**Cache Lab实验报告**

## **实验简介**

Cache Lab - Understanding Cache Memories主要是有关缓存的实验，对应于书本的第6章：存储器层次结构。主要阐明了缓存对于C语言程序的性能影响。

本实验主要分为两个部分。第一个部分要求完成一个C语言程序用来模拟缓存的行为；而第二个部分要求优化一个小的矩阵变换函数，使其具有尽可能小的缓存不命中率。

关于本实验的具体介绍详见 [实验讲义](http://csapp.cs.cmu.edu/3e/archlab32-handout.tar" \t "https://www.tuicool.com/articles/_blank) 。

**实验目的：**

使用c语言的来模拟cpu对cach的访问，然后统计hits,misses和eviction的次数，更好的理解高速缓存对于c语言程序性能的影响

## **实验要求 - Part A**

在Part A中你需要在csim.c中实现一个LRU驱逐机制的缓存模拟器，该模拟器接收valgrind的trace作为输入，模拟一个缓存在该情况下的命中/不命中情况，并且输出所有的命中，不命中以及驱逐的次数。

本实验已经提供了一个用来参考的缓存模拟器csim-ref，其使用方式以及输出如下：

Usage: ./csim-ref [-hv] -s <s> -E <E> -**b** <b> -t <tracefile>

• -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

linux> ./csim-ref -s 4 -E 1 -**b** 4 -t traces/yi.trace

hits:4 misses:5 evictions:3

linux> ./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

对于Part A有以下的要求和限制：

1. 所写的程序编译时不能有任何的警告
2. 能应对不同s，E和b的缓存生成对应的结果，这要求使用malloc来为你的数据结构分配空间
3. 不需要考虑指令加载的缓存情况
4. 调用printSummary打印最终的结果
5. 假定数据已经适当对齐，不会出现一次数据加载跨区块的情况。这样，你可以忽略trace中的大小

除此以外，实验的自学者讲义中还包含了一些提示，这里就不再赘述。

### **实验思路**

整个模拟器在思路上可以进行如下的拆分：

#### 1.命令行参数解析

由于该模拟器是命令行程序并且接受命令行参数，因此需要能对命令行参数进行解析和处理，这里推荐使用getopt进行参数的解析。并且根据不同的参数执行不同的控制流，并且处理一些基本的错误如参数缺失以及参数类型错误。

#### 2.数据结构

这是整个模拟器最基本的部分。我们需要创建合适的数据结构来模拟缓存，该数据结构不仅要能模拟缓存的数据的实际组织方式（有效位，标志位以及行和块等等），还需要考虑到LRU的驱逐机制。

#### 3.文件中内存访问记录的处理和解析

这个是模拟器中核心的部分。模拟器的功能就是从文件中接收内存访问记录，并且根据这些记录来模拟缓存的行为，操作我们所设计的缓存的数据结构。

**1. part A**

part A比较简单，就是一个C语言的模拟，不是真的写一个Cache，功能比较简易。

有如下要求，运行时有6个参数，需要处理命令行参数。有两种方法，一种如下

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

直接处理main函数的参数，第一个是命令行参数的个数，第二个是每个参数，不过都是字符串类型，如下代码处理参数：

int s,S,E,b;

FILE \*fp;

for(int i=1;i<argc;){

//printf("%s",argv[i]);

if(argv[i][0]=='-'){

if(argv[i][1] == 's'){

i++;

s = change2number(argv[i]);

S = (1<<s);

i++;

}

else if(argv[i][1] == 'E'){

i++;

E = change2number(argv[i]);

i++;

}

else if(argv[i][1] == 'b'){

i++;

b = change2number(argv[i]);

i++;

}

else if(argv[i][1] == 't'){

i++;

fp = fopen(argv[i],"r");

i++;

}

}

}

第二种方式是用getopt方法，需要头文件getopt.h，这个方法网上都是用这个的，所以我就不用了。这个方法比我上面的简单好写点。

然后是模拟算法实现，先处理读入的文件，每一行的操作指令，address，size都处理出来。

接着就是对于每条指令，模拟Cache的操作，我这边写的不太优美，没写结构体，就开了几个数组。因为是模拟 ，很多东西可以不用实现，只有几个数组是有用的。

只需要开有效位，标记位，以及最近使用时间三个数组即可，然后对于三种操作，load，store，modify，也很简单，根本不需要单独实现三个函数。

load就是加载一个数据，如果不在Cache中就miss，然后放进Cache（可能需要移除一个），store也是相同操作，所以这俩个就是一样的。modify就是load+store，执行两次同一个函数就行了，但是因为load之后，数据肯定在Cache中，所以直接输出hit就行了。

这样就大大简化了这个题目。代码不够优美，不过还是贴一下

#include "cachelab.h"

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <getopt.h>

#include <unistd.h>

int miss,hit,evictions;

char input[50];

long change2number(char \*ss){

int len=strlen(ss);

long ret=0;

for(int i=0;i<len;i++){

ret=ret\*10+(ss[i]-'0');

}

return ret;

}

int hexnumber(char c){

if(c>='0'&&c<='9') return c-'0';

if(c>='a'&&c<='f') return c-'a'+10;

if(c>='A'&&c<='F') return c-'A'+10;

return 0;

}

void op\_load(int index,int ttab,

int S,int E, int \*vaild,

long \*tab,int \*recenttime,int count){

for(int i=index\*E;i<(index+1)\*E;i++){

if(vaild[i]&&tab[i]==ttab){

printf("%s hit\n",input);

hit++;

recenttime[i]=count;

return;

}

}

for(int i=index\*E;i<(index+1)\*E;i++){

if(!vaild[i]){

vaild[i]=1;

tab[i]=ttab;

printf("%s miss\n",input);

miss++;

recenttime[i]=count;

return;

}

}

int flag=0;

int maxn=1e9;

for(int i=index\*E;i<(index+1)\*E;i++){

if(recenttime[i]<maxn){

maxn=recenttime[i];

flag=i;

}

}

printf("%s miss eviction\n",input);

miss++;evictions++;

tab[flag]=ttab;

recenttime[flag]=count;

}

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

int s,S,E,b;

FILE \*fp;

for(int i=1;i<argc;){

//printf("%s",argv[i]);

if(argv[i][0]=='-'){

if(argv[i][1] == 's'){

i++;

s = change2number(argv[i]);

S = (1<<s);

i++;

}

else if(argv[i][1] == 'E'){

i++;

E = change2number(argv[i]);

i++;

}

else if(argv[i][1] == 'b'){

i++;

b = change2number(argv[i]);

i++;

}

else if(argv[i][1] == 't'){

i++;

fp = fopen(argv[i],"r");

i++;

}

}

}

miss=0,hit=0,evictions=0;

//printf("%d %d %d\n",s,E,b);

int \*vaild = (int \*)malloc(sizeof(int)\*S\*E);

long \*tab = (long \*)malloc(sizeof(long)\*S\*E);

int \*recenttime = (int \*)malloc(sizeof(int)\*S\*E);

for(int i=0;i<S\*E;i++){

vaild[i] = 0;

tab[i] = 0;

recenttime[i] = 0;

}

int count=0;

while(fgets(input, 100,fp)){

//printf("%s",input);

count++;

int len=strlen(input)-1;

input[len]=0;

int op;

int dou=0;

int size=0;

long address = 0;

for(int i=0;i<len;i++){

if(input[i]=='I') op=0;

else if(input[i]=='L') op=1;

else if(input[i] == 'M') op=2;

else if(input[i] == 'S') op=3;

else if(input[i]==' ') continue;

else if(input[i] == ',') dou=1;

else{

if(!dou){

address = address\*16+hexnumber(input[i]);

}

else size = size\*10+(input[i]-'0');

}

}

int offset = 0;

int indexgroup = 0;

for(int i=0;i<b;i++){

offset=offset\*2+(address&1);

address>>=1;

}

for(int i=0;i<s;i++){

indexgroup = indexgroup\*2 + (address&1);

address>>=1;

}

if(op==1||op==3) op\_load(indexgroup, address, S, E, vaild, tab, recenttime,count);

if(op==2){

op\_load(indexgroup, address, S, E, vaild, tab, recenttime,count);

printf("hit\n");

hit++;

}

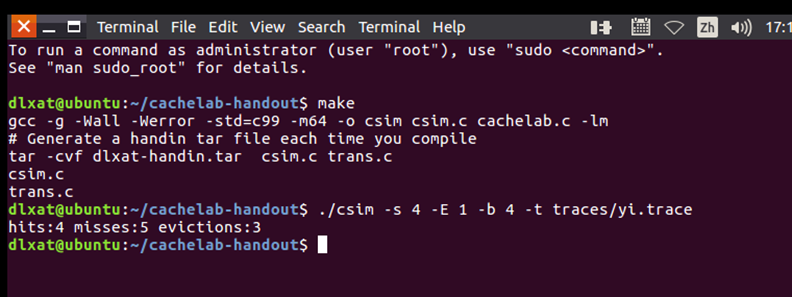
}

printSummary(hit, miss, evictions);

return 0;

}

其结果为：



最后要提的一点是这个题目，必须解决所有的warning，不然编译时warning就会变成error。

## **实验总结**

由于在寒假的课程设计中已经使用过了命令行参数解析以及复杂的数据结构设计与实验，Part A总体上来说不是很难，我主要的精力花费在代码风格上，希望代码能尽量具有良好的架构以及可读性。

本部分中犯的错误有，直接根据s设置缓存的组数，而实际上的组数是2^s个。循环中由于手误导致出现死循环。没有搞清楚inline static关键字而导致的编译错误等等。存在知识的盲区也存在着粗心导致的错误，以后还需要更加注意。

## **2. part B**

part B主要涉及提升矩阵转置的cache命中率。主要用到的将大矩阵块分解为小的矩阵来计算。   
仅在32\*32 、61 \* 67中拿到了full score。等之后对于64 \* 64有解法再来详细补充撰写该部分。

part B是优化矩阵转置算法，使用分块的方法，让Cache的miss次数越少越好

题目给的Cache大小只有 1024B，s=5，E=1，b=5。给的数据大小分别是32×32,64×64,61×67

然后只能只用12个int变量

**32×32**

第一题要求miss次数在300以下，首先观察，Cache的一个块只有32B，也就是只能容纳8个int。这个Cache可以容纳这个matrix的前8行。分块的话，肯定是取8×8的比较合适。先读取A的一行，然后放入B的一列。12个int变量，4个用来循环，其余8个用来存A中块的一行。

对于在对角线上的块，A中每读一行，会有一次miss，也就是miss次数是读取操作的1/8，对于B数组的话，第一次读取这行会产生一次miss，之后对于第i行，只有A中读到第i行的时候，会被移除出Cache，然后存的时候会产生一次miss。可以粗略计算为miss次数是读取次数的1/4。

对于不在对角线上的块，做转置的时候，A还是1/8的miss率，B的每行在Cache中和A的行不冲突 ，所以也是1/8的miss率，我们计算下最后大概多少次miss呢？

大概是

最后跑出来的答案是287，非常接近。

**64×64**

这题比较难，因为大小和前面变化了，并且次数卡的比较紧，必须1300次以下。

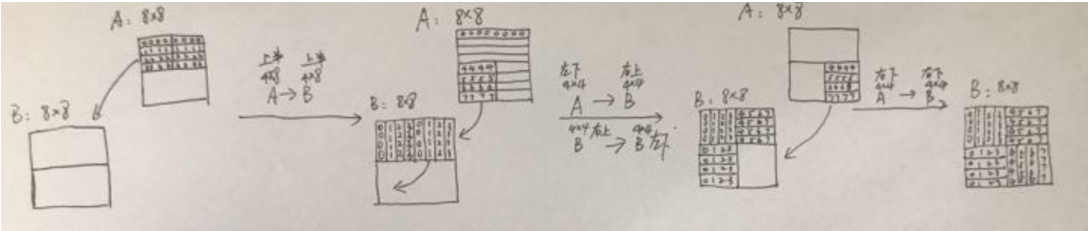
首先考虑Cache中只能放4行A中的行，如果再用8×8的块，前面4行可以填入，后面4行会在Cache中发生冲突，导致miss次数增加。

如果只用4×4的块呢？那么每次Cache中放入8个int，我们却只用4个，浪费严重，我用这个方法最少也只能做到1677次miss。

有一种很巧妙的方法，就是还用8×8的块来做，题目说A数组不能变换，但是说B数组可以任意操作啊。我们必须要一步到位嘛？可否考虑先把数字移动到B中，然后在B中自己做变化。

考虑用同样的miss次数，把更多的数据移动到B中，但是不一定是正确的位置，然后再用同样的miss次数，把A中部分数据移动到B中时，完成把B中前面位置错误数据的纠正。

如图：



画的线的分割是读入到Cache中的行以及写入到B中的顺序。（第二步有些画错了，A左下角应该是按列取数据）

1.先考虑把A的上半部分存入到B，但是为了考虑Cache不冲突，所以把右上角的4×4的区域也存在B的右上角。对于在对角线上的块，A的miss率是1/8，B的左上角部分miss率是1/2。对于不在对角线上的块，A的miss率还是1/8，B左上角部分的miss率为1/4.

2. 接下来这步是减少miss率的关键，把A左下角的一列4个数据读出，B右上角的一行4个数据读出，都用int变量暂存，然后把前四个填入B右上角行中，后四个填入B的左下角行中。

因为从B右上角读取的时候，把块放入了Cache，然后从A往B中填的时候，就不会出现miss操作。

来计算一下miss率，对于在对角线上的块，从A左下角读取miss率为1，B的右上角的操作miss率为1/4，B的左下角miss率为1/4。对于不在对角线的快，A的miss率为1/4，B右上角miss率为0，左下角miss率为1/4。

3. 最后一步就是把A的右下角填入B的右下角，对于在对角线上的块，A的miss率为1/4，B的miss率为1/2.不在对角线上的块，A，B的miss率都为0.

最后我们来计算下miss的次数吧，计算出来近似是1280次，实际我们代码跑出来是1219次 。

**61×67**

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，但是由于要求比较松，我们可以尝试一些分块的大小，直接进行转置操作。尝试到16左右 ，可以小于2000次miss。

三个题的代码如下：

char transpose\_submit\_desc[] = "Transpose submission";

void transpose\_submit(int M, int N, int A[N][M], int B[M][N])

{

int a1,a2,a3,a4,a5,a6,a7,a8;

int i,j,k,h;

if(N==32){

for(i=0;i<4;i++){

for(j=0;j<4;j++){

for(k=i\*8;k<(i+1)\*8;k++){

h=j\*8;

a1=A[k][h];a2=A[k][h+1];a3=A[k][h+2];a4=A[k][h+3];

a5=A[k][h+4];a6=A[k][h+5];a7=A[k][h+6];a8=A[k][h+7];

B[h][k]=a1;B[h+1][k]=a2;B[h+2][k]=a3;B[h+3][k]=a4;

B[h+4][k]=a5;B[h+5][k]=a6;B[h+6][k]=a7;B[h+7][k]=a8;

}

}

}

}

else if(N==64){

for(i=0;i<64;i+=8){

for(j=0;j<64;j+=8){

for(k=j;k<j+4;++k){

a1=A[k][i];a2=A[k][i+1];a3=A[k][i+2];a4=A[k][i+3];

a5=A[k][i+4];a6=A[k][i+5];a7=A[k][i+6];a8=A[k][i+7];

B[i][k]=a1;B[i][k+4]=a5;B[i+1][k]=a2;B[i+1][k+4]=a6;

B[i+2][k]=a3;B[i+2][k+4]=a7;B[i+3][k]=a4;B[i+3][k+4]=a8;

}

for(k=i;k<i+4;++k){

a1=B[k][j+4];a2=B[k][j+5];a3=B[k][j+6];a4=B[k][j+7];

a5=A[j+4][k];a6=A[j+5][k];a7=A[j+6][k];a8=A[j+7][k];

B[k][j+4]=a5;B[k][j+5]=a6;B[k][j+6]=a7;B[k][j+7]=a8;

B[k+4][j]=a1;B[k+4][j+1]=a2;B[k+4][j+2]=a3;B[k+4][j+3]=a4;

}

for(k=i+4;k<i+8;++k){

a1=A[j+4][k];a2=A[j+5][k];a3=A[j+6][k];a4=A[j+7][k];

B[k][j+4]=a1;B[k][j+5]=a2;B[k][j+6]=a3;B[k][j+7]=a4;

}

}

}

}

else{

for(i=0;i<N;i+=16){

for(j=0;j<M;j+=16){

for(k=i;k<i+16&&k<N;k++){

for(h=j;h<j+16&&h<M;h++){

B[h][k]=A[k][h];

}

}

}

}

}

}

**完整代码**：

/\*

\* Harbin Institute of Technology 1603002 1150810613 Qiuhao Li

\*

\* Anyone can modify this code and then redistribute it to help others.

\*

\* -std=c99 -m64 -O0

\*/

#include <stdio.h>

#include "cachelab.h"

char transpose\_submit\_desc[] = "Transpose submission";

void transpose\_submit(int M, int N, int A[N][M], int B[M][N])

{

if (N == 32)

{

for (int i = 0; i < N; i += 8)

{

for (int j = 0; j < M; j += 8)

{

for (int k = i; k < i + 8; ++k)

{

int temp\_value0 = A[k][j];

int temp\_value1 = A[k][j+1];

int temp\_value2 = A[k][j+2];

int temp\_value3 = A[k][j+3];

int temp\_value4 = A[k][j+4];

int temp\_value5 = A[k][j+5];

int temp\_value6 = A[k][j+6];

int temp\_value7 = A[k][j+7];

B[j][k] = temp\_value0;

B[j+1][k] = temp\_value1;

B[j+2][k] = temp\_value2;

B[j+3][k] = temp\_value3;

B[j+4][k] = temp\_value4;

B[j+5][k] = temp\_value5;

B[j+6][k] = temp\_value6;

B[j+7][k] = temp\_value7;

}

}

}

}

else if (N == 64)

{

for (int i = 0; i < N; i += 8)

{

for (int j = 0; j < M; j += 8)

{

for (int k = i; k < i + 4; ++k)

{

// 1 2

int temp\_value0 = A[k][j];

int temp\_value1 = A[k][j+1];

int temp\_value2 = A[k][j+2];

int temp\_value3 = A[k][j+3];

int temp\_value4 = A[k][j+4];

int temp\_value5 = A[k][j+5];

int temp\_value6 = A[k][j+6];

int temp\_value7 = A[k][j+7];

B[j][k] = temp\_value0;

B[j+1][k] = temp\_value1;

B[j+2][k] = temp\_value2;

B[j+3][k] = temp\_value3;

B[j][k+4] = temp\_value7;

B[j+1][k+4] = temp\_value6;

B[j+2][k+4] = temp\_value5;

B[j+3][k+4] = temp\_value4;

}

for (int l = 0; l < 4; ++l)

{

int temp\_value0 = A[i+4][j+3-l];

int temp\_value1 = A[i+5][j+3-l];

int temp\_value2 = A[i+6][j+3-l];

int temp\_value3 = A[i+7][j+3-l];

int temp\_value4 = A[i+4][j+4+l];

int temp\_value5 = A[i+5][j+4+l];

int temp\_value6 = A[i+6][j+4+l];

int temp\_value7 = A[i+7][j+4+l];

B[j+4+l][i] = B[j+3-l][i+4];

B[j+4+l][i+1] = B[j+3-l][i+5];

B[j+4+l][i+2] = B[j+3-l][i+6];

B[j+4+l][i+3] = B[j+3-l][i+7];

B[j+3-l][i+4] = temp\_value0;

B[j+3-l][i+5] = temp\_value1;

B[j+3-l][i+6] = temp\_value2;

B[j+3-l][i+7] = temp\_value3;

B[j+4+l][i+4] = temp\_value4;

B[j+4+l][i+5] = temp\_value5;

B[j+4+l][i+6] = temp\_value6;

B[j+4+l][i+7] = temp\_value7;

}

}

}

}

else

{

for (int i = 0; i < N; i += 16)

{

for (int j = 0; j < M; j += 16)

{

for (int k = i; k < i + 16 && k < N; ++k)

{

int temp\_position = -1;

int temp\_value = 0;

int l;

for (l = j; l < j + 16 && l < M; ++l)

{

if (l == k)

{

temp\_position = k;

temp\_value = A[k][k];

}

else

{

B[l][k] = A[k][l];

}

}

if (temp\_position != -1)

{

B[temp\_position][temp\_position] = temp\_value;

}

}

}

}

}

}

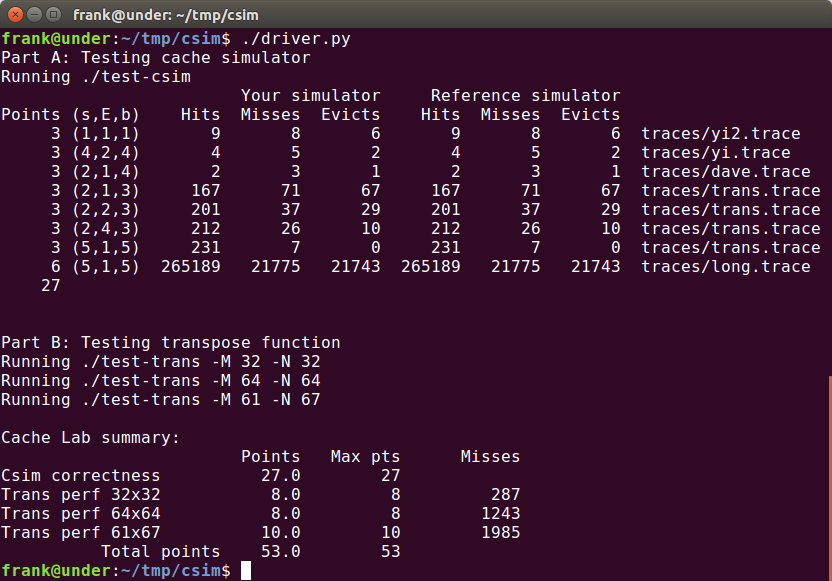
void registerFunctions()

{

registerTransFunction(transpose\_submit, transpose\_submit\_desc);

}

最终结果为满分：



**3. 总结**

part A比较简单，主要是模拟出Cache的功能。part B比较有难度，思考了很久，同时在做出来之后，还通过定量的计算miss次数，和实验结果接近，让我对Cache理解更深了。Cache这套理论非常重要，要在以后编程中编写对Cache友好的代码，我还有很多路需要走。