were "string\bitset of 4-bits\bitset of 32-bits".

```
98      while(file_input>>input&&input.size()!=0){
99          int counting=31;
100         if(input.size()==7){
101             while(counting!=27){
102                 address[counting]=0;
103                 counting--;
104             }
105         }
106         for(char*itt=&input[0];itt<=&input[input.size()-1];itt++){
107             in=simple_hash(*itt);
108             for(int i=3;i>=0;i--){
109                 address[counting]=in[i];
110                 counting--;
111             }
112         }
113
114         string address_str=address.to_string();
115
116         string tag,ind;
117         int i=0;
118         for(;i<tag_num;i++){
119             tag+=address_str[i];
120         }
121         for(;i<=tag_num+cache_num-1;i++){
122             ind+=address_str[i];
123         }
```

In the while loops, we got one address and stored in the variable "input" for each loop and check if its size was zero. If so, break the loops, which meant we finished reading the file.

If the input size was equal to seven, we needed to pad the address by 4 bits of '0' in the variable "address".

Then, we transform the char we read in the "input" to the bitset with using the function "simple_hash" and push the result into the address.

Since we have already calculated the number of bits of tag and index, we just copy the bits to the new variable "tag/ind".

(next page)

```
125         int tag_tag=string_to_int(tag);
126         int ind_ind=string_to_int(ind);
127
128         if(table[ind_ind]==-1){
129             table[ind_ind]=tag_tag;
130         }
131         else{
132             if(table[ind_ind]==tag_tag){
133                 hit_num++;
134             }
135             else{
136                 table[ind_ind]=tag_tag;
137             }
138         }
139         total_num++;
140     }
```

At the most important step, we transform the varibles “tag/ind” from bitset in to integer and go to the mapping stage.

If the value of “table[ind_ind]” is -1, which means there was nothing, we push the value “tag_tag” into the place of “table[ind_ind]. If there has already been something, we needed to check whether it was equal to “tag_tag”. If so, it was “hit”; otherwise, the value would be replace by “tag_tag”.

## 2. Set-associative cache

```
 9 > inline int simplehash(char in){ ···
59
60 > int stringtoint(string in){ ···
```

These two functions were same as above.

```
76      cache_size/=(block_size*way);
77      int table_size=cache_size;
78      int block_num=-1;
79      int cache_num=-1;
80      while(block_size!=0){
81          block_size/=2;
82          block_num++;
83      }
84      while(cache_size!=0){
85          cache_size/=2;
86          cache_num++;
87      }
88      int tag_num=(32-cache_num-block_num);
89
90      vector<int>*table=new vector<int>[table_size];
91
92      ifstream file_input;
93      file_input.open(filename);
94
95      string input;
96      bitset<4> in;
97      bitset<32> address;
```

There were some different with the step in the direct-mapped cache.

The “cache_size” not only divided by the “block_size” but also the “way”. The result will be our “table_size”.

And the definition of the table was also different. In the direct-mapped cache, we just used the integer arr. But in here, we used the vector<int> array, that is we might store more than one value in each index.

Others code were same as direct-mapped cache.

```
99      while(file_input>>input&&input.size()!=0){
100         int counting=31;
101         if(input.size()==7){
102             while(counting!=27){
103                 address[counting]=0;
104                 counting--;
105             }
106         }
107         for(char*itt=&input[0];itt<=&input[input.size()-1];itt++){
108             in=simplehash(*itt);
109             for(int i=3;i>=0;i--){
110                 address[counting]=in[i];
111                 counting--;
112             }
113         }
114
115         string address_str=address.to_string();
116
117         string tag,ind;
118         int i=0;
119         for(;i<tag_num;i++){
120             tag+=address_str[i];
121         }
122         for(;i<=tag_num+cache_num-1;i++){
123             ind+=address_str[i];
124         }
```

In the first part of the while loop. There were all the same as direct-mapped cache.

```
126         int tag_tag=stringtoint(tag);
127         int ind_ind=stringtoint(ind);
128
129         int finding=0;
130         bool flag=false;
131         if(table[ind_ind].size()!=0){
132             for(finding;finding<=table[ind_ind].size()-1;finding++){
133                 if(table[ind_ind][finding]==tag_tag){
134                     flag=true;
135                     break;
136                 }
137             }
138         }
139
140         if(flag){
141             hit_num++;
142             table[ind_ind].erase(table[ind_ind].begin()+finding);
143             table[ind_ind].push_back(tag_tag);
144         }
145         else{
146             if(table[ind_ind].size()==way){
147                 table[ind_ind].erase(table[ind_ind].begin());
148             }
149             table[ind_ind].push_back(tag_tag);
150         }
151         total_num++;
152     }
```

In the second part of while loop, it was different. As we might have many (up to the number of “way”) value in a single index, we needed to find if the value we want “tag_tag” was in the vector. And record its index.

This idea is like a queue, but there is something different. If the value is we use recently, it would be the back of the vector. In the other words, if this value was least recently used (LRU), it would be the front of the vector. So, if we find the value in the vector, we would erase the original place and push back to the vector again which means it was the recently used value. If we didn’t find it, we just push the value back to the vector. However, in the case that the vector was full, which means the size of vector was equal to the “way”, we would pop the first item in the vector to delete the LRU item.

3. Results

```
dufeng@dufeng: /mnt/d/大二
dufeng@dufeng:/mnt/d/大二下/計算機組織/Lab06$ ./demo.sh
rm -f *.o
g++ -I./include main.cpp direct_mapped_cache.cpp set_associative_cache.cpp -o main.o
============================ 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
dufeng@dufeng:/mnt/d/大二下/計算機組織/Lab06$
```

4. Some improvements and problems

We may skip the step of “bitset” and transform the char in to a string composed by ‘0’ and ‘1’.