

实验：Cachelab

一、实验目标

通过完成cachelab，以提高对计算机cache的理解和掌握（csim）和对其的优化方法（trans）。

二、实验思路

（一）、csim.c

第一部分的任务是编写C代码以模拟cache在计算机硬件层面的工作过程，主要是判断输入的地址在cache中的hit/miss/eviction情况。

既然要模拟cache，那么在一开始就要为代码开辟出相应大小的空间。根据输入的s(set)、E(line)、b(block)为模拟的cache分配空间。因为本题仅要求判断命中情况，而每次是访问一个set中的一line的，因此在分配空间时不需要考虑b。所以得到的应该是一个二维数组。又因为s表示的是set的二进制位数，故二维数组的行数应为 2^s ，所以在代码中可表示为 `s=1<<s;`，最后得到一个S*E的二维数组。

根据题意，该数组的每个元素应该存放一个标识tag。又因为此题还需考虑到eviction情况，而miss与eviction的主要区别就在于当该位置为空时，第一次访问该位置是miss，而不需要计数eviction。因此还需定义一个valid来判断该位置是否为空。此外，题目要求在冲突时，用LRU（最近最少使用）策略，为了找出同一个组中，哪个line未被访问的时间最长，还需要定义一个unused_times来记录未被访问的次数。因此定义结构体：

```
struct CACHE
{
    int valid;
    int tag;
    int unused_times;
};
```

接下来处理每个地址的命中情况。因为一个地址从左到右是由tag、set和block组成的，现在set和block位数已知，总地址长度已知（32），可知tag位数为 $32-s-b$ 。故可得tag的值 `int t_=address>>(s+b);` 和set的地址 `int s_=(address<<(32-b-s))>>(32-s);`

一个地址在cache中只有两种情况：hit和miss，而在本代码中判断它命中情况的办法便是遍历该set中的每一个line。若先判断是否miss，则在遍历所有line后才能得出结论，且还需使用一个标记来记录在遍历过程中是否命中，再根据这个进行之后的操作；而若先判断是否hit，则在其hit后可以直接返回，且当遍历完后，说明其必定miss，因此先判断是否hit更好。

若该位置不为空且tag一致，说明其hit，则将最后要打印的hit值加一，并将该位置的元素unused_times清零。

若没有hit，则再遍历每一line判断是否有空的位置以减少不必要的eviction，若有，则将miss值加一，并将其valid设为1表示非空，将tag设为t_，并将unused_times设为0。

若miss且所有位置都非空，除了将miss加一外，还需将eviction加一。遍历该set中所有line，找出unused_times最大的元素，将其tag赋值为t_，并将unused_times清零。

因此该部分代码为

```
void cache_simulator(unsigned int address)
{
```

```

int t_=address>>(s+b);
int s_=(address<<(32-b-s))>>(32-s);
for(int i=0;i<E;i++)          //case:hit
{
    if(cache_address[s_][i].tag==t_&&cache_address[s_][i].valid==1)
    {
        hit++;
        cache_address[s_][i].unused_times=0;
        return;
    }
}
for(int i=0;i<E;i++)          //case:not hit but miss
{
    if(cache_address[s_][i].valid==0)
    {
        miss++;
        cache_address[s_][i].tag=t_;
        cache_address[s_][i].valid=1;
        cache_address[s_][i].unused_times=0;
        return;
    }
}
miss++;                        //case:not hit but cache's full - eviction
int max=0;
int loc;
for(int i=0;i<E;i++)
{
    if(cache_address[s_][i].unused_times>max)
    {
        max=cache_address[s_][i].unused_times;
        loc=i;
    }
}
eviction++;
cache_address[s_][loc].tag=t_;
cache_address[s_][loc].unused_times=0;
return;
}

```

接下来处理输入的参数。 首先利用实验说明中给到的getopt函数处理输入的参数，因为输入的参数有s、E、b、t以及选择性输入的参数v因此该部分代码为：

```

while((initial=getopt(argc, argv, "vs:E:b:t:"))!=-1)
{
    switch(initial)
    {
        case 'v':
            v_show=1;
            break;
        case 's':
            s=atoi(optarg);
            break;
        case 'E':
            E=atoi(optarg);
            break;
    }
}

```

```

        case 'b':
            b=atoi(optarg);
            break;
        case 't':
            strcpy(file_address,optarg);
            break;
    }
}

```

为了能够实现像样例中的-v功能（即输出每个地址的hit/miss/eviction情况），还需定义一个名为output的字符串数组，并遍历文件中的所有行来确定该分配内存的大小（结束后需将文件指针指回第一行）。

```

while(fgets(input,100,fp))
    MAXIMUN++;
rewind(fp);
output=(char**)calloc(MAXIMUN,sizeof(char*));
for(int i=0;i<MAXIMUN;i++)
    output[i]=(char*)calloc(50,sizeof(char));

```

接下来处理文件。对于每次从文件读入的行input，是由命令、地址和大小组成，而根据题意，M相当于执行一次S和L指令，在代码层面相当于是S和L执行一次cache_simulator，而M执行两次。又因为输入中有杂项I，因此只要储存命令的值operation不为I，就执行一次cache_simulator，同时对output赋值为该输入行。执行完后，还需判断operation是否为M，若是，则再执行一次cache_simulator。

```

sscanf(input,"%s %x",&operation,&address);
if(operation!='I')
{
    strcpy(output[num],input);
    strtok(output[num],"\\n");
    output[num]=output[num]+1;
    cache_simulator(address,output[num]);
    num++;
}
if(operation=='M')
    cache_simulator(address,output[num-1]);

```

最后，更新unused_times，将所有非空的地址的unused_times加一

```

for(int i=0;i<S;i++)
{
    for(int j=0;j<E;j++)
    {
        if(cache_address[i][j].valid==1)
            cache_address[i][j].unused_times++;
    }
}

```

我们发现，在处理文件部分的代码的cache_simulator多了个参数output[num]，这是因为-v要求的输出是每个hit/miss/eviction情况，因此在cache_simulator判断地址的三种情况后还要对该情况进行记录，故将cache_simulator函数更新为：

```

void cache_simulator(unsigned int address,char* output)
{

```

```

int t_=address>>(s+b);
int s_=(address<<(32-b-s))>>(32-s);
for(int i=0;i<E;i++)          //case:hit
{
    if(cache_address[s_][i].tag==t_&&cache_address[s_][i].valid==1)
    {
        hit++;
        strcat(output," hit"); //记录hit情况
        cache_address[s_][i].unused_times=0;
        return;
    }
}
for(int i=0;i<E;i++)          //case:not hit but miss
{
    if(cache_address[s_][i].valid==0)
    {
        miss++;
        strcat(output," miss"); //记录miss情况
        cache_address[s_][i].tag=t_;
        cache_address[s_][i].valid=1;
        cache_address[s_][i].unused_times=0;
        return;
    }
}
miss++;                      //case:not hit but cache's full - eviction
strcat(output," miss");
int max=0;
int loc;
for(int i=0;i<E;i++)
{
    if(cache_address[s_][i].unused_times>max)
    {
        max=cache_address[s_][i].unused_times;
        loc=i;
    }
}
eviction++;
strcat(output," eviction"); //记录eviction情况
cache_address[s_][loc].tag=t_;
cache_address[s_][loc].unused_times=0;
return;
}

```

实验完整代码见文末

(二)、trans.c

根据输入，s E b分别为5 1 5，算出共有32个set，每个set只有一个line（直接映射），一个line（set）的大小为32字节，因为一个int变量占4个字节的大小，故一个line（set）可以存放8个int类型的变量。

首先处理32*32的情况。首先考虑到，在执行B[0][0]=A[0][0]时，该两个元素对应的是同一个set，因此cache会先读取A[0][0]及之后的七个元素，再用B[0][0]及之后的七个元素将其覆盖掉，那么在下一次读取A[0][1]时，cache又会重新覆盖掉B[0][0]，这就造成了两次冲突不命中。因此可以先将A[0][0]到A[0][7]的八个元素全都读出来，放入寄存器中，之后可用B覆盖掉他们。如下

```

if(M==32&&N==32)
{
    for(i=0;i<32;i++)
    {
        for(j=0;j<32;j+=8)
        {
            for(k=j;k<j+8;k++)
                tmp[k-j]=A[i][k];
            for(k=j;k<j+8;k++)
                B[k][i]=tmp[k-j];
        }
    }
}

```

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:898, misses:1157, evictions:1125

但发现miss率只比直接赋值降低了32（样例的misses值为1189）。

通过计算得出该数组一行需要占用4个set，一个cache最多可以存放数组8行的元素。尽管对于A来说，列号从0~8,9~15.....都在一个set中，不会有不命中的情况，但对B来说，k对应的是他的行号，当k从7变成8后，B[8]行会覆盖掉B[0]行，那么B[0]行其实只被用到了B[0][0]元素之后就被覆盖掉了，对于该行的利用率极低。因此该方法与样例中的直接赋值并无太大区别，只在于开头所说的对于小部分情况两次不命中和一次不命中的区别，这也是为什么miss率降低不多的原因。

所以，只要提高B的行的利用率，miss率就能够有极大的下降。我们除了要对B[0][0]元素进行赋值，还要对相同set中的B[0][1]、B[0][2].....B[0][7]进行赋值而不让B[7]覆盖掉该部分。因此可以限定i，八个为一循环，当将B所在块的八个元素全部赋值完后，再将j+8，进行下一轮的赋值。

```

if(M==32&&N==32)
{
    for(i=0;i<32;i+=8)
    {
        for(j=0;j<32;j+=8)
        {
            for(a=i;a<i+8;a++)
            {
                for(b=j;b<j+8;b++)
                    B[b][a]=A[a][b];
            }
        }
    }
}

```

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:1710, misses:345, evictions:313

这比之前的一种方法有了极大的提升，但未达到300以下。考虑到是当i等于j时造成一开始所说的两次冲突不命中，因此可以将两种方法相结合：

```

if(M==32&&N==32)

```

```

{
    for(i=0;i<32;i+=8)
    {
        for(j=0;j<32;j+=8)
        {
            for(a=i;a<i+8;a++)
            {
                if(i==j)
                {
                    for(b=j;b<j+8;b++)
                        tmp[b-j]=A[a][b];
                    for(b=j;b<j+8;b++)
                        B[b][a]=tmp[b-j];
                }
                else
                {
                    for(b=j;b<j+8;b++)
                        B[b][a]=A[a][b];
                }
            }
        }
    }
}

```

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:289, evictions:257

之后处理64*64的情况。当数组扩展之后，储存一行需要用到8个块，一个cache也只能储存最多4行的内容，故一开始类比32*32的情况：

```

void test_64(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,a,b,tmp[4];
    for(i=0;i<64;i+=4)
    {
        for(j=0;j<64;j+=4)
        {
            for(a=i;a<i+4;a++)
            {
                if(i==j)
                {
                    for(b=j;b<j+4;b++)
                        tmp[b-j]=A[a][b];
                    for(b=j;b<j+4;b++)
                        B[b][a]=tmp[b-j];
                }
                else
                {
                    for(b=j;b<j+4;b++)
                        B[b][a]=A[a][b];
                }
            }
        }
    }
}

```

```
Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (test 64*64): hits:6402, misses:1797, evictions:1765
```

因为当j由0变成4后，B[4]行也会覆盖A[0]行，故在前一个i==j条件下在增加i-j==4 || j-i==4:

```
void test_64(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,a,b,tmp[4];
    for(i=0;i<64;i+=4)
    {
        for(j=0;j<64;j+=4)
        {
            for(a=i;a<i+4;a++)
            {
                if(i==j || i-j==4 || j-i==4)
                {
                    for(b=j;b<j+4;b++)
                        tmp[b-j]=A[a][b];
                    for(b=j;b<j+4;b++)
                        B[b][a]=tmp[b-j];
                }
                else
                {
                    for(b=j;b<j+4;b++)
                        B[b][a]=A[a][b];
                }
            }
        }
    }
}
```

```
Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (test 64*64): hits:6498, misses:1701, evictions:1669
```

发现miss率只降低了一点。

一个块中可存8个int，但因为现在循环范围缩小到了4，故对于A来说，每块的后四个没有被用上，造成浪费。

若将j的范围扩展成8，则依次循环可用上A[i][j]~A[i][j+8]，即该八个里面只有一个强制不命中。

但对应的，B[j+4][i]会将B[j][i]覆盖掉，因此要先对j~j+3的部分赋完值后再赋j+4~j+7的部分，代码如下：

```
void test_64(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,a,b,tmp[4];
    for(i=0;i<64;i+=4)
    {
        for(j=0;j<64;j+=8)
        {
```

```

        for(a=i;a<i+4;a++)
        {
            if(i==j || i-j==4 || j-i==4)
            {
                for(b=j;b<j+4;b++)
                    tmp[b-j]=A[a][b];
                for(b=j;b<j+4;b++)
                    B[b][a]=tmp[b-j];
            }
            else
            {
                for(b=j;b<j+4;b++)
                    B[b][a]=A[a][b];
            }
        }
        for(a=i;a<i+4;a++)
        {
            for(b=j+4;b<j+8;b++)
                B[b][a]=A[a][b];
        }
    }
}

```

Function 0 (1 total)

Step 1: Validating and generating memory traces

Step 2: Evaluating performance (s=5, E=1, b=5)

func 0 (test 64*64): hits:6402, misses:1797, evictions:1765

miss率反而变回一开始的情况。

分析：B的一行被读入两次(A[i][j+4]和A[i+4][j+4]对B[j+4]行赋值时)造成miss率增加。

将i的范围也扩成8。使得B[j+4]行能在一次循环内被赋值完成。

但A[i][j+4]和A[i+4][j+4]的对应位置相同也会造成冲突不命中，miss概率还是会增加。且虽然A[i]行所在块的元素都能被使用，但A[i+4]行所在块的元素只有前四个会在该次循环中被使用。

矛盾点：要完成对B[j+4]行在一个块中所有元素的赋值，既要用到A[i]行，又要用到A[i+4]行，且两行会产生冲突。

8 0	8 1	8 2	8 3	8 4	8 5	8 6	8 7
0 8	0 9	0 10	0 11	0 12	0 13	0 14	0 15
9 0	9 1	9 2	9 3	9 4	9 5	9 6	9 7
1 8	1 9	1 10	1 11	1 12	1 13	1 14	1 15
10 0	10 1	10 2	10 3	10 4	10 5	10 6	10 7
2 8	2 9	2 10	2 11	2 12	2 13	2 14	2 15
11 0	11 1	11 2	11 3	11 4	11 5	11 6	11 7
3 8	3 9	3 10	3 11	3 12	3 13	3 14	3 15

以该图为例（用一行来模拟一块，整个图为cache 的大小，数字分别为行号和列号），当 $i=0$ ， $j=8$ 时。黄色所在行为A的块，蓝色的为B。第一次赋完值后，被用掉的部分已用黄色和蓝色标记。要利用A的每行后四个元素，则B所在的行 $B[j]$ 都会被替换成 $B[j+4]$ ，即如下：

12 0	12 1	12 2	12 3	12 4	12 5	12 6	12 7
0 8	0 9	0 10	0 11	0 12	0 13	0 14	0 15
13 0	13 1	13 2	13 3	13 4	13 5	13 6	13 7
1 8	1 9	1 10	1 11	1 12	1 13	1 14	1 15
14 0	14 1	14 2	14 3	14 4	14 5	14 6	14 7
2 8	2 9	2 10	2 11	2 12	2 13	2 14	2 15
15 0	15 1	15 2	15 3	15 4	15 5	15 6	15 7
3 8	3 9	3 10	3 11	3 12	3 13	3 14	3 15

此时A所在行全部被使用，但为了让空白部分全部被赋值，还需加入A[i+4]行：

8 0	8 1	8 2	8 3	8 4	8 5	8 6	8 7		12 0	12 1	12 2	12 3	12 4	12 5	12 6	12 7
0 8	0 9	0 10	0 11	0 12	0 13	0 14	0 15		4 8	4 9	4 10	4 11	4 12	4 13	4 14	4 15
9 0	9 1	9 2	9 3	9 4	9 5	9 6	9 7		13 0	13 1	13 2	13 3	13 4	13 5	13 6	13 7
1 8	1 9	1 10	1 11	1 12	1 13	1 14	1 15		5 8	5 9	5 10	5 11	5 12	5 13	5 14	5 15
10 0	10 1	10 2	10 3	10 4	10 5	10 6	10 7		14 0	14 1	14 2	14 3	14 4	14 5	14 6	14 7
2 8	2 9	2 10	2 11	2 12	2 13	2 14	2 15		6 8	6 9	6 10	6 11	6 12	6 13	6 14	6 15
11 0	11 1	11 2	11 3	11 4	11 5	11 6	11 7		15 0	15 1	15 2	15 3	15 4	15 5	15 6	15 7
3 8	3 9	3 10	3 11	3 12	3 13	3 14	3 15		7 8	7 9	7 10	7 11	7 12	7 13	7 14	7 15

且发现A[i+4]行的前四个元素（右边部分标灰的地方）可以为之前第一张图中的B的后四行（左边部分）赋值。

为了解决这种需要“互补”的情况，可以先对一开始的B[j]行全部完成赋值，但A[i]行就会出现元素（每行后四个）丢失而无法被利用的情况。但是在第一轮赋值时，B的一块中每后四个元素是没有被用到的，故可以暂时将A的后四个元素存放到这些地方，且不会发生miss。

之后再用A[i+4]覆盖A[i]，并将B每行后四个元素取出并存入寄存器中，用A[i+4]对其赋正确的值，此时，该行已被全部赋值，可认为是“没用”的了。因此，用B[j+4]覆盖B[j]行，用取出的四个值和A[i+4]中的四个值完成对B[j+4]行的赋值。

为了防止产生像32*32时i和j相同而造成冲突不命中的情况，这里全部都用寄存器作为中间变量。

```

void test_64(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,a,b,tmp[8];
    for(i=0;i<N;i+=8)
    {
        for(j=0;j<M;j+=8)
        {
            for(a=i;a<i+4;a++)
            {
                for(b=j;b<j+8;b++)           //取出A[i]一行所在块的所有八个元素
                    tmp[b-j]=A[a][b];
                for(b=j;b<j+4;b++)           //将它们赋值给对应的B[j]行
                {
                    B[b][a]=tmp[b-j];
                    B[b][a+4]=tmp[b+4-j];
                }
            }
            for(b=j;b<j+4;b++)
            {
                for(a=i;a<i+4;a++)           //用A[i+4]行覆盖A[i]行并取出前四个元
                //{
                    tmp[a-i]=A[a+4][b];
                for(a=i;a<i+4;a++)           //取出B[j]行后四个储存的有关A的元素
                    tmp[a+4-i]=B[b][a+4];
                //}
                for(a=i;a<i+4;a++)
                    B[b][a+4]=tmp[a-i];           //将B[j]行后四个用A[i+4]的元素赋值完
                for(a=i;a<i+4;a++)
                    B[b+4][a]=tmp[a+4-i];       //用B[j+4]行覆盖B[j]行并对前四个元素
                //}
            }
            for(a=i+4;a<i+8;a++)
            {
                for(b=j+4;b<j+8;b++)           //取出A[i+4]行的后四个元素
                //{
                    tmp[b-j]=A[a][b];
                for(b=j+4;b<j+8;b++)           //对B[j+4]行最后的四个元素赋值
                    B[b][a]=tmp[b-j];
                //}
            }
        }
    }
}

```

```

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (test 64*64): hits:9066, misses:1181, evictions:1149

```

在过程中发现，原本在代码中注释掉的括号之间的可以用一个循环完成，但当i等于j时该方法会出现冲突不命中而提高miss率，故将其分开写后，前一部分的循环完成后对应的值已被存入寄存器中，因此可以被舍弃，不影响后一半循环的值被存入块中。修改前的结果：

```
Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (test 64*64): hits:8970, misses:1277, evictions:1245
```

最后考虑61*67的情况。因为61和67都是质数，且与8、4之类的没有什么关联，故只能“死马当活马医”，先套用32*32的情况(8*8)：

```
void test_61_0(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,a,b;
    for(i=0;i<N;i+=8)
    {
        for(j=0;j<M;j+=8)
        {
            for(a=i;a<N&&a<i+8;a++)
            {
                for(b=j;b<M&&b<j+8;b++)
                    B[b][a]=A[a][b];
            }
        }
    }
}
```

```
Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (test 61*67): hits:5954, misses:2225, evictions:2193
```

发现miss率已经接近要求，故又套用64*64的分块情况（4*4）：

```
void test_61_1(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,a,b;
    for(i=0;i<N;i+=4)
    {
        for(j=0;j<M;j+=4)
        {
            for(a=i;a<N&&a<i+4;a++)
            {
                for(b=j;b<M&&b<j+4;b++)
                    B[b][a]=A[a][b];
            }
        }
    }
}
```

```
Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (test 61*67): hits:5736, misses:2443, evictions:2411
```

miss率上升，考虑到可能分块比较大时miss率会下降，故使用16*16的分块方式：

```
void test_61_2(int M, int N, int A[N][M], int B[M][N])
```

```

{
    int i,j,a,b;
    for(i=0;i<N;i+=16)
    {
        for(j=0;j<M;j+=16)
        {
            for(a=i;a<N&&a<i+16;a++)
            {
                for(b=j;b<M&&b<j+16;b++)
                    B[b][a]=A[a][b];
            }
        }
    }
}

```

```

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (test 61*67): hits:6171, misses:2008, evictions:1976

```

已经基本接近2000，再使用18*18的分块：

```

void test_61_2(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,a,b;
    for(i=0;i<N;i+=18)
    {
        for(j=0;j<M;j+=18)
        {
            for(a=i;a<N&&a<i+18;a++)
            {
                for(b=j;b<M&&b<j+18;b++)
                    B[b][a]=A[a][b];
            }
        }
    }
}

```

```

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (test 61*67): hits:6207, misses:1972, evictions:1940

```

满足要求。

三、实验结果截图

```

2018202196@VM-0-46-ubuntu: ~/cachelab-handout
2018202196@VM-0-46-ubuntu:~/cachelab-handout$ ./driver.py
Part A: Testing cache simulator
Running ./test-csim

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      289
Trans perf 64x64     8.0     8     1181
Trans perf 61x67    10.0    10    1963
Total points    53.0    53

2018202196@VM-0-46-ubuntu:~/cachelab-handout$

```

四、实验完整代码

(一)、csim.c

```

#include "cachelab.h"
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<getopt.h>
#include<unistd.h>

struct CACHE
{
    int valid;
    int tag;
    int unused_times;
};

struct CACHE** cache_address;
int s,E,b;
char file_address[100];
char** output;
int hit=0,miss=0,eviction=0;

void cache_simulator(unsigned int address,char* output)
{
    int t_=address>>(s+b);
    int s_=(address<<(32-b-s))>>(32-s);
    for(int i=0;i<E;i++)          //case:hit
    {
        if(cache_address[s_][i].tag==t_&&cache_address[s_][i].valid==1)
        {
            hit++;
            strcat(output," hit");
            cache_address[s_][i].unused_times=0;
            return;
        }
    }
    for(int i=0;i<E;i++)          //case:not hit but miss
    {

```

```

        if(cache_address[s_][i].valid==0)
        {
            miss++;
            strcat(output," miss");
            cache_address[s_][i].tag=t_;
            cache_address[s_][i].valid=1;
            cache_address[s_][i].unused_times=0;
            return;
        }
    }
    miss++; //case: not hit but cache's full - eviction
    strcat(output," miss");
    int max=0;
    int loc;
    for(int i=0;i<E;i++)
    {
        if(cache_address[s_][i].unused_times>max)
        {
            max=cache_address[s_][i].unused_times;
            loc=i;
        }
    }
    eviction++;
    strcat(output," eviction");
    cache_address[s_][loc].tag=t_;
    cache_address[s_][loc].unused_times=0;
    return;
}

int main(int argc,char* argv[])
{
    int MAXIMUN=0;
    int v_show=0;
    char initial,input[60];
    while((initial=getopt(argc, argv, "vs:E:b:t:"))!=-1)
    {
        switch(initial)
        {
            case 'v':
                v_show=1;
                break;
            case 's':
                s=atoi(optarg);
                break;
            case 'E':
                E=atoi(optarg);
                break;
            case 'b':
                b=atoi(optarg);
                break;
            case 't':
                strcpy(file_address,optarg);
                break;
        }
    }
    FILE* fp=fopen(file_address,"r");
    if(fp==NULL)
    {

```

```

        printf("Filename Wrong!\n");
        return 0;
    }
    int S=1<<s;
    cache_address=(struct CACHE**)calloc(S,sizeof(struct CACHE*));
    for(int i=0;i<S;i++)
        cache_address[i]=(struct CACHE*)calloc(E,sizeof(struct CACHE));
    for(int i=0;i<S;i++)
    {
        for(int j=0;j<E;j++)
        {
            cache_address[i][j].unused_times=0;
            cache_address[i][j].valid=0;
        }
    }
    while(fgets(input,100,fp))
        MAXIMUN++;
    rewind(fp);
    output=(char**)calloc(MAXIMUN,sizeof(char*));
    for(int i=0;i<MAXIMUN;i++)
        output[i]=(char*)calloc(50,sizeof(char));
    char operation;
    unsigned int address;
    int num=0;
    while(fgets(input,100,fp))
    {
        sscanf(input,"%s %x",&operation,&address);
        if(operation!='I')
        {
            strcpy(output[num],input);
            strtok(output[num],"\\n");
            output[num]=output[num]+1;
            cache_simulator(address,output[num]);
            num++;
        }
        if(operation=='M')
        {
            cache_simulator(address,output[num-1]);
        }
        for(int i=0;i<S;i++)
        {
            for(int j=0;j<E;j++)
            {
                if(cache_address[i][j].valid==1)
                    cache_address[i][j].unused_times++;
            }
        }
    }
    fclose(fp);
    if(v_show==1)
    {
        for(int i=0;i<num;i++)
            printf("%s\\n",output[i]);
    }
    printSummary(hit,miss,eviction);
    for(int i=0;i<S;i++)
        free(cache_address[i]);
    free(cache_address);

```



```

    free(output);
    return 0;
}

```

(二) 、trans.c

```

void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,a,b,tmp[8];
    if(M==32&&N==32)
    {
        for(i=0;i<32;i+=8)
        {
            for(j=0;j<32;j+=8)
            {
                for(a=i;a<i+8;a++)
                {
                    if(i==j)
                    {
                        for(b=j;b<j+8;b++)
                            tmp[b-j]=A[a][b];
                        for(b=j;b<j+8;b++)
                            B[b][a]=tmp[b-j];
                    }
                    else
                    {
                        for(b=j;b<j+8;b++)
                            B[b][a]=A[a][b];
                    }
                }
            }
        }
    }
    if(M==64&&N==64)
    {
        for(i=0;i<N;i+=8)
        {
            for(j=0;j<M;j+=8)
            {
                for(a=i;a<i+4;a++)
                {
                    for(b=j;b<j+8;b++)
                        tmp[b-j]=A[a][b];
                    for(b=j;b<j+4;b++)
                    {
                        B[b][a]=tmp[b-j];
                        B[b][a+4]=tmp[b+4-j];
                    }
                }
                for(b=j;b<j+4;b++)
                {
                    for(a=i;a<i+4;a++)
                        tmp[a-i]=A[a+4][b];
                    for(a=i;a<i+4;a++)
                        tmp[a+4-i]=B[b][a+4];
                    for(a=i;a<i+4;a++)

```

```

        B[b][a+4]=tmp[a-i];
        for(a=i;a<i+4;a++)
            B[b+4][a]=tmp[a+4-i];
    }
    for(a=i+4;a<i+8;a++)
    {
        for(b=j+4;b<j+8;b++)
            tmp[b-j]=A[a][b];
        for(b=j+4;b<j+8;b++)
            B[b][a]=tmp[b-j];
    }
}
}
}
if(M==61&&N==67)
{
    for(i=0;i<N;i+=18)
    {
        for(j=0;j<M;j+=18)
        {
            for(a=i;a<N&&a<i+18;a++)
            {
                for(b=j;b<M&&b<j+18;b++)
                    B[b][a]=A[a][b];
            }
        }
    }
}
}
}
}

```