上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

计算机系统结构

实验名称：Cache Lab: Understanding Cache Memories

姓　　名：洪瑄锐

学　　号：517030910227

班　　级：F1703302

手　　机：15248246044

邮　　箱：1204378645@qq.com

目录

1. PartA..... 错误!未定义书签。

1.1 实现想法和代码..... 错误!未定义书签。

1.2 实验结果和解释 ..... 错误!未定义书签。

2. PartB..... 错误!未定义书签。

2.1 32×32..... 错误!未定义书签。

2.2 64×64..... 错误!未定义书签。

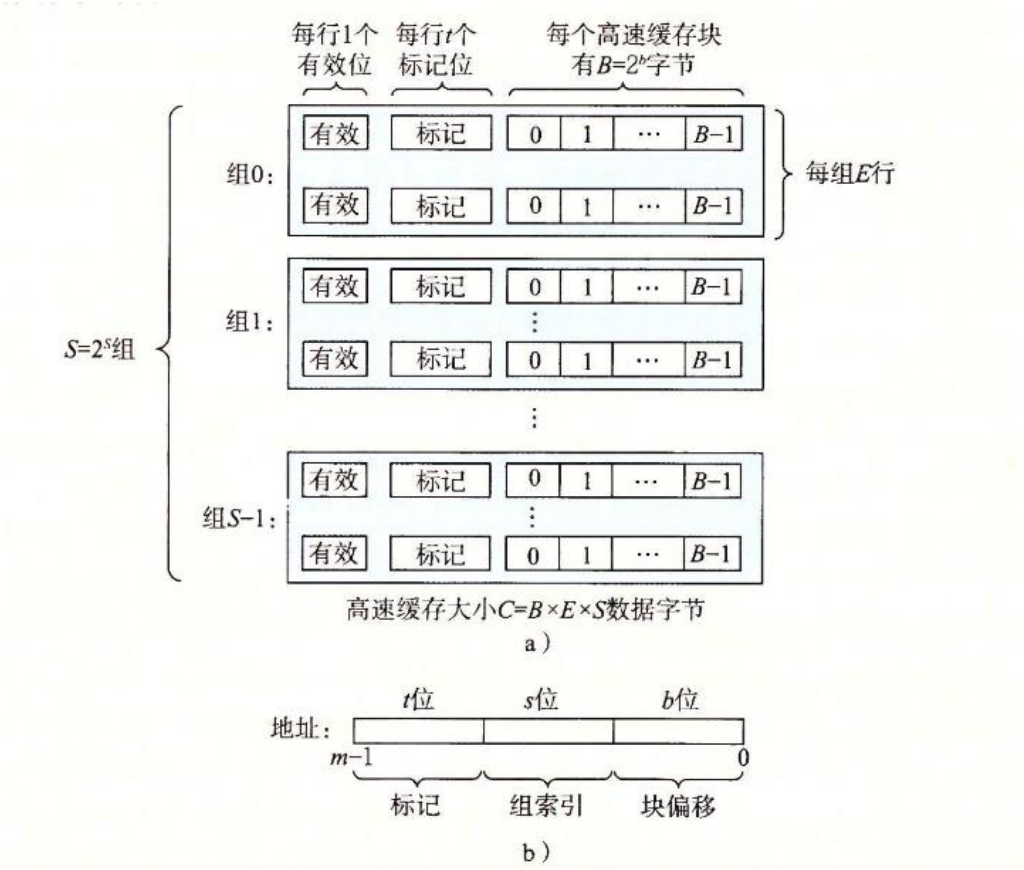
2.3 61×67

3. 总测试..... 错误!未定义书签。

1.Part A

1.1 实验想法和代码

1.1.1 Cache 的组织结构



如图所示是高速缓存（S,E,B,m）的通用组织。

1） 高速缓存是一个高速缓存的数组，每个组包含一个或多个行，每个行包含一个有效位，一些标记位，以及一个数据块；

2） 高速缓存的结构将 m 个地址位划分为 t 个标记位，s 个组索引位和 b 个块偏移位；

3） t、s、b、m、S、B、E 的数学关系为：

S = 2^b, B = 2^b, C = S×E×B, t = m – s – b。

1.1.2 模拟 cache 的数据结构

Cache 由若干 set 组成，每个 set 由若干行组成，每行由若干字节的数据块以及标志位、有效位组成。由于 cache 涉及数据替换，因此另设置一个 use 变量用于记录相对使用时间。

```
/*cache组成*/
typedef struct cache_line //定义cache每组的一行，有效位，标志位和用于替换的use变量
{
    char valid;//有效位
    unsigned long long int tag;//标志位
    unsigned long long int use;//用于LRU替换策略
} cache_e;

typedef cache_e* cache_s;//每组有很多行
typedef cache_s* cache_all;//每个cache有很多组

cache_all cache;
```

1.1.3 cache 的初始化

由于实验要求 cache 的 S,E,B 可以按要求变化，因此使用 malloc 函数为 cache 分配存储空间。此外，各行有效位置 0，use 置 0，标志位 tag 置 0。

```
cache = (cache_s*)malloc(sizeof(cache_s) * S);
for (int i = 0; i < S; i++)
{
    cache[i] = (cache_e*)malloc(sizeof(cache_e) * E);
    for (int j = 0; j < E; j++)
    {
        cache[i][j].valid = 0;
        cache[i][j].tag = 0;
        cache[i][j].use = 0;
    }
}
```

1.1.4 cache 工作

首先根据所给地址分离出标志位 tag 和组索引 set

```
//获取标志位和组索引
unsigned long long int set = (addr >> b) & set_mask;
unsigned long long int tag = (addr >> (b + s)) & tag_mask;
```

然后在组 set 进行行匹配。数据在 cache 中当且仅当某行有效位 valid 为 1 且该行标志位 tag 为求出的 tag。变量 use 的作用是，如果当前数据在某行，则该行的 use 置 0，意思是它现在刚被使用，在遍历的过程中，如果某行不是该数据的匹配，则该行的 use+1，意思是它的使用时间距现在已经过了一个单位时间。

```
int hit = 0;//标记是否命中

for (int i = 0; i < E; i++)
{
    if (cache[set][i].valid == 1 && cache[set][i].tag == tag)
    {
        hit = 1;
        cache[set][i].use = 0;
    }
    else
        cache[set][i].use++;
}
```

如果 hit 为 1，代表命中，hit+1, 如果输入命令行存在-v 代表需要输出相关信息，即 verbose=1，输出“hit”，并且直接 return，工作结束。

如果 hit 为 0，代表失效，miss+1，找到为空的行进行填充或者最近最少被使用的行进行替换，两种情况可以归为一种做法，即寻找该组中 use 最大的行，因为对于空行来说，它在初始化后对于每一次寻找行匹配时都被+1，一定比有数据的行的 use 大。如果没有空行，那 use 最大的就是最近最少被使用的行。

此外，在找到最大 use 行后，还需要判断有效位是否为 0，为 0 代表这是一个空行，为 1 代表此刻发生了替换，eviction+1。

```
if (hit)
{
    if (verbose)
        printf("hit");
    hit_number++;
    return;
}

if (verbose)
    printf("miss ");
miss_number++;
```

```
            //替换
            //找到LRU
            int max = 0;
            int lru_index = 0;

            for (int i = 0; i < E; i++)
            {
                if (cache[set][i].use > max)
                {
                    max = cache[set][i].use;
                    lru_index = i;
                }
            }
```

```
if (cache[set][lru_index].valid == 1)//如果该块不是空的
{
    if (verbose)
        printf("eviction ");
    eviction_number++;
}
    cache[set][lru_index].valid = 1;
    cache[set][lru_index].tag = tag;
    cache[set][lru_index].use = 0;
```

1.1.5 文件输入

文件内容共有四种，I,M,L,S。实验要求我们将 I 型忽略，对于 M,L,S,因前有空格，所以读入文件的每一行后通过 tmp[1]辨认是 M,L 还是 S。如果命令行有-v，则需要输入相关信息，所以还需要提取字节数和地址。对于 M，因为它需要两次访存，所以调用两次 cache_work(addr)函数。

```
/*将trace文件输入*/
void input_trace(char *file)
{
    char tmp[500];
    unsigned long long int addr;
    int len;

    FILE* trace = fopen(file, "r");

    while (fgets(tmp, 500, trace) != NULL)
    {
        if (tmp[1] == 'S' || tmp[1] == 'L' || tmp[1] == 'M')
        {
            sscanf(tmp+3, "%llx,%u", &addr,&len);
            if (verbose)
            {
                printf("%c ", tmp[1]);
                printf("%llx,%u ", addr, len);
            }
            cache_work(addr);

            //M要访问两次
            if (tmp[1] == 'M')
            {
                cache_work(addr);
            }
            if (verbose)
                printf("\n");
        }

    }

    fclose(trace);
}
```

1.1.6 打印帮助

仿照实验指导文件的帮助输出即可。

```
void printUsage(char* argv[])
{
    printf("Usage: %s [-hv] -s <num> -E <num> -b <num> -t <file>\n", argv[0]);
    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(S=2^s is the number of sets\n");
    printf("  -E <E>: Associativity(number of lines per set)\n");
    printf("  -b <b>: Number of block bits(B=2^b is the block size)\n");
    printf("  -t <tracefile>: Name of the valgrind trace to replay\n");
    printf("\nFor examples:\n");
    printf("  linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace\n");
    printf("  linux> ./csim-ref -v -s 4 -E 4 -b 4 -t traces/yi.trace\n");
}
```

1.1.7 主函数

因为涉及命令行输入，所以需要包含头文件<getopt.h>。从命令行获取 s,E,b,v,h 的信息，进行 cache 初始化，文件输入，cache 释放，最后将全局变量 hit，miss，eviction 输出。

```
int main(int argc,char* argv[])
{
    char tmp;

    while ((tmp = getopt(argc, argv, "s:E:b:t:vh")) != -1)
    {
        switch (tmp)
        {
        case 'h':
            printUsage(argv);
            break;
        case 'v':
            verbose = 1;
            break;
        case 's':
            s = atoi(optarg);
            break;
        case 'E':
            E = atoi(optarg);
            break;
        case 'b':
            b = atoi(optarg);
            break;
        case 't':
            input_file = optarg;
            break;
        }
    }
```

```
    S = (unsigned int)(pow(2, s));
    B = (unsigned int)(pow(2, b));

    init_cache();

    input_trace(input_file);

    free_cache();

    printSummary(hit_number, miss_number, eviction_number);
    return 0;

}
```

1.2 实验结果

1.2.1

命令行输入./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace

命令行输入./csim -v -s 4 -E 1 -b 4 -t traces/yi.trace

将模拟 cache 和老师给出的示例 cache 运行结果相比较，完全相同。

```
hongxuanrui@ubuntu:~/cs/cachelab-handout$ ./csim-ref -v -s 4 -E 1 -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 eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
hongxuanrui@ubuntu:~/cs/cachelab-handout$ ./csim -v -s 4 -E 1 -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 eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

1.2.2

命令行输入./csim -h -s 4 -E 1 -b 4 -t traces/yi.trace

测试输出帮助的效果。

```
hongxuanrui@ubuntu:~/cs/cachelab-handout$ ./csim -h -s 4 -E 1 -b 4 -t traces/yi.trace
Usage: ./csim [-hv] -s <num> -E <num> -b <num> -t <file>
  -h  Optional help flag that prints usage info
  -v  Optional verbose flag that displays trace info
  -s <s>: Number of set index bits(S=2^s is the number of sets
  -E <E>: Associativity(number of lines per set)
  -b <b>: Number of block bits(B=2^b is the block size)
  -t <tracefile>: Name of the valgrind trace to replay

For examples:
  linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
  linux> ./csim-ref -v -s 4 -E 4 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

1.2.3 完整测试 csim

命令行输入./test-csim，由图可知 cache 模拟成功。

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

TEST_CSIM_RESULTS=27
```

2.PartB

PartB 要求我们写 cache 友好的转置矩阵函数，测试用 cache 总容量为 1KB,E=1, 每块 32bytes，即共有 32 组，每组一行，每行可容纳 8 个 int 型数据。采用针对不同型号的矩阵分别优化算法的方法。

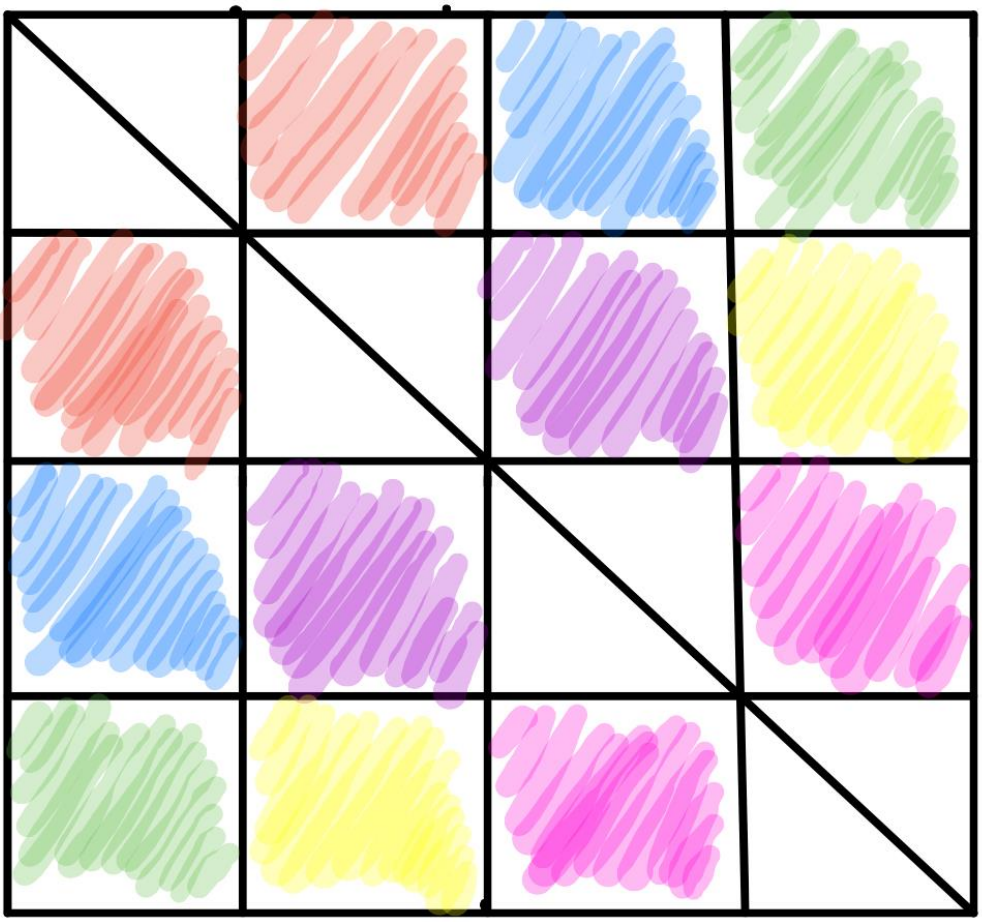
老师给出的基础转置矩阵函数，因为对于 A 的第 i 行和 B 的第 i 行，会被放入 cache 的相同组，所以在转置的过程中不断发生冲突失 效，导致最终失效次数非常多。例如，访问 A[0][0], 产生一次 cold miss，A[0][0]~A[0][7]被放入 chache 的第 i 组中，因为要执行 B[0][0]=A[0][0]，而 B[0][0]在 cache 中也对应第 i 组，产生 conflict miss，驱逐 A[0][0]~A[0][7]，B[0][0]~B[0][7]被放入第 i 组，接下来访问 A[0][1], 而因为刚才被驱逐了，所以再次产生 conflict miss…… 所以基础转置矩阵函数在循环的每一步都会产生两次 miss。

2.1 32×32

2.1.1 实现思想和代码

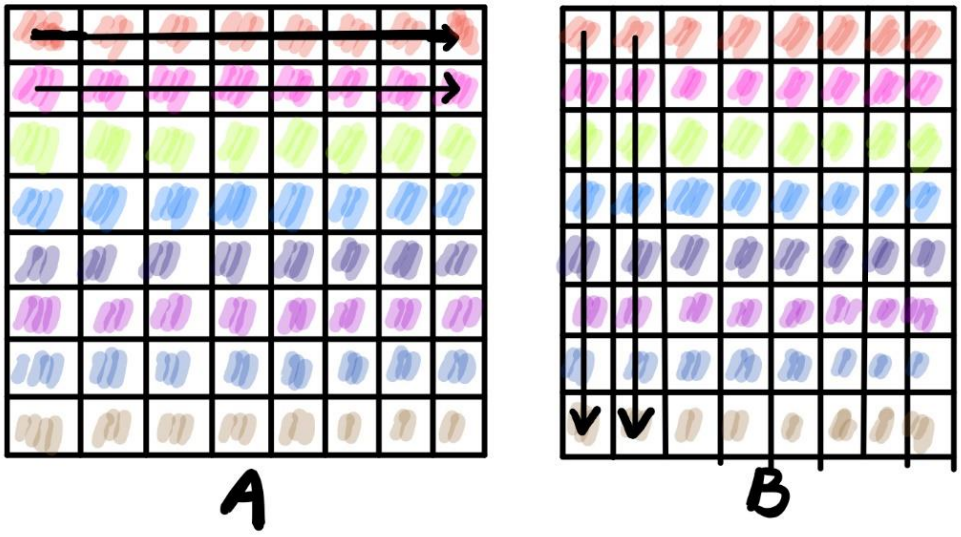
按照分块的思想，32×32 的矩阵可分为 16 个 8×8 的矩阵，因为 cache 每行容纳 8 个 int，每块 8×8 则刚好可以使 cache 容纳一个块。分析矩阵分块后 A 与转置

矩阵 B 元素的对应关系如图：



所以除了白色含对角线的部分，其他矩阵块可以正常转置，A 与 B 不会产生冲突失效。

现在**针对含对角线**的部分，**抽取某一个白色矩阵块**进行分析，相同颜色意味着在 cache 中对应相同的组。箭头为对 8×8 矩阵块正常转置时的走向，由图可知对于恰好位于对角线上的数据，会导致冲突失效。例如 A[0][0]，解决方法为在访问 A[i][i]时，先不对 B[i][i]进行处理，用一个 tmp 变量暂存 A[i][i]，当对于该矩阵块 A[i]这行全部处理完后，再令 B[i][i]=tmp。



代码如下：

```
if (N == 32) {
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            if (i == j) {
                for (k = j * 8; k < j * 8 + 8; k++) {
                    for (l = i * 8; l < i * 8 + 8; l++) {
                        if (k != l)
                            B[l][k] = A[k][l];
                        else
                            tmp0 = A[k][l];
                    }
                    B[k][k] = tmp0;
                }
            }
            else {
                for (k = j * 8; k < j * 8 + 8; k++) {
                    for (l = i * 8; l < i * 8 + 8; l++) {
                        B[l][k] = A[k][l];
                    }
                }
            }
        }
    }
}
```

2.1.2 实验结果和解释

```
hongxuanrui@ubuntu:~/cs/cachelab-handout$ ./test-trans -M 32 -N 32

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

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

TEST_TRANS_RESULTS=1:287
```

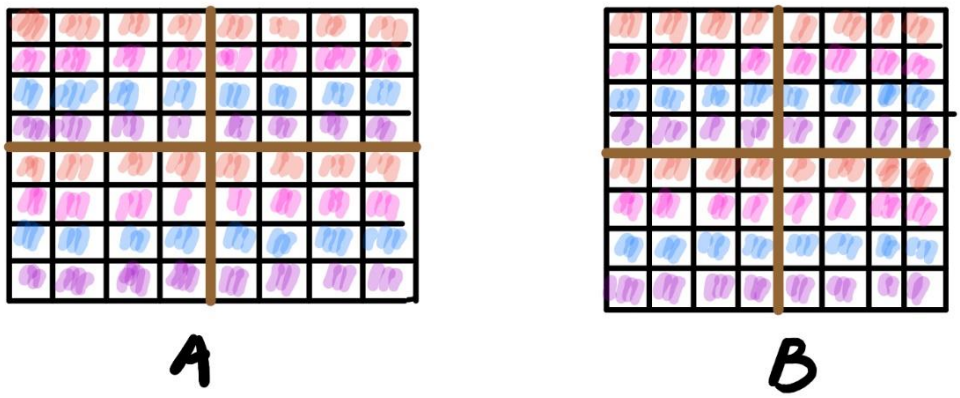
转置结果正确并且失效次数<300。

2.2 64×64

2.2.1 实现思想和代码

对于 64×64 的矩阵，矩阵每行需要 8 个组才能容纳，因此第 i 行与第 i+4 行相同列的数据会对应到 cache 的同一个组，所以不能简单采用 32×32 矩阵的分成 8×8 矩阵块的做法。如果分为 4×4 矩阵块，cache 发生一次 cold miss 后读入 8 个数据，但是用了 4 个，浪费了资源而且最终测试结果超过 1300。最后采用利用 B 数组暂存 A 数组数据的做法，仍将矩阵分为 8×8 的矩阵块，分块处理，但是对于每块，不是直接转置，而分为以下几步：

1) A 左上角与 B 左上角直接转置
2) A 右上角转置后存在 B 的右上角而不是本应放入的 B 左下角（假如 A 右上角转置后直接放入 B 的左下角，那么会不断的产生冲突失效）
3) 连续访问 A 的左下角一列，存入 4 个变量 tmp0,tmp1,tmp2,tmp3 中
4) 连续访问 B 右上角一行，存入 4 个变量 tmp4，tmp5，tmp6，tmp7 中
5) 将 tmp0,tmp1,tmp2,tmp3 存入 B 右上角一行（B 右上角已经因为步骤 4 而放入缓存了，所以步骤 5 不会产生失效）
6) 将 tmp4,tmp5,tmp6,tmp7 存入 B 左下角一列
7) A 右下角与 B 右下角直接转置。（B 右下角已经因为步骤 6 而放入缓存了，所以步骤 7 使得 B 部分不产生冷失效）



（A 和 B 8×8 矩阵块各行元素对应 cache 中的组的情况如上图，同一颜色则为一个组）

此外，由于设置了 8 个 tmp 变量，所以在左上角和右下角规模为 4×4 的矩阵块直接转置的时候可以直接存下待转置的 A[i]行的四个元素，再赋值给 B，也可以避免对角线问题。

```
else if (M == 64)
{
        for (i = 0; i < N; i += 8) {
            for (j = 0; j < M; j += 8) {
                for (k = i; k < i + 4; k++) {
                    tmp0 = A[k][j];
                    tmp1 = A[k][j + 1];
                    tmp2 = A[k][j + 2];
                    tmp3 = A[k][j + 3];
                    tmp4 = A[k][j + 4];
                    tmp5 = A[k][j + 5];
                    tmp6 = A[k][j + 6];
                    tmp7 = A[k][j + 7];

                    B[j][k] = tmp0;  //A左上角直接转置到B左上角
                    B[j + 1][k] = tmp1;
                    B[j + 2][k] = tmp2;
                    B[j + 3][k] = tmp3;

                    B[j][k + 4] = tmp4; //将A右上角的转置暂存B右上角
                    B[j + 1][k + 4] = tmp5;
                    B[j + 2][k + 4] = tmp6;
                    B[j + 3][k + 4] = tmp7;
                }
```

```
                for (l = j + 4; l < j + 8; l++) {

                    tmp0 = A[i + 4][l - 4]; //取A左下角一列
                    tmp1 = A[i + 5][l - 4];
                    tmp2 = A[i + 6][l - 4];
                    tmp3 = A[i + 7][l - 4];

                    tmp4 = B[l - 4][i + 4]; //取B右上角一行
                    tmp5 = B[l - 4][i + 5];
                    tmp6 = B[l - 4][i + 6];
                    tmp7 = B[l - 4][i + 7];

                    B[l - 4][i + 4] = tmp0; //将A左下角一列赋值给B右上角一行
                    B[l - 4][i + 5] = tmp1;
                    B[l - 4][i + 6] = tmp2;
                    B[l - 4][i + 7] = tmp3;

                    B[l][i] = tmp4;         //将B右上角一行赋值给B左下角一列
                    B[l][i + 1] = tmp5;
                    B[l][i + 2] = tmp6;
                    B[l][i + 3] = tmp7;

                    B[l][i + 4] = A[i + 4][l]; //A的右下角直接转置为B右下角
                    B[l][i + 5] = A[i + 5][l];
                    B[l][i + 6] = A[i + 6][l];
                    B[l][i + 7] = A[i + 7][l];
                }
            }
        }
    }
```

2.2.2 实验结果和解释

```
hongxuanrui@ubuntu:~/cs/cachelab-handout$ ./test-trans -M 64 -N 64

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:9074, misses:1171, evictions:1139

Summary for official submission (func 0): correctness=1 misses=1171

TEST_TRANS_RESULTS=1:1171
```

如图所示，转置结果正确并且 miss<1300。

2.3 61×67

2.3.1 实现想法和代码

虽然该型号的矩阵不能被分为完整的 8×8，4×4 等矩阵块，但经试验直接分块方法仍然有效，注意不要超过矩阵边界即可。

代码：

```
    else {
        for (i = 0; i < N / 8 + 1; i++) {
            for (j = 0; j < M / 8 + 1; j++) {
                if (i == j) {
                    for (k = j * 8; k < M && k < j * 8 + 8; k++) {
                        for (l = i * 8; l < N && l < i * 8 + 8; l++) {
                            if (k != l)
                                B[k][l] = A[l][k];
                            else
                                tmp0 = A[l][k];
                        }
                        B[k][k] = tmp0;
                    }
                }
                else {
                    for (k = j * 8; k < M && k < j * 8 + 8; k++) {
                        for (l = i * 8; l < N && l < i * 8 + 8; l++) {
                            B[k][l] = A[l][k];
                        }
                    }
                }
            }
        }
    }
}
```

2.3.2 实验结果和解释

```
hongxuanrui@ubuntu:~/cs/cachelab-handout$ ./test-trans -M 61 -N 67

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6250, misses:1929, evictions:1897

Summary for official submission (func 0): correctness=1 misses=1929

TEST_TRANS_RESULTS=1:1929
```

如图所示，转置结果正确并且 miss<2000。

3 总测试

输入 python driver.py,得到如下结果。

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


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

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           8.0         8        1171
Trans perf 61x67          10.0        10        1929
          Total points    53.0        53
hongxuanrui@ubuntu:~/cs/cachelab-handout$
```