

**计算机系统结构实验报告**

|  |
| --- |
| 姓 名：江易星 |
| 学 院：计算机科学与技术学院 |
| 专 业：计算机科学与技术 |
| 班 级：ACM1601 |
| 学 号：U201614756 |
| 指导教师：吴非，曹强 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2019 年. 04 月. 26 日

**目 录**

[1. 第一部分：Cache模拟器 3](#_Toc7382839)

[1.1. 实验目的 3](#_Toc7382840)

[1.2. 实验环境 3](#_Toc7382841)

[1.3. 实验思路 3](#_Toc7382842)

[1.4. 实验结果和分析 6](#_Toc7382843)

[2. 第二部分：矩阵转置优化 7](#_Toc7382844)

[2.1实验目的 7](#_Toc7382845)

[2.2实验环境 7](#_Toc7382846)

[2.3实验思路 7](#_Toc7382847)

[2.4实验结果和分析 17](#_Toc7382848)

[3. 总结和体会 17](#_Toc7382849)

[4. 对实验课程的改进建议 17](#_Toc7382850)

# 第一部分：Cache模拟器

### 实验目的

1理解cache工作原理。  
2加深cache缓存组成结构对c程序性能的影响的理解。

### 实验环境

linux 64bit+c语言。

### 实验思路

1 cache的模拟：

给出的代码中对于一个cache行的结构体的定义如图1所示：

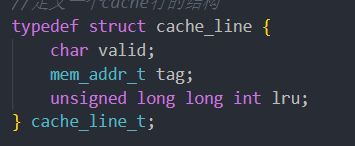


图1：cache行结构体定义

由定义可知：这里没有存储对应内存区的具体数据，只保存了所有的标志信息：valid：表示cache行内的数据是否有效。tag：内存地址的高位部分，表示cache行中存储的数据对应哪一个内存区。lru：计时器，用来是按lru替换算法。

代码中对于cache组和cache的定义如图2所示：



图2：cache组和cache 的定义

由给出的定义我们可以知道：实现模拟cache 的逻辑是：用一个cache\_line的数组来模拟一个cache组：cache\_set，然后用一个cache\_set的数组来模拟整个cache。

2 输入输出参数

给出的代码中对于程序执行所需的输入输出参数的定义如图3所示：

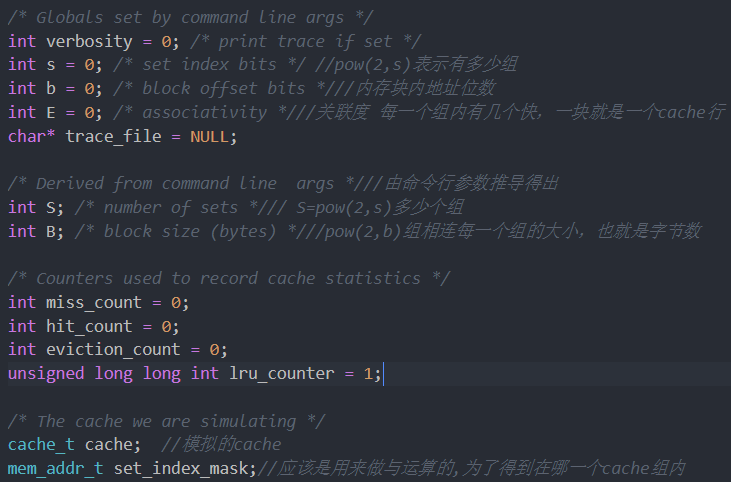


图3：输入输出参数的定义

对于访存的一个地址：按照从高位到低位可以划分为 tag,index,offset三个部分，offset表示块内偏移，index表示组索引，也就是映射到cache的哪一组中，tag为标识。s，b，E，trace\_file为四个输入的参数，s表示index的位数，b表示offset的位数，E表示每一个cache组中由多少个cache行。trace\_file是一个文件指针。指向需要测试的轨迹文件。S表示cache组数，由s决定：S=2的s次方。B表示划分cache和内存块的大小：B = 2的b次方。miss\_count,hit\_count和eviction\_count分别表示缺失次数，命中次数和淘汰次数。cache全局变量是模拟的cache，空间需要根据输入的参数动态的分配。set\_index\_mask是为了通过与运算得到某个内存地址对应的cache组。

3 freeCache

freeCache的功能是在完成整个模拟过程之后，需要将分配给cache的内存释放。对应着initCache 函数的实现我们可以知道：initCache先分配了一个cache组指针构成的数组，然后每一个cache数组在分配物理空间。所以在freeCache中，需要先释放每一个cache组的物理空间，然后释放存放每一个cache组指针的空间。

4accessData

accessData函数是对CPU访问数据的过程的模拟。当需要访问一个内存地址时，根据内存地址的index部分可以找到其cache映射块的组号，然后在组内遍历每一个cache块。如果遇到有效的cache块，则比较cache块的tag位与访问地址的tag位是否相等，相等则说明cache命中，这时需要将对应cache块的计时器lru重新清零，如果不命中则将该cache块的计时器lru加1，如果是无效的cache块则看不对其进行任何操作。  
如果在整个cache组内没有找到对应的数据块。则说明cache块缺失，需要从内存中将数据块调入，如果cache组内有空闲的块，则选择一块空闲块将内存数据存入，然后将这个cache快标记为有效块。如果cache组满了，则需要淘汰掉最近未使用的cache块，选出计时器lru最大的一块，然后将数据替换。accessData函数的流程图如图4所示：

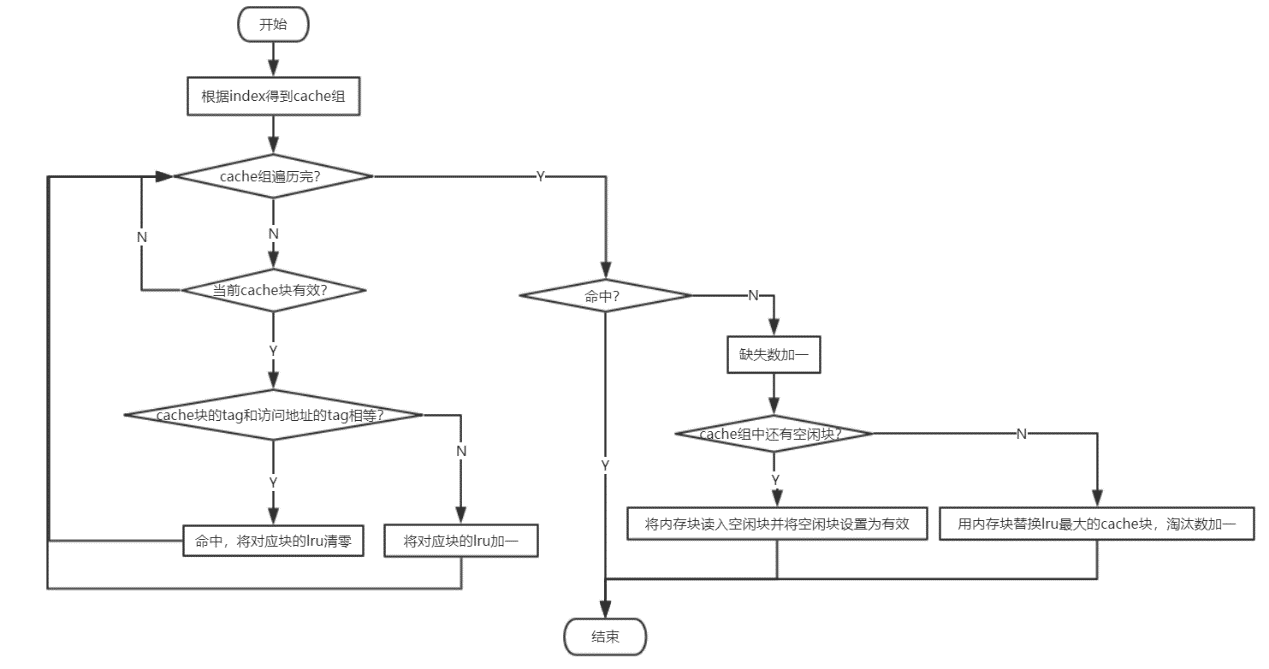


图4：accessData函数的流程图

5 replayTrace

replayTrace函数用来读取trace轨迹文件并根据文件内容模拟内存访问的过程。

设置三个参数，operation表示需要执行的操作指令，addr表示指令所访问的内存地址，len表示指令访问的地址空间的长度。依次读取每一行的三个参数，根据进来的参数进行处理：由于这里的内存访问不会超过边界，所以可以忽略第三个参数len。如果operation是I，说明会访问数据cache，这时不会访问数据cache。如果是M，表示需要进行依次读操作和依次写操作，调用两次accessData函数即可。如果是S或者L，则只需要进行一次数据访问，调用一次accessData即可。replayTrace函数的流程图如图5所示：

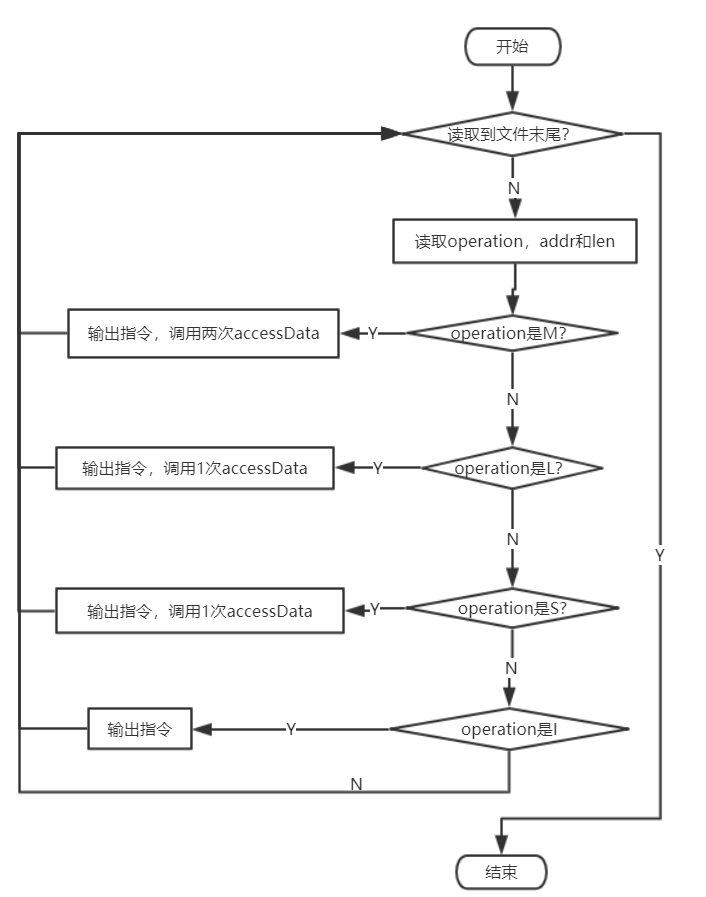


图5：replayTrace函数流程图

### 实验结果和分析

使用实验包中的自带的test-csim测试程序来测试实现的cache模拟器的正确性，结果如图6所示：

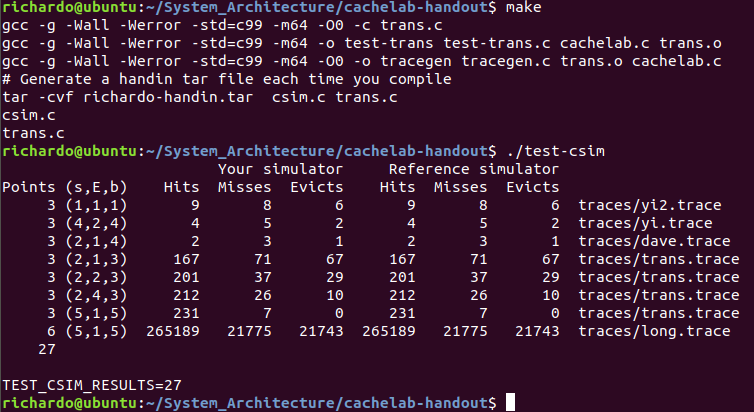


图6：cache模拟器测试结果

由图可知：实现的cache模拟程序和给出的标准的cache模拟程序的测试结果是一致的，所以可知实现的cache模拟程序是正确的。

# 第二部分：矩阵转置优化

## 2.1实验目的

一理解cache工作原理。

二加深cache缓存组成结构对c程序性能的影响的理解。

三理解各种造成cache的不命中的原因和优化cache不命中的方法。

## 2.2实验环境

linux64位+valgrind软件包+c语言。

## 2.3实验思路

由于这里的cache采用的是直接映射的cache，一个cache行为32个字节，总共由32个cache行。所以一行cache可以存储8个int型的数据，矩阵中每8个数会映射到同一个cache块中。

### 2.3.1 32\*32矩阵转置

一行32个int型的数据，一个cache行只能存下8个，所以矩阵的一行就需要4个cache行才能存下。由于矩阵在计算机内部是连续存放的，所以可以知道32\*32的矩阵，与cache块的映射关系如图7所示：

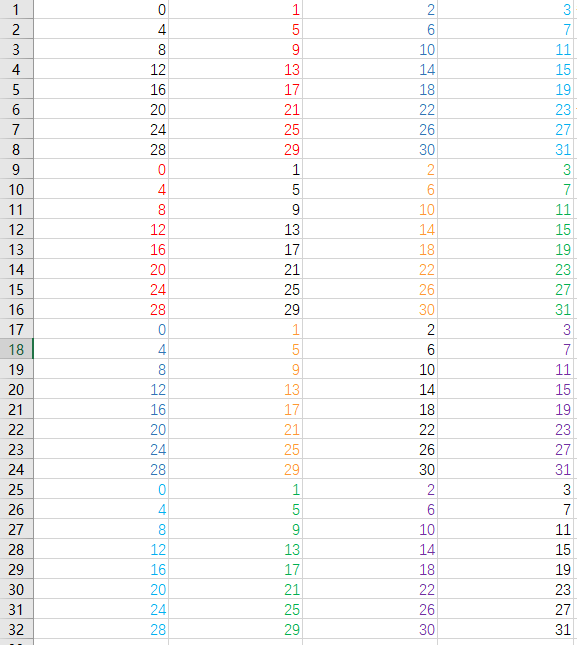


图7：32\*32矩阵数据与cache块映射关系

图中每一个格子代表8个相邻的int，数字代表这8个int映射到的cache的块号。由矩阵转置之后元素的对应关系可以知道：转置之前和转置之后同一个数字的位置是关于对角线对称的。即图中用相同颜色标注的块，转置之后相互对应，对角线上的块转置之后仍然在对角线上。

我们考虑8\*8的分块方式：由上图可知，互相对应的两个8\*8的块，他们映射到的cache块没有冲突，所以这时如果我们对两个块中的数据进行转置，则不会发生冲突不命中的情况。

代码如图8所示：

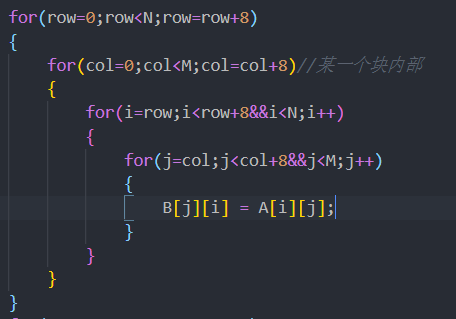


图8：8\*8分块交换代码

运行的结果如图9所示：

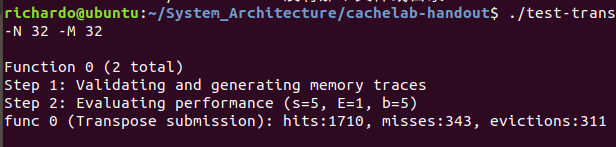


图9：8\*8分块运行结果

这时我们发现，还没有能够达到300以内的miss。

具体的原因在于对角线上块：对角线上的块没有进行优化，会发生很多的冲突不命中。取第一个对角线上的块进行分析：

1）取A[0][0]：一次强制不命中。

2）写B[0][0]：一次冲突不命中。

3）取A[0][1]：一次冲突不命中。

4）写B[1][0]：一次强制不命中。

5）取A[0][2]：命中。

6）写B[2][0]：强制不命中。

7）取A[0][3]：命中。

8）写B[3][0]：强制不命中。

9）取A[0][4]：命中。

10）写B[4][0]：强制不命中。

11）取A[0][5]：命中。

12）写B[5][0]：强制不命中。

13）取A[0][6]：命中。

14）写B[6][0]：强制不命中。

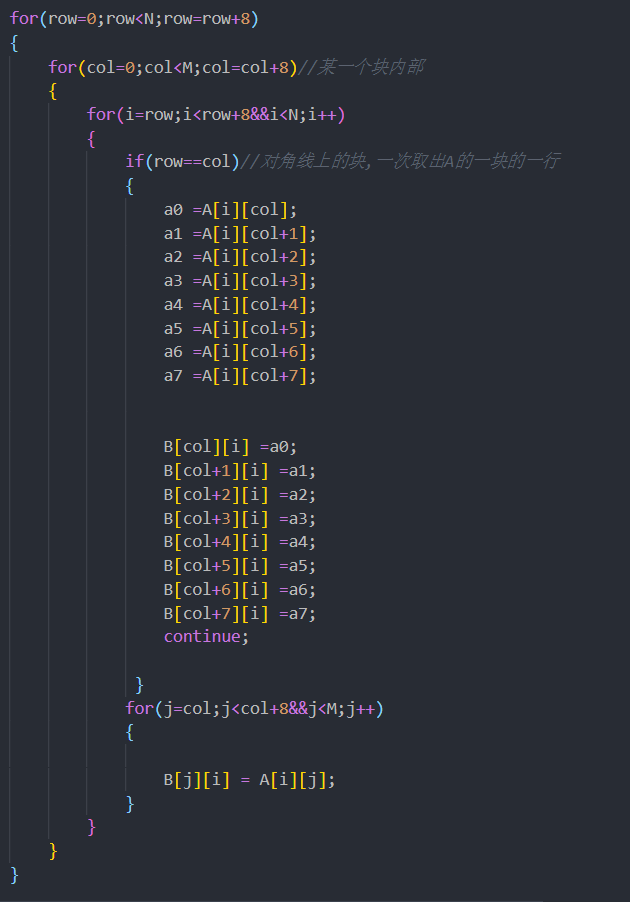
15）取A[0][7]：命中。

16）写B[7][0]：强制不命中。

。。。。。。之后每次当A的数据和B的数据映射到相同块时都会发生一次冲突不命中。

可以知道一个对角线上块，最后会发生很多的冲突不命中。

所以我们采用一次取出一个cache块，8个数字的方法，将一个cache块中的数据充分利用，然后再将这个块淘汰的思路。用8个临时变量存储8个值，然后用这8个临时变量来进行对B矩阵赋值。代码如图10所示：



如图10：优化对角块后的代码

运行结果如图11所示：

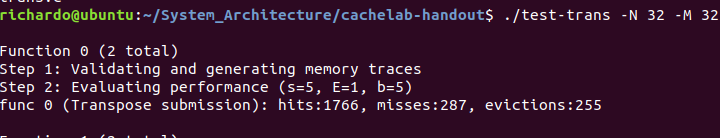


图11：优化之后的运行结果

由运行结果可以知道，这样miss的数量就优化到了300以内。

分析这里交换一个对角块的不命中：

A的一个对角块会发生8次强制不命中，

B的第一行会发生一次冲突不命中，其余的7行每一行发生一次强制不命中，发生一次冲突不命中，总15次。

总共加起来为23次。

### 2.3.2 64\*64这矩阵转置

我们先按照8\*8分块，这时数据和cache的映射关系如图12所示：

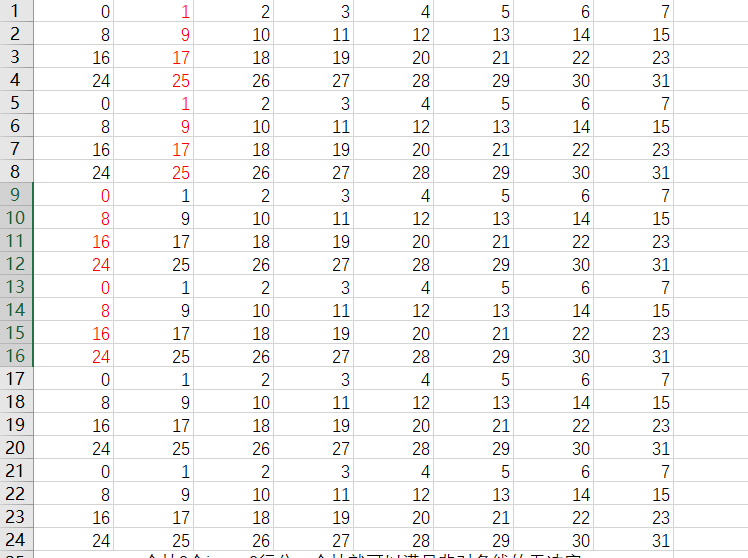


图12：cache的映射关系

这里没有画出整个64行的情况，只画出了24行的情况。这时我们发现，如果按照8\*8分块并且一次取出8个数字再淘汰的方法：

以两个对角线对称的块来分析：

1）A第一行，强制不命中一次

2）B：前四行四次强制不命中，后四行四次冲突不命中

3）A第二行,一次强制不命中

4）B：8次冲突不命中

5）A第三行，强制不命中一次

6）B：8次冲突不命中

7）A第四行，强制不命中一次

8）B：8次冲突不命中

9）A第五行，冲突不命中一次

10）B：8次冲突不命中

11）A第六行，冲突不命中一次

12）B：8次冲突不命中

13）A第七行，冲突不命中一次

14）B：8次冲突不命中

15）A第八行，冲突不命中一次

16）B：8次冲突不命中

这样，当访问B的一个块的时候，每一个数据都会发生一次冲突不命中，共64次。访问A的一个块的时候会发生8次的不命中。其中绝大多是都是冲突不命中造成的。显然不能满足优化的条件。

于是我们考虑在8\*8的大块内进行4\*4的分块，将一个8\*8的大块分成4个4\*4的块，分别称为左上块，右上块，左下块和右下块。如图13所示：

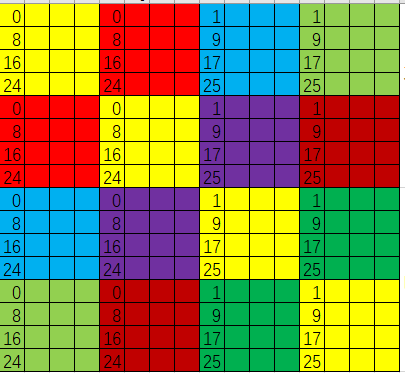


图13：8\*8内分出四个4\*4的块

如图，当我们读取第一行第三列的蓝色块时，它旁边的绿色块也会被读取到cache中。我们为了充分利用cache块的数据，分如下三步处理这个矩阵。

（1）将A左上块转置到B的左上块，将A的右上块转置到B的右上块暂时作为存储。示意图如图14所示：

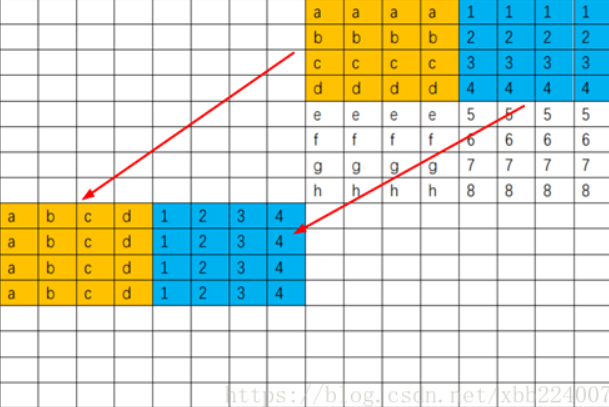


图14：第一步示意图

这样我们可以发现这一步：A会发生4次强制不命中，B会发生4次强制不命中。

（2）将B的右上块移动到B的左下块，同时将A的左下块转置到B的右上块。

示意图如图15所示：

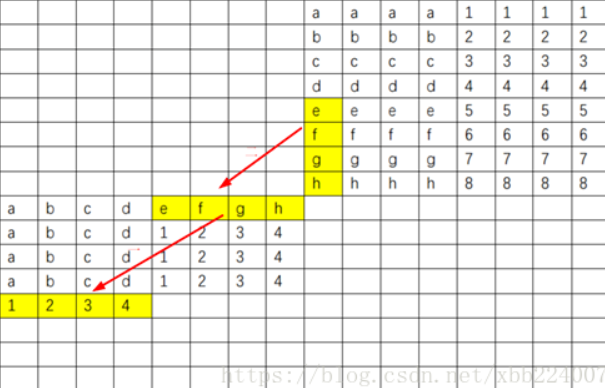


图15：第二步示意图

这里我们可以知道：A会发生4次冲突不命中，B会发生4次冲突不命中。

（3）将A的右下块转置到B的右下块，示意图如图16所示：

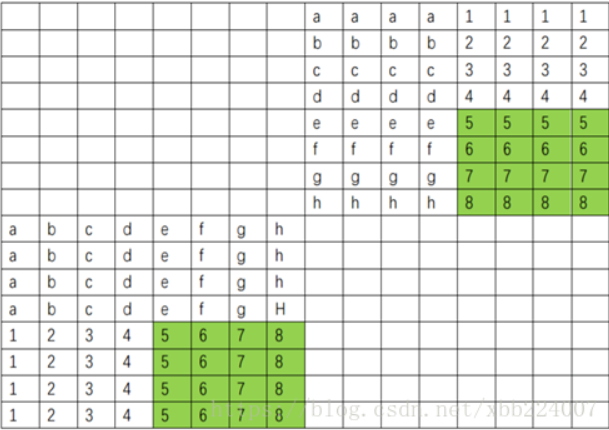


图16：第三步示意图

这一步A不会发生缺失，B也不会发生缺失。

实现的代码如下：

|  |
| --- |
| for(row=0;row<N;row=row+8)  {  for(col=0;col<M;col=col+8)//对应每一个8\*8的块,8\*8k块的内部需j=要按照4\*4来处理  {  for(i=row;i<row+4;i++)  {  a0 = A[i][col];  a1 = A[i][col+1];  a2 = A[i][col+2];  a3 = A[i][col+3];  a4 = A[i][col+4];  a5 = A[i][col+5];  a6 = A[i][col+6];  a7 = A[i][col+7];  B[col][i] = a0;  B[col+1][i] = a1;  B[col+2][i] = a2;  B[col+3][i] = a3;  B[col][i+4] = a4;  B[col+1][i+4] = a5;  B[col+2][i+4] = a6;  B[col+3][i+4] = a7;  }  for(j=col;j<col+4;j++)  {  a0 = A[row+4][j];  a1 = A[row+5][j];  a2 = A[row+6][j];  a3 = A[row+7][j];  a4 = B[j][row+4];  a5 = B[j][row+5];  a6 = B[j][row+6];  a7 = B[j][row+7];  B[j][row+4] = a0;  B[j][row+5] = a1;  B[j][row+6] = a2;  B[j][row+7] = a3;    B[j+4][row] = a4;  B[j+4][row+1] = a5;  B[j+4][row+2] = a6;  B[j+4][row+3] = a7;  }  for(i=row+4;i<row+8;i++)  {  a4 = A[i][col+4];  a5 = A[i][col+5];  a6 = A[i][col+6];  a7 = A[i][col+7];  B[col+4][i] = a4;  B[col+5][i] = a5;  B[col+6][i] = a6;  B[col+7][i] = a7;  }    } |

运行结果如图17所示：

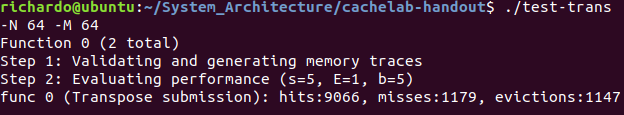


图17：64\*64矩阵转置运行结果

### 2.3.3 67\*61矩阵转置

对于67\*61的矩阵转置，由于每一行无法按照cache行的大小刚好分割，所以选择一种最简单的优化方式，逐一测试不同的分块方式所造成的miss数量。如图18所示：

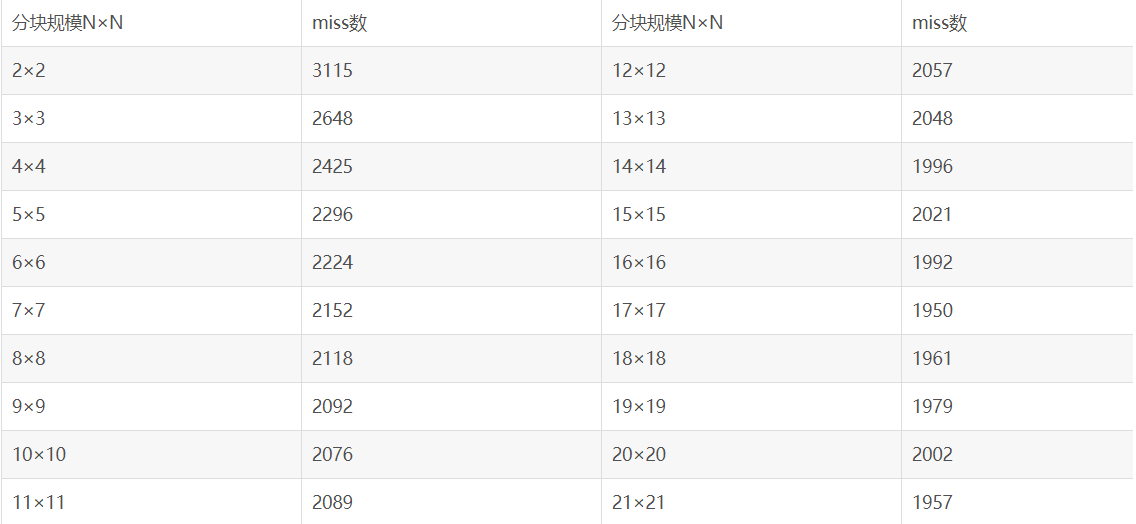


图18：不同分块方式的miss数量

如图可以发现：利用17分块时，有一个极小值，所以采用17的分块。代码如下：

|  |
| --- |
| for(row=0;row<N;row=row+17)  {  for(col=0;col<M;col=col+17)  {  for(i=row;i<row+17&&i<N;i++)  {  for(j=col;j<col+17&&j<M;j++)  {  B[j][i]=A[i][j];  }  }  }  } |

运行的结果如图19所示：

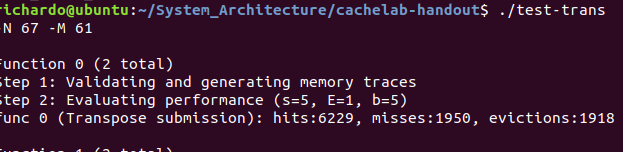


图19：17\*17分块的运行结果

## 2.4实验结果和分析

利用实验包中的driver.py脚本来测试整个优化方案的性能，结果如图20所示：

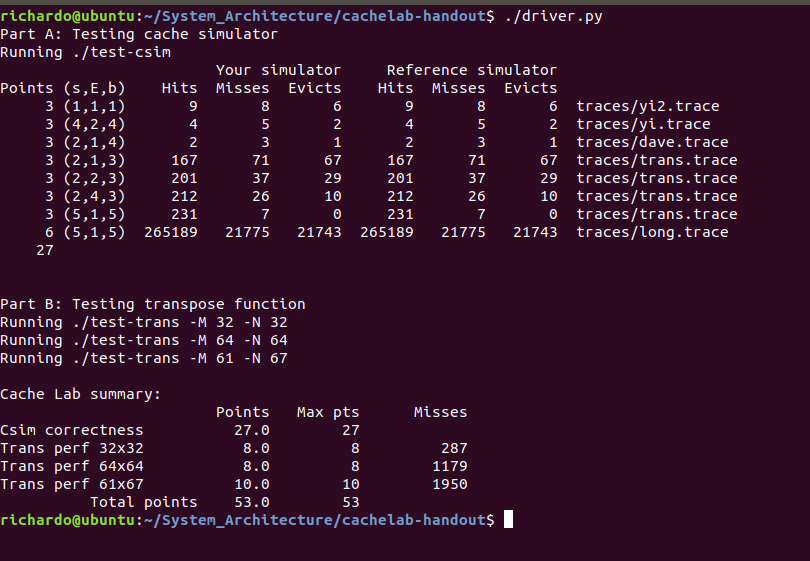


图20：整个优化测试结果

由图可知：对32\*32、64\*64以及67\*61的优化都已经达到了要求。

# 总结和体会

觉得这门课程的实验很有趣，也有一点挑战性。第一个实验复习了cache工作的基本工作原理和流程，非常的贴近我们课堂上所讲过的内容。第二个实验感觉课本上没有给出具体的例子来说明该如何分块优化cache，所以这部分如果只凭借思考时具有难度的，几乎想不到。这时就需要多取寻找一些其他的讲解，这样便于自己更加深入的理解cache分块优化的原理和实现。整个实验有很多收获，即结合和课堂上的理论知识，又通过动手加深了理解，我觉得是一个非常好的实验。

# 对实验课程的改进建议

（1）实验报告的模板不太好用，感觉格式很奇怪，建议更换一个模板。

（2）希望可以增加一些实验内容，只涉及cache这一部分，两个实验感觉量有点少。