Project 2: Understanding Cache Memories

521021910604, Xiaohan Qin, galaxy-1@sjtu.edu.cn

May 23, 2023

1 Introduction

In this project, we are asked to complete two experiments about cache. By finishing this project, we can understand cache memories from a higher perspective. The two parts of this project are as follows:

- **Part A: Writing a Cache Simulator**
  In this part we will write a cache simulator in `csim.c` that takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.
- **Part B: Optimizing Matrix Transpose**
  In this part we will wite a transpose function in `trans.c` that causes as few cache misses as possible. In order to achieve this goal, we need to have a clear understanding of the cache structure and replacement strategy, and minimize misses by drawing simulations.

2 Experiments

2.1 Part A

2.1.1 Analysis

We are required to implement a cache simulator in Part A. First we should know the cache replacement strategy and the meaning of command line arguments. After careful reading of `project2.pdf`, I think the following three are the most critical parameters:

- `-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)

As we know, there are three cache implement ways: **fully associative, directly mapped, set associative**, which are very similar essentially. And actually the three parameters above determine the structure of the cache. That is: the cache has 2^s sets, each set has E rows, and the size of each row is 2^b.
After knowing the architecture of the cache, we should know how the instructions be like. So we just take a simple look at an test example `yi.trace`:

```
L 10,1      //load
M 20,1      //modify
L 22,1
S 18,1      //store
L 110,1
L 210,1
M 12,1
```

In this trace file, there are 3 instructions, and each of them has the form like **{Operation, Address, Size}**:

- **L(Load)**: load *Size* bytes data from *Address*.
- **M(Modify)**: load *Size* bytes data from *Address*, modify them in cache, then store them back to *Address*.(L+S)
- **S(Store)**: store *Size* bytes data from *cache* to *Address*.

In this part, we use **LRU**(least-recently used) replacement policy when choosing which cache line to evict. That means we should set a time stamp for each element in cache, and update them when visit cache again.

2.1.2 Code

First, we define a data structure **cache_line**, and then use this to define **cache_asso** and **cache**. After cache initialization, we define the operation for **L,M,S** to calculate cache hits/misses/evictions each time.

```
void update(unsigned int address)
{
        int setindex_add = (address >> b) & ((1 << s) - 1);
        //calculate the index
        int tag_add = address >> (b + s);
        //the tag
        int max_stamp = -1;
        int max_stamp_index = -1;

        for(int i = 0; i < E; ++i)
        {
                if(cache_sim[setindex_add][i].tag == tag_add &&
            cache_sim[setindex_add][i].valid == 1)
                {
                        cache_sim[setindex_add][i].stamp = 0;
                        ++hits;
                        return ;
                }
        }
                                            //if tags are same, hit++
```

```
        for(int i = 0; i < E; ++i)
        {
                if(cache_sim[setindex_add][i].valid == 0)
                {
                        cache_sim[setindex_add][i].valid = 1;
                        cache_sim[setindex_add][i].tag = tag_add;
                        cache_sim[setindex_add][i].stamp = 0;
                ++misses;                   //if has empty block, miss++
                        return ;
                }
        }
        ++evictions;
        ++misses;

        for(int i = 0; i < E; ++i)
        {
                if(cache_sim[setindex_add][i].stamp > max_stamp)
                {
                        max_stamp = cache_sim[setindex_add][i].stamp;
                        max_stamp_index = i;
                }
        }
        cache_sim[setindex_add][max_stamp_index].tag = tag_add;
        cache_sim[setindex_add][max_stamp_index].stamp = 0;
        return ;
}
```

After calculating the hits, misses and evictions each time, we need to update all the time stamps in the cache:

```
void update_stamp()
{
        for(int i = 0; i < S; ++i)
                for(int j = 0; j < E; ++j)
                        if(cache_sim[i][j].valid == 1)
                                ++cache_sim[i][j].stamp;
}
```

Finally, we use `getopt()` to get the arguments and execute instructions sequentially.

```
while(fscanf(fp, " %c %xu,%d\n", &operation, &address, &size) > 0)
{

        switch(operation)
        {
                case 'L':
                        update(address);
                        break;
```

```
                case 'M':
                        update(address);
                case 'S':
                        update(address);
        }
        update_stamp();
}
```

2.1.3 Evaluation

```
jianke@ubuntu:~/Desktop/CS2305-project/project2$ ./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
```

Figure 1: evaluation result

In Figure 1, we show our evaluation results about the cache simulator. We can see that all the hit, miss, eviction times are the same as the reference simulator, and we get the full score **27**.

2.2 Part B

2.2.1 Analysis

In Part B of this project, we will write a transpose function in `trans.c` that causes as few cache misses as possible. The difficulty of this problem lies in how to reduce the number of cache misses. Here are some main implementation techniques when I implement this part:

- **Blocking**: Divide the matrix into small blocks, and then transpose each block. This can take advantage of the principle of locality so that each block can be cached, thereby reducing the number of cache misses. The block size is different for matrices of different sizes and will be carefully designed by calculating and experimenting.
- **Lazy tranposing**: We can use up to 12 local variables, which means we can use some variables to copy A to B first, then the data in B is in the cache after copy, so we can reduce misses when we transpose B later.
- **Avoiding conflicts**: In the transpose process, it is necessary to avoid conflicts between different blocks. One way is to use different variable names to store different blocks.

In `test-trans`, the cache has the following parameters:

$$s = 5, E = 1, b = 5$$

Which means the number of sets is $2^5 = 32$, the number of line per set is 1, and the block size is $2^5 = 32$ bytes. (Actually this cache is directly mapped)
For this part, we need to consider three different sizes of the matrix: $32 \times 32, 64 \times 64, 61 \times 67$. In order to achieve a lower miss rate, we need to take different measures for matrices of different sizes. Actually in my code, I write three different functions for these three matrices, and the final `trans_sumbit()` function looks like this:

```
1  void transpose_submit(int M, int N, int A[N][M], int B[M][N])
2  {
3      if (M == 32)
4      {
5          trans_32x32(M, N, A, B);
6      }
7      if (M == 64)
8      {
9          trans_64x64(M, N, A, B);
10     }
11     if (M == 61)
12     {
13         trans_61x67(M, N, A, B);
14     }
15 }
```

`trans_32x32()`,`trans_64x64()`,`trans_61x67()` are three different functions for different sizes, each of which has no more than 12 local variables, and more details can be seen below.

2.2.2 Code and Design

32×32

32×32 matrix is the first matrix I implement, so I spend a lot time on it to have a deeper understanding of the cache so that it is easier to complete the subsequent experiments.
The original number of misses reached **1183**, which is not a small distance from the full score below 300 we require. This requires us to use block technology, because a line of cache can hold 8, so it is best to use multiples of 8 for block. In a 32 x 32 matrix, one row has 32 ints, that is, 4 cache rows, so the cache can store a total of 8 rows of the matrix. It happens that blocks with a length and width of 8 can be used without causing conflicts. (By the way, I notice that the blocks on the diagonal cause additional misses, so I use different implementation when the block is on the diagonal)

First Version of 32×32 Matrix

```
1  void trans_32x32(int M, int N, int A[N][M], int B[M][N])
2  {
```

```
    //version 1:287 misses
    int i, j, k, l, tmp;                    //5 local variables
    for (i = 0; i < N; i += 8)
    {
        for (j = 0; j < M; j += 8)
        {
            for (k = i; k < i + 8; k++)
            {
                for (l = j; l < j + 8; l++)
                {
                    if (k != l)
                    {
                        B[l][k] = A[k][l];  //not on diagonal
                    }
                    else
                    {
                        tmp = A[k][l];
                    }
                }
            if (i == j)             //on diagonal, use tmp as write buffer
                {
                    B[k][k] = tmp;
                }
            }
        }
    }
}
```

The first version of implementation get **287** misses, which is less than 300, so I get the full score. But I'm still not very clear about how do these misses come about, then I spent some more time doing simulations and calculations. The following is the specific calculation process of miss:

- when the block is on the diagonal (let i=j=0 as an example)

  misses for one block: $3 \times 7 + 2 = 23$
  (See Table 1 for details)

- when the block is not on the diagonal (let i=0, j=8 as an example)
  misses for one block: $1 \times 16 = 16$
  (See Table 2 for details)

Cache line	data
0	A[0]→B[0]
4	B[1]→A[1]→B[1]
8	B[2]→A[2]→B[2]
12	B[3]→A[3]→B[3]
16	B[4]→A[4]→B[4]
20	B[5]→A[5]→B[5]
24	B[6]→A[6]→B[6]
28	B[7]→A[7]→B[7]

Table 1: block on the diagonal

Cache line	data
0	B[8][0]∼B[8][7]
1	A[0][8]∼A[0][15]
4	B[9][0]∼B[9][7]
5	A[1][8]∼A[1][15]
8	B[10][0]∼B[10][7]
9	A[2][8]∼A[2][15]
12	B[11][0]∼B[11][7]
13	A[3][8]∼A[3][15]
16	B[12][0]∼B[12][7]
17	A[4][8]∼A[4][15]
20	B[13][0]∼B[13][7]
21	A[5][8]∼A[5][15]
24	B[14][0]∼B[14][7]
25	A[6][8]∼A[6][15]
28	B[15][0]∼B[15][7]
29	A[7][8]∼A[7][15]

Table 2: block not on the diagonal

After searching, I learned that the function overhead will cause 3 misses fixedly. There are 4 blocks on the diagonal and 12 blocks not on the diagonal, so the total misses is:

$$23 \times 4 + 16 \times 12 + 3 = 287$$

Which is the same as our result! So our calculation is correct. By the way, there are two matrices A and B, each of their row will cause at lest 4 miss due to 1 row is mapped to 4 lines in the cache, so theoretically the minimum number of misses is $4 \times 32 \times 2 = 256$. If we consider the additional function overhead, the minimum number of misses is $256 + 3 = 259$.
As we know that the extra misses are caused by blocks on the diagonal, we can think of ways to optimize the performance. Actually, here I use a strategy called **lazy-tranposing**, which use more local variables to copy data from A to B first and then transpose B.

Final Version of 32×32 Matrix

```
1  void trans_32x32(int M, int N, int A[N][M], int B[M][N])
2  {
3      //version 2:259 misses
4      int i, j, k, l, v1, v2, v3, v4, v5, v6, v7, v8;  //8 extra variables
5
6      for (i = 0; i < N; i += 8)
7      {
8          for (j = 0; j < M; j += 8)
9          {
10             // copy 8x8 matrix first: 16 misses each time
11             for (int k=i, l=j; k<i+8; k++, l++)
12             {
13                 v1 = A[k][j];
```

```
                v2 = A[k][j+1];
                v3 = A[k][j+2];
                v4 = A[k][j+3];
                v5 = A[k][j+4];
                v6 = A[k][j+5];
                v7 = A[k][j+6];
                v8 = A[k][j+7];
                B[l][i] = v1;
                B[l][i+1] = v2;
                B[l][i+2] = v3;
                B[l][i+3] = v4;
                B[l][i+4] = v5;
                B[l][i+5] = v6;
                B[l][i+6] = v7;
                B[l][i+7] = v8;
            }
            // then transpose: 0 misses
            for (k=0; k<8; k++)
            {
                for (l=k+1; l<8; l++)
                {
                    v1 = B[j+k][i+l];
                    B[j+k][i+l] = B[j+l][i+k];
                    B[j+l][i+k] = v1;
                }
            }
        }
    }
}
```

In this version, we get **259** misses, which means the **theoretical minimum** and **SOTA** of this part.

64×64

Due to size limitation, I will not describe the implementation method in detail. The idea is similar to 32x32, but the specific implementation has changed because the number of matrix rows that can be stored in the cache has changed from 8 to 4. I still use 8x8 blocks, but I subdivide each block into 4x4 blocks for detailed operations. The schematic diagram of the operation is as Figure 2:

Final Misses: 1083

```
void trans_64x64(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, k, tmp, v1, v2, v3, v4, v5, v6, v7, v8;
    for (i = 0; i < N; i += 8)
        for (j = 0; j < M; j += 8)
        {
```

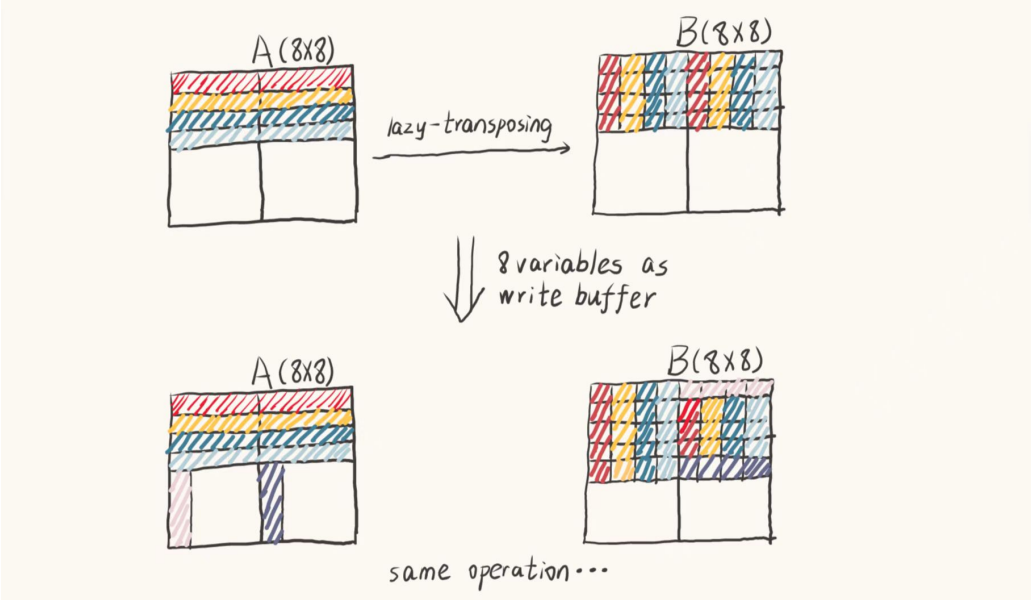

Figure 2: 64x64 impletation

```
7           for (k = 0; k < 4; k++)
8           {
9               // A top left
10              v1 = A[k + i][j];
11              v2 = A[k + i][j + 1];
12              v3 = A[k + i][j + 2];
13              v4 = A[k + i][j + 3];
14
15              // copy
16              // A top right
17              v5 = A[k + i][j + 4];
18              v6 = A[k + i][j + 5];
19              v7 = A[k + i][j + 6];
20              v8 = A[k + i][j + 7];
21
22              B[j + k][i + 0] = v1;
23              B[j + k][i + 1] = v2;
24              B[j + k][i + 2] = v3;
25              B[j + k][i + 3] = v4;
26              B[j + k][i + 4] = v5;
27              B[j + k][i + 5] = v6;
28              B[j + k][i + 6] = v7;
29              B[j + k][i + 7] = v8;
30          }
```

```
            for (k = 0; k < 4; k++)
            {
                for (v1 = k + 1; v1 < 4; v1++)
                {
                    tmp = B[j + k][i + v1], B[j + k][i + v1] = B[j + v1][i + k],
                    B[j + v1][i + k] = tmp;
                    tmp = B[j + k][i + 4 + v1],
                    B[j + k][i + 4 + v1] = B[j + v1][i + 4 + k],
                    B[j + v1][i + 4 + k] = tmp;
                }
            }
            for (k = 0; k < 4; k++)
            {
                // step 1 2
                v1 = A[i + 4][j + k], v5 = A[i + 4][j + k + 4];
                v2 = A[i + 5][j + k], v6 = A[i + 5][j + k + 4];
                v3 = A[i + 6][j + k], v7 = A[i + 6][j + k + 4];
                v4 = A[i + 7][j + k], v8 = A[i + 7][j + k + 4];
                // step 3
                tmp = B[j + k][i + 4], B[j + k][i + 4] = v1, v1 = tmp;
                tmp = B[j + k][i + 5], B[j + k][i + 5] = v2, v2 = tmp;
                tmp = B[j + k][i + 6], B[j + k][i + 6] = v3, v3 = tmp;
                tmp = B[j + k][i + 7], B[j + k][i + 7] = v4, v4 = tmp;
                // step 4
                B[j + k + 4][i + 0] = v1, B[j + k + 4][i + 4 + 0] = v5;
                B[j + k + 4][i + 1] = v2, B[j + k + 4][i + 4 + 1] = v6;
                B[j + k + 4][i + 2] = v3, B[j + k + 4][i + 4 + 2] = v7;
                B[j + k + 4][i + 3] = v4, B[j + k + 4][i + 4 + 3] = v8;
            }
        }
}
```

61×67

This is all the same with above, I just do some test and choose a better parameter to divide block.

Final Misses: 1758

```
void trans_61x67(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, k, s, v1, v2, v3, v4, v5, v6, v7, v8;
    for (i = 0; i < N; i += 8)
    for (j = 0; j < M; j += 1)
    {
        if (i + 8 <= N && j + 1 <= M)
        {
```

```
            for (s = j; s < j + 1; s++)
            {
                v1 = A[i][s];
                v2 = A[i + 1][s];
                v3 = A[i + 2][s];
                v4 = A[i + 3][s];
                v5 = A[i + 4][s];
                v6 = A[i + 5][s];
                v7 = A[i + 6][s];
                v8 = A[i + 7][s];
                B[s][i + 0] = v1;
                B[s][i + 1] = v2;
                B[s][i + 2] = v3;
                B[s][i + 3] = v4;
                B[s][i + 4] = v5;
                B[s][i + 5] = v6;
                B[s][i + 6] = v7;
                B[s][i + 7] = v8;
            }
        }
        else
        {
            for (k = i; k < min(i + 8, N); k++)
            {
                for (s = j; s < min(j + 1, M); s++)
                {
                    B[s][k] = A[k][s];
                }
            }
        }
    }
}
```

2.2.3 Evaluation

The final evaluation results are shown in Figure 3, we completed all tasks and got full marks.

3 Conclusion

3.1 Problems

In this project, my problems can be can be summarized as follows:

- in Part A, I am not clear with the cache structure and LRU algorithm, which takes me some time to understand and learn.

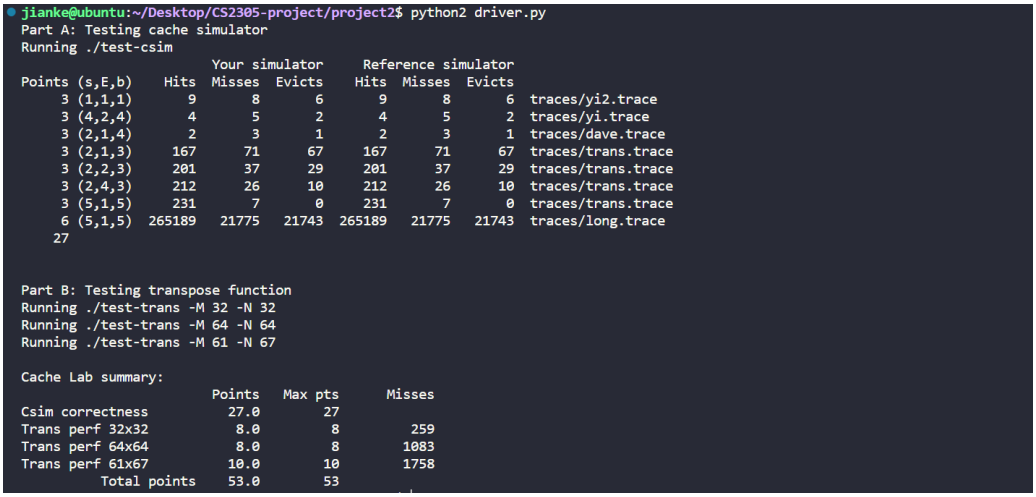

Figure 3: Final evaluation results

- in Part B, I just know how to reduce misses but I do not know how the precise misses number is, so it takes me some extra time to do simulations and calculations.
- about 32x32 matrix, I get full score with 287 easily, but it takes me more time to think and get 259(the **SOTA**)

3.2 Achievements

In this project, my achievements can be summarized as follows:

- in Part A, I successfully implemented the cache simulator, which works the same as csim-ref, and get full scores.
- about 32x32 matrix, I write 2 versions for the transposition, which reached **287** misses and **259** misses respectively. What's more, I **precisely calculated why misses are like this** and **made detailed flow sheets**. Finally, I got **SOTA** for this part.(256+3)
- about 64x64 matrix, I drew a diagram of the operations implemented (Figure 2), and got **1083** misses. Although this is not the theoretical optimal solution(1024+3), I think this is much better than many other results.
- about 61x67 matrix, I did many experiments to get a better performance. Finally I got **1758** misses, which I think is much better than many other results as well.
- All the codes I write are readable, and I add many comments when necessary, which means the codes are explained very specifically.