

**计算机系统结构实验报告**

姓 名：吕鹏泽

学 院：计算机科学与技术

专 业：计算机科学与技术

班 级：CS1601

学 号：U201614532

指导教师：童薇

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

2019 年. 4 月. 23 日

**目 录**

[1. 第一部分：Cache模拟器 3](#_Toc6859990)

[**1.1.** **实验目的** 3](#_Toc6859991)

[**1.2.** **实验环境** 3](#_Toc6859992)

[**1.3.** **实验思路** 3](#_Toc6859993)

[**1.4.** **实验结果和分析** 8](#_Toc6859994)

[2. 第二部分：矩阵转置优化 10](#_Toc6859995)

[**2.1.** **实验目的** 10](#_Toc6859996)

[**2.2.** **实验环境** 10](#_Toc6859997)

[**2.3.** **实验思路** 10](#_Toc6859998)

[**2.4.** **实验结果和分析** 16](#_Toc6859999)

[3. 总结和体会 17](#_Toc6860000)

[4. 对实验课程的改进建议 17](#_Toc6860001)

# 第一部分：Cache模拟器

* 1. **实验目的**

编写一个 Cache 模拟器，其输入为内存访问轨迹，输出为命中、缺失和（缓存行）淘汰/驱逐的总数。实验要求：

* 模拟器必须在输入参数 s、E、b 设置为任意值时均能正确工作——即需要使用 malloc 函数（而 不是代码中固定大小的值）来为模拟器中数据结构分配存储空间。
* 由于实验仅关心数据 Cache 的性能，因此模拟器应忽略所有指令 cache 访问（即轨迹中“I”起始 的行）
* 假设内存访问的地址总是正确对齐的，即一次内存访问从不跨越块的边界——因此可忽略访问轨 迹中给出的访问请求大小
* main 函数最后必须调用 printSummary 函数输出结果，并如下传之以命中 hit、缺失 miss 和淘 汰/驱逐 eviction 的总数作为参数：
* 在编写完成后，使用 test-csim 程序进行测试以及评分。Cache 模拟器使用的策略应为 LRU 替换
  1. **实验环境**

操作系统 Ubuntu 18.04.1 LTS

编辑器 VSCode 1.30.1

编译器 gcc version 7.3.0

内存调试工具 valgrind 3.13.0

函数调用关系分析工具 graphviz 2.38，doxygen 1.8.14

* 1. **实验思路**

需要修改的文件是csim.c，观察源文件框架，可以得知包含以下函数：

accessData(mem\_addr\_t addr)

freeCache()

initCache()

main(int argc,char \* argv[])

printUsage(char \* argv[])

replayTrace(char \* trace\_fn)

利用doxygen对函数间的关系进行分析，可以得知各函数的调用结果如图所示：

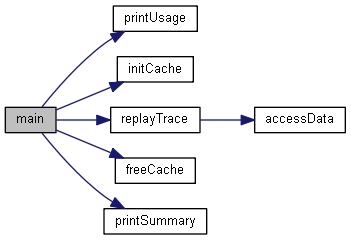


图 1‑1函数关系分析

其中printUsage用于打印帮助信息，initCache和freeCache用于初始化Cache和释放Cache，replayTrace用于分析参数并调用accessData来模拟Cache，printSummary数据模拟结果。在本实验中需要实现freeCache函数和accessData函数。

框架中给出了如下所示的Cache结构定义：

typedef struct cache\_line

{

char valid; //有效位

mem\_addr\_t tag; //标志位

unsigned long long int lru; //LRU计数器

} cache\_line\_t;

typedef cache\_line\_t \*cache\_set\_t;

typedef cache\_set\_t \*cache\_t;

valid用于标记Cache Line中的数据是否有效，tag用于查找数据时进行比较，lru用于替换。其结构如图 1‑2所示：

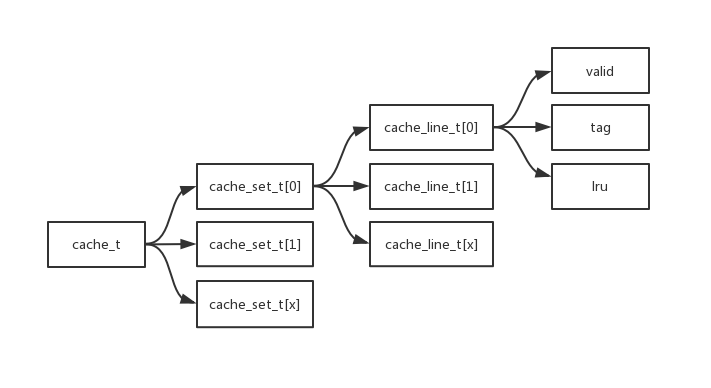


图 1‑2 Cache结构

了解了框架的结构后，接下来需要实现freeCache、 accessData和replayTrace函数，freeCache比较简单，将动态分配空间free释放掉即可，不再过多赘述。

该实验主要内容是根据LRU的替换规则设计accessData中具体的cache替换流程。首先，根据传入的地址计算组索引index以及标记位 tag，通过移位实现，并根据组索引获得 cache 组。接下来对于组中的所有cache line进行遍历，如果某一个Cache Line的有效位为1并且tag字段与计算得到的的tag字段相等，则命中，将hit计数器加1，置该Cache Line的lru为0并将该组内的其余Cache Line的lru值加1。如果遍历完毕后没有命中，miss计数器加1，然后分类讨论：

1. 当前Cache组未填满

不需要替换，寻找一个空的Cache Line插入记录，然后更新组内其他Cache Line的lru。

1. 当前Cache组已填满

eviction计数器加1。然后遍历所有Cache Line，寻找出其中lru值最大的Cache Line，将新的纪录插入到该Cache Line中，并更新其他Cache Line的lru。

具体的代码实现如下。freeCache函数用于回收内存空间，首先遍历组内所有的Cache行，使用free释放掉动态分配的内存，在最后释放Cache组指针数组。实现代码如下：

void freeCache()

{

for (int i = 0; i < S; i++) //释放cache行

{

free(cache[i]);

}

free(cache); //释放cache组指针

}

accessData函数首先要计算出index和tag，使用移位操作即可实现，实现代码如下：

mem\_addr\_t set\_index = (addr >> b) & set\_index\_mask; //cache组号

mem\_addr\_t tag = addr >> (s + b);

获得组索引index后，遍历该组内的所有Cache行，判断是否命中，如果命中则递增hits计数器，并修改Cache行的lru计数器，命中的条件是某一Cache行的有效位为1且该行的tag字段等于计算得到的tag字段。此外，本实现支持对参数-v的支持，当verbosit为1时输出执行操作。具体的代码实现如下：

cache\_set\_t cache\_set = cache[set\_index]; //获取cache组

/\*判断是否命中\*/

for (i = 0; i < E; i++)

{

if (cache\_set[i].valid && cache\_set[i].tag == tag) //命中

{

hit\_count++;

cache\_set[i].lru = 0;

if (verbosity)

printf("hit ");

for (int j = 0; j < E; j++)

if (j != i && cache\_set[j].valid) //更新其他行的计数器

cache\_set[j].lru++;

return;

}

}

如果遍历过程中命中，则修改完lru计数器后函数就会返回，否则会继续执行for循环之后的语句，即处理未命中情况的代码。

当Cache未命中时，会递增miss计数器，然后根据Cache组内是否有空行来决定是否替换。为了加快执行效率，同时查找空行和lru值最大的行，如果找到了空行就跳出循环，然后将数据加载到空行中；如果没有空行，在便利完毕后就会得到lru计数器值最大的行，将数据加载到该行。查找过程的实现如下所示：

/\*不命中\*/

if (verbosity)

printf("miss ");

miss\_count++;

int j;

/\*当cache组满时，替换掉lru计数器值最大的那一行，否则替换掉无效行\*/

for (j = 0; j < E; j++)

{

if (cache\_set[j].valid == 0) //存在无效行

{

eviction\_line = j;

break;

}

if (eviction\_lru < cache\_set[j].lru)

{

eviction\_lru = cache\_set[j].lru;

eviction\_line = j;

}

}

如果没有找到空行，需要递增eviction\_count计数器，然后组内所有Cache Line的lru计数器值加1，并将加载数据的Cache Line的lur值置0并修改valid和tag字段。具体的代码实现如下：

if (j == E) //cache组已满，需要淘汰

{

if (verbosity)

printf("eviction ");

eviction\_count++;

}

for (int k = 0; k < E; k++)

if (cache\_set[k].valid)

cache\_set[k].lru++;

cache\_set[eviction\_line].lru = 0;

cache\_set[eviction\_line].valid = 1;

cache\_set[eviction\_line].tag = tag;

replayTrace函数使用fscanf每次读入一行，然后根据访问类型进行对于 accessData 的调用：忽略 I 型访问，对于 L 型和 S 型访问调用 accessData 一次，而对于 M 型访问调用 accessData 两次。代码如下:

void replayTrace(char \*trace\_fn)

{

char buf[1000];

mem\_addr\_t addr = 0;

unsigned int len = 0;

FILE \*trace\_fp = fopen(trace\_fn, "r");

int count = 0;

while (fscanf(trace\_fp, " %c %llx,%d", buf, &addr, &len) > 0) {

if (verbosity && buf[0] != 'I')

printf("%c %llx,%d ", buf[0], addr, len);

switch (buf[0]) {

case 'I':

break;

case 'L':

case 'S':

accessData(addr);

count++;

break;

case 'M':

accessData(addr);

accessData(addr);

count++;

break;

default:

break;

}

if (verbosity && buf[0] != 'I')

putchar('\n');

}

fclose(trace\_fp);

}

* 1. **实验结果和分析**

使用make命令对代码进行编译，然后直接运行./test-csim程序对代码进行测试。实验的测试结果如图 1‑3所示，测试结果和标准测试程序csim-ref结果一致。

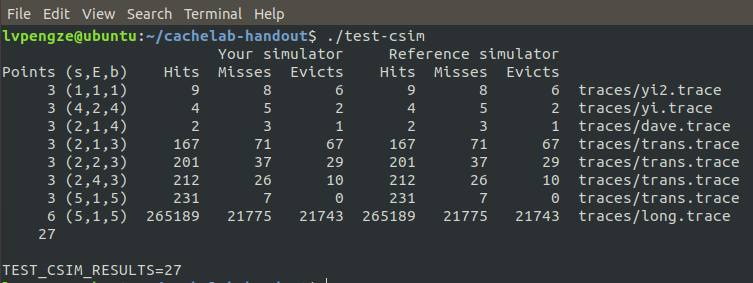


图 1‑3测试结果

# 第二部分：矩阵转置优化

* 1. **实验目的**

在trans.c中编写实现一个矩阵转置函数transpose\_submit，要求其在参考Cache模拟器csim-ref上运行时对不同大小的矩阵均能最小化缓存缺失的数量

实验需要满足以下要求：

* 限制对栈的引用——在转置函数中最多定义和使用12个int类型的局部变量，同时不能使用任何long类型的变量或其他位模式数据以在一个变量中存储多个值。
* 不允许使用递归。如果定义和调用辅助函数，在任意时刻，从转置函数的栈帧到辅助函数的栈帧之间最多可以同时存在12个局部变量。
* 转置函数不允许改变矩阵A，但可以任意操作矩阵B。
* 不允许在代码中定义任何矩阵或使用malloc及其变种。
* 32×32：如果m<300得8分，如果m>600得0分，对其他m得(600-m)\*8/300分。
* 64×64：如果m<1300得8分，如果m>2000得0分，对其他m得(2000-m)\*8/700分。
* 61×67：如果m<2000得10分，如果m>3000得0分，对其他m得(3000-m)\*10/1000分。
  1. **实验环境**

操作系统 Ubuntu 18.04.1 LTS

编辑器 VSCode 1.30.1

编译器 gcc version 7.3.0

内存调试工具 valgrind 3.13.0

函数调用关系分析工具 graphviz 2.38，doxygen 1.8.14

* 1. **实验思路**

在该实验中，s = 5，b = 5，E = 1，即Cache采用直接相连的映射方式。对于该Cache，tag字段为5位，总共存在32个组。组内偏移字段为5位，共32个字节，可以装入8个int型变量

观察程序中矩阵存储的位置即可以发现，矩阵A和矩阵B的同一行实际上被映射到了同一个缓存组。且所需优化的矩阵的总大小超出了缓存的总大小，需要冲分利用程序的局部性原理来减少Cache Miss次数。

优化过程根据时间和空间局部性原理进行优化。

* 时间局部性：尽可能的利用当前的 cache block。
* 空间局部性：在尽可能小的范围内访问数据。

**32\*32矩阵**

对于32 \* 32的矩阵，一行是32个int，一个block可以存入8个int，即一行等于4个block，Cache有32个block，可以存入8行，所以A[0][0]和A[8][0]的会映射到同一Cache行。此外，由于32\*32是Cache总容量的整数倍，因此A[i][j]的数据和B[i][j]（处理时只有i=j时才会同时访问A[i][j]和B[j][i]）的数据也会映射到同一行。由此可以推出映射冲突的情况：**只要两个int之间相差8行的整数倍，或者i=j时，那么读取这两个元素所在的block就会发生替换**，再读后面连续的元素也会不断发生替换。

将矩阵分割成8\*8的矩阵块进行转置，每次读取A的8\*8矩阵的一行数据到8个临时变量中，将读取的A[i][j]位置的数据存放到B[j][i]中，代码实现如下：

/\*将32×32的矩阵分为8×8的矩阵进行运算，由于b=5, cache每一行大小为32字节\*/

for (row = 0; row < N; row += 8)

for (col = 0; col < M; col += 8)

for (line = row; line < row + 8; line++) //访问8×8矩阵的某一行

{

//从A中取出数据

temp0 = A[line][col + 0];

temp1 = A[line][col + 1];

temp2 = A[line][col + 2];

temp3 = A[line][col + 3];

temp4 = A[line][col + 4];

temp5 = A[line][col + 5];

temp6 = A[line][col + 6];

temp7 = A[line][col + 7];

//将数据存放到B中

B[col + 0][line] = temp0;

B[col + 1][line] = temp1;

B[col + 2][line] = temp2;

B[col + 3][line] = temp3;

B[col + 4][line] = temp4;

B[col + 5][line] = temp5;

B[col + 6][line] = temp6;

B[col + 7][line] = temp7;

}

对于32 × 32的矩阵，总共存在32\*32=1024次读和1024次写。

因此对于非对角线上的12个分块， 读+写的cache miss数为8×12+8×12 = 192次，对于对角线上的分块读取A发生8×4 = 32次cache miss，B写入则发生16×4 = 64次cache miss，总共192 + 32 + 64 = 288次miss。

测试结果如图 2‑1所示，为287次miss，与理论相符。

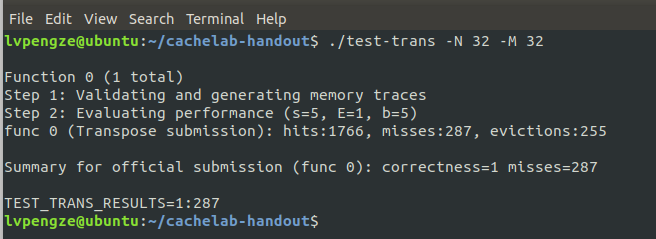


图 2‑1 32×32矩阵测试结果

**64\*64矩阵**

当矩阵规模增到到64x64的时候，每一行的数据规模增大一倍，我们发现对于任意的A中的一个块会与以下位置的数据冲突：

因此不能再使用8\*8的矩阵分块。首先考虑将矩阵分块化为4\*4，此时每次Cache中会放入8个int，我们却只用4个，浪费严重，该方法只能做到1651次miss，与实验要求的1300相差甚远。

为了提高效率，将8\*8的矩阵分为4个4\*4的矩阵，分别记为左上、右上、左下、右下。

仍然按照8\*8的矩阵块进行处理，我们首先对A和B的上半部分(4\*8的矩阵块)操作，将左上角的矩阵正常转置，将右上角的数据临时存放到B的右上角这样除了第一次以外缓存全部命中。我们把B的上半部分填满。示意图如图 2‑2所示：

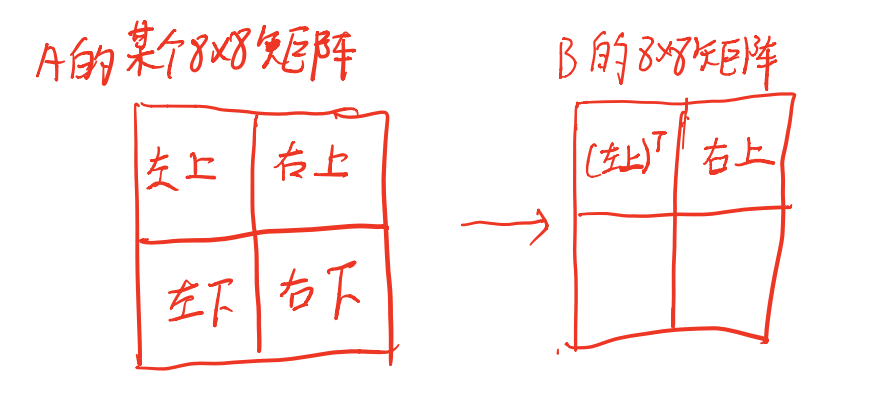


图 2‑2右上矩阵转置示意图

首先B的左上角已经正确转置。而B的右上角也填上A的右上角中的数。这里放上去的也是转置过的，我们最后只需要把这部分平移到B的左下角就好。

整个过程按照A和B的行来遍历操作。

1. 首先用四个变量存储A的左下角的一列。
2. 再用四个变量存储B的右上角的一行。
3. 把四个变量存储的A的左下角的一列移动到B右上角的一行
4. 把四个变量存储的B的右上角的一行平移到B的右下角的一行
5. 经过上面的操作，矩阵B的左上、右上、左下都完成转置。

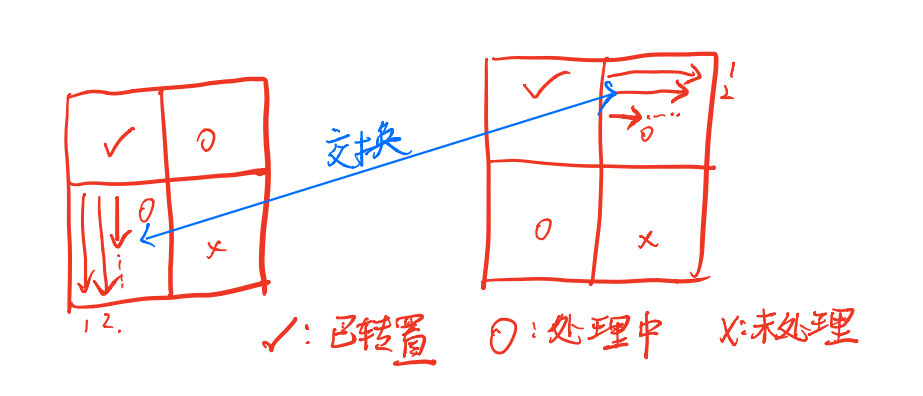


图 2‑3转置过程示意

最后，我们直接对右下角4\*4的矩阵进行转置即可。具体的代码实现如下：

for (row = 0; row < N; row += 8)

{

for (col = 0; col < M; col += 8)

{

for (line = row; line < row + 4; ++line)

{ //首先处理8×8矩阵的上半部分(4×8的矩阵)

temp0 = A[line][col + 0];

temp1 = A[line][col + 1];

temp2 = A[line][col + 2];

temp3 = A[line][col + 3];

temp4 = A[line][col + 4];

temp5 = A[line][col + 5];

temp6 = A[line][col + 6];

temp7 = A[line][col + 7];

B[col + 0][line] = temp0;

B[col + 1][line] = temp1;

B[col + 2][line] = temp2;

B[col + 3][line] = temp3;

B[col + 0][line + 4] = temp4; //临时存放

B[col + 1][line + 4] = temp5; //临时存放

B[col + 2][line + 4] = temp6; //临时存放

B[col + 3][line + 4] = temp7; //临时存放

}

for (line = 0; line < 4; line++)

{

temp0 = B[col + line][row + 4];

temp1 = B[col + line][row + 5];

temp2 = B[col + line][row + 6];

temp3 = B[col + line][row + 7];

temp4 = A[row + 4][col + line];

temp5 = A[row + 5][col + line];

temp6 = A[row + 6][col + line];

temp7 = A[row + 7][col + line];

B[col + line][row + 4] = temp4;

B[col + line][row + 5] = temp5;

B[col + line][row + 6] = temp6;

B[col + line][row + 7] = temp7;

B[col + 4 + line][row + 0] = temp0;

B[col + 4 + line][row + 1] = temp1;

B[col + 4 + line][row + 2] = temp2;

B[col + 4 + line][row + 3] = temp3;

}

for (line = 4; line < 8; line++)

for (index = 4; index < 8; index++)

B[col + index][row + line] = A[row + line][col + index];

}

}

测试结果如所示，miss数为1275，低于1300，符合要求。

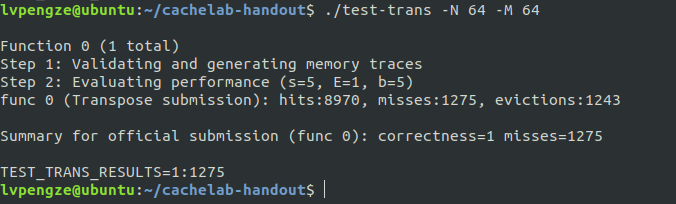
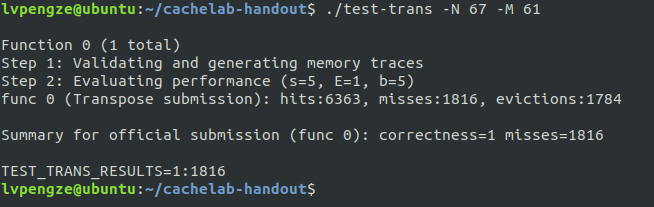


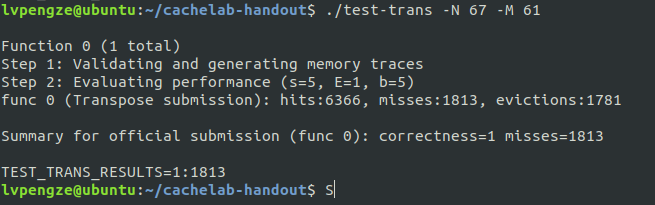
图 2‑4 64×64矩阵测试结果

**67\*61矩阵**

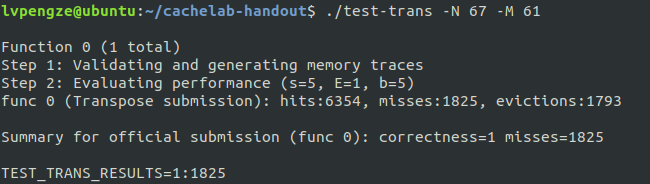
不规则的matrix，本质也是用分块来优化Cache的读写，但是不能找到比较显然的规律看出来间隔多少可以填满一个Cache。为了尽量使行之间的数据不保存到同一Cache行，按照列优先进行遍历，直接进行转置操作，尝试换用不同的边长分块即可。使用16\*16，17\*17，18\*18，19\*19的分块进行测试之后，发现按照采用17\*17的方式进行分块的得到miss数最少，为1813。测试结果如图 2‑5所示。



1. 16\*16矩阵块



1. 17\*17矩阵块



1. 18\*18矩阵块

图 2‑5 67\*61矩阵测试

程序的代码实现如下：

const int T = 17;

for (row = 0; row < 61; row += T)

for (col = 0; col < 67; col += T)

for (line = col; line < col + T && line < 67; line++)

for (index = row; index < row + T && (index < 61); index++)

{

temp0 = A[line][index];

B[index][line] = temp0;

}

* 1. **实验结果和分析**

运行测试程序driver.py，测试的运行结果如图所示，Part A部分和Part B部分均通过了测试，测试结果如图 2‑6所示。

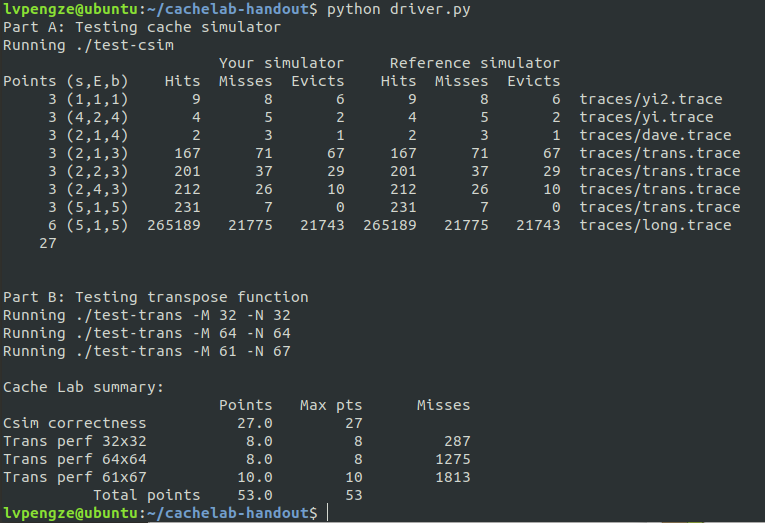


图 2‑6总体测试

# 总结和体会

通过第一个实验，我模拟了cache的行为并获得了直观的理解和感受，更加深入地了解了cache 的组成方式、与内存地址的对应关系以及 cache 组相连的工作方式，同时对cache替换算法lru的原理有了直接的使用和理解。这对我在今后编写出Cache友好型的程序会有很大的帮助。

第二个实验，从简单的矩阵转置入手，展示了Cache 是如何影响代码性能的，以及我们应该如何优化代码使之能够在 Cache 层面有更好的工作效率，以提升程序运行效率。通过对32\*32、64\*64、67\*61三个矩阵进行转置，我对cache的理解也逐步加深，尤其是第二部分的64\*64很有趣，利用B本身来作为缓存，很不好想出来，只有对Cache结构有着充分的了解才能做出来。

# 对实验课程的改进建议

希望增加Cache访问一致性的相关实验，以帮助我们充分理解写穿、写回法的策略。