Project 2: Understanding Cache Memories

[518021910604-Shengyuan Hou-445164577@qq.com]

1. Introduction

This project is mainly about the cache simulation and blocking techniques to reduce cache miss. Part A is about simulating the behavior of a cache memory with C code. Part B is about optimizing a small matrix transpose function, with the goal of minimizing cache misses.

Shengyuan Hou finishes both parts.

2. Experiments

2.1 Part A

2.1.1 Analysis

(1)Generalization

In this part , we are requested to simulate the behavior of a cache memory. Four types of memory access operation, I(instruction load),L(load),S(store),M(modify) are provided by the input trace file, while I(instruction load) should be ignored. The overall code thinking is as follows.

1. Make use of getopt() function to fetch the parameters help, verbosemmode, E,b,t, tracefile.
2. Construct and initialize the cache using malloc() and memset().
3. Read in the tracefile, fetch the access address, request size(no use in this project) and judge the operation type,.
4. If operation is I ,ignore it, otherwise call the corresponding function Load(), Store() or modify() to simulate cache access.
5. Simulate the cache access and linearly search the set to determine hit or miss, if there is a miss and the corresponding line is valid, call the eviction() function to rewrite the line.
6. Do 2->3->4->5 until the end of trace file

(2)Difficulties

1.Integrate L(load),S(store),M(modify) three types of cache access operation into one cache_access function.

When the CPU is to read a word from address A of main memory, it sends the address A to the cache. If the cache is holding a copy of the word at address A, we have a cache hit. Otherwise we have a cache miss, and the CPU must wait until the data is copied from the main memory to the corresponding cache line. If the cache line is already filled, an eviction happens.

When the CPU is to store a word to address A of main memory, it sends the address A to the cache. If the cache is holding a copy of the word at address A, we have a cache hit and modify the cache. Otherwise we have a cache miss, and if the corresponding cache line is already filled, an eviction happens.(We take write-back strategy).

The data modify operation (M) is treated as a load followed by a store to the same address. We could simply regard it as a special L+S.

According to the precedent analysis, although L and S perform different operations on data, their cache access operations are the same except for writing or reading, so their hit, miss and eviction are respectively the same as well. Therefore, we could define a union function cache_access to count the three indexes.

2.organization of cache struct

In reality, cache is organized into the following structure.

Consider a computer system where each memory address has m bits that form M=2^m unique addresses, which is illustrated in Figure 2.

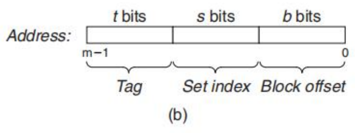


Figure 2

As illustrated in Figure 1, a cache for such a machine is organized as an array of S=2^s cache sets. Each set consists of E cache lines. Each line consists of a data block of B=2^b bytes, a valid bit that indicates whether or not the line contains meaningful information and t=m-(b+s) tag bits(a subset of the bits from address) that uniquely identify the block stored in the cache line.

In general, a cache's organization can be characterized by the tuple (S,E,B,m). The size(or capacity) of a cache, C, is stated in terms of the aggregate size of all the blocks. The tag bits and valid bit are not included. Thus C=S×E×B.

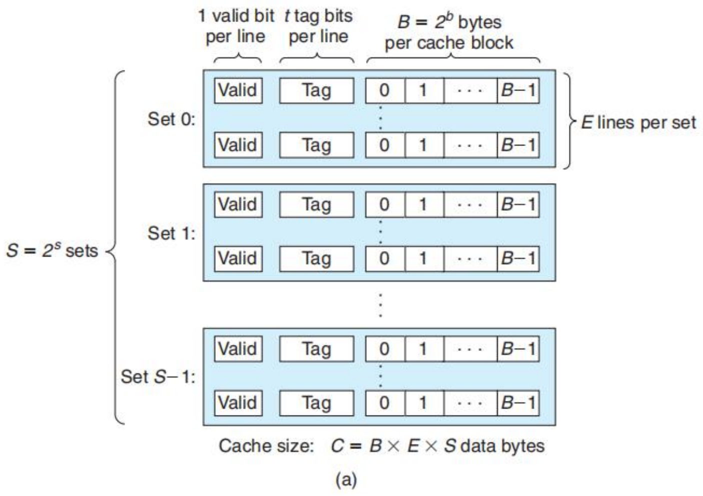
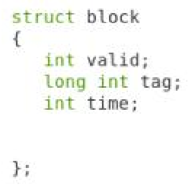


Figure 1

Nevertheless, simulation differs from the reality, a cache simulator is not a cache!

1. We do not store the memory contents.
2. We do not use the block offset in the address.
3. We simply want to count the cache hits, cache misses and cache evictions.
4. We want to record the recent access time of every line for LRU algorithm.
5. We do not have to store the components of cache line into binary or hexadecimal form. Int or long int is okay.

Due to precedent analysis, we could ignore some components of cache line and add new ones. Since we do not store memory contents, we do not have to store the data area in simulative structure. Also, every valid line should record the most recent access time, which is represented by a timestamp integer "clock", for LRU algorithm, so we add "time" variable. Valid bit and tag index are necessary to match the address, therefore we retain them. Cache line struct is displayed in Figure 3.



```
struct block
{
   int valid;
   long int tag;
   int time;


};
```

Figure3

(3)Core technique

1.LRU cache replacement algorithm

We use the timestamps to represent the access time, which is stored in the "block" struct in every line of cache. When it turns to the usage of LRU in some cache set, we just sequentially search all lines in the set and select one with the least recent access time, and then modify corresponding variables.

2.read in the parameter

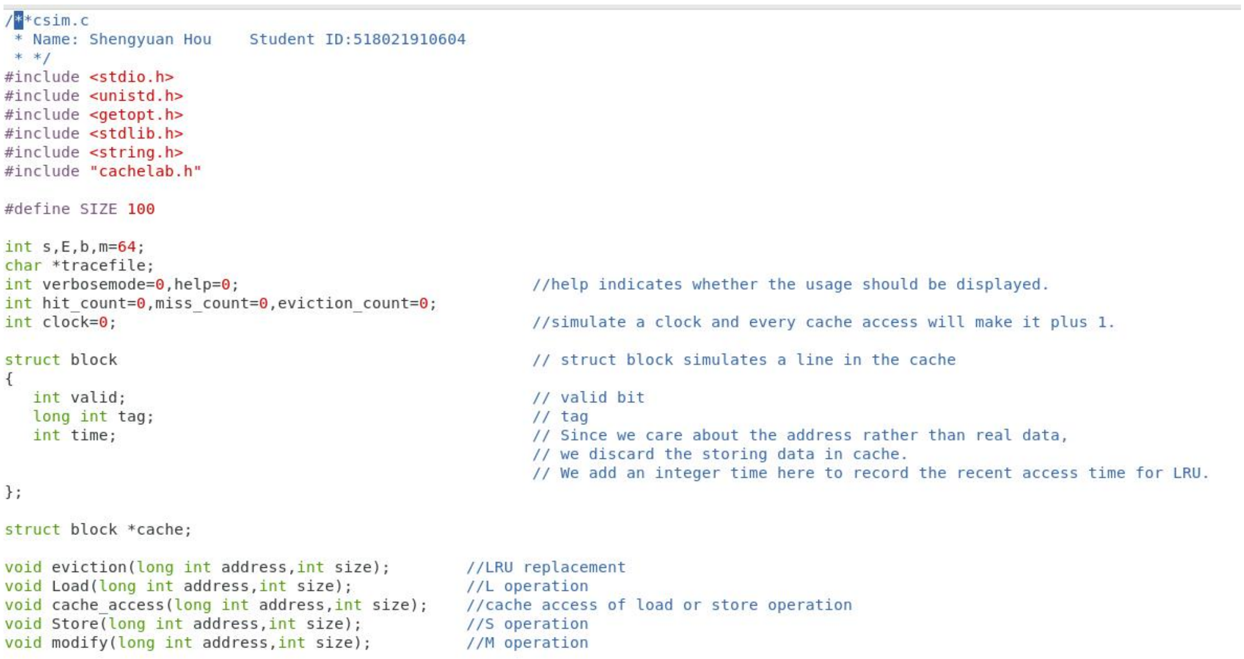
We use the Linux getopt() function to interpret the parameters

3.judgement of cache hit, cache miss and cache eviction

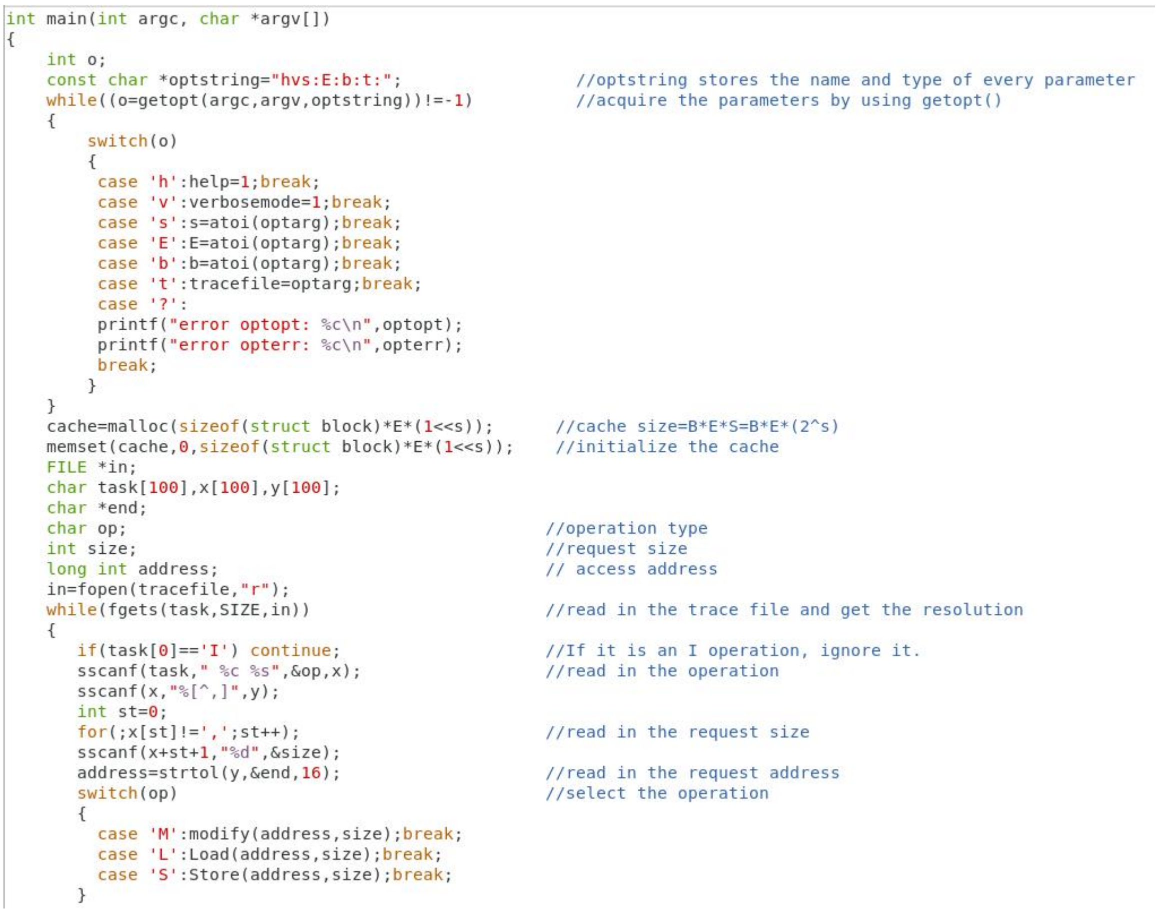
Firstly we calculate the set index by the address. Secondly we linearly retrieve the lines to check whether the valid bit is set and the tag index is matched. If we find a matching, then it's a hit, otherwise there is a miss. If there is no empty line in the set, then there is a cache eviction.

2.1.2 Code

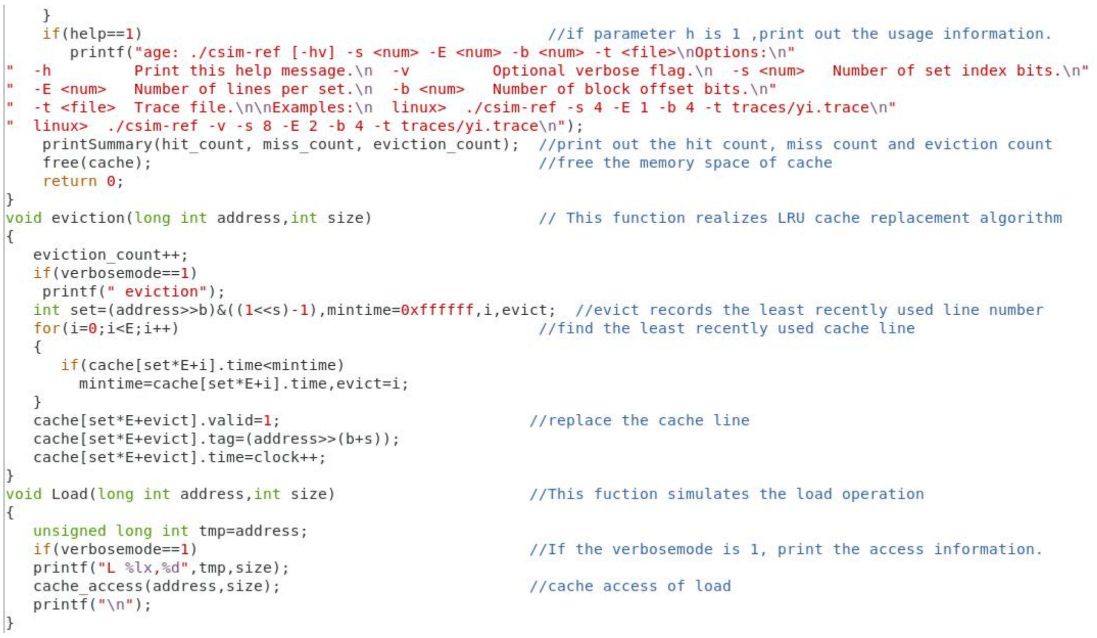
csim.c



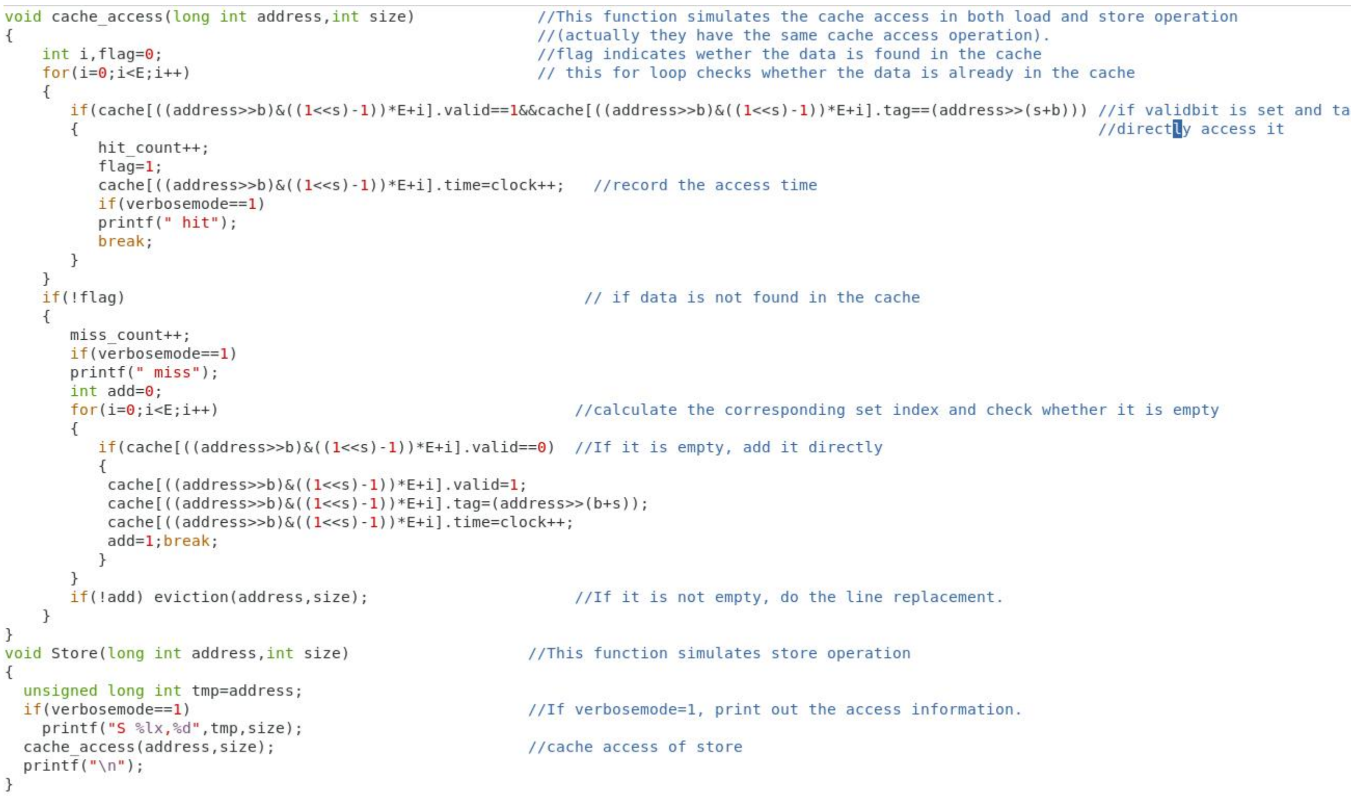
```c
/**csim.c
 * Name: Shengyuan Hou    Student ID:518021910604
 * */
#include <stdio.h>
#include <unistd.h>
#include <getopt.h>
#include <stdlib.h>
#include <string.h>
#include "cachelab.h"

#define SIZE 100

int s,E,b,m=64;
char *tracefile;
int verbosemode=0,help=0;                                //help indicates whether the usage should be displayed.
int hit_count=0,miss_count=0,eviction_count=0;
int clock=0;                                             //simulate a clock and every cache access will make it plus 1.

struct block                                             // struct block simulates a line in the cache
{
   int valid;                                            // valid bit
   long int tag;                                         // tag
   int time;                                             // Since we care about the address rather than real data,
                                                         // we discard the storing data in cache.
                                                         // We add an integer time here to record the recent access time for LRU.
};

struct block *cache;

void eviction(long int address,int size);        //LRU replacement
void Load(long int address,int size);            //L operation
void cache_access(long int address,int size);    //cache access of load or store operation
void Store(long int address,int size);           //S operation
void modify(long int address,int size);          //M operation
```



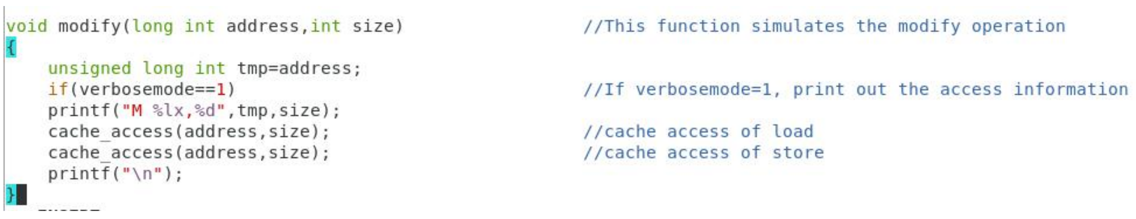
```c
int main(int argc, char *argv[])
{
    int o;
    const char *optstring="hvs:E:b:t:";                  //optstring stores the name and type of every parameter
    while((o=getopt(argc,argv,optstring))!=-1)           //acquire the parameters by using getopt()
    {
        switch(o)
        {
         case 'h':help=1;break;
         case 'v':verbosemode=1;break;
         case 's':s=atoi(optarg);break;
         case 'E':E=atoi(optarg);break;
         case 'b':b=atoi(optarg);break;
         case 't':tracefile=optarg;break;
         case '?':
         printf("error optopt: %c\n",optopt);
         printf("error opterr: %c\n",opterr);
         break;
        }
    }
    cache=malloc(sizeof(struct block)*E*(1<<s));      //cache size=B*E*S=B*E*(2^s)
    memset(cache,0,sizeof(struct block)*E*(1<<s));    //initialize the cache
    FILE *in;
    char task[100],x[100],y[100];
    char *end;
    char op;                                         //operation type
    int size;                                        //request size
    long int address;                                // access address
    in=fopen(tracefile,"r");
    while(fgets(task,SIZE,in))                       //read in the trace file and get the resolution
    {
       if(task[0]=='I') continue;                    //If it is an I operation, ignore it.
       sscanf(task," %c %s",&op,x);                  //read in the operation
       sscanf(x,"%[^,]",y);
       int st=0;
       for(;x[st]!=',';st++);                        //read in the request size
       sscanf(x+st+1,"%d",&size);
       address=strtol(y,&end,16);                    //read in the request address
       switch(op)                                    //select the operation
       {
         case 'M':modify(address,size);break;
         case 'L':Load(address,size);break;
         case 'S':Store(address,size);break;
       }
```



```c
    }
    if(help==1)                                              //if parameter h is 1 ,print out the usage information.
       printf("age: ./csim-ref [-hv] -s <num> -E <num> -b <num> -t <file>\nOptions:\n"
"  -h         Print this help message.\n  -v         Optional verbose flag.\n  -s <num>   Number of set index bits.\n"
"  -E <num>   Number of lines per set.\n  -b <num>   Number of block offset bits.\n"
"  -t <file>  Trace file.\n\nExamples:\n  linux>  ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace\n"
"  linux>  ./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace\n");
    printSummary(hit_count, miss_count, eviction_count);  //print out the hit count, miss count and eviction count
    free(cache);                                          //free the memory space of cache
    return 0;
}
void eviction(long int address,int size)                  // This function realizes LRU cache replacement algorithm
{
   eviction_count++;
   if(verbosemode==1)
    printf(" eviction");
   int set=(address>>b)&((1<<s)-1),mintime=0xffffff,i,evict;  //evict records the least recently used line number
   for(i=0;i<E;i++)                                       //find the least recently used cache line
   {
      if(cache[set*E+i].time<mintime)
        mintime=cache[set*E+i].time,evict=i;
   }
   cache[set*E+evict].valid=1;                           //replace the cache line
   cache[set*E+evict].tag=(address>>(b+s));
   cache[set*E+evict].time=clock++;
}
void Load(long int address,int size)                     //This fuction simulates the load operation
{
   unsigned long int tmp=address;
   if(verbosemode==1)                                    //If the verbosemode is 1, print the access information.
   printf("L %lx,%d",tmp,size);
   cache_access(address,size);                           //cache access of load
   printf("\n");
}
```



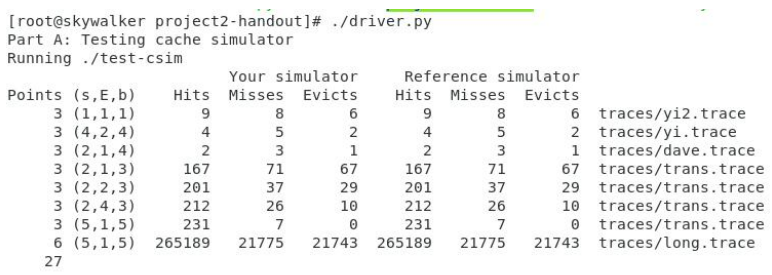
```c
void cache_access(long int address,int size)             //This function simulates the cache access in both load and store operation
{                                                        //(actually they have the same cache access operation).
    int i,flag=0;                                        //flag indicates wether the data is found in the cache
    for(i=0;i<E;i++)                                     // this for loop checks whether the data is already in the cache
    {
       if(cache[((address>>b)&((1<<s)-1))*E+i].valid==1&&cache[((address>>b)&((1<<s)-1))*E+i].tag==(address>>(s+b))) //if validbit is set and ta
       {                                                                                                              //directly access it
          hit_count++;
          flag=1;
          cache[((address>>b)&((1<<s)-1))*E+i].time=clock++;   //record the access time
          if(verbosemode==1)
          printf(" hit");
          break;
       }
    }
    if(!flag)                                                 // if data is not found in the cache
    {
       miss_count++;
       if(verbosemode==1)
       printf(" miss");
       int add=0;
       for(i=0;i<E;i++)                                      //calculate the corresponding set index and check whether it is empty
       {
          if(cache[((address>>b)&((1<<s)-1))*E+i].valid==0)  //If it is empty, add it directly
          {
           cache[((address>>b)&((1<<s)-1))*E+i].valid=1;
           cache[((address>>b)&((1<<s)-1))*E+i].tag=(address>>(b+s));
           cache[((address>>b)&((1<<s)-1))*E+i].time=clock++;
           add=1;break;
          }
       }
       if(!add) eviction(address,size);                      //If it is not empty, do the line replacement.
    }
}
void Store(long int address,int size)                    //This function simulates store operation
{
  unsigned long int tmp=address;
  if(verbosemode==1)                                     //If verbosemode=1, print out the access information.
    printf("S %lx,%d",tmp,size);
  cache_access(address,size);                            //cache access of store
  printf("\n");
}
```



```c
void modify(long int address,int size)                //This function simulates the modify operation
{
    unsigned long int tmp=address;
    if(verbosemode==1)                                //If verbosemode=1, print out the access information
    printf("M %lx,%d",tmp,size);
    cache_access(address,size);                       //cache access of load
    cache_access(address,size);                       //cache access of store
    printf("\n");
}
```

2.1.3 Evaluation

Obviously we have passed the test because we get a full score on the auto graders.



```
[root@skywalker 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
```

2.2 Part B

2.2.1 Analysis

(1)Generalization

In this part, we are requested to optimize a transpose function for the purpose of minimizing cache misses. Three matrices are provided, which are respectively 32×32, 64×64, 61×67, and the overall strategy is as follows.

●32×32: Use cache blocking technique and the size of block matrix is 8×8---Assign 8 elements one time to reduce miss.We use another trick "copy first, transpose second" and it will be illustrated in the following part.

●64×64: Use cache blocking technique and the size of block matrix is 8×8. Different from 32×32 matrix, we use a ingenious method to reduce the miss conflict, which is called as "Step by step" by myself.

●61×67: Simply do some trials and choose the best size for block matrix. Use normal blocking technique and the size of block matrix is 18×14.

(2)Difficulties

●32×32: Reduce the conflict miss on the diagonal block matrix. When calculating the diagonal transpose matrix B of diagonal block matrix A, we find that A and B have the same cache set because their set index are the same. This will increase the cache miss and we use "copy first, transpose second" strategy to resolve it.

●64×64: Reduce the conflict miss within the block matrix. A 8×8 block matrix here will cause a conflict miss, because the upper half side(4 rows) has the same set index with the lower half side(4 rows). We use the "Step by step" technique to resolve it.

(3)Core technique

1. normal blocking technique

$$\begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} = \begin{pmatrix} A_1^T & A_3^T \\ A_2^T & A_4^T \end{pmatrix}$$

Blocking a matrix transpose routine works by partitioning the matrices into submatrices and then exploiting the mathematical fact that these submatrices can be manipulated just like scalars.

By partitioning the matrix into submatrices, the number of rows written to the B matrix each time is limited, and the portion of the B matrix in the cache is fully taken advantage.

2.cache blocking

The implementation of the normal blocking will cause too many conflicts on the diagonal blocks, because the cache blocks of the A and B matrices are replaced with each other too often. We can consider using 8 local variables to store a line of A, and then copy it to B, that is, use the local variables as a cache to store the contents of each cache block.

```
a0=A[k][j]      a1=A[k][j+1]   a2=A[k][j+2]   a3=A[k][j+3]
a4=A[k][j+4]   a5=A[k][j+5]   a6=A[k][j+6]   a7=A[k][j+7]
B[j][k]=a0      B[j+1][k]=a1   B[j+2][k]=a2   B[j+3][k]=a3
B[j+4][k]=a4   B[j+5][k]=a5   B[j+6][k]=a6   B[j+7][k]=a7
```

3.“copy first, transpose second” technique in Matrix 32×32

Although we have taken blocking strategy, the cache conflicts between the A and B matrices will inevitably occur at the diagonal blocks because their corresponding elements have the same set index. However, the experiment requires that the A matrix cannot be changed, but B can be handled. We can consider a way to eliminate the conflict between the two matrices on the B matrix.

In order to eliminate the mutual replacement of the block lines on the diagonal, each line of the block A is first cached with 8 local variables, and then copied to the line corresponding to the block B. After the copy is completed, all of B's blocks are in the cache, and there is no miss in the transposition process.

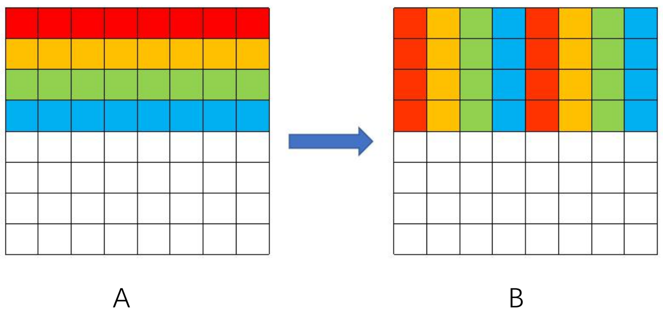
```
a0=A[k][j]      a1=A[k][j+1]   a2=A[k][j+2]   a3=A[k][j+3]
a4=A[k][j+4]   a5=A[k][j+5]   a6=A[k][j+6]   a7=A[k][j+7]
B[j][k]=a0      B[j][k+1]=a1   B[j][k+2]=a2   B[j][k+3]=a3
B[j][k+4]=a4   B[j][k+5]=a5   B[j][k+6]=a6   B[j][k+7]=a7
```

Next transpose every block on the B matrix(no miss) and we get the transpose matrix B=A^T.

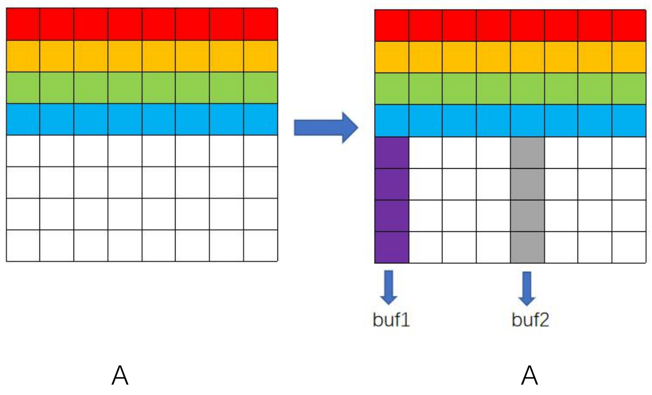
4.”Step by step” technique in Matrix 64×64

The 64×64 matrix requires 8 cache blocks per line, and the cache index is repeated every four lines. Obviously, with the 8 × 8 block matrix the cache blocks in the same matrix will conflict, so we need to change the transpose method. Also we must make full use of the two ideas mentioned above: use local variables as cache, copy first and then transpose. Here are the four steps of ”Step by step”.

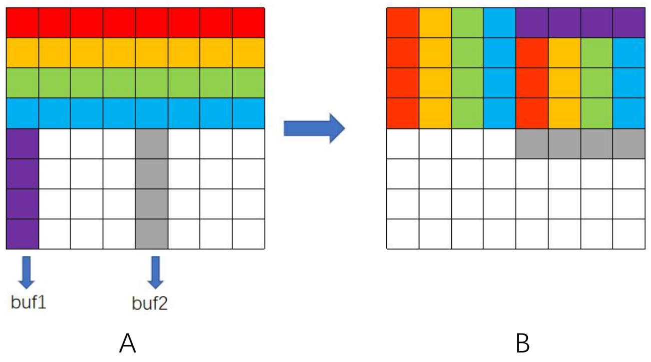
(1)First copy the first four lines of A into B and transpose two 4×4 matrices in B respectively.



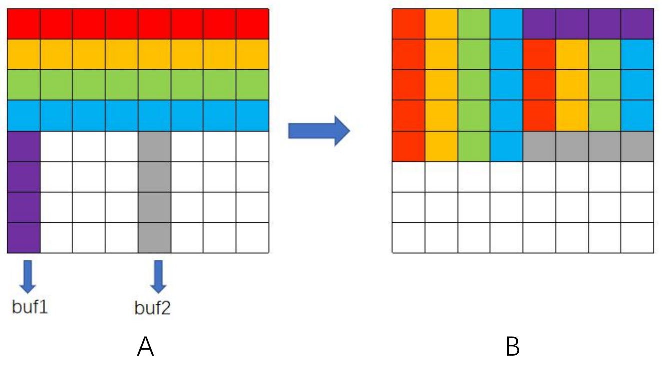
(2)Store the element at the corresponding position(buf1 and buf2) in A into 8 local variables.



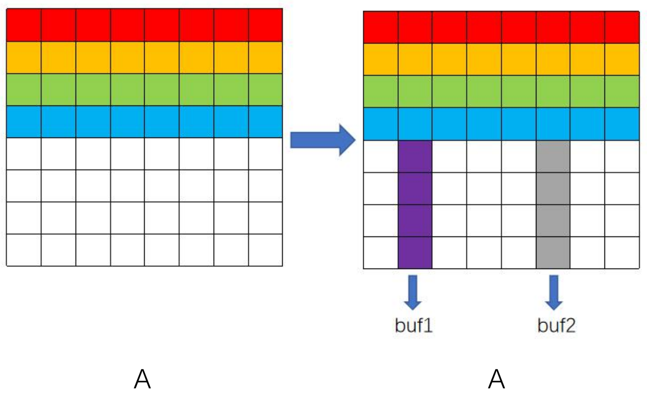
(3)The four elements of buf1 are exchanged with the first line in the upper right corner of B, and the value in buf2 is stored in the corresponding position in the lower right corner of B. At this time, B[4] in the cache replaces B[0].



(4)Store the element in buf1(Now buf1 has exchanges the value with four elements respectively in the upper right corner of B matrix in (3)) to the corresponding position in the lower left corner of B.



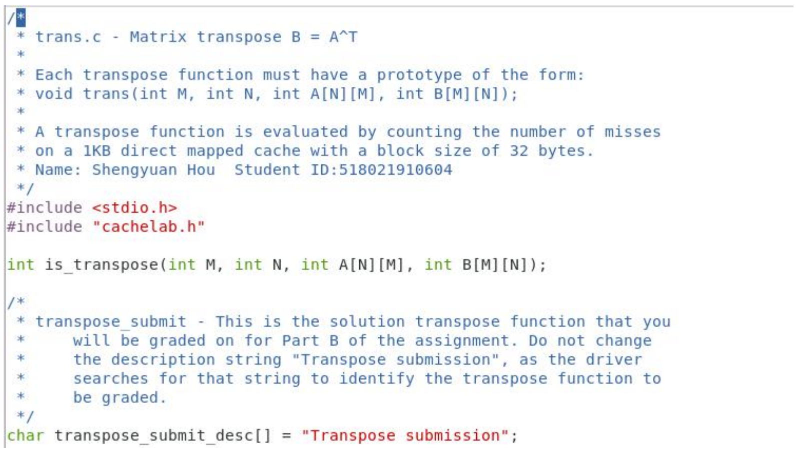
(5)Change buf1 and buf2 to the new position, execute (2),(3),(4) cyclically until all elements are transported to the correct position. (The following figure is the execution of (2), which is the beginning of the second iteration.)



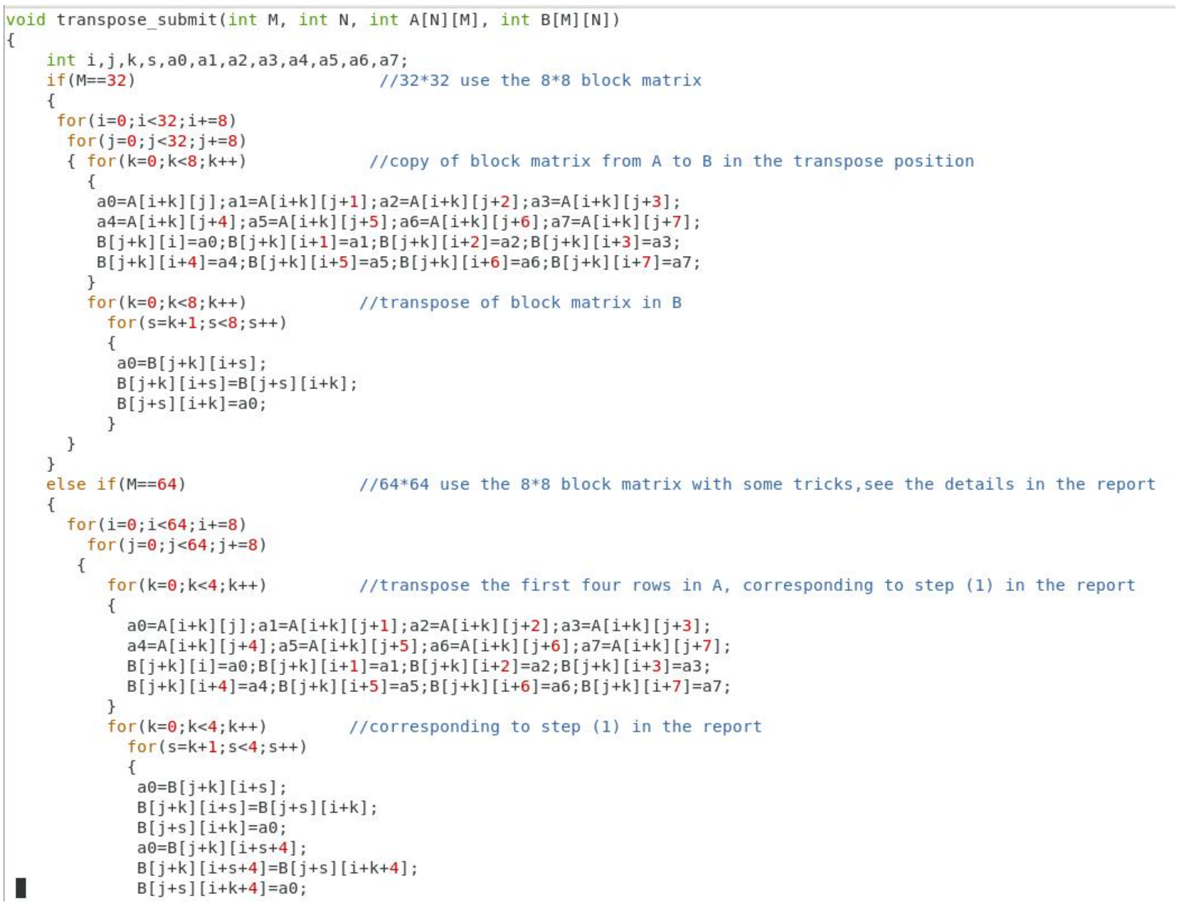
5. Select the size of block matrix. Size of block matrix will greatly influence the cache misses. After enough numerical experiments we select the best size 18×14.

2.2.2 Code

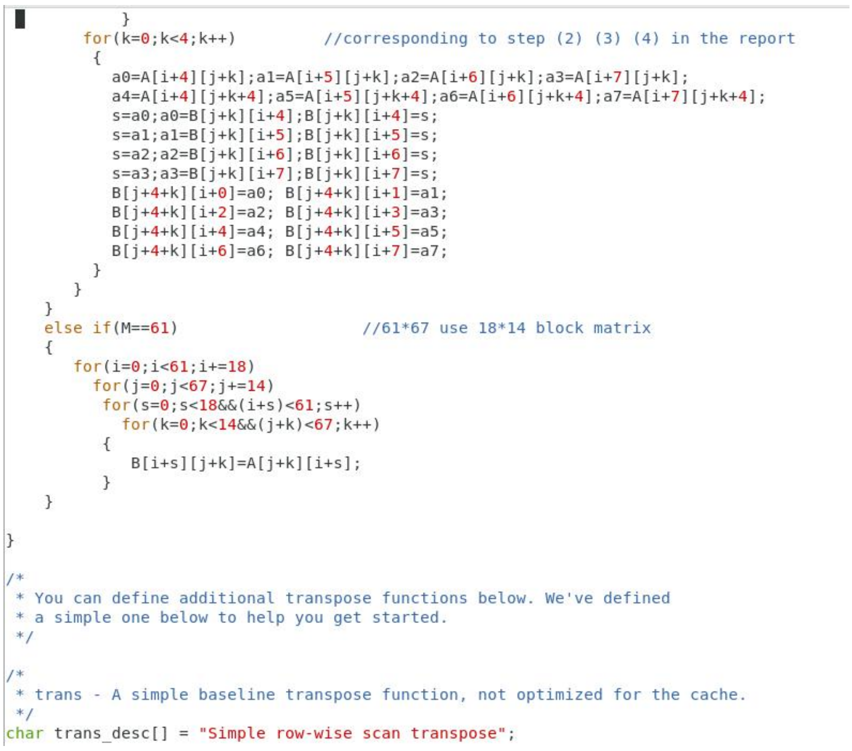
trans.c



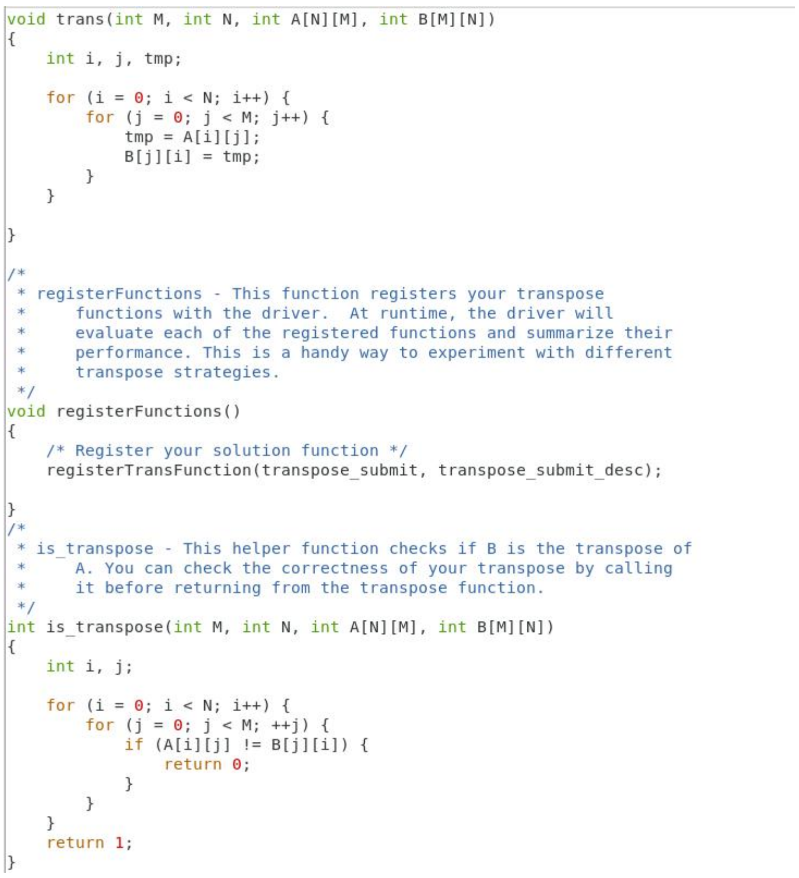
```
/*
 * trans.c - Matrix transpose B = A^T
 *
 * Each transpose function must have a prototype of the form:
 * void trans(int M, int N, int A[N][M], int B[M][N]);
 *
 * A transpose function is evaluated by counting the number of misses
 * on a 1KB direct mapped cache with a block size of 32 bytes.
 * Name: Shengyuan Hou  Student ID:518021910604
 */
#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";
```



```c
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,k,s,a0,a1,a2,a3,a4,a5,a6,a7;
    if(M==32)                         //32*32 use the 8*8 block matrix
    {
     for(i=0;i<32;i+=8)
      for(j=0;j<32;j+=8)
      { for(k=0;k<8;k++)            //copy of block matrix from A to B in the transpose position
        {
         a0=A[i+k][j];a1=A[i+k][j+1];a2=A[i+k][j+2];a3=A[i+k][j+3];
         a4=A[i+k][j+4];a5=A[i+k][j+5];a6=A[i+k][j+6];a7=A[i+k][j+7];
         B[j+k][i]=a0;B[j+k][i+1]=a1;B[j+k][i+2]=a2;B[j+k][i+3]=a3;
         B[j+k][i+4]=a4;B[j+k][i+5]=a5;B[j+k][i+6]=a6;B[j+k][i+7]=a7;
        }
        for(k=0;k<8;k++)           //transpose of block matrix in B
          for(s=k+1;s<8;s++)
          {
           a0=B[j+k][i+s];
           B[j+k][i+s]=B[j+s][i+k];
           B[j+s][i+k]=a0;
          }
      }
    }
    else if(M==64)                 //64*64 use the 8*8 block matrix with some tricks,see the details in the report
    {
      for(i=0;i<64;i+=8)
        for(j=0;j<64;j+=8)
       {
          for(k=0;k<4;k++)         //transpose the first four rows in A, corresponding to step (1) in the report
          {
            a0=A[i+k][j];a1=A[i+k][j+1];a2=A[i+k][j+2];a3=A[i+k][j+3];
            a4=A[i+k][j+4];a5=A[i+k][j+5];a6=A[i+k][j+6];a7=A[i+k][j+7];
            B[j+k][i]=a0;B[j+k][i+1]=a1;B[j+k][i+2]=a2;B[j+k][i+3]=a3;
            B[j+k][i+4]=a4;B[j+k][i+5]=a5;B[j+k][i+6]=a6;B[j+k][i+7]=a7;
          }
          for(k=0;k<4;k++)        //corresponding to step (1) in the report
            for(s=k+1;s<4;s++)
            {
             a0=B[j+k][i+s];
             B[j+k][i+s]=B[j+s][i+k];
             B[j+s][i+k]=a0;
             a0=B[j+k][i+s+4];
             B[j+k][i+s+4]=B[j+s][i+k+4];
             B[j+s][i+k+4]=a0;
```



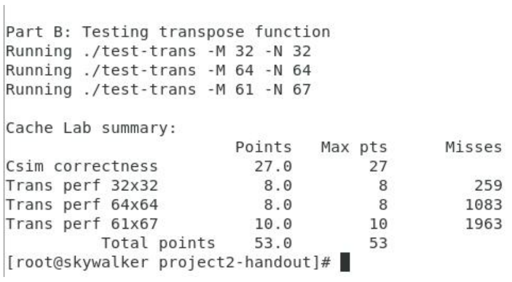
```c
            }
        for(k=0;k<4;k++)        //corresponding to step (2) (3) (4) in the report
         {
           a0=A[i+4][j+k];a1=A[i+5][j+k];a2=A[i+6][j+k];a3=A[i+7][j+k];
           a4=A[i+4][j+k+4];a5=A[i+5][j+k+4];a6=A[i+6][j+k+4];a7=A[i+7][j+k+4];
           s=a0;a0=B[j+k][i+4];B[j+k][i+4]=s;
           s=a1;a1=B[j+k][i+5];B[j+k][i+5]=s;
           s=a2;a2=B[j+k][i+6];B[j+k][i+6]=s;
           s=a3;a3=B[j+k][i+7];B[j+k][i+7]=s;
           B[j+4+k][i+0]=a0; B[j+4+k][i+1]=a1;
           B[j+4+k][i+2]=a2; B[j+4+k][i+3]=a3;
           B[j+4+k][i+4]=a4; B[j+4+k][i+5]=a5;
           B[j+4+k][i+6]=a6; B[j+4+k][i+7]=a7;
         }
       }
    }
    else if(M==61)                  //61*67 use 18*14 block matrix
    {
       for(i=0;i<61;i+=18)
         for(j=0;j<67;j+=14)
          for(s=0;s<18&&(i+s)<61;s++)
            for(k=0;k<14&&(j+k)<67;k++)
          {
             B[i+s][j+k]=A[j+k][i+s];
          }
    }

}

/*
 * 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";
```



```c
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);

}
/*
 * 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.
 */
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

Obviously we have passed the test because we get a full score on the autograders.



```
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         259
Trans perf 64x64           8.0         8        1083
Trans perf 61x67          10.0        10        1963
          Total points    53.0        53
[root@skywalker project2-handout]# █
```

3. Conclusion

3.1 Problems

(1)In part B, I meet with “Program timed out” error. It takes me tons of time to resolve it. At first I attribute it to the endless loop from my program, but afterwards I still cannot find out the error. Eventually I search the Internet and find that I ignore the hint in the documentation that this project should not be run in the shared folder with other system.
(2)In part A, I was confused about with which radix to express the tag index and do the calculation. Afterwards, I find that we could make use of ‘shift’ and ‘bitwise and’ operation to gain the corresponding tag index and set index.

3.2 Achievements

(1)We have a great solution because our experimental result is brilliant. 1083 cache misses in 64×64 matrix transpose and 259 cache misses in 32×32 matrix transpose are both significantly low, which excel most of the existing solutions.
(2)We take advantage of many practical and comprehensible techniques to improve the performance, such as cache blocking, 'copy first, transpose second' and 'Step by step'.
(3) We simplify part A by analyzing and integrating M,L,S operation into one cache_access function. It greatly simplifies my code and my code length is only about 160 lines, while the documentation declares that we have to write 200~300 lines of simulation code. This greatly enhances my code readability.
(4)Along this project, I get a deep understanding of cache structure and some corresponding parameters. Furthermore, I have learned more about cache write strategy and many blocking techniques.

4. Mathematical analysis of techniques in Part B

4.1normal blocking

Non-diagonal blocking matrix transpose produces 16 cache misses, because matrix A and B both have 8 new lines to import the corresponding cache line and they do not conflict with each other.

However, in diagonal blocking matrix when we execute B[n][n]=A[n][n], cache line for A should be replaced by B and B is replaced by A again because A will continue along the row.

The first line of B, B[0], is loaded into the cache for the first time, which should be counted in those 16 times, but the reload of A[0] will also occur, so the number of additional misses is 1. The last line A[7] is replaced, but the copy has been completed, there is no need to reload A[7] , so the additional miss is also 1.

After B[m] is replaced by A[m], when the next line A[m+1] is copied, B[m] needs to be reloaded into the cache (except the first line), so there will be one more miss in each line except the first line.

Diagonal_miss=16+8*2-2+7=37.

Total_miss=37*4+16*12+3(function call)=343.

4.2 cache blocking

Basic 16 misses will appear.

Since 8 elements in a line is previously stored in 8 temporary variables, there is no longer conflicts between A and B, but when copying A[n], B[n] will be replaced(except the first line), and B[n] will be reloaded when writing data to B[n]. This causes additional 7 misses.

Diagonal_miss=16+7=23.

Total_miss=23*4+16*12+3(function call)=287.

4.3 “copy first, transpose second”

Basic 16 misses will appear.

Each time, use 8 local variables to cache a line of block A, and then copy them to the line corresponding to block B. After the copy is completed, all of B's blocks are in the cache, and there is no miss in the transposition process.

Diagonal_miss=16.

Total_miss=16*16+3(function call)=259.

4.4”Step by step”

Basic 16 misses will appear(8 misses in step (1),4 misses in first step (2), 1 misses every iteration (2)->(3)->(4)->(5)(totally 4 iterations)).

Since we use “copy first, transpose second” strategy in step 1, diagonal and non-diagonal block matrices have the same count of misses in step (1).

In (2),(3),(4),(5) diagonal blocking matrix will lead to 7 additional misses.

Diagonal_miss=7+16=23.

Total_miss=7*8+64*16+3(function call)=1083.