cachelab实验报告

罗思佳

2021201679

Part A

一、实验要求

在csim.c里实现一个cache模拟器，使用`LRU`策略，最终结果要和`csin-ref`的一样。

输入是`s E b`，以及`trace`文件，输出模拟的`hits`数、`miss`数和`eviction`数，如果输入的参数有`v`，则打印出模拟的中间过程。

对于`trace`文件，如果遇到`'I'`开头的指令，可以忽略；`'M'`开头的指令相当于一次`'L'`和`'S'`指令。

```
I 0400d7d4,8
M 0421c7f0,4
L 04f6b868,8
S 7ff0005c8,8
```

二、设计思想

为实现输入-v显示中间过程的功能，我定义了一个bool变量`isDisplay`，输入`-v`时置为`true`,否则为`false`，类似于一个开关，在函数里相应的地方控制是否打印。

1.cache结构体

因为`cache`有S个`set`，每个`set`有E个`cacheline`,所以可以设计成一个二维结构体数组`cache[S][E]`,因为s和E不确定，所以可以设为动态分配的数组，即使用指针。

数组中的元素是结构体，由`valid_bit`、`tag`、`LRU_counter`组成，`cache_set`代表由E个`cache line`组成的`set`，`cache`为二维数组的指针。

```
typedef struct
{
    int valid_bit;//有效位
    int tag;//标识
    int LRU_counter;//时间戳
} cache_line, *cache_set, **cache;
```

```
cache Cache = NULL;//定义一个空的二维数组
```

2.操作函数

init_cache()和delete_cache()

进行cache的初始化和释放

```
void init_cache() //初始化cache
{
    Cache = (cache)malloc(sizeof(cache_set) * S);
```

```
    int i,j;
    for (i = 0; i < S; i++)
    {
        Cache[i] = (cache_set)malloc(sizeof(cache_line) * E);
        for (j = 0; j < E; j++)
        {
            Cache[i][j].valid_bit = 0;
            Cache[i][j].tag = -1;
            Cache[i][j].LRU_counter = -1;
        }
    }
}
```

```
void delete_cache() //释放cache
{
    int i;
    for (i = 0; i < S; i++)
    {
        free(Cache[i]);
    }
    free(Cache);
}
```

getLRU_block()

获取最久未使用的块的`index`

```
int getLRU_block(int set_index) //获取最久未使用的index
{
    int max_LRU_counter = INT_MIN;
    int max_LRU_index = -1;
    int i;
    for (i = 0; i < E; i++)
    {
        if (Cache[set_index][i].LRU_counter > max_LRU_counter)
        {
            max_LRU_counter = Cache[set_index][i].LRU_counter;
            max_LRU_index = i;
        }
    }
    return max_LRU_index;
}
```

update_cache()

每读一条指令后进行对`cache`的更新：

先由传入的`address`计算出`set`的序号`set_index`和行的标识`_tag`，然后查找该`set_index`对应的`set`里看有没有`tag`和`_tag`相同的，如果找到了说明命中，`hit_count`加1，时间戳设置为0；如果没找到，就看有没有未使用过的`cache line`，有的话就放进去，同时`miss_count`也要加1；如果连空的line也没找到，就需要进行替换，此时先`miss_count`和`eviction_count`都加1，然后调用`getLRU_block()`函数得到最久未使用的块的`index`，将对应的块的`tag`改为`_tag`，时间戳改为0。

```
void update_cache(unsigned int address)
{
```

```
    int set_index = (address >> b) & (0xffffffffU >> (64 - s));
    int _tag = address >> (s + b);

    int i;
    for (i = 0; i < E; i++)
    {
        if (Cache[set_index][i].tag == _tag) //找到了
        {
            Cache[set_index][i].LRU_counter = 0;
            hit_count++;
            if (isDisplay)
                printf("hit ");
            return;
        }
    }
    int j;
    for (j = 0; j < E; j++) //看有没有未使用的line
    {
        if (Cache[set_index][j].valid_bit == 0)
        {
            Cache[set_index][j].valid_bit = 1;
            Cache[set_index][j].tag = _tag;
            Cache[set_index][j].LRU_counter = 0;
            miss_count++;
            if (isDisplay)
                printf("miss ");
            return;
        }
    }

    //没有空的cache_line,需要进行替换
    miss_count++;
    eviction_count++;
    int LRU_index = getLRU_block(set_index);
    Cache[set_index][LRU_index].tag = _tag;
    Cache[set_index][LRU_index].LRU_counter = 0;
    if (isDisplay)
        printf("miss eviction ");
}
```

update_LRU_counter()

用于每次更新全部block的时间戳，即全都加1。

stimulate()

该函数通过每次调用`update_cache()`实现全过程的模拟，函数逻辑为：先打开对应的trace文件，然后逐行读取，运用`switch-case`语句，如果是L或者S开头的指令，就调用一次`update_cache()`，如果是M开头的指令，就调用两次`update_cache()`（因为M相当于L加S）。

```
void stimulate(char *filename)//总的模拟函数
{
    FILE *pFile;
    pFile = fopen(filename, "r");
    if (pFile == NULL) //未成功打开
```

```
    {
        printf("fail to open");
        exit(1);
    }

    char identifier;
    unsigned int address;
    int size;

    while (fscanf(pFile, " %c %x,%d\n", &identifier, &address, &size) > 0)
    {
        if (isDisplay)
        {
            printf("%c %x,%d ", identifier, address, size);
        }
        switch (identifier)
        {
        case 'L':
            update_cache(address);
            break;
        case 'S':
            update_cache(address);
            break;
        case 'M':
            update_cache(address);//需要调用2次
            update_cache(address);
        }

        if (isDisplay)
            printf("\n");
        update_LRU_counter();//每次要整体更新时间戳
    }
    fclose(pFile);
    delete_cache();//用完释放空间
}
```

3.数据的输入

包括命令行输入参数的解析，和trace文件的输入。关于命令行的解析，文档里已经有提示了，需要用`getopt`函数取出参数的值，然后用`atoi`函数将字符转换为整数；文件的读取要用到`fscanf()`函数，包含在`stimulate()`函数中。如果输入了无效的参数（比如s<=0），就输出`“invalid input"`并返回`-1`。

```
int main(int argc, char *argv[])
{
    hit_count = 0;
    miss_count = 0;
    eviction_count = 0;
    int opt;
    char *tracefile;
    while (-1 != (opt = (getopt(argc, argv, "hvs:E:b:t:"))))
    {
        switch (opt)
        {
        case 'v':
            isDisplay = true;
            break;
```

```
        case 's':
            s = atoi(optarg);
            break;
        case 'E':
            E = atoi(optarg);
            break;
        case 'b':
            b = atoi(optarg);
            break;
        case 't':
            strcpy(tracefile, optarg);
            break;
        }
    }
    if (s <= 0 || E <= 0 || b <= 0 || tracefile == NULL) //处理无效输入
    {
        printf("invalid input\n");
        return -1;
    }
    S = 1 << s;//S = 2^s
    init_cache();
    stimulate(tracefile);
    printSummary(hit_count, miss_count, eviction_count);
    return 0;
}
```

三、实验结果

下图是不加-v和加-v的结果，加-v后输出了中间过程。

```
[2021201679@work122 cachelab-handout]$ ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:2
[2021201679@work122 cachelab-handout]$ ./csim -v -s 4 -E 2 -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
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:2
```

整体的模拟结果和`csim-ref`相同，得到了27points。

```
[2021201679@work122 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
```

Part B

实验要求

编写一个矩阵转置的函数，对于给定的`A[N][M]`，得到A的转置矩阵`B[M][N]`，并且使cache的miss数尽可能的少。测试样例有3个，分别为`32x32`、`64x64`、`61x67`

一些规则:

一共只能使用不超过12个的int型局部变量;

不能用递归;

不能改变A数组的内容;

不能定义其他数组，不能使用`malloc`函数。

PartB使用的是`csim-ref`进行评估的，`cache`的`s=5`，`E=1`，`b=5`，即有32个`set`，每个`set`有1行，每行可以存储32字节的数据。

32 × 32

1. 先看一下示例代码的miss数

```
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
```

为1183个，与目标的300相差很大。

为什么呢?

0	8	16	24	32
	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
8	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
16	0	1	2	3

假设`A[0][0]`到`A[0][8]`映射到序号为4的块，后面以此类推，如果用朴素的转置算法，举个例子，在完成了`A[0][0]`到`A[0][7]`的转置后，`B[0][0]`到`B[7][0]`也被加载到了缓存中，当访问`A[0][8]`到`A[0][15]`时，`B[8][0]`到`B[15][0]`也需要被加载到缓存，而由上图可以看到，`B[8][0]`到`B[15][0]`的set和`B[0][0]`到`B[7][0]`的set是一样的，因此在这个过程中会发生大量的eviction，miss数也会增加。

总之miss数过多的原因是在访问两个数组的过程中出现太多的冲突不命中。

2. ppt中给了提示让我们采用分块的思想，因为int是4字节，一行可以存32字节，所以一行可以存储8个数，32×32的矩阵一行需要4个cache line，所以cache可以存矩阵的8行，因此可以用8×8的分块来做。

```
if(M == 32)
    {
       int i,j,m,n;
       for( i = 0; i < N; i+=8)
         for(j = 0; j < M; j+=8)
           for(m = i; m < i+8; m++)
             for(n = j; n < j+8; n++)
                 B[n][m] = A[m][n];
    }
```

看一看结果如何：

```
Function 1 (3 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (try1): hits:1710, misses:343, evictions:311
```

343次，比之前少很多，但离300还差了一点，说明还有优化的地方。

3. 可以发现，对于处于对角线上的分块，不论是在A还是B中，都会映射到相同的set，转置期间会发生冲突不命中。对于这个情况，可以设8个局部变量暂存A里的数，然后再放到B的对应的位置。

```
if(M == 32)
    {
        int i,j,m,n,a1,a2,a3,a4,a5,a6,a7,a8;
        for(i = 0; i < N; i += 8)
          for(j = 0; j < M; j += 8)
          {
            if(i != j)
            {
              for(m = i; m < i+8; m++)
               for(n = j; n < j+8; n++)
                  B[n][m] = A[m][n];
            }
            else
            {
              for(m = i; m < i+8; m++)
              {
                a1 = A[m][j];
                a2 = A[m][j+1];
                a3 = A[m][j+2];
                a4 = A[m][j+3];
                a5 = A[m][j+4];
```

```
                a6 = A[m][j+5];
                a7 = A[m][j+6];
                a8 = A[m][j+7];

                B[j][m] = a1;
                B[j+1][m] = a2;
                B[j+2][m] = a3;
                B[j+3][m] = a4;
                B[j+4][m] = a5;
                B[j+5][m] = a6;
                B[j+6][m] = a7;
                B[j+7][m] = a8;
            }
        }
    }
}
```

运行一下看看结果，减少到了287次，小于300。

```
Function 0 (3 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
```

64 × 64

1. 先沿用上面8分块的方法试一试，结果如下，和朴素的转置算法的效果差不多。

```
Function 0 (3 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:3586, misses:4611, evictions:4579
```

为什么会这样呢?

0	8	16	24	32				64
	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
8	28	29	30	31	0	1	2	3

通过画图可以发现，每个8分块的前四行和后四行对应的set是相同的，因此在将A数组的一行向B数组搬运时，B数组的后四行会和前四行发生冲突不命中，反复下去，B数组访问的所有元素都会不命中。

2. 既然前四行和后四行的set相同，那对矩阵进行4分块呢?

```
else if(M == 64)//4分块
    {
    int i, j, k, a1, a2, a3, a4;
```

```
        for (i = 0; i < M; i += 4)
            for(j = 0; j < M; j += 4)
                for(k = i; k < (i + 4); ++k)
                {
                    a1 = A[k][j];
                    a2 = A[k][j+1];
                    a3 = A[k][j+2];
                    a4 = A[k][j+3];
                    B[j][k] = a1;
                    B[j+1][k] = a2;
                    B[j+2][k] = a3;
                    B[j+3][k] = a4;
                }
    }
```

结果如下，已经减少了许多，但离1300还是有差距，因为cache中本来可以放8个int，却只放了4个，对cache的利用效率不高。

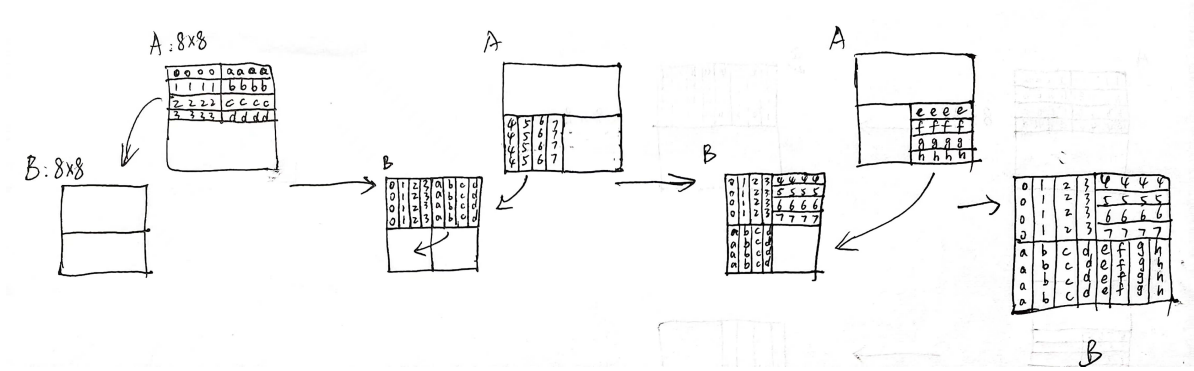
```
Function 2 (4 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 2 (try2): hits:6498, misses:1699, evictions:1667
```

3. 之后我又尝试了下8×4分块，结果是1651，并没有改善多少。

```
if(M == 64)//8×4分块
    {
        int i,j,m,n,a1,a2,a3,a4;
        for(i = 0; i < N; i += 8)
          for(j = 0; j < M; j += 4)
          {
            if(j-i != 0 && j-i != 4)
            {
              for(m = i; m < i+8; m++)
               for(n = j; n < j+4; n++)
                  B[n][m] = A[m][n];
            }
            else
            {
              for(m = i; m < i+8; m++)
              {
                a1 = A[m][j]; a2 = A[m][j+1]; a3 = A[m][j+2];a4 = A[m][j+3];
                B[j][m] = a1;B[j+1][m] = a2;B[j+2][m] = a3;B[j+3][m] = a4;
              }
            }
          }
    }
```

4. 如果还是用8分块来做，可以换一个思路，就是不像之前那样一步到位，而是先把元素移到B数组中，再调整位置。

   具体做法：先将A的前4行放到B对应的前4行，同时两个4分块进行转置，然后利用局部变量，将A的后4行的前4列的转置放到B的前4行的后4列，B的前4行的后4列移动到后4行的前4列，最后将A的后4行的后4列转置后移到B的后4行的后4列。手绘的一个简单的示意图如下：




```
else if(M == 64)
    {
        int i,j,m,n,a1,a2,a3,a4,a5,a6,a7,a8;
        for(i = 0; i < N; i += 8)
          for(j = 0; j < M; j += 8)
          {
            for(m = i; m < i+4; m++)//处理前四行
            {
              a1 = A[m][j];a2 = A[m][j+1];a3 = A[m][j+2];a4 = A[m][j+3];
              a5 = A[m][j+4];a6 = A[m][j+5];a7 = A[m][j+6];a8 = A[m][j+7];

              B[j][m] = a1;B[j+1][m] = a2; B[j+2][m] = a3;B[j+3][m] = a4;
              B[j][m+4] = a5;B[j+1][m+4] = a6;B[j+2][m+4] = a7;B[j+3][m+4] = a8;
            }
            for(n = j; n < j + 4; n++)//逐行进行后4行前四列的转置
            {
              a1 = A[i+4][n];a2 = A[i+5][n];a3 = A[i+6][n];a4 = A[i+7][n];

              a5 = B[n][i+4];a6 = B[n][i+5];a7 = B[n][i+6];a8 = B[n][i+7];

              B[n][i+4] = a1;B[n][i+5] = a2;B[n][i+6] = a3;B[n][i+7] = a4;

              B[n+4][i] = a5;B[n+4][i+1] = a6;B[n+4][i+2] = a7;B[n+4][i+3] = a8;
            }
            for(m = i+4; m < i+8; m++)//进行后四行后四列的转置
            {
              a1 = A[m][j+4];a2 = A[m][j+5];a3 = A[m][j+6];a4 = A[m][j+7];

              B[j+4][m] = a1;B[j+5][m] = a2;B[j+6][m] = a3;B[j+7][m] = a4;
            }
          }
    }
```

结果如下，达到了1179次，小于1300.

```
Function 0 (4 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:9066, misses:1179, evictions:1147
```

61 × 67

这一关也是用分块来做，因为61和67都是质数，不能明显看出要用多大的分块，所以可以一个一个地试。我是从4分块尝试到了20分块，已经有小于2000次miss的了，如16x16、17x17、18x18，从中任选一个即可，后面的就没有再尝试。

我选的是18分块，代码如下

```
else
    {
       int step = 18;
      int i,j,m,n;
      for(i = 0; i < N; i+=step)
       for(j = 0; j < M; j += step)
       {
        int min_1 = (i+step) > N ? N : (i+step);
        int min_2 = (j+step) > M ? M : (j+step);
        for(m = i; m < min_1; m++)
           for(n = j; n < min_2;n++)
             B[n][m] = A[m][n];
       }
    }
```

1961次，已经满足要求。

```
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:6218, misses:1961, evictions:1929
```

最终得分

运行./driver.py，PartA和PartB都拿到了满分。

```
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        1179
Trans perf 61x67          10.0        10        1961
          Total points    53.0        53
```

实验收获

PartA

在实现cache模拟器的过程中，我对cache的组织结构和工作原理有了更深刻的理解，也学会了如何用`getopt`函数进行命令行参数的解析，用`fscanf`函数读取文件（之前习惯用c++的方式读文件，而c比较少用），以及对于动态二维数组如何正确地`malloc`和`free`，组织代码的能力有进一步的提升。

PartB

PartB的难度比PartA大很多，需要对cache的工作细节非常熟悉，需要知道每一步数据是怎么搬运的，什么情况会出现冷不命中、冲突不命中等等，以及对于不佳的算法如何利用cache的特点进行优化来减少miss次数。这个过程很好地锻炼了我分析问题、解决问题的能力。