Project 2: Understanding Cache Memories

518021910609 Yifan Liu 2602843473@qq.com

1. Introduction

This project is aimed to help us understand the impact that cache memories can have on the performance of our C programs. It contains two parts. In part A, we will implement a cache simulator that simulates the behavior of a cache memory. In part B, we will optimize a small matrix transpose function, with the goal of minimizing the number of cache misses in some particular size of matrices.

In part A, I implement a C program that simulates the behavior of cache memory. It could correctly detect the number of misses, hits and evictions. In part B, I implement a small matrix transpose function that could decrease the misses by changing the logic when transposing.

2. Experiments

2.1 Part A

2.1.1 Analysis

In Part A I write a cache simulator in ***csim.c*** that takes a ***valgrind*** memory trace as input, simulates the behavior of a cache memory, and outputs the total number of hits, misses, and evictions.

I define a struct ***Scache***, which contains variables that represent the number of sets, the number of lines and many sets. In a set, there are many lines,. In a line, there are three variables: valid bit, LRU number and tag number. The structure of ***Scache*** is similar to the real cache.

In ***main*** function, the program calls ***get_Opt*** to get variables from command line, then calls ***init_cache*** to initialize the cache and open the file. Then it continuously reads the line in the trace file to get the tag and set of the current data by calling ***get_tag*** and ***get_set*** and execute the corresponding load, store or modify operation. At last, it call ***printSummary*** to output the total number of hits, misses and evictions.

In ***get_Opt*** function, the program determines the parameters by checking the values after s, E, b, and t and store them to corresponding variables. It uses ***getopt*** function and ***optarg*** to get these values. I also define a check function ***checkOptarg*** to check whether these values are valid.

In ***init_cache*** function, it first checks the exceptions to avoid some errors. Then the function uses variables ***s*** and ***E*** to set the number of sets and lines. After that, it calls ***malloc*** function to initial each set and each line.

When initializing each line, it sets the LRU number and valid bit to 0.

Since the cache behavior of store operation is just like load operation, and modify operation is actually one load operation and one store operation, so we just need to implement load operation ***load_op***.

In ***load_op*** function, it calls ***miss_checking*** function to check whether there is a miss in current operation. If not, the number of hits plus 1. If so, the number of misses plus 1. Then it calls ***cache_full*** function to check whether the cache is full and need to do eviction. If so, the number of evictions plus 1.

In ***miss_checking*** function, it locates the target line in the set by ***tag*** and check the valid bit. If the valid bit is equal to 1, it means that there are some data in this address. So there does not exist a miss of this operation. Then, it needs to call ***update_LRU*** to update LRU number of this address. If we could not find the target data, it means that there is a miss and return 1.

In ***cache_full*** function, it checks whether all valid bits are 1. If so, it means that we need to replace some data with new one. Therefore, it find the smallest LRU number by calling ***find_smallest_LRU***. After that, it updates the valid bit and tag of the line that has the smallest LRU number and calls ***update_LRU*** to update LRU number. If the cache is not full, we simply update valid bit, tag and LRU number of one empty line.

In ***find_smallest_LRU*** function, it traverses the whole set to find the line that has the smallest LRU number. This line is just the one needs to be replaced.

In ***update_LRU*** function, it sets the LRU number of current line to the maximum number (I set it to 1000) and "-1" with LRU number of other lines.

2.1.2 Code

```c
// Name: Yifan Liu
// ID:   518021910609
#include <getopt.h>
#include <stdlib.h>
#include <unistd.h>
#include "cachelab.h"
#include <stdio.h>
#include <strings.h>
#include <string.h>

#define LRUMAX 1000
```

```c

typedef struct{
    int valid;//valid bit
    int LRU;//lru number
    int tag;//tag number
} Line;//the struct of one line in cache

typedef struct{
    Line* lines;//many lines in a set
} Set;//the struct of one set in cache

typedef struct{
    int set_cnt;  //the number of sets
    int line_cnt; //the number of lines
    Set* sets;     //many sets
} Scache;//struct of simulate cache


//output variables: misses, hits and evictions
int miss_count;
int hit_count;
int eviction_count;

//print help messages
void printHelp()
{
    printf("Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -
t <tracefile>\n");
    printf("-h: Optional help flag that prints usage info\n");
    printf("-v: Optional verbose flag that displays trace info\n");
    printf("-s <s>: Number of set index bits\n");
    printf("-E <E>: Associativity (number of lines per set)\n");
    printf("-b <b>: Number of block bits\n");
    printf("-t <tracefile>: Name of the valgrind trace to replay\n");
}

//check whether the command is valid

void checkOptarg(char *curOptarg){
    if(curOptarg[0]=='-'){
        printf("Wrong command format\n");
        printHelp();
        exit(0);
    }
```

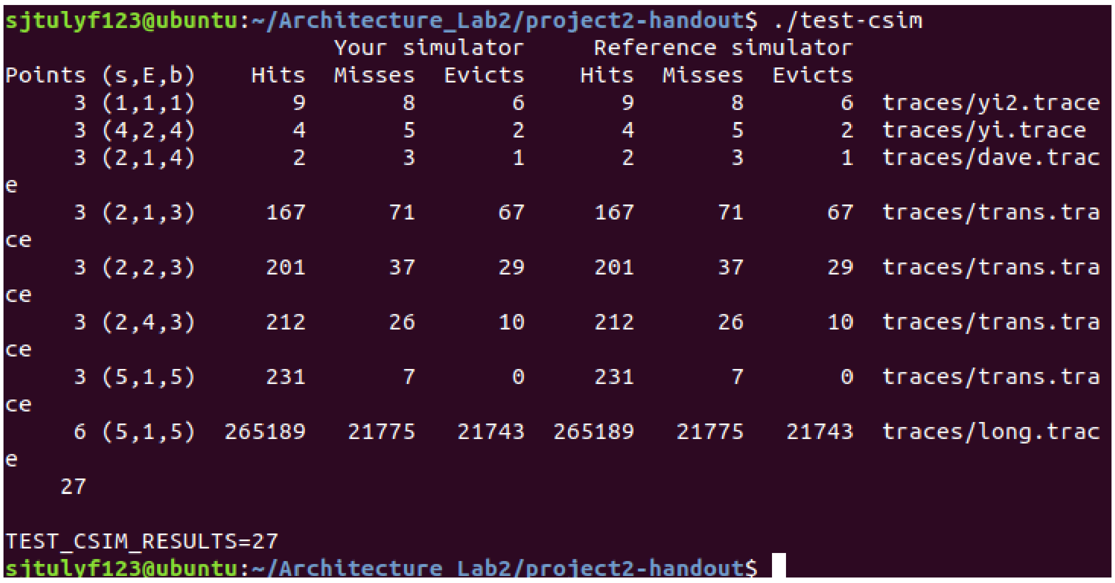
```c
}


//get variables from command

int get_Opt(int argc,char **argv,int *s,int *E,int *b,char *traceName,int *Verbose_mode){
    int c;
    while((c = getopt(argc,argv,"hvs:E:b:t:"))!=-1)
    {
        switch(c)
        {
        case 'v':
            *Verbose_mode = 1;
            break;
        case 's':
            checkOptarg(optarg);
            *s = atoi(optarg);
            break;
        case 'E':
            checkOptarg(optarg);
            *E = atoi(optarg);
            break;
        case 'b':
            checkOptarg(optarg);
            *b = atoi(optarg);
            break;
        case 't':
            checkOptarg(optarg);
            strcpy(traceName,optarg);
            break;
        case 'h':
        default:
            printHelp();
            exit(0);
        }
    }
    return 1;
}

//initial cache
void init_cache(int s,int E,int b, Scache *cache)
{
    //exceptions
    if(s<0)
    {
        printf("sets number must larger than 0\n!");
                exit(0);
     }

     cache->set_cnt=2<<s;
     cache->line_cnt=E;
     cache->sets=(Set *)malloc(cache->set_cnt * sizeof(Set));//initial cache

     int i,j;
     for(i=0;i<cache->set_cnt;++i)
     {
         cache->sets[i].lines=(Line *)malloc(E*sizeof(Line));//allocate line
 space

         for(j=0;j<E;++j)//initial lines
         {
             cache->sets[i].lines[j].valid = 0;
                     cache->sets[i].lines[j].LRU = 0;
         }
     }
     return ;
 }

 //get set number of current address
 int get_set(int addr,int s,int b)
 {
     addr=addr>>b;
     int k=(1<<s)-1;
     return addr &k;
 }

 //get tag number of current address
 int get_tag(int addr,int s,int b)
 {
     int x=s+b;
     return addr>>x;
 }

 //update LRU number, current LRU becomes the max number, others -1
 void update_LRU(Scache *cache, int setaddr, int current_line)
 {
```

```c
    cache->sets[setaddr].lines[current_line].LRU=LRUMAX;
    int i;
    for(i=0;i<cache->line_cnt;++i)//update other LRU
    {
        if(i!=current_line)
        cache->sets[setaddr].lines[i].LRU--;
    }
}

//check whether current data is missing in cache
int miss_checking(Scache *cache,int setaddr,int tagaddr)
{
    int miss_flag=1;
    int i;
    for(i=0;i<cache->line_cnt;++i)
    {
        if(cache->sets[setaddr].lines[i].valid==1 && cache->sets[setaddr].lines[i].tag==tagaddr)//not miss
        {
            miss_flag=0;
            update_LRU(cache, setaddr, i);
        }
    }
    return miss_flag;
}

//find smallest LRU to do replacement
int find_smallest_LRU(Scache *cache,int setaddr)
{
    int target_line=0;//the smallest LRU's index
    int i;
    int min_LRU=LRUMAX;//smallest LRU
    for(i=0;i<cache->line_cnt;++i)//find
    {
        if(cache->sets[setaddr].lines[i].LRU < min_LRU)//update smallest LRU
        {
            target_line=i;
            min_LRU=cache->sets[setaddr].lines[i].LRU;
        }
    }
    return target_line;
}

```

```c
//check whether the cache is full and need replacement
int cache_full(Scache *cache, int setaddr, int tagaddr)
{
    int full_flag=1;
    int i;
    //check full or not
    for(i=0;i<cache->line_cnt;++i)
    {
        if(cache->sets[setaddr].lines[i].valid==0)//there is an empty space
        {
            full_flag=0;
            break;
        }
    }

    if(full_flag==0)//not full
    {
        cache->sets[setaddr].lines[i].valid=1;//update valid bits
        cache->sets[setaddr].lines[i].tag=tagaddr;//update tag
        update_LRU(cache, setaddr,i);

    }
    else//cache is full
    {
        int eviction_line;
        eviction_line=find_smallest_LRU(cache,setaddr);//find the least recent used data
        cache->sets[setaddr].lines[eviction_line].valid=1;//update valid bits
        cache->sets[setaddr].lines[eviction_line].tag=tagaddr;//update tag
        update_LRU(cache,setaddr,eviction_line);
    }
    return full_flag;
}

//load operation
void load_op(Scache *cache,int addr,int sizee,int setaddr,int tagaddr,int Verbose_mode)
{
    if(miss_checking(cache,setaddr,tagaddr)==1)//there is a miss
    {
        miss_count++;
        if(Verbose_mode==1)
            printf("miss ");
        if(cache_full(cache,setaddr,tagaddr)==1)//the cache is full and need eviction
        {
            eviction_count++;
            if(Verbose_mode==1)
                printf("eviction ");
        }
    }
    else//hit
    {
        hit_count++;
            if(Verbose_mode==1) printf("hit ");
    }
}

//store operation
void store_op(Scache *cache,int addr,int sizee,int setaddr,int tagaddr,int Verbose_mode)
{
    load_op(cache,addr,sizee,setaddr,tagaddr,Verbose_mode);
}

//modify operation
void modify_op(Scache *cache,int addr,int sizee,int setaddr,int tagaddr,int Verbose_mode)
{
    load_op(cache,addr,sizee,setaddr,tagaddr,Verbose_mode);
    store_op(cache,addr,sizee,setaddr,tagaddr,Verbose_mode);
}

int main(int argc,char **argv)
{
    int s,E,b=0;
    int Verbose_mode=0;
    miss_count=0;
    hit_count=0;
    eviction_count=0;
    int addr,sizee;

    Scache cache;

    char cur_filename[100];
    char opt[10];

    get_Opt(argc,argv,&s,&E,&b,cur_filename,&Verbose_mode);//get variables from command
    init_cache(s,E,b,&cache);//initial cache sim
    FILE *trace=fopen(cur_filename,"r");//open file and execute

    while(fscanf(trace,"%s %x,%d",opt,&addr,&sizee)!=EOF)
    {
        if(strcmp(opt,"I")==0) continue;//jump I line
        int setaddr = get_set(addr,s,b);
            int tagaddr = get_tag(addr,s,b);
        if(Verbose_mode==1)
            printf("%s %x,%d ",opt,addr,sizee);
        if(strcmp(opt,"S")==0)//store operation
            store_op(&cache,addr,sizee,setaddr,tagaddr,Verbose_mode);
        if(strcmp(opt,"L")==0)//load operation
            load_op(&cache,addr,sizee,setaddr,tagaddr,Verbose_mode);
        if(strcmp(opt,"M")==0)//modify operation
            modify_op(&cache,addr,sizee,setaddr,tagaddr,Verbose_mode);
        if(Verbose_mode==1)
            printf("\n");
    }
    printSummary(hit_count, miss_count, eviction_count);
    return 0;
}
```

2.1.3 Evaluation

I use the ***test-csim*** program to automatically check the correctness of Part A. From the figure below, it could be concluded that I successfully solve Part A.



```
sjtulyf123@ubuntu:~/Architecture_Lab2/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.trac
e
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.tra
ce
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.tra
ce
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.tra
ce
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.tra
ce
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trac
e
    27

TEST_CSIM_RESULTS=27
sjtulyf123@ubuntu:~/Architecture_Lab2/project2-handout$ 
```

2.2 Part B

2.2.1 Analysis

In Part B I modify the function ***transpose_submit*** in ***trans.c*** to implement transpose function of matrix and decrease the number of misses.

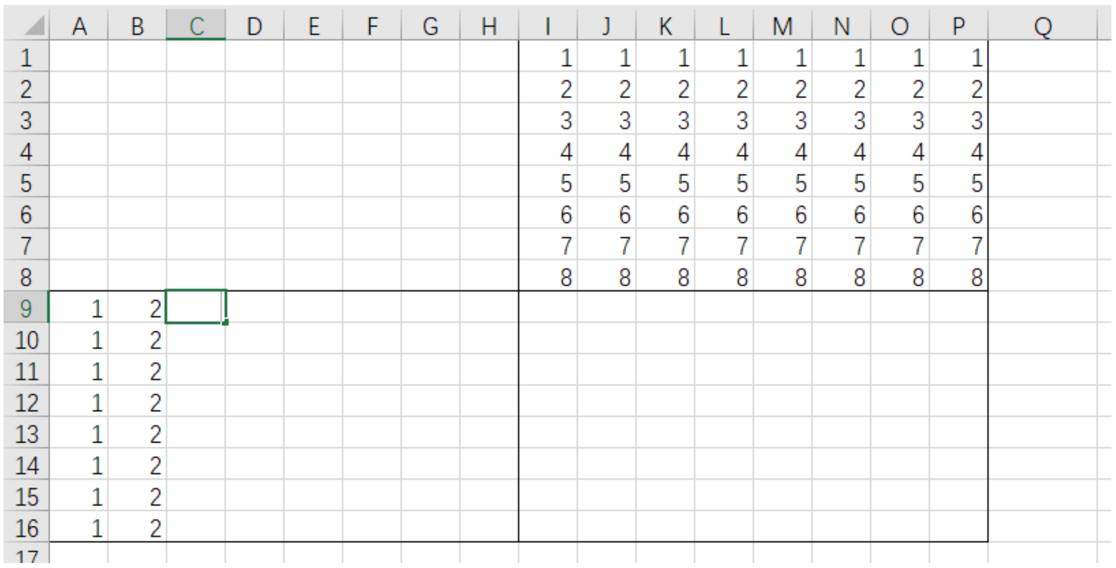
Let me briefly analyze the hits and misses when transposing the matrix. Since the size of block is 32 byte and one int element is 4 byte, so one block contains 8 elements in a matrix. We know that one cache contains 32 blocks, so it contains 256 elements in a matrix. If we simply visit each row and transpose these elements, it would cause a lot of conflict misses because the visit of matrix B would cause the eviction of some block in matrix A. However, this block needs to be visited in later operation. So we need to add this block to the cache again, which causes many misses. To reduce this kind of misses, I split the original 32*32 matrix into sixteen 8*8 small matrices and transpose each small matrix at a time, other than transpose the whole 32 elements in a row at a time. This could reduce the misses obviously, but still larger than 300.

After that, I notice that when it transposes the elements on the diagonal, it would also cause misses because the elements in A and B are mapped to the same block. Therefore, I store the element on the diagonal in block ***b*** to a variable and do transpose of this diagonal element after the program executes the other 7 elements in block ***b***. In this way, the number of misses is fewer than 300.

In 32*32 matrix, I **separate it into 8*8 small matrices**, **execute one small matrix at a time** and **optimize the operation on diagonal elements**; In 61*67 matrix, I **separate it into 16*16 small matrices**, **execute one small matrix at a time** and **optimize the operation on**

diagonal elements. The number of misses of these two matrix sizes is fewer than the requirements. However, the number of misses in 64*64 matrix is still large.

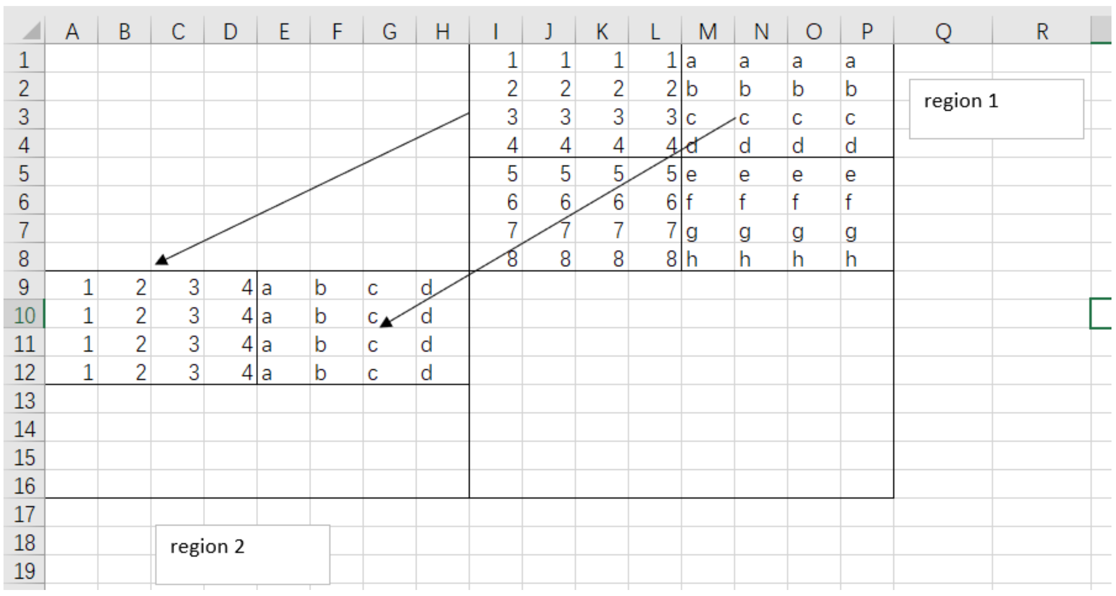
In 64*64 matrix, 4 rows of 64 elements would occupy the whole cache. Therefore, as shown in the figure below, if we transpose the elements in the second small 8*8 matrix and simply visit A by row and visit B by column, it would cause many misses: After visiting 9A, 10A, 11A 12A in matrix B, it wants to visit 13A, 14A, 15A, 16A in matrix B. However, the first 4 elements and last 4 elements in the same column are mapped onto the same blocks. Therefore, when we visit blocks in row 9,10,11,12, the blocks in row 13,14,15,16 would be replaced; when we visit blocks in row 13,14,15,16, the blocks in row 9,10,11,12 would also be replaced. So visiting every element in matrix B would cause a miss.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1									1	1	1	1	1	1	1	1	
2									2	2	2	2	2	2	2	2	
3									3	3	3	3	3	3	3	3	
4									4	4	4	4	4	4	4	4	
5									5	5	5	5	5	5	5	5	
6									6	6	6	6	6	6	6	6	
7									7	7	7	7	7	7	7	7	
8									8	8	8	8	8	8	8	8	
9	1	2															
10	1	2															
11	1	2															
12	1	2															
13	1	2															
14	1	2															
15	1	2															
16	1	2															
17																	

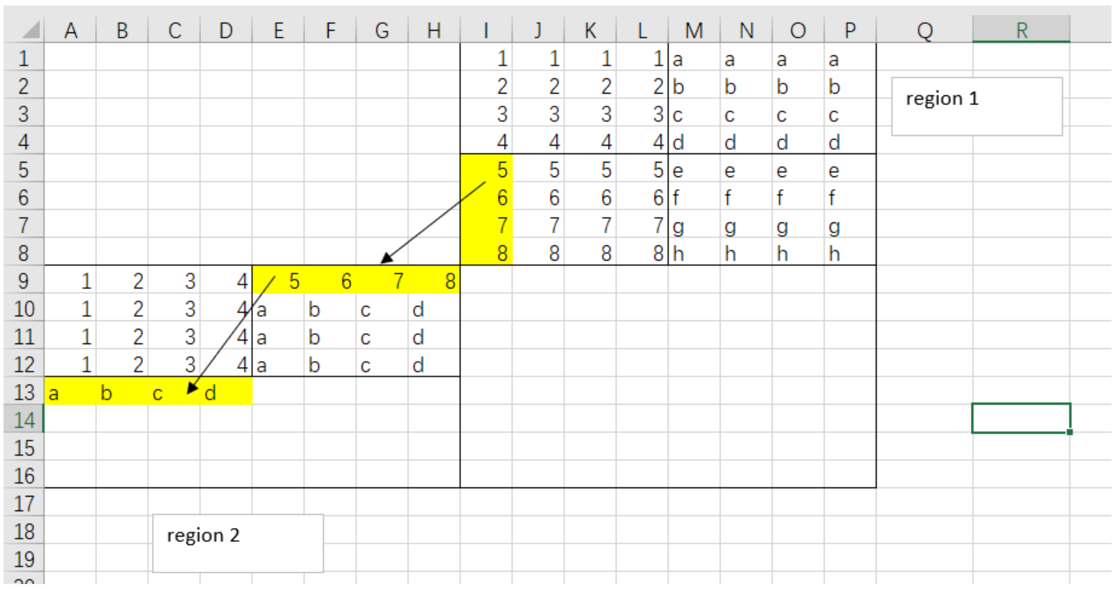
I optimize the operation in the following way (region 1 is in matrix A and region 2 is in matrix B). The key idea is to reduce the repeatedly visit of first 4 rows and last 4 rows in matrix B:

Step 1: Store first 4 rows in region 1 into region 2. During this process, the first 4 columns in first 4 rows are transposed and put into the correct places in region 2, the last 4 columns in first 4 rows are transposed but put into other empty places in region 2 in order not to visit the same block later.

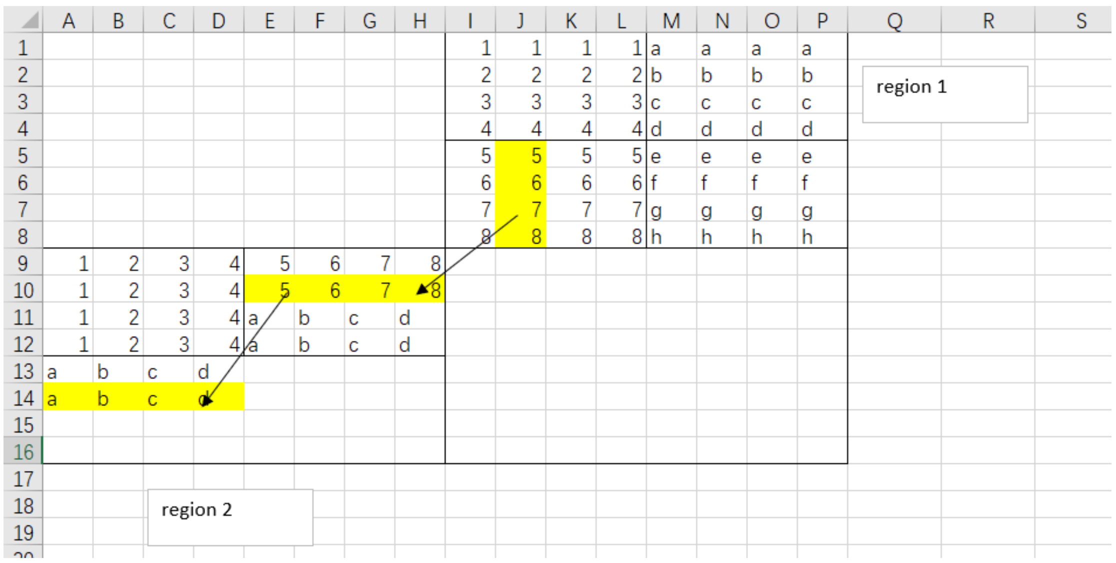


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1									1	1	1	1	a	a	a	a
2									2	2	2	2	b	b	b	b
3									3	3	3	3	c	c	c	c
4									4	4	4	4	d	d	d	d
5									5	5	5	5	e	e	e	e
6									6	6	6	6	f	f	f	f
7									7	7	7	7	g	g	g	g
8									8	8	8	8	h	h	h	h
9	1	2	3	4	a	b	c	d								
10	1	2	3	4	a	b	c	d								
11	1	2	3	4	a	b	c	d								
12	1	2	3	4	a	b	c	d								

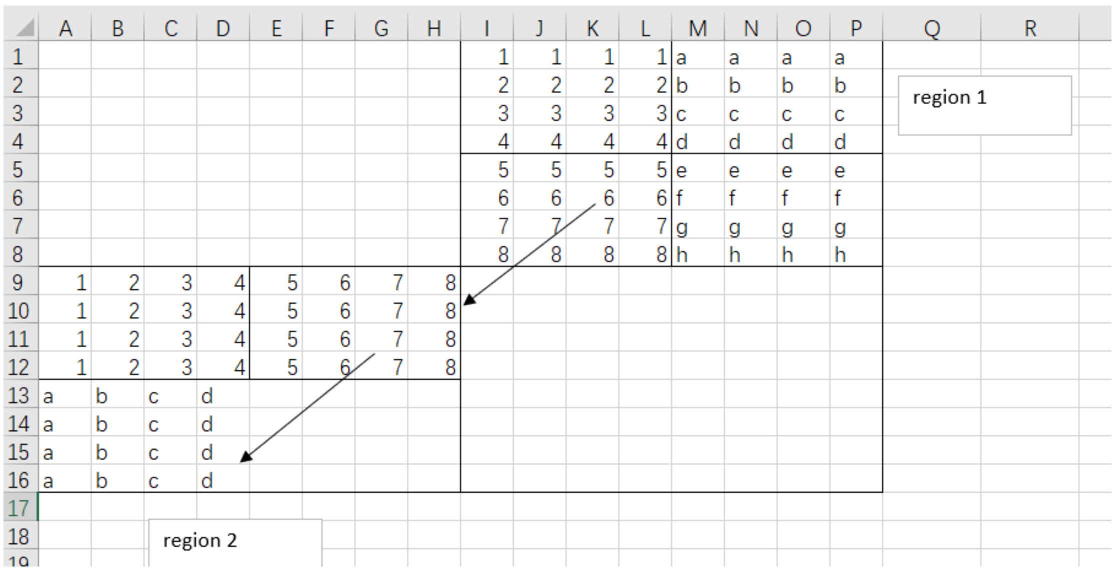
Step 2: Put the last 4 columns in first 4 rows into correct places in region 2 by each row. Transpose the first 4 columns in last 4 rows and put them into the correct places in region 2. After this step, 3 small 4*4 matrices is transposed correctly.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1									1	1	1	1	a	a	a	a
2									2	2	2	2	b	b	b	b
3									3	3	3	3	c	c	c	c
4									4	4	4	4	d	d	d	d
5									5	5	5	5	e	e	e	e
6									6	6	6	6	f	f	f	f
7									7	7	7	7	g	g	g	g
8									8	8	8	8	h	h	h	h
9	1	2	3	4	5	6	7	8								
10	1	2	3	4	a	b	c	d								
11	1	2	3	4	a	b	c	d								
12	1	2	3	4	a	b	c	d								
13	a	b	c	d												

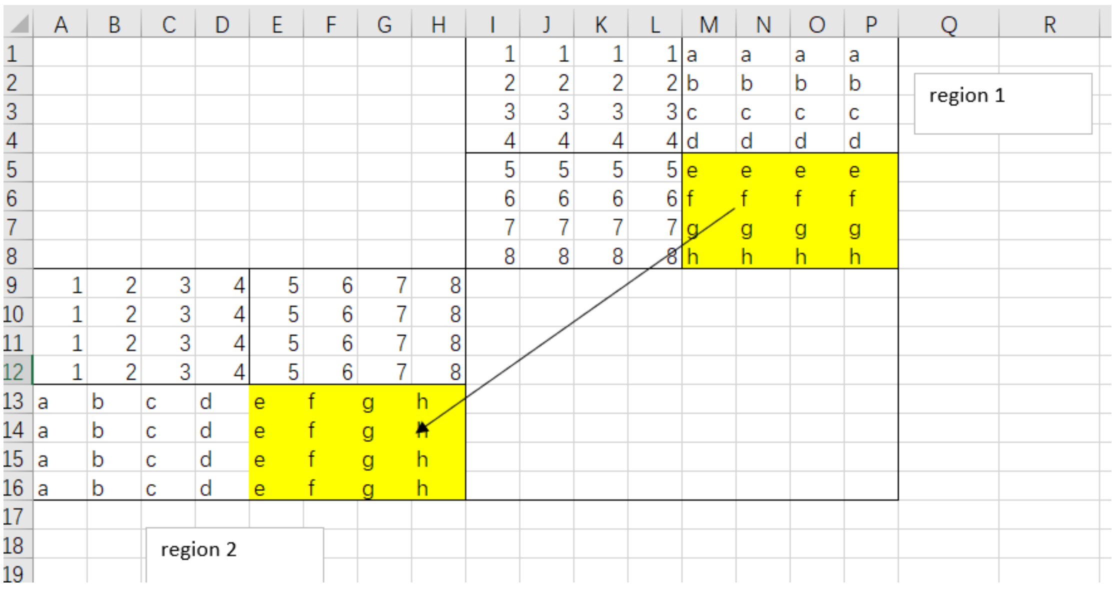


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1									1	1	1	1	a	a	a	a	
2									2	2	2	2	b	b	b	b	region 1
3									3	3	3	3	c	c	c	c	
4									4	4	4	4	d	d	d	d	
5									5	5	5	5	e	e	e	e	
6									6	6	6	6	f	f	f	f	
7									7	7	7	7	g	g	g	g	
8									8	8	8	8	h	h	h	h	
9	1	2	3	4	5	6	7	8									
10	1	2	3	4	5	6	7	8									
11	1	2	3	4	a	b	c	d									
12	1	2	3	4	a	b	c	d									
13	a	b	c	d													
14	a	b	c	d													
15																	
16																	
17																	
18			region 2														
19																	



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1									1	1	1	1	a	a	a	a	
2									2	2	2	2	b	b	b	b	region 1
3									3	3	3	3	c	c	c	c	
4									4	4	4	4	d	d	d	d	
5									5	5	5	5	e	e	e	e	
6									6	6	6	6	f	f	f	f	
7									7	7	7	7	g	g	g	g	
8									8	8	8	8	h	h	h	h	
9	1	2	3	4	5	6	7	8									
10	1	2	3	4	5	6	7	8									
11	1	2	3	4	5	6	7	8									
12	1	2	3	4	5	6	7	8									
13	a	b	c	d													
14	a	b	c	d													
15	a	b	c	d													
16	a	b	c	d													
17																	
18			region 2														

Step 3: Transpose the last 4 columns in last 4 rows.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1									1	1	1	1	a	a	a	a	
2									2	2	2	2	b	b	b	b	region 1
3									3	3	3	3	c	c	c	c	
4									4	4	4	4	d	d	d	d	
5									5	5	5	5	e	e	e	e	
6									6	6	6	6	f	f	f	f	
7									7	7	7	7	g	g	g	g	
8									8	8	8	8	h	h	h	h	
9	1	2	3	4	5	6	7	8									
10	1	2	3	4	5	6	7	8									
11	1	2	3	4	5	6	7	8									
12	1	2	3	4	5	6	7	8									
13	a	b	c	d	e	f	g	h									
14	a	b	c	d	e	f	g	h									
15	a	b	c	d	e	f	g	h									
16	a	b	c	d	e	f	g	h									
17																	
18			region 2														
19																	

After executing the processes, 1 block in region 2 (matrix B) would just cause 1 miss. This kind of operation decreases the number of misses dramatically.

2.2.2 Code

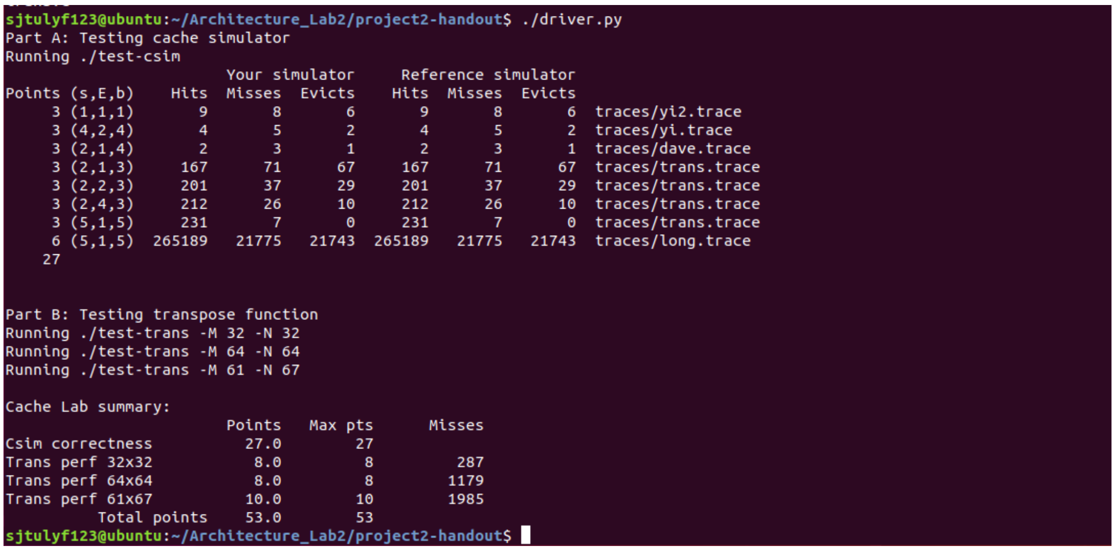
```c
//Name: Yifan Liu
//ID:   518021910609

/*
 * trans.c - Matrix transpose B = A^T
 *
 * Each transpose function must have a prototype of the form:
 * void trans(int M, int N, int A[N][M], int B[M][N]);
 *
 * A transpose function is evaluated by counting the number of misses
 * on a 1KB direct mapped cache with a block size of 32 bytes.
 */
#include <stdio.h>
#include "cachelab.h"

int is_transpose(int M, int N, int A[N][M], int B[M][N]);

/*
 * transpose_submit - This is the solution transpose function that you
 *     will be graded on for Part B of the assignment. Do not change
 *     the description string "Transpose submission", as the driver
 *     searches for that string to identify the transpose function to
 *     be graded.
 */
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    //12 variables in total
    //row is the row block
    //clo is the column block
    int i,j,row,col;
    int x1,x2,x3,x4,x5,x6,x7,x8;

    if(M==32)//seperate 32*32 into 8*8, so the miss is smaller
    {
        for(row=0;row<N;row=row+8)
        for(col=0;col<M;col=col+8)
        for(i=row;i<row+8;++i)
        {
        for(j=col;j<col+8;++j)
         {
            if(i!=j) B[j][i]=A[i][j];
            else
            {
                //i==j,the diagonal element,it need to save to B later,otherwise the miss will increase
                x1=A[i][j];
                x2=i;
            }
         }
         if(col==row)
            {
                B[x2][x2]=x1;//save element to B in this time. This would not cause more miss
            }
        }
    }
    else if(M==64)
    {
        for(row=0;row<N;row=row+8)
        for(col=0;col<M;col=col+8)
        {
            for(i=row;i<row+4;i++)
            {
                x1=A[i][col];x2=A[i][col+1];x3=A[i][col+2];x4=A[i][col+3];
                x5=A[i][col+4];x6=A[i][col+5];x7=A[i][col+6];x8=A[i][col+7];//first line
                //first 4 element in first 4 lines:do transpose
                B[col][i]=x1;B[col+1][i]=x2;B[col+2][i]=x3;B[col+3][i]=x4;
                //last 4 elements in first 4 lines:transpose and move these to ''some empty B space''
                B[col][i+4]=x5;B[col+1][i+4]=x6;B[col+2][i+4]=x7;B[col+3][i+4]=x8;
            }
            for(j=col;j<col+4;++j)
            {

                x1=A[row+4][j];x2=A[row+5][j];x3=A[row+6][j];x4=A[row+7][j];

                x5=B[j][row+4];x6=B[j][row+5];x7=B[j][row+6];x8=B[j][row+7];

                //last 4 elements in first 4 columns: do transpose
                B[j][row+4]=x1;B[j][row+5]=x2;B[j][row+6]=x3;B[j][row+7]=x4;
                //transpose data in ''some empty B space'' to correct space
                B[j+4][row]=x5;B[j+4][row+1]=x6;B[j+4][row+2]=x7;B[j+4][row+3]=x8;
            }
            for(i=row+4;i<row+8;++i)
            {
                //other data:do transpose per line
                x1=A[i][col+4];x2=A[i][col+5];x3=A[i][col+6];x4=A[i][col+7];
                B[col+4][i]=x1;B[col+5][i]=x2;B[col+6][i]=x3;B[col+7][i]=x4;
            }
        }
    }
    else//67*61,separate it into 16*16
    {
        for(row=0;row<N;row=row+16)
        for(col=0;col<M;col=col+16)
        for(i=row;i<row+16 &&(i<N);++i)
        {
        for(j=col;j<col+16&&(j<M);++j)
         {
            if(i!=j) B[j][i]=A[i][j];
            else
            {
                //i==j,the diagonal element,it need to save to B later,otherwise the miss will increase
                x1=A[i][j];
                x2=i;
            }
         }
         if(col==row)
            {
                B[x2][x2]=x1;//save element to B in this time. This would not cause more miss
            }
        }
    }
}

/*
```

```c
 * You can define additional transpose functions below. We've defined
 * a simple one below to help you get started.
 */

/*
 * trans - A simple baseline transpose function, not optimized for the cache.
 */
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, tmp;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            tmp = A[i][j];
            B[j][i] = tmp;
        }
    }

}

/*
 * registerFunctions - This function registers your transpose
 *     functions with the driver.  At runtime, the driver will
 *     evaluate each of the registered functions and summarize their
 *     performance. This is a handy way to experiment with different
 *     transpose strategies.
 */
void registerFunctions()
{
    /* Register your solution function */
    registerTransFunction(transpose_submit, transpose_submit_desc);

    /* Register any additional transpose functions */
    registerTransFunction(trans, trans_desc);

}

/*
 * is_transpose - This helper function checks if B is the transpose of
 *     A. You can check the correctness of your transpose by calling
 *     it before returning from the transpose function.
 */
```

```c
int is_transpose(int M, int N, int A[N][M], int B[M][N])
{
    int i, j;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; ++j) {
            if (A[i][j] != B[j][i]) {
                return 0;
            }
        }
    }
    return 1;
}
```

2.2.3 Evaluation

I use driver.py to check both Part A and Part B. From the figure below, it could be concluded that I successfully decrease the number of misses to the goal in 32*32, 64*64 and 61*67 matrices. I solve Part B successfully.



```
sjtulyf123@ubuntu:~/Architecture_Lab2/project2-handout$ ./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        1985
          Total points    53.0        53
sjtulyf123@ubuntu:~/Architecture_Lab2/project2-handout$ 
```

3. Conclusion

3.1 Problems

In part A, I meet a problem of how to correctly get the variables s, E, and b from the command line. After searching for the use of ***getopt*** function and ***optarg*** variable, I solve this problem.

When writing the store operation, I am confused with what I need to do in this operation. To solve this, I refer to the CSAPP book to check what the cache does when it needs to store some data. It turns out that the store operation is just like the load operation, especially when counting the number of hits, misses and evictions. Similarly, the modify operation is just one load and one store operation.

I also overcome a problem when I try to test my Part A program by using ***valgrind***. The command says: “valgrind: not found”. I thought that there are some problems with my program or segmentation fault, but I could not figure out what happens. After carefully check the hints in the project guide, I notice that I forget to install ***valgrind*** on my virtual machine. After installing it, the problem is solved.

In part B, it is easy to discover that separating matrix into some small matrices is useful, but determining the size of small matrix takes me a lot of time. I tried 4*4, 8*8 and 16*16 and compare the number of misses. Finally, I decide to use 8*8 size in 32*32 and 64*64 matrix, use 16*16 size in 61*67 matrix. But the number of misses is still larger that the requirement. After analyzing the behavior of transpose operation, I discover that the diagonal elements would also cause misses.

When optimizing 64*64 transpose matrix, the former method is not enough. After searching for some matrix optimizing methods, I finally realize that storing every element in B would cause a miss, so I continue to optimize the operation. The difficulty of Part B is to find a valid method to reduce the number of misses.

3.2 Achievements

1. I successfully implement the simulated cache and it could correctly report the number of misses, hits and evictions. I also successfully implement the matrix transpose function to reduce the number of misses when transposing a matrix.
2. Both of the programs have good robustness. They could detect the invalid input and send the corresponding error messages.
3. I have a better understanding of how cache works and how the misses happen when transposing the matrix.