CS359:计算机体系结构实验 Cache lab

一.实验目的

本次实验 PartA 要求写一个高速缓存 Cache 模拟器, PartB 要求通过优化矩阵转置达到尽可能少的 miss，使得 cache 的性能最优。

二.实验 PartA:

2.1 具体要求：

写一个 cache 模拟器在 csim.c 程序中，并使用 vargrind memory trace 文件作为输入，模拟 cache 的 hit 与 miss 行为，并输出所有的 hit,miss 以及 eviction 次数。

2.2 实验步骤：

2.2.1 首先使用 vargrind 查看 filename.trace 文件内容格式，如:

```
cachelab-handout$ valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

可以在输出窗口中看到 filename.trace 文件格式如下:

```
I  04e85f9b,4
I  04e85f9f,3
I  04e85fa2,2
I  04e85fa4,2
I  04e85fa6,5
 S ffefff928,8
I  04f18710,3
I  04f18713,7
 L 0520fe78,8
I  04f1871a,6
I  04f18720,5
I  04f18725,2
I  04f18740,3
I  04f18743,3
I  04f18746,2
==4975==
==4975== Counted 0 calls to main()
==4975==
==4975== Jccs:
==4975==   total:         170,807
==4975==   taken:         71,946 (42%)
==4975==
==4975== Executed:
==4975==   SBs entered:   181,282
==4975==   SBs completed: 118,625
==4975==   guest instrs:  959,708
==4975==   IRStmts:       5,715,409
==4975==
==4975== Ratios:
==4975==   guest instrs : SB entered  = 52 : 10
==4975==        IRStmts : SB entered  = 315 : 10
==4975==        IRStmts : guest instr = 59 : 10
==4975==
==4975== Exit code:       0
```

即：[space]operation address,size

相应的字符代表："I"取指令，"L"取数据，"S"存数据，"M"数据移动（数据取后再存）；中间的十六进制数 address 代表相应的内存地址，size 操作访问的 byte 数量。

2.2.2 依据参考的二进制可执行文件 csim-ref 执行后的输出编写自己的 csim.c 程序。

★基本参数

命令行中的信息：

h 表示打印出 usage 信息，v 表示打印出详细的 trace 信息，s 的参数表示 cache 的组索引位数，E 的参数表示 cache 的关联度，b 的参数表示块大小的偏移位，t 的参数表示输入文件名。

关键的几个函数：

1. 使用结构体嵌套模拟 cache 空间

```
typedef struct{
    int valid;
    int tag;
    int lru_flag;
}line;
typedef struct{
    line* cache_line;
}set;
typedef struct{
    set* sets;
    int set_number;
    int line_number;
}cache;
```

2. 获取组索引函数（通过移位以及逻辑与来实现）

```
long get_set_index(long address){return (address>>b)&((0x1<<s)-1);}
```

3. 获取标记位（通过移位实现）

```
long get_tag(long address){return address>>(s+b);}
```

4. 获取命令行参数（利用 C 语言的自带的 getopt()函数获取命令行参数）

```
while((temp=getopt(argc,argv,"hvs:E:b:t:"))!=-1)
```

5. 判断是否命中函数（比较有效位 valid 以及标记位 tag）

```
bool data_hit(cache *my_cach int __unnamed_struct_490e_3::line_number
    for(int i=0;i<my_cache->line_number;++i){
        if(my_cache->sets[set_ind].cache_line[i].valid==1&&my_cache->sets[set_ind].cache_line[i].tag==ln_tag){
            update_cache_lru_flag(my_cache,set_ind,ln_tag,i);
            return 1;
        }
    }
    return 0;
}
```

6. Lru 替换算法函数 （cache 中没加入一个新的元素就赋给它一个替换最后替换等级初值，其他元素的等级减一，每次选出最小等级元素将其替换）

```
int choose_replaced_cache_by_lru(cache *my_cache,int set_ind,int ln_tag){
    int min_tag=1000000;
    int rep_item;
    for(int i=0;i<my_cache->line_number;++i){
        if(my_cache->sets[set_ind].cache_line[i].lru_flag<min_tag){
            min_tag=my_cache->sets[set_ind].cache_line[i].lru_flag;
            rep_item=i;
        }
    }
    return rep_item;
}
void update_cache_lru_flag(cache *my_cache,int set_ind,int ln_tag,int rep_cache){
    my_cache->sets[set_ind].cache_line[rep_cache].lru_flag=999999;
    for(int i=0;i<my_cache->line_number;++i){
        if(i!=rep_cache){
            --my_cache->sets[set_ind].cache_line[i].lru_flag;
        }
    }
}
```

7. Cache 更新函数(如果命中，只需要更新命中元素的替换等级，不命中，检查该元素对应的组 cache 是否有空位，如果有就放

入空位置，没有则选择该替换的元素）

```
bool update_cache(cache *my_cache,int set_ind,int ln_tag){
    //judge cache full
    bool full_flag=1;
    int empty_location=-1;
    int eviction_flag=0;
    for(int i=0;i<my_cache->line_number;++i){
        if(my_cache->sets[set_ind].cache_line[i].valid==0){
            full_flag=0;//unfull
            empty_location=i;
            break;
        }
    }
    //unfull
    if(full_flag==0){
        my_cache->sets[set_ind].cache_line[empty_location].valid=1;
        my_cache->sets[set_ind].cache_line[empty_location].tag=ln_tag;
        update_cache_lru_flag(my_cache,set_ind,ln_tag,empty_location);
    }
    //full
    else{
        if(!data_hit(my_cache,set_ind,ln_tag)){
           eviction_flag=1;
        }
        int rep_cache_lru=choose_replaced_cache_by_lru(my_cache,set_ind,ln_tag);
        my_cache->sets[set_ind].cache_line[rep_cache_lru].valid=1;
        my_cache->sets[set_ind].cache_line[rep_cache_lru].tag=ln_tag;
        update_cache_lru_flag(my_cache,set_ind,ln_tag,rep_cache_lru);
    }
    return eviction_flag;
}
```

8. 数据指令存取（利用以上编写好的函数进行 miss 与 hit 判断）

★两者输出结果比较如下：

csim-ref: 执行./cism-ref 后出现 usage 信息

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-handout$ ./csim-ref
./csim-ref: Missing required command line argument
Usage: ./csim-ref [-hv] -s <num> -E <num> -b <num> -t <file>
Options:
  -h          Print this help message.
  -v          Optional verbose flag.
  -s <num>    Number of set index bits.
  -E <num>    Number of lines per set.
  -b <num>    Number of block offset bits.
  -t <file>   Trace file.

Examples:
  linux>  ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
  linux>  ./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace
```

csim: 执行./cism 后出现 usage 信息

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-handout$ ./csim
Usage: ./csim [-hv] -s <num> -E <num> -b <num> -t <file>
Options:
-h         Print this help message.
-v         Optional verbose flag.
-s <num>   Number of set index bits.
-E <num>   Number of lines per set.
-b <num>   Number of block offset bits.
-t <file>  Trace file.


Examples:
linux>  ./csim -s 4 -E 1 -b 4 -t traces/yi.trace
linux>  ./csim -v -s 8 -E 2 -b 4 -t traces/yi.trace
```

csim-ref: 执行./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-handout$ ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

csim: 执行./csim -s 4 -E 1 -b 4 -t traces/yi.trace

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-handout$ ./csim -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

csim-ref: 执行./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-handout$ ./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss
L 210,1 miss
M 12,1 hit hit
hits:5 misses:4 evictions:0
```

csim: 执行./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-handout$ ./csim -v -s 8 -E 2 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss
L 210,1 miss
M 12,1 hit hit
hits:5 misses:4 evictions:0
```

执从以上的 csim-ref 与 csim 执行结果看，两者的基本模拟效果相同，接下来还需要使用./driver.py 测试自己写的 csim.c 真正是否正确。

执行./driver.py 对自己写的 cache 模拟器进行测试得出一下结果

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-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
```

测试能够得出 27 分，说明自己的 cache 基本上到达模拟要求。

三.实验 PartB:

2.1 具体要求：

编写一个矩阵转置函数，使得函数再调用过程达到 miss 数尽可能的少，实现最优化的矩阵转置。

实验进行了三组不同大小的矩阵进行测试，分别是：

32×32（M=32,N=32）矩阵

64×64（M=64,N=64）矩阵

61×67（M=61,N=67）矩阵

根据实验提供的 cache 参数（s=5,E=1,b=5）可以知道这是一个关联度为 1 的有 32 组的直接映射 cache，并且 cache 每个块的大小为 32byte 即 8int 类型大小。

2.3 实验步骤：

2.3.1 32×32（M=32,N=32）矩阵

首先如果不对矩阵进行分块，A[0][0]转置到 B[0][0]后，A[0][0]所在的块就会被 B[0][0]所在块替换掉，导致访问 A[0][1]就会 miss，这

样一来 miss 率等于 1.而根据要求 miss 率基本上得为 1/8 才会达到要求，所以我们必须跟换策略，使用分块方式降低 miss 率。

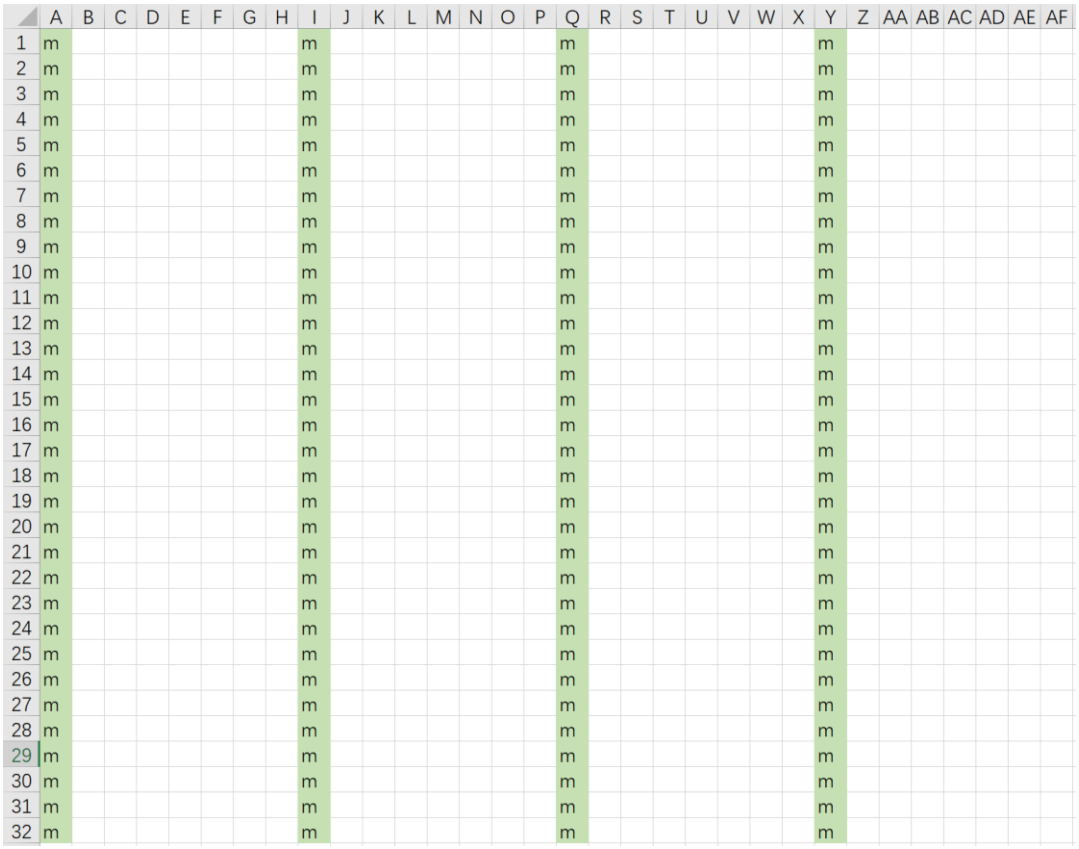
根据 cache 的块大小为 8int，我们可以将矩阵分成 8×8 矩阵。

每次使用中间变量将从矩阵中取出 8 个数据存好，然后存好得这个数据对应的块就不会被再访问到，就不会导致 miss；代码如下：

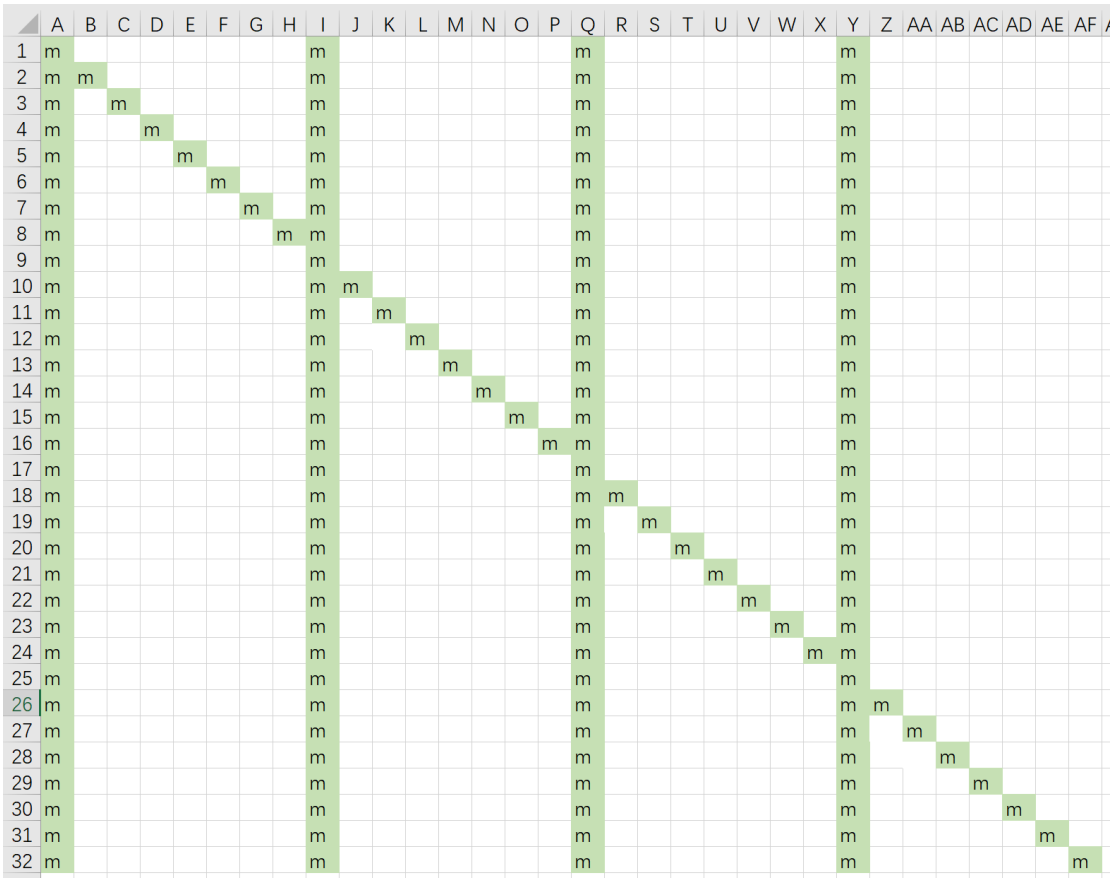
```
for(k=i;k<i+8;++k){
    temp0=A[k][j];
    temp1=A[k][j+1];
    temp2=A[k][j+2];
    temp3=A[k][j+3];
    temp4=A[k][j+4];
    temp5=A[k][j+5];
    temp6=A[k][j+6];
    temp7=A[k][j+7];
    B[j][k]=temp0;
    B[j+1][k]=temp1;
    B[j+2][k]=temp2;
    B[j+3][k]=temp3;
    B[j+4][k]=temp4;
    B[j+5][k]=temp5;
    B[j+6][k]=temp6;
    B[j+7][k]=temp7;
}
```

这样 A[N][M]矩阵 miss 率就达到 1/8；B[N][M]由于在去 A[k][k]对角线元素时，在得到对角线元素时必须先从 A 矩阵中取，所以此前在 cache 中的对角线对应的块就会被 A 的相应的对角线块替换掉，下次再给 B 对角线上元素复制的时候就会导致 miss；虽然这样能够得到 miss 数位 287<300；但是还可以优化。

Miss 数为 287 情况下矩阵 A miss 情况：

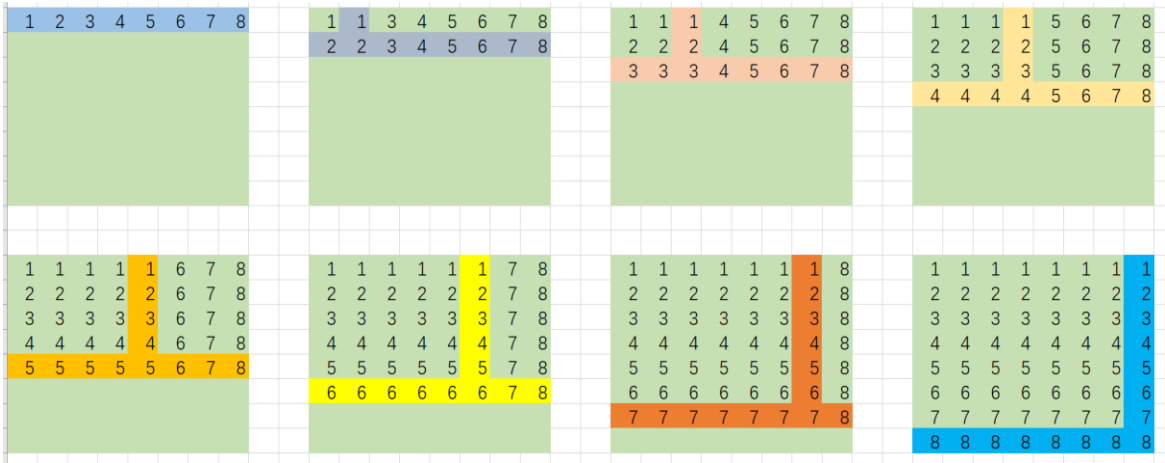


矩阵 B 的 miss 情况：



消除 B 矩阵对角线 miss 对于处于对角线上的 8×8 的矩阵块，我们进行以下策略：先将 A 矩阵的一行赋值给 temp 变量，然后用 B 的行对 A 的这一样进行存储；然后再 B 的内部就行局部转置；从 1×1，2×2，……到 8×8 的转置，这样一来 A 的一行元素取完之后全放再 temp 变量里，然后再访问相应的 B 矩阵的行，这时遇到对角线元素时不需要再从 A 矩阵中取，就不会再出现 A 与 B 之间的块相互交叉替换导致 miss 了。下面的是 A 矩阵一行一行赋值给 temp 过程：

Temp 赋值给 B 矩阵以及 B 矩阵内部局部转置过程：



通过优化就可以达到 A 矩阵与 B 矩阵的 miss 都只是 1/8：

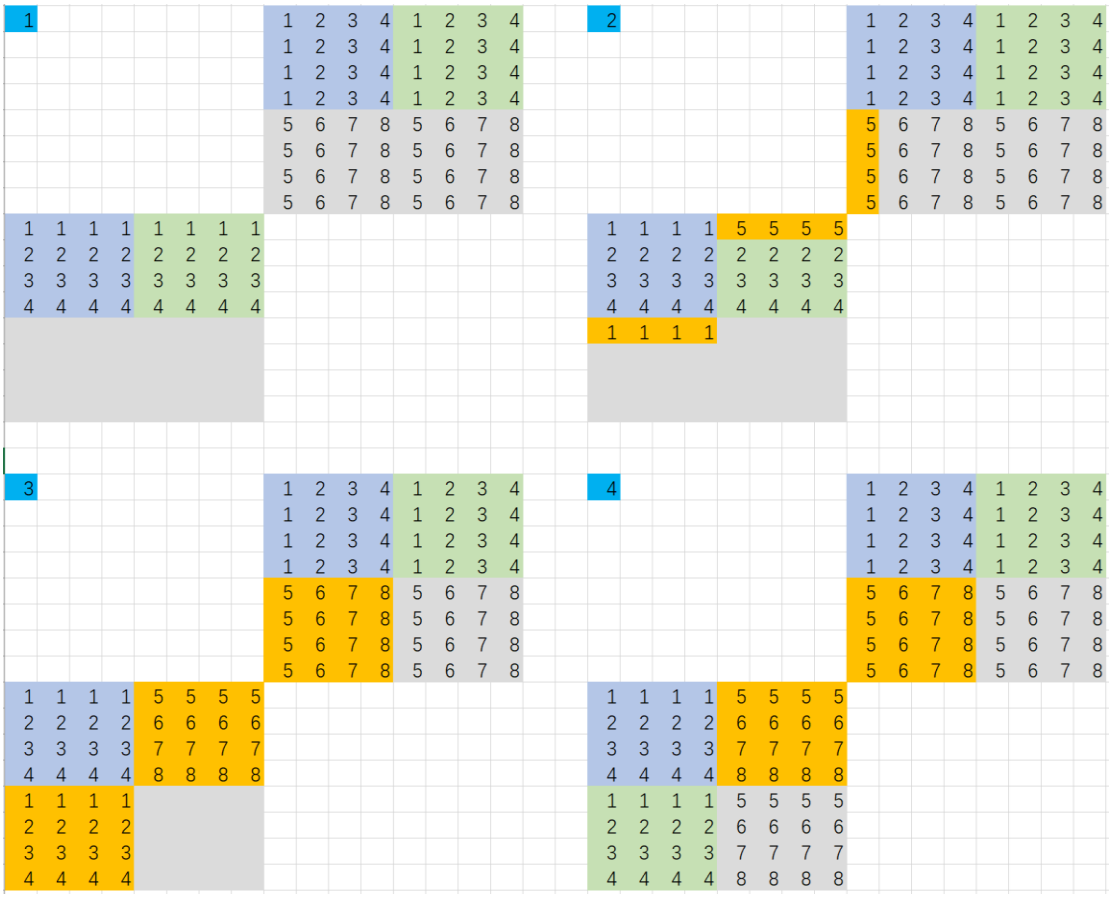
下面是./test-trans -M 32 N -32 测试结果（miss 只有 259）：

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-handout$ ./test-trans -M 32 -N 32

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:2018, misses:259, evictions:227

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151

Summary for official submission (func 0): correctness=1 misses=259

TEST_TRANS_RESULTS=1:259
```

2.3.2 64×64（M=64,N=64）矩阵

通过分析知道如果在 64×64 矩阵仍然是用 8×8 分块是达不到要求的；原因在于在该矩阵每行有 64 个 int，cache 只能容下其 4 行；所以转置过程中 B[4][0]所在的块就会将 B[0][0]所在的块所替换掉导致访问 B[0][5]就会 miss 等等。

这是更换策略，在 8×8 矩阵内部进行 4×4 矩阵转置，具体如下图：



这样一来 miss 率大大降低，能够达到实验要求：

下面是执行./test-trans -M 64 N -64 结果：

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-handout$ ./test-trans -M 64 -N 64

Function 0 (2 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

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3474, misses:4723, evictions:4691

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

TEST_TRANS_RESULTS=1:1179
```

2.3.3 61×67（M=61,N=67）矩阵

对于 61×67 矩阵不再是 8 的倍数，不方便再使用之前的分块策略，所以通过尝试不停分块大小来确定是否能满足要求。最终可以得出块为 18×17 的时候可以说达到最小 miss: 1949

程序实现如下：

```
283     else if(M==61&&N==67){
284         for(i=0;i<67;i=i+18){
285             for(j=0;j<61;j=j+17){
286                 for(k=i;k<67&&k<i+18;++k){
287                     for(h=j;h<61&&h<j+17;++h){
288                         B[h][k]=A[k][h];
289                     }
290                 }
291             }
292         }
293     }
```

执行./test-trans -M 61 N -67 结果：

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-handout$ ./test-trans -M 61 -N 67

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6230, misses:1949, evictions:1917

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3756, misses:4423, evictions:4391

Summary for official submission (func 0): correctness=1 misses=1949

TEST_TRANS_RESULTS=1:1949
```

当然还有 17×17 分块（miss:1950）：

```
Trans perf 61x67          10.0          10          1950
```

18×18 分块（miss:1961）：

```
Trans perf 61x67          10.0          10          1961
```

16×18 分块（miss:1951）：

```
Trans perf 61x67          10.0          10          1951
```

简单的分块优化有多种方案能达到要求。

总的 PartB 运行./driver.py 结果：

```
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        1179
Trans perf 61x67          10.0        10        1949
          Total points    53.0        53
```

四.实验 PartA 与 PartB 结果

执行./driver.py:

```
tiansnowfly@txf:~/cache_project/txf_cache/cachelab-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         259
Trans perf 64x64           8.0         8        1179
Trans perf 61x67          10.0        10        1949
          Total points    53.0        53
```

五.实验感想

对于实验 PartA 总的来说不算难，只需要利用 lru 算法模拟 cache 替换策略；在写 cache 之前也参考课一些网上的东西；所以了解原理后代码就很容易写出来；但是自己在实现的过程种也出现了一些问题，就是不知道是电脑原因还是什么，就是使用 switch 语句处理命令行输入参数就会出错，自己也是一开始也不知道为什么，在其他地方一直找原因，但是没有发现其他问题；利用单步调试的方法在 switch 里面输出一些字符观察最终发现 switch 语句不能输出，也不知道为什么，最后将 switch 修改成为 if else 语句，顺利完成了 PartA。

对于 PartB 部分，总的来说是比较难的，特别实在 32×32 矩阵的 B 矩阵对角线处理；64×64 矩阵的块内分块处理需要动很大的脑筋；自己开始也是尝试了很多种分块，测试结果达不到要求才发觉自己有些地方忽略了；最后也是在网上参考了一些别人的方法策略，自己再进一步理解，然后才顺利完成实验。

在这个 Cache 实验过程遇到了很多问题，但是自己能够不断去克服并且从中学到了很多知识，通过实验，我自己对于 cache 的理解程度可以说是加深了很多；在 cache 性能提升方面也学到了不少的知识，所以，感谢老师与助教给了我这么好的一次机会取接触并深入理解这些知识。自己也会继续关注了解计算机体系结构其他方面的一些知识。

姓名：田雪飞

学号：515030910347