Project 2: Understanding Cache Memories

Yanjie Ze, 519021910706
zeyanjie@sjtu.edu.cn

June 1, 2021

1 Introduction

The project: **Understanding Cache Memories** is the second project for CS359: Computer Architecture. By finishing this project, we will understand cache memories from a higher perspective. The project consists of two parts:

- **Part A: Writing A Cache Simulator**

  In this part, we will finish a C program to simulate the cache's behaviours and count the number of **cache hits**, **cache misses**, **evictions**. We take the *valgrind* memory trace as the input of this simulator, to observe the actions of the cache.

  This part aims to get us better and deeper understand the cache.

- **Part B: Optimizing Matrix Transpose**

  In this part, we focus on how to optimize the performance of **Matrix Transpose**, which is to store the transpose of matrix A into matrix B. To optimize the function, we need to make the cache misses as few as possible and have a full comprehension in the cache miss.

  This part aims to teach us the methods of optimizing a program by attaching special attention on the cache.

2 Experiments

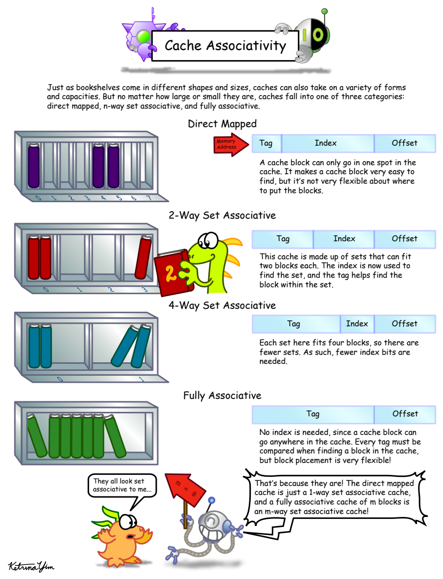
2.1 Part A

2.1.1 Analysis

We are required to implement a cache simulator in Part A. Let's figure out how to finish this part by the following analysis.

Cache Placement: The most important thing for us is to know how the cache works before constructing the frame of codes. In our class, we have learned about three block placement way: **fully associative, direct mapped, set associative**, which are very similar essentially. By showing Fig.1 we know that they are actually all set associative, different in the set size. [1].

As shown in Fig. 1, direct-mapped is 1-way set associative and fully associative is m-way set associative, where m is the number of blocks. Thus, what we care about is how to implement **set associative**.



Just as bookshelves come in different shapes and sizes, caches can also take on a variety of forms and capacities. But no matter how large or small they are, caches fall into one of three categories: direct mapped, n-way set associative, and fully associative.

A cache block can only go in one spot in the cache. It makes a cache block very easy to find, but it's not very flexible about where to put the blocks.

This cache is made up of sets that can fit two blocks each. The index is now used to find the set, and the tag helps find the block within the set.

Each set here fits four blocks, so there are fewer sets. As such, fewer index bits are needed.

No index is needed, since a cache block can go anywhere in the cache. Every tag must be compared when finding a block in the cache, but block placement is very flexible!

They all look set associative to me...

That's because they are! The direct mapped cache is just a 1-way set associative cache, and a fully associative cache of m blocks is an m-way set associative cache!

Figure 1: Illustration of Cache Associativity[1]

Simulation: To know in detail how to simulate the cache, let's take a look at **trace files** generated in the form same as **valgrind**:

```
L 10,1
M 20,1
L 22,1
S 18,1
L 110,1
L 210,1
M 12,1
```

In this trace file, there are 3 instructions, and each of them has the form like: `Operation, Address, Size`:

- **L(load)**. Load *size* bytes from *address*.
- **M(modify)**. Load *size* bytes from *address*, modify them, and store them back into *address*.(load + store)

- **S(store)**. Store *size* bytes into *address*.

Thus, when we are parsing each operation, it's very important to consider the **M** operation may lead to hit/miss twice.

LRU: When a miss happens, we use **LRU** to replace the blocks, whose full name is **Least Recently Used**. LRU algorithm is shown in Alg. 1. In Alg. 1, $b_i.t$ refers to the recently used time of the block i.

Algorithm 1: LRU Replacement

```
1 Input: Blocks $B = \{b_1, b_2, ..., b_n\}$
2 Output: Replaced Block $b$
3 Define $t_{min} = \infty$;
4 Define $idx = 0$;
5 for each $b_i$ in $B$ do
6     if $b_i.t < t_{min}$ then
7         $t_{min} = b_i.t$;
8         $idx = i$;
9 return $b_{idx}$
```

2.1.2 Structure

In this section, we briefly show the structure of Cache Simulator, which mainly consists of three parts: **parsing, cache initialization, cache updating, cache free**, as shown in Fig. 2.

In the next section, we will give the detailed implementation.

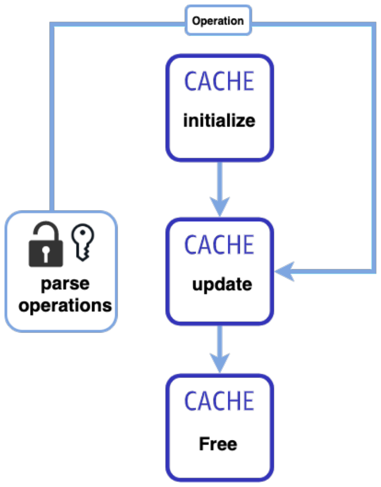


Figure 2: Structure of Cache Simulator

2.1.3 Code

In this section, we show the specific implementation of our simulator.

Basically, we define a data structure called **cache line**, the unit of the cache:

```c
/* basic data structure for caches*/
typedef struct CACHE_LINE
{
    int valid_bit;
    int tag;
    int time;
}cache_line;
```

Then our cache is made up of the cache lines:

```c
cache_line **cache;
```

And the initialization of the cache is shown below. We initialize each elements in a cache line.

```c
void cache_init()
{
    cache = (cache_line**)malloc(sizeof(cache_line*) * S);
    for(int i=0; i<S; ++i)
    {
        cache[i] = (cache_line*)malloc(sizeof(cache_line) * E);
        for(int j=0; j<E; ++j)
        {
            cache[i][j].tag = -1;
            cache[i][j].time = -1;
            cache[i][j].valid_bit = 0;// set to invalid
        }
    }
}
```

First, we need to parse the operations read from the trace files. There are three operations as introduced before: **L,M,S**. Note that the order we place them: $L \rightarrow M \rightarrow S$. This is because Modify Operation needs to access the cache twice and there's no `break` after `case 'M'`. The function `operation_parse` is shown below.

```c
void operation_parse()
{
    char operation;
    unsigned int address;
    int size;
    char comma;

    while (fscanf(file_pointer, " %c %xu%c%xu", &operation, &address,&
    comma, &size)>0)
    {
        switch (operation)
        {
        case 'L': // Load Data
            cache_update(address);
            break;
        case 'M':// Modify Data
            cache_update(address);// Modiy also need Store

        case 'S': // Store Data
            cache_update(address);
            break;

        default:
            break;
        }
        time_update();
    }

}
```

Second, we provide the implementation of `cache_update`, which takes the address as the input and access the corresponding cache line in the cache. The process of this functio is:

1. Parse the address into `tag_idx` and `set_idx`.
2. Search in the set for the existence.
   (a) If hit, update and return.
   (b) If not hit, go to next step.
3. Search in the set for the empty space.
   (a) If found, update and return.
   (b) If not found, go to next step.
4. Use **LRU** to replace the block. Update and return.

Thus, the code below implements this process.

```c
 void cache_update(unsigned int address)
{
    int tag_idx = (address>>(num_b+num_s));
    int set_idx = (address>>num_b) & ((0xFFFFFFFF)>>(64-num_s));

    for(int i=0; i<E; ++i)// search in set
    {   // if hit
        if((cache[set_idx][i].tag == tag_idx) && cache[set_idx][i].
    valid_bit==1)
        {
            hits ++;// hit
            cache[set_idx][i].time = current_time;
            return;
        }
    }

    for(int i=0; i<E; ++i)// not hit
    {
        if(cache[set_idx][i].valid_bit == 0)//  invalid, means empty
        {
            cache[set_idx][i].tag = tag_idx;
            cache[set_idx][i].time = current_time;
            cache[set_idx][i].valid_bit = 1;
            misses ++;// miss
            return;
        }
    }

    // not hit and not empty, need eviction
    evictions ++;
    misses ++;
    // We use LRU here
    int min_time = 10000;
    int min_idx;
    for(int i=0; i<E; ++i)
    {
        if(cache[set_idx][i].time<min_time)
        {
            min_time = cache[set_idx][i].time;
            min_idx = i;
        }
    }
    cache[set_idx][min_idx].tag = tag_idx;
    cache[set_idx][min_idx].time = current_time;
}
```

Third, we provide the implementation of `time_update`, which is used to update the current time simply, and `cache_free`, which is to free the cache after all the operations.

```
/*update current time*/
 void time_update()
{
    current_time ++;
}

/* free the cache */
void cache_free()
{
    for(int i=0; i<S; ++i)
        free(cache[i]);
    free(cache);
}
```

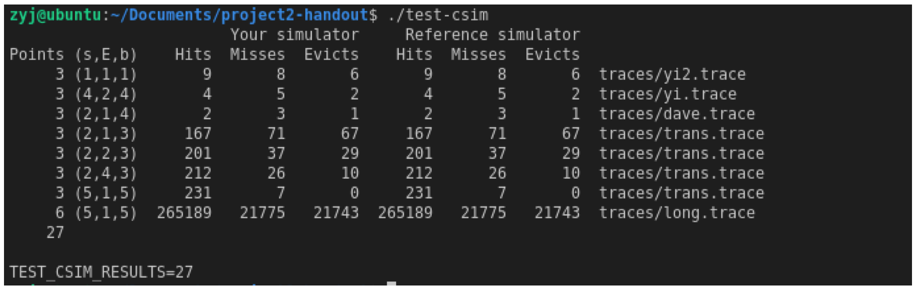
Finally, we give our `main` function, which parse the command line parameters and call functions we create before:

```
int main(int argc, char*argv[])
{
    int command_param; // get the command from getopt()

    num_b = -1;
    num_E = -1;
```

```c
    num_s = -1;

    // parse command
    while ((command_param = getopt(argc,argv,"s:E:b:t:hv"))!= -1)
    {
        switch (command_param)
        {
        case 'h':
            print_help();
            break;

        case 's':
            num_s = atoi(optarg);
            break;

        case 'E':
            num_E = atoi(optarg);
            break;

        case 'b':
            num_b = atoi(optarg);
            break;

        case 't':
            strcpy(tgt_file, optarg);
            break;

        default:
            printf("./csim: Wrong required command line argument\n");
            print_help();
            break;
        }
    }

    // Basic Check
    if(num_b<= 0 || num_E<=0 || num_s<=0)
    {
        return -1;
    }
    file_pointer = fopen(tgt_file, "r");
    if(file_pointer==NULL)
    {
        printf("File Error: Can't Open File.\n");
        return -1;
    }


    hits = 0;
    misses = 0;
    evictions = 0;

    S = 1<<num_s;
    B = 1<<num_b;
    E = num_E;

    current_time = 0;
    cache_init();
    operation_parse();
    cache_free();
    fclose(file_pointer);
    printSummary(hits, misses, evictions);

    return 0;
}
```

2.1.4 Evaluation

In Fig. 3, we show the results of our program, autograded by **test-csim**. Thus, we successfully build our Cache Simulator.



```
zyj@ubuntu:~/Documents/project2-handout$ ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27

TEST_CSIM_RESULTS=27
```

Figure 3: Evaluation Results

2.2 Techniques in Part B

This section is used to introduce several tricks we apply in Part B. We first introduce them here and show how we apply them in the next section.

2.2.1 Blocking

Blocking is a technique introduced in [2], which utilizes large chunks called `blocks` to improve the temporal locality of inner loops.

For example, to calculate a matrix multiplication, i.e. $C = AB$, we can partition A and C into $1 \times bsize$ row silvers and B into $bsize \times bsize$ blocks, as shown in Fig. 4.

In order to improving the performance of our function **Matrix Transpose**, we also apply this technique.

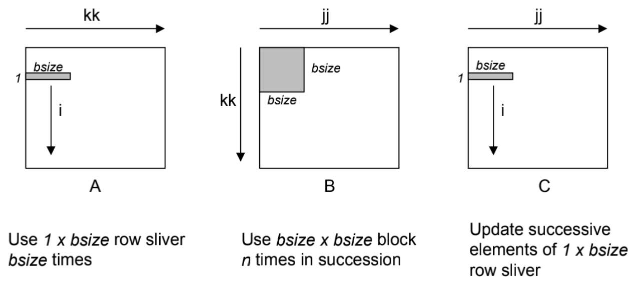

kk
jj
jj
bsize
1
i
kk
bsize
bsize
bsize
1
i
A
B
C
Use *1 x bsize* row sliver *bsize* times
Use *bsize x bsize* block *n* times in succession
Update successive elements of *1 x bsize* row sliver

Figure 4: Illustration of Matrix Multiplication using Blocking[2]

2.2.2 Read All Only Once

We propose a trick called **Read All Only Once**, to further improve the performance.

Read All Only Once is a trick that when doing blocking, we read the full line of the block.

Then we explain why this trick works.

For example, the matrix A has the size 32×32. When CPU reads $A[0][0]$, one cache line is filled with $A[0][0], A[0][1], ..., A[0][7]$. If we just read one element then transpose the element, the cache line is replaced with $B[0][0], ..., B[0][7]$. Thus, there are extra elements that are not read but placed in the cache. If we not use them in this time, they are still in requirement. We can read these elements at one time, which is called **Read All Only Once**.

2.2.3 Save and Load

We propose a trick called **Save and Load**, to make the use of the data which has been read into the cache more efficiently.

We use an example of how we apply **Save and Load** to illustrate why we propose this trick and how this works.

For example, the matrix A has the size 64×64.

Thus, the cache can store 4 lines of the matrix ($64 \times 4 = 32 \times 8$). We need to refresh the cache every 4 lines.

On such a condition, if we perform *blocking* of size 8, as shown in Fig. 5, the cache can only be filled with $line1$ to $line4$ or $line5$ to $line8$. A natural idea is to use **Read All Only Once**, but it can not work since the number of variables is greater than 8.

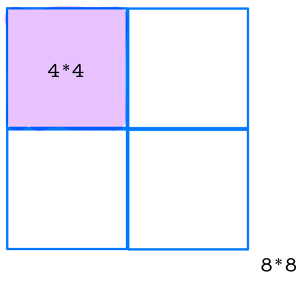


Figure 5: Illustration of Save and Load in 8 Blocking

To solve this problem, we first read 4 elements in A's lower left quarter and store the transposition of them in B's lower left quarter. This is called *Save*. Then, when transposing A's upper right corner, we first *load* the elements in B we store before and do the transposition of the upper right corner and the lower left corner of B. This trick is named as **Save and Load**.

2.3 Part B

2.3.1 Analysis

In Part B, we are required to optimize the performance of the function `Matrix Transpose`, to make the cache misses as few as possible. `Matrix Transpose` stores the transposition of matrix $A_{n\times m}$ into matrix $B_{m\times n}$.

In `test-trans`, the cache has the following parameters:

$$s = 5, E = 1, b = 5$$

Which means the number of sets is $2^5 = 32$, the number of line per set is 1, and the block size is $2^5 = 32$. Thus, the cache size is:

$$cache\ size = \#sets \times \#line\ per\ set \times block\ size = 2^{10} byte = 1kB$$

$$set\ size = \#line\ per\ set \times block\ size = 32byte$$

One integer demands $4byte$. Thus one cache set can store 8 integers at a time, and one cache can store 32×8 integers at a time.

When we consider how to optimize the performance, we merely consider three different sizes of the matrix: 32×32, 64×64, 61×67. Different sizes have different features and we utilize the techniques we propose before to optimize them one by one.

2.3.2 Code

In this section, we give the implementation of the function with the best performance for three sizes separately. Some other trys will be shown in the next section, in compared with the best functions.

First, for $A_{32\times32}$, we apply **Blocking 8** (we use the size as the suffix) and **Read All Only Once**, which has the performance of **hits:1766, misses:287, evictions:255**, reaching the full score.

```c
char blocking8_readall[] = "Blocking 8 Read All Only Once transpose";
void transpose_blocking8_readall(int M, int N, int A[N][M], int B[M][N
    ])
{   // using 3 + 8 = 11 variable
    int t0,t1,t2,t3,t4,t5,t6,t7;
    for(int i=0; i<N; i+=8)
        for(int j=0; j<M; j+=8)
            for(int k=i; k<i+8; k++)
            {
                // read all only once
                t0 = A[k][j];
                t1 = A[k][j+1];
                t2 = A[k][j+2];
                t3 = A[k][j+3];
                t4 = A[k][j+4];
                t5 = A[k][j+5];
                t6 = A[k][j+6];
                t7 = A[k][j+7];

                // update
                B[j][k] = t0;
                B[j+1][k] = t1;
                B[j+2][k] = t2;
                B[j+3][k] = t3;
                B[j+4][k] = t4;
                B[j+5][k] = t5;
                B[j+6][k] = t6;
                B[j+7][k] = t7;
            }
}
```

Second, for $A_{64\times64}$, we apply **Blocking 8+Read All Only Once+Save and Load**, which has the performance of **hits:9066, misses:1179, evictions:1147**, reaches the full score.

```c
char blocking8_readall_improve[] = "Blocking 8 Read All Only Once
    Improved transpose";
void transpose_blocking8_readall_improve(int M, int N, int A[N][M], int
     B[M][N])
{   // using 4 + 8 = 12 variable
    int t,t1,t2,t3,t4,t5,t6,t7,t8;
    for(int i=0; i<N; i+=8)
        for(int j=0; j<M; j+=8)
         {      // save and load
                for (t = i; t < i + 4; ++t)
        {
        // read all only once
          t1 = A[t][j]; t2 = A[t][j+1]; t3 = A[t][j+2]; t4 = A[t][j+3];
          t5 = A[t][j+4]; t6 = A[t][j+5]; t7 = A[t][j+6]; t8 = A[t][j+7
    ];

```

```c
          B[j][t] = t1; B[j+1][t] = t2; B[j+2][t] = t3; B[j+3][t] = t4;
          B[j][t+4] = t5; B[j+1][t+4] = t6; B[j+2][t+4] = t7; B[j+3][t+4] = t8;
        }
        for (t = j; t < j + 4; ++t)
        {
          t1 = A[i+4][t]; t2 = A[i+5][t]; t3 = A[i+6][t]; t4 = A[i+7][t];
          t5 = B[t][i+4]; t6 = B[t][i+5]; t7 = B[t][i+6]; t8 = B[t][i+7];

          B[t][i+4] = t1; B[t][i+5] = t2; B[t][i+6] = t3; B[t][i+7] = t4;
          B[t+4][i] = t5; B[t+4][i+1] = t6; B[t+4][i+2] = t7; B[t+4][i+3] = t8;
        }
        for (t = i + 4; t < i + 8; ++t)
        {
          t1 = A[t][j+4]; t2 = A[t][j+5]; t3 = A[t][j+6]; t4 = A[t][j+7];
          B[j+4][t] = t1; B[j+5][t] = t2; B[j+6][t] = t3; B[j+7][t] = t4;
        }
         }
}
```

Third, for $A_{61\times 67}$, we apply **Blocking 16+Read All Only Once**. Since the shape of the matrix is not a square, we need to first transpose the square part then transpose the remaining part, which is called *Compensate*.This function has the performance of **hits:6264, misses:1993, evictions:1961**, reaching the full score.

```c

char blocking16_readall_shape[] = "Blocking 16 Read All Random Shape transpose";
void transpose_blocking16_readall_shape(int M, int N, int A[N][M], int B[M][N])
{
    int t0,t1,t2,t3,t4,t5,t6,t7;

    int i,j;
    for(i=0; i<N-N%16; i+=16)
        for(j=0; j<M-M%16; j+=16)
            for(int k=i; k<i+16; k++)
            {
                // read all only once
                t0 = A[k][j];
                t1 = A[k][j+1];
                t2 = A[k][j+2];
                t3 = A[k][j+3];
                t4 = A[k][j+4];
                t5 = A[k][j+5];
                t6 = A[k][j+6];
                t7 = A[k][j+7];

                // update
                B[j][k] = t0;
                B[j+1][k] = t1;
                B[j+2][k] = t2;
                B[j+3][k] = t3;
                B[j+4][k] = t4;
                B[j+5][k] = t5;
                B[j+6][k] = t6;
                B[j+7][k] = t7;

                // read all only once
                t0 = A[k][j+8];
                t1 = A[k][j+9];
                t2 = A[k][j+10];
                t3 = A[k][j+11];
                t4 = A[k][j+12];
                t5 = A[k][j+13];
                t6 = A[k][j+14];
                t7 = A[k][j+15];

                // update
                B[j+8][k] = t0;
                B[j+9][k] = t1;
                B[j+10][k] = t2;
                B[j+11][k] = t3;
                B[j+12][k] = t4;
                B[j+13][k] = t5;
                B[j+14][k] = t6;
                B[j+15][k] = t7;
            }

     // compenstate
    for (i = 0; i < N; ++i)
    for (j = M-M%16; j < M; ++j)
      {
        t0 = A[i][j];
        B[j][i] = t0;
      }
  for (i = N-N%16; i < N; ++i)
    for (j = 0; j < M; ++j)
      {
        t0 = A[i][j];
        B[j][i] = t0;
      }
}
```

2.3.3 Evaluation

In this section, we compare the results of the experiments we do with different techniques applied in different size. Note that Blocking is short for **B**, Read All Only Once is short for **RAOO** and Save and Load is short for **LS**.

First, for matrix $A_{32\times32}$, we test different blockings and RAOO, shown in table. 1.

technique	blocking 4	blocking 8	blocking 16	blocking 8+RAOO
hit	1566	1710	870	1766
miss	487	343	1183	287
eviction	455	311	1151	255

Table 1: Results for $A_{32\times 32}$

Second, for matrix $A_{64\times 64}$, we test different blockings ,RAOO and LS, as shown in table. 2

technique	b4	b 8	b 16	b 4+RAOO	b 8+RAOO	b 8+RAOO+LS
hit	6306	3474	3474	6498	3586	9066
miss	1891	4723	4723	1699	4611	1179
eviction	1859	4691	4691	1667	4579	1147

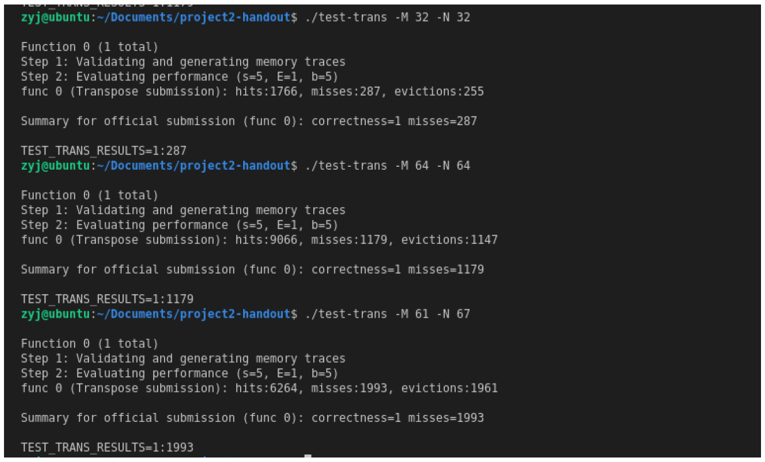
Table 2: Results for $A_{64\times 64}$

Third, for matrix $A_{61\times 67}$, we test different blockings and RAOO, as shown in table. 3

technique	b 15	b 16	b 17	b 8+RAOO	b 16+RAOO
hit	5922	6194	5902	7888	6264
miss	2271	2063	2597	2113	1993
eviction	2239	2031	2565	2081	1961

Table 3: Results for $A_{61\times 67}$

Finally we show the results of our program with the best performance, as shown in Fig. 6.



```
zyj@ubuntu:~/Documents/project2-handout$ ./test-trans -M 32 -N 32

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255

Summary for official submission (func 0): correctness=1 misses=287

TEST_TRANS_RESULTS=1:287
zyj@ubuntu:~/Documents/project2-handout$ ./test-trans -M 64 -N 64

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:9066, misses:1179, evictions:1147

Summary for official submission (func 0): correctness=1 misses=1179

TEST_TRANS_RESULTS=1:1179
zyj@ubuntu:~/Documents/project2-handout$ ./test-trans -M 61 -N 67

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6264, misses:1993, evictions:1961

Summary for official submission (func 0): correctness=1 misses=1993

TEST_TRANS_RESULTS=1:1993
```

Figure 6: Running Results for 3 sizes

3 Conclusion

3.1 Problems

In Project 2 **Understanding Cache Memories**, we encounter many difficulties and problems and solve them in the end, and we list the problems below:

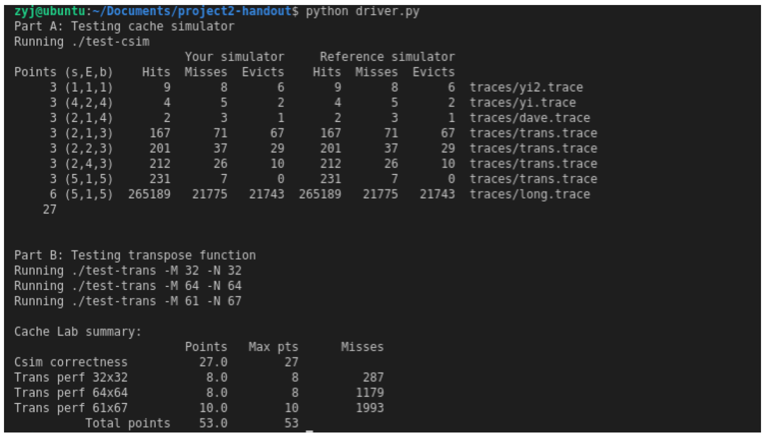
- In Part A, building the structure of the cache simulator is somewhat difficult in the beginning, and it takes some effort to further understand the cache and construct the code structure.
- In Part B, the first problem is that why **Blocking** technique works. After we dig into this technique and understand the principle of **Blocking**, we actually know how to optimize the function further.
- In Part B, when we are optimizing $A_{64\times64}$, it's difficult to get the full score, and after many experiments we propose **LS**(load and use) to improve the performance further.

3.2 Achievements

Our achievements in this project include:

- In Part A, we successfully simulates the cache by constructing a cache simulator which works the same as **csim-ref**. What's more, we visualize the structure of our codes.
- In Part B, we propose three techniques: **Blocking, Read All Only Once, Save and Load** and introduce the principles of them in detail.
- In Part B, we achieve the full score performance on all matrix sizes: 32×32, 64×64, 61×67. And we do a number of experiments to compare different techniques.
- In both Part A and Part B, we write the codes with readability, which means our codes are explained very specifically.

The total evaluation of two parts is shown below, using `driver.py`, which shows our simulator and the optimized function both work well.



```
zyj@ubuntu:~/Documents/project2-handout$ python driver.py
Part A: Testing cache simulator
Running ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27


Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           8.0         8        1179
Trans perf 61x67          10.0        10        1993
          Total points    53.0        53
```

Figure 7: Total Evaluation for Part A and Part B

4 Acknowledgements

- Thanks for the teacher of CS359, Prof.Yanyan Shen to teach us the basic knowledge of the cache and give us the chance to do this interesting project.
- Thanks for Bryant and other writers of CS:APP, who provide the material about Blocking and the basic design of the challenging lab.
- Thanks for my roommate David Guo to give some hints and guidance to understand the cache well.

References

[1] Cache associativity. http://csillustrated.berkeley.edu/PDFs/handouts/cache-3-associativity-handout.pdf.

[2] R. E.Bryant and D. R. O. Hallaron. Cs:app2e web aside mem:blocking: Using blocking to increase temporal locality, 2012.