

CacheLab实验报告

软73 金昕祺 2016010524

一、PartA

1. 使用双向链表数组模拟使用LRU策略的高速缓存

在csim.c中，首先定义了双向链表的结点struct Node，然后定义struct DoubleLinkedList作为双向链表。每个双向链表有一个头结点、一个尾结点和一个size信息。在本人的代码中，每个双向链表对象对应于高速缓存中的一个组。为了理解后面的代码，首先解释如何使用双向链表来模拟LRU策略的高速缓存：

设高速缓存中共有num_of_Sets组，我们用DoubleLinkedList list[num_of_Sets]这个双向链表组成的数组模拟整个高速缓存，其中list[i]表示高速缓存的第i组。每个双向链表list[i]初始时只有一个头结点list[i].head、一个尾结点list[i].tail和一个list[i].size（初始化时设为0）。为了实现LRU策略，我们的目标是每个双向链表中的每个中间结点（即头、尾结点以外的结点）对于有效位为1的高速缓存的一行，并且在每次调整任意一个双向链表后，保证这个双向链表中越最近被使用的结点（即对应的行越最近被读写的结点）离头结点越近。为了实现这个目标，我们具体按如下方式操作：

每当我们处理trace文件的一行时，只要发现该行不是以“I”开头，就解析出该行的地址对应的组索引setIndex和标记位。然后，我们去检查list[setIndex]中是否存在一个中间结点（即头结点和尾结点以外的结点）的标记位和trace文件这一行的地址的有效位相同，若存在某个中间结点的标记位相同就调整该中间结点为紧跟在头结点之后（其他结点的先后顺序不变），并让list[setIndex].size++；若不存在相同的就比较list[setIndex].size是否等于程序的参数E（E表示每个组的行数），若等于就删除list[setIndex].tail->prev对应的结点（即上次被使用的时间最久远的行对应的结点）并在头结点后插入新结点（同时将该结点的tag设为处理的trace文件的这一行的地址对应的标记位），若list[setIndex].size < 参数E就直接在头结点后插入新结点（同时将该结点的tag设为处理的trace文件的这一行的地址对应的标记位）并令list[setIndex].size++。不难看出，这样在处理trace文件之前每个双向链表被初始化为越最近被使用的结点（即对应的行越最近被读写的结点）离头结点越近（因为这时没有中间结点，自然满足这个性质），而每处理完一行我们都能保证这个顺序的要求依然被满足。

2. 程序结构解析

csim.c的第17-27行typedef了双向链表struct DoubleLinkedList和链表的结点struct Node。

第29-36行声明了一个全局结构体globalArgs，用于存储程序运行时的命令行参数。

在第42-75行，程序通过<getopt.h>提供的函数处理了命令行参数并保存在globalArgs中。

在第77-81行，计算出标记位的位数t、组数num_of_Sets。

在第83-86行，定义了记录结果的三个变量。

在第91-100行，定义并初始化了用来模拟高速缓存的DoubleLinkedList list[num_of_Sets]。

在第102-273行，是一个while循环，每次循环处理trace文件的一行：处理trace文件的每一行时，每一行的内容被写入char buf[1024]中，每一行对应的地址被存储到char address[64]中（第105-182行）。在第184-189行，求出trace文件该行中地址对应的组号setIndex，并将address中除标记位以外的每个char都设成8个bit全0的字符。在193-221行，检查是否为cache hit，若为cache hit则根据buf[1]的值对记录结果的变量hits作相应调整，并且调整双向链表list[setIndex]使得最近被访问的结点被移到离头结点最近）。若不是cahce hit，在程序的第222-269行进一步处理，根据list[setIndex].size和globalArgs.E的大小关系判断是否需要eviction，然后根据是否需要eviction和buf[1]的值作出不同的操作。

最后，在第276行输出结果。

二、Part B

1. Case 1: M=32, N=32

依次执行如下命令：

```
linux> make
linux> ./test-trans -M 32 -N 32
linux> ./csim-ref -v -s 5 -E 1 -b 5 -t trace.f1
```

通过输出可以判断出，数组A的起始地址是10d080，数组B的起始地址是14d080。

由于b=5，E=1，并且每个int是4个字节大小，因而高速缓存中每一组可以存储数组中8个连续的元素（并且每一组只有一行。据此，我们可以列出下表（表中每一个单元格对应矩阵A或B中的同一行连续8个元素，如最左上方单元格对应A[0][0:8]，其右方的单元格对应A[0][8:16]。由于高速缓存中每一组只能存储8个int，因而每一单元格也对应高速缓存中的一个组，单元格中的数字为该组的组号）：

4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
0	1	2	3
4	5	6	7

8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
0	1	2	3

首先分析trans.c中提供的trans函数，找出trans函数中miss次数过多的原因并针对其进行改善。通过分析./csim-ref -v -s 5 -E 1 -b 5 -t trace.f1命令的输出发现，大部分的miss发生在对矩阵B进行写操作时。具体分析发现，在trans函数最内层的for循环中，会遍历B的第i列的各行。我们以外层for循环中i=0为例具体分析，对B的写操作对应的高速缓存的组号依次为4、8、12、16、20、24、28、0、4、8、12、16、20、24、28、0、4、8、12、16、20、24、28、0，而每相邻两次访问第i组（i=4, 8, 12, 16, 20, 24, 28, 32, 0）时，标记位总是不同，这导致了大量的cache miss。

为了解决这个问题，本人设计了如下方案：

将32*32的矩阵分成16块，每块大小为8*8（上表中每一个颜色的块对应一个8*8的数据块，关于上表的含义前文中有提及）。为了避免访问B矩阵带来的cache miss，每次将A矩阵的这样一个8*8的子矩阵转置后拷贝到B矩阵中，这样就避免了直接遍历B矩阵的全部32行带来的大量的cache miss。

由此设计代码如下：

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,k,h;
    if(M==32){
        for(i=0;i<32;i+=8){
            for(j=0;j<32;j+=8){
                for(k=i;k<i+8;k++){
                    for ( h = j; h < j + 8; h ++ ) {
                        B[h][k] = A[k][h];
                    }
                }
            }
        }
    }else{
        //TODO
    }
}
```

测试该代码，结果如下：

```
jxqhhh@jxqhhh-VirtualBox:~/Desktop/cachelab-handout$ make
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -o test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -cvf jxqhhh-handin.tar csim.c trans.c
csim.c
trans.c
jxqhhh@jxqhhh-VirtualBox:~/Desktop/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:1710, misses:343, evictions:311

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=343
TEST_TRANS_RESULTS=1:343
```

发现仍然没有达到题目的要求，因而我们利用csim-ref程序保存下来结果，并对保存的结果中操作B矩阵时的cache hit/miss情况可视化如下图（下图中第i行第j列表示B的第i行第j列被写时是否发生cache miss，若为1则发生，若为0则未发生），以便进一步分析：

1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1

可以发现，每次访问B矩阵的对角线元素右边紧邻的一个元素时，都会发生cache miss。这是因为在程序访问B[8*i+j][8*i+j+1]之前($i=0,1,2,3; j=0,1,\dots,6$)，必须先访问B的第(8*i+j)列，

而访问完B的该列时需要访问A的第 $(8*i+j)$ 行，故访问完B的该列时A的第 $(8*i+j)$ 行必然被存储在高速缓存中。之后程序访问B的第 $(8*i+j+1)$ 列时，直到访问到 $B[8*i+j][8*i+j+1]$ 前都不会去修改存储A的第 $(8*i+j)$ 行的那组高速缓存的内容，这样在访问 $B[8*i+j][8*i+j+1]$ 时，存储A的第 $(8*i+j)$ 行的那组高速缓存就会发生cache miss。

为了消除这种cache miss，我们可以进行如下改进：处理位于对角线上的 $8*8$ 子矩阵时，每次处理A的对角线上子矩阵某一行时，使用8个临时变量存下来这一行的内容，然后用这8个临时变量给B中相应子矩阵的相应列赋值。

这样处理后，经检验发现确实能消除 $B[8*i+j][8*i+j+1]$ 处的cache miss($i=0,1,2,3; j=0,1,\dots,6$)。实际上，这样的处理还可以减少读写A矩阵时的cache miss次数，具体的代码和实验结果如下：

```

char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int a1, a2, a3, a4, a5, a6, a7, a8;
    int i, j, k, h;
    if (M == 32) { // Case1: M=32, N=32
        for (i = 0; i < 32; i += 8) {
            for (j = 0; j < 32; j += 8) {
                for (k = i; k < i + 8; k++) {
                    if (i == j) {
                        a1 = A[k][j];
                        a2 = A[k][j+1];
                        a3 = A[k][j+2];
                        a4 = A[k][j+3];
                        a5 = A[k][j+4];
                        a6 = A[k][j+5];
                        a7 = A[k][j+6];
                        a8 = A[k][j+7];
                        B[j][k] = a1;
                        B[j+1][k] = a2;
                        B[j+2][k] = a3;
                        B[j+3][k] = a4;
                        B[j+4][k] = a5;
                        B[j+5][k] = a6;
                        B[j+6][k] = a7;
                        B[j+7][k] = a8;
                        continue;
                    }
                    for (h = j; h < j + 8; h++) {
                        B[h][k] = A[k][h];
                    }
                }
            }
        }
    } else if (M==64) { // Case2: M=64, N=64
    }
}

```

此时已达到要求，但我们可以进一步优化。通过分析发现，在访问 $B[i][i]$ 时($i \in \{0,1,2,3,\dots,63\} - \{0,8,16,\dots,56\}$)总会发生缓存不命中。以 $B[1][1]$ 为例，这里之所以会缓存不命中，是因为为了写 $B[1][1]$ 必须要读 $A[1][1]$ ，而按照目前设计的算法，读 $A[1][1]$ 之前写过 $B[1][0]$ ，所以这里写 $B[1][1]$ 时必然会出现cache miss。为了解决这一问题，我们可以在对A的对角线上的 $8*8$ 子矩阵进行操作时，首先把A的这个子矩阵的第一行存到 $a1-a8$ 中，再把 $a1-a8$ 赋值给B的相应行，然后再继续这样处理A的这个子矩阵的下一行，如此循环直到把A的子矩阵拷贝给B矩阵。通过这样的方式处理A的这个子矩阵，对B的写操作每一行只会发生一次cache miss。但我们这里只是把A的这个对角线上子矩阵拷贝到了B矩阵里，为了得到正确的结果，我们再对B的相应的子矩阵进行就地转置即可。只要我们在拷贝完子矩阵后立即进行就地（in-place）转置，自然不会导致新的cache miss。

具体来说，改进的代码如下：

```

char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i,j,k,h;
    int a1,a2,a3,a4,a5,a6,a7,a8;
    if ( M == 32 ) { // Case1: M=32, N=32
        for ( i = 0; i < 32; i += 8) {
            for ( j = 0; j < 32; j += 8 ) {
                if ( i == j ) {
                    // copy the submatrix:
                    for ( k = i; k < i + 8; k ++ ) {
                        a1 = A[k][j];
                        a2 = A[k][j+1];
                        a3 = A[k][j+2];
                        a4 = A[k][j+3];
                        a5 = A[k][j+4];
                        a6 = A[k][j+5];
                        a7 = A[k][j+6];
                        a8 = A[k][j+7];
                        B[k][j] = a1;
                        B[k][j+1] = a2;
                        B[k][j+2] = a3;
                        B[k][j+3] = a4;
                        B[k][j+4] = a5;
                        B[k][j+5] = a6;
                        B[k][j+6] = a7;
                        B[k][j+7] = a8;
                    }
                    // transpose in-place:
                    for( k = i; k < i + 8 ; k ++ ){
                        for( h = k + 1 ; h < i + 8; h ++ ){
                            a1=B[k][h];
                            a2=B[h][k];
                            B[k][h]=a2;
                            B[h][k]=a1;
                        }
                    }
                    continue;
                }
                for ( k = i; k < i + 8; k ++ ) {
                    for ( h = j; h < j + 8; h ++ ) {
                        B[h][k] = A[k][h];
                    }
                }
            }
        }
    } else if ( M == 64 ) {
        for ( i = 0; i < 64; i += 16) {
            for ( j = 0; j < 64; j += 16 ) {
                if ( i == j ) {
                    // copy the submatrix:
                    for ( k = i; k < i + 16; k ++ ) {
                        a1 = A[k][j];
                        a2 = A[k][j+1];
                        a3 = A[k][j+2];
                        a4 = A[k][j+3];
                        a5 = A[k][j+4];
                        a6 = A[k][j+5];
                        a7 = A[k][j+6];
                        a8 = A[k][j+7];
                        a9 = A[k][j+8];
                        a10 = A[k][j+9];
                        a11 = A[k][j+10];
                        a12 = A[k][j+11];
                        a13 = A[k][j+12];
                        a14 = A[k][j+13];
                        a15 = A[k][j+14];
                        a16 = A[k][j+15];
                        B[k][j] = a1;
                        B[k][j+1] = a2;
                        B[k][j+2] = a3;
                        B[k][j+3] = a4;
                        B[k][j+4] = a5;
                        B[k][j+5] = a6;
                        B[k][j+6] = a7;
                        B[k][j+7] = a8;
                        B[k][j+8] = a9;
                        B[k][j+9] = a10;
                        B[k][j+10] = a11;
                        B[k][j+11] = a12;
                        B[k][j+12] = a13;
                        B[k][j+13] = a14;
                        B[k][j+14] = a15;
                        B[k][j+15] = a16;
                    }
                    // transpose in-place:
                    for( k = i; k < i + 16 ; k ++ ){
                        for( h = k + 1 ; h < i + 16; h ++ ){
                            a1=B[k][h];
                            a2=B[h][k];
                            B[k][h]=a2;
                            B[h][k]=a1;
                        }
                    }
                    continue;
                }
                for ( k = i; k < i + 16; k ++ ) {
                    for ( h = j; h < j + 16; h ++ ) {
                        B[h][k] = A[k][h];
                    }
                }
            }
        }
    }
}

```

测试的结果如下，最终cache miss数只有259：

```

jxqhhh@jxqhhh-VirtualBox:~/Desktop/cachelab-handout$ make
# Generate a handin tar file each time you compile
tar -cvf jxqhhh-handin.tar  csim.c trans.c
csim.c
trans.c
jxqhhh@jxqhhh-VirtualBox:~/Desktop/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:2242, 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

```

最终，操作B矩阵时的cache hit/miss情况可视化如下图（下图中第i行第j列表示B的第i行第j列被写时是否发生cache miss，若为1则发生，若为0则未发生）：

对于A矩阵，只在访问 $A[i][8*j]$ 这些位置($i=0,1,\dots,31$; $j=0,1,2,3$)时会发生cache miss (并且这些位置中每个位置只发生一次)，其余位置不发生cache miss。

2. Case 2: M=64, N=64

由于列数从Case 1中的32增大到了64，因而矩阵A中每一行需要8个组号连续的高速缓存组

```
trans.c
jxqhh@jxqhh-VirtualBox:~/Desktop/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:6498, misses:1699, evictions:1667

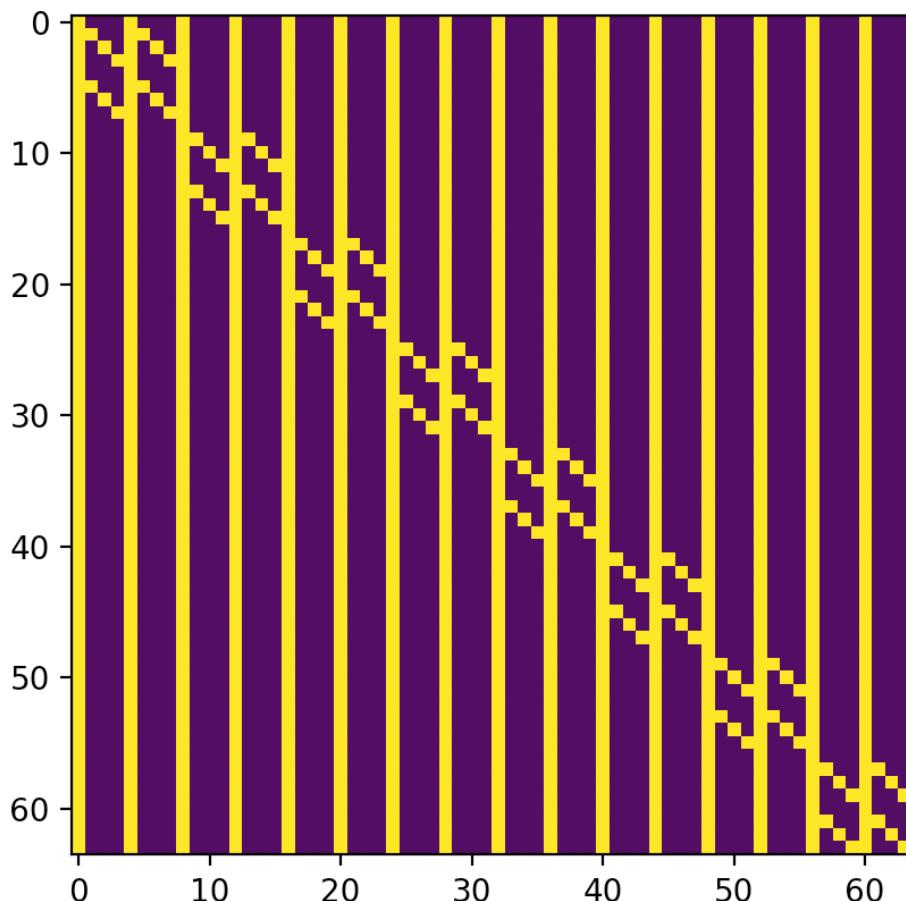
Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3474, misses:4723, evictions:4691

Summary for official submission (func 0): correctness=1 misses=1699
TEST_TRANS_RESULTS=1:1699
jxqhh@jxqhh-VirtualBox:~/Desktop/cachelab-handout$
```

来存储（矩阵B同理），也即矩阵A的每四行就能填满整个高速缓存。这时，若仍然采用依次转置 $8*8$ 子矩阵的思路，会发生很多的cache miss，具体分析类似于之前的Case 1中的分析（此处有超链接）。因而，我们首先尝试使用依次转置 $4*4$ 矩阵的思路，代码和结果如下：

发现结果未达到本题的要求，

我们将矩阵B的缓存命中和未命中情况绘制成下图：



观察上图可以发现，每次访问存储B的某个 1×8 的子矩阵的某个高速缓存组时，会发生两次不命中（不考虑对角线上的情况）。这是为什么呢？我们不妨研究B矩阵的第i行、第j~j+3列($j=0, 8, 16, \dots, 56$)的子矩阵，访问该子矩阵时会将其存入一个高速缓存组。按照我们已有的代码，在访问完B的这个 1×4 子矩阵后，一段时间后会去访问B矩阵的第i+4行、第j~j+3列的子矩阵，这会导致被存在高速缓存组中的 $B[i][j:j+4]$ 被弹出。经历一定的其他步骤后，我们设计的代码需要去访问 $B[i][j+4:j+8]$ ，而高速缓存组在之前弹出存储的 $B[i][j:j+4]$ 时也同时把存储的 $B[i][j+4:j+8]$ 弹出了，这样就导致了两次缓存不命中。

为了减少访问B时出现的cache miss数，我们考虑使用之前在M=32、N=32的条件下使用过的策略，即不追求直接将A中每个元素移动到B中正确的位置，而是在之后再补救。具体来说，我们的策略如下：

考虑A的一个 8×8 子矩阵 $A[8*i:8*i+8][8*j:8*j+8]$ ，我们需要将其转置到 $B[8*j:8*j+8][8*i:8*i+8]$ 处，此处分三步完成：

Step 1: 先把 $A[8*i:8*i+4][8*j:8*j+8]$ 存入到 $B[8*j:8*j+4][8*i:8*i+8]$ 处，存完后保证 $B[8*j:8*j+4][8*i:8*i+4]$ 等于 $A[8*i:8*i+4][8*j:8*j+4]$ 的转置、 $B[8*j:8*j+4][8*i+4:8*i+8]$ 等于 $A[8*i:8*i+4][8*j+4:8*j+8]$ 的转置。这一步对于B矩阵的读写共有四次cache miss。这一步具体实现方式是依次按行处理A的 8×8 子方阵上四行，用八个变量a1 ~ a8存A的这一行，再放到B矩阵中的相应位置。Step 1的最终效果如下：

0	0	0	0	a	a	a	a
1	1	1	1	b	b	b	b
2	2	2	2	c	c	c	c
3	3	3	3	d	d	d	d
e	e	e	e	4	4	4	4
f	f	f	f	5	5	5	5
g	g	g	g	6	6	6	6
h	h	h	h	7	7	7	7

A的 8×8 子矩阵



0	1	2	3	a	b	c	d
0	1	2	3	a	b	c	d
0	1	2	3	a	b	c	d
0	1	2	3	a	b	c	d

B的 8×8 子矩阵

Step 2: 这一步是减少总的cache miss的关键。在这一步中，把A的 8×8 子矩阵左下角（即 4×4 块）的某一列4个数据读出，B右上角（即 4×4 块）的相应行4个数据读出，用8个int变量暂存它们，然后把前四个int变量填入B右上角相应行中，后四个int变量填入B的左下角相应行中。具体过程如下图（由图可知，Step 3对于B的读写仅发生四次cache miss）（下图中左边一列为A的子矩阵，右边一列为B的子矩阵）：

0	0	0	0	a	a	a	a
1	1	1	1	b	b	b	b
2	2	2	2	c	c	c	c
3	3	3	3	d	d	d	d
e	e	e	e	4	4	4	4
f	f	f	f	5	5	5	5
g	g	g	g	6	6	6	6
h	h	h	h	7	7	7	7



0	1	2	3	e	f	g	h
0	1	2	3	a	b	c	d
0	1	2	3	a	b	c	d
0	1	2	3	a	b	c	d
a	b	c	d				

0	0	0	0	a	a	a	a
1	1	1	1	b	b	b	b
2	2	2	2	c	c	c	c
3	3	3	3	d	d	d	d
e	e	e	e	4	4	4	4
f	f	f	f	5	5	5	5
g	g	g	g	6	6	6	6
h	h	h	h	7	7	7	7



0	1	2	3	e	f	g	h
0	1	2	3	e	f	g	h
0	1	2	3	a	b	c	d
0	1	2	3	a	b	c	d
a	b	c	d				
a	b	c	d				

0	0	0	0	a	a	a	a
1	1	1	1	b	b	b	b
2	2	2	2	c	c	c	c
3	3	3	3	d	d	d	d
e	e	e	e	4	4	4	4
f	f	f	f	5	5	5	5
g	g	g	g	6	6	6	6
h	h	h	h	7	7	7	7



0	1	2	3	e	f	g	h
0	1	2	3	e	f	g	h
0	1	2	3	e	f	g	h
0	1	2	3	a	b	c	d
a	b	c	d				
a	b	c	d				
a	b	c	d				

0	0	0	0	a	a	a	a
1	1	1	1	b	b	b	b
2	2	2	2	c	c	c	c
3	3	3	3	d	d	d	d
e	e	e	e	4	4	4	4
f	f	f	f	5	5	5	5
g	g	g	g	6	6	6	6
h	h	h	h	7	7	7	7



0	1	2	3	e	f	g	h
0	1	2	3	e	f	g	h
0	1	2	3	e	f	g	h
0	1	2	3	e	f	g	h
a	b	c	d				
a	b	c	d				
a	b	c	d				
a	b	c	d				

Step 3: 把A的右下角转置到B的右下角，这一步不发生cache miss，其效果如图：

0	0	0	0	a	a	a	a
1	1	1	1	b	b	b	b
2	2	2	2	c	c	c	c
3	3	3	3	d	d	d	d
e	e	e	e	4	4	4	4
f	f	f	f	5	5	5	5
g	g	g	g	6	6	6	6
h	h	h	h	7	7	7	7



0	1	2	3	e	f	g	h
0	1	2	3	e	f	g	h
0	1	2	3	e	f	g	h
0	1	2	3	e	f	g	h
a	b	c	d	4	5	6	7
a	b	c	d	4	5	6	7
a	b	c	d	4	5	6	7
a	b	c	d	4	5	6	7

综上分析，这四步最终能把A的一个8*8子方阵正确地转置到B的相应位置处，并且读写B的相应8*8子方阵时只发生4次cache miss，比之前的情况有所提升。

我们将该算法写成代码如下：

```
    } else if ( M == 64 ) { // Case2: M=64, N=64
        for ( i = 0; i < 64; i += 8 ) {
            for ( j = 0; j < 64; j += 8 ) {

                // Step 1:
                for ( k = 0; k < 4; k ++ ) {
                    a1 = A[i+k][j];
                    a2 = A[i+k][j+1];
                    a3 = A[i+k][j+2];
                    a4 = A[i+k][j+3];
                    a5 = A[i+k][j+4];
                    a6 = A[i+k][j+5];
                    a7 = A[i+k][j+6];
                    a8 = A[i+k][j+7];
                    B[j][i+k] = a1;
                    B[j+1][i+k] = a2;
                    B[j+2][i+k] = a3;
                    B[j+3][i+k] = a4;
                    B[j][i+k+4] = a5;
                    B[j+1][i+k+4] = a6;
                    B[j+2][i+k+4] = a7;
                    B[j+3][i+k+4] = a8;
                }

                // Step 2:
                for ( k = 0; k < 4; k ++ ) {
                    a1 = A[i+4][j+k];
                    a2 = A[i+5][j+k];
                    a3 = A[i+6][j+k];
                    a4 = A[i+7][j+k];
                    a5 = B[j+k][i+4];
                    a6 = B[j+k][i+5];
                    a7 = B[j+k][i+6];
                    a8 = B[j+k][i+7];
                    B[j+k][i+4] = a1;
                    B[j+k][i+5] = a2;
                    B[j+k][i+6] = a3;
                    B[j+k][i+7] = a4;
                    B[j+k+4][i] = a5;
                    B[j+k+4][i+1] = a6;
                    B[j+k+4][i+2] = a7;
                    B[j+k+4][i+3] = a8;
                }

                // Step 3:
                for ( k = i + 4; k < i + 8; k ++ ) {
                    for ( h = j + 4; h < j + 8; h ++ ){
                        B[h][k]=A[k][h];
                    }
                }
            }
        }
    } else if ( M == 61 ) { // Case3: M=61, N=67
```

最终测试结果如下：

```
trans.c
jxqhhh@jxqhhh-VirtualBox:~/Desktop/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:9018, misses:1227, evictions:1195

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=1227

TEST_TRANS_RESULTS=1:1227
```

3. Case 3: M=61, N=67

对于这种情况，分析起来较为复杂。经过尝试，直接使用分块（块的大小为16*16）的方法即可实现2000次以下的cache miss。具体代码如下：

```
        }
    } else if ( M == 61 ) { // Case3: M=61, N=67
        for ( i = 0; i < 80; i += 16 ) {
            for ( j = 0; j < 64; j += 16 ) {
                for ( k = i; ( k < i + 16 ) && ( k < 67 ); k ++ ) {
                    for ( h = j; ( h < j + 16 ) && ( h < 61 ); h ++ ) {
                        B[h][k]=A[k][h];
                    }
                }
            }
        }
    } else { // Case4: M=48, N=48
```

测试结果如下：

```
jxqhhh@jxqhhh-VirtualBox:~/Desktop/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:6187, misses:1992, evictions:1960

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=1992

TEST_TRANS_RESULTS=1:1992
```

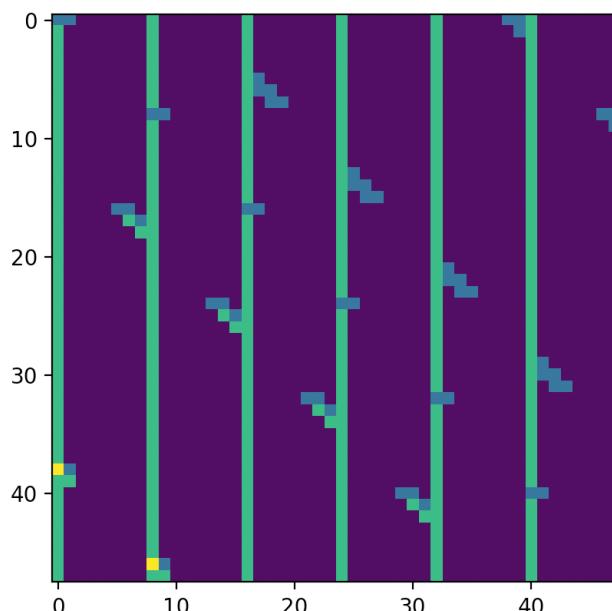
4. Case 4: M=48, N=48

采用如下思路：

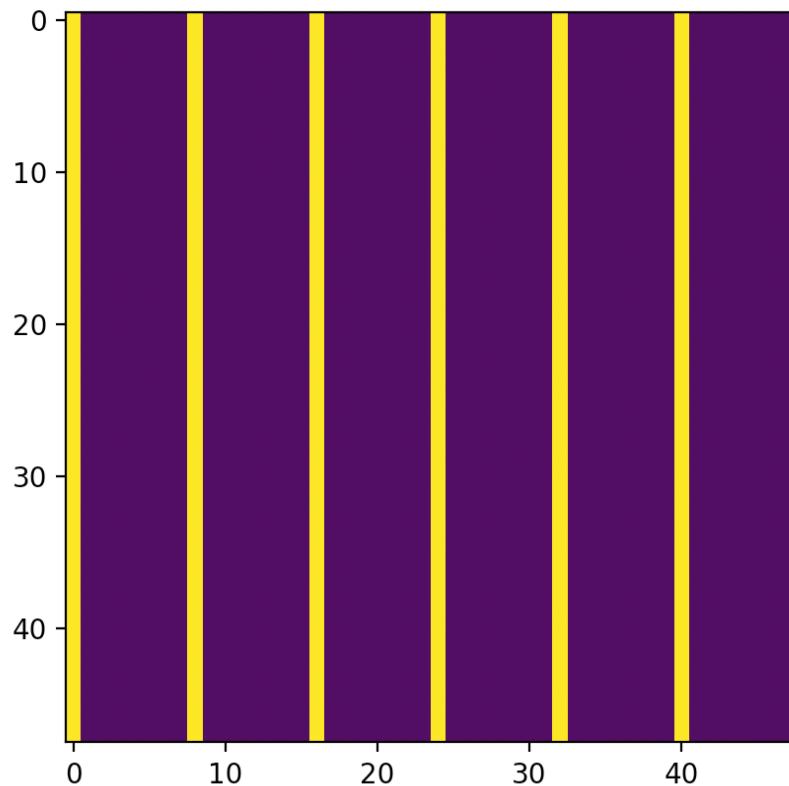
每次将A的一个8*8子矩阵转置到B的相应位置处，通过两层for循环（第二层循环内转置A的一个子矩阵）完成全部转置。处理A的每个8*8子矩阵时，按行依次处理这个子矩阵，每一行的值使用a1-a8这8个变量存下来，再放到B的相应位置处。具体代码如下：

```
        } else { // Case4: M=64, N=64
    for ( i = 0; i < 48; i += 8 ) {
        for ( j = 0; j < 48; j += 8 ) {
            // transpose the 8*8 submatrix of A:
            for ( k = i ; k < i + 8; k ++ ) {
                a1 = A[k][j+0];
                a2 = A[k][j+1];
                a3 = A[k][j+2];
                a4 = A[k][j+3];
                a5 = A[k][j+4];
                a6 = A[k][j+5];
                a7 = A[k][j+6];
                a8 = A[k][j+7];
                B[j+0][k] = a1;
                B[j+1][k] = a2;
                B[j+2][k] = a3;
                B[j+3][k] = a4;
                B[j+4][k] = a5;
                B[j+5][k] = a6;
                B[j+6][k] = a7;
                B[j+7][k] = a8;
            }
        }
    }
}
```

分析B矩阵的cache miss情况，如下图（绿色、蓝色、黄色、紫色位置处cache miss分别为2、1、3、0）：



分析A矩阵的cache miss情况，如下图（黄色位置处cache miss数为1，紫色位置处未发生cache miss）：



测试结果如下：

```
jxqhh@jxqhh-VirtualBox:~/Desktop/cachelab-handout$ ./test-trans -M 48 -N 48
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:3966, misses:647, evictions:615

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:1950, misses:2663, evictions:2631

Summary for official submission (func 0): correctness=1 misses=647
```