Project 2: Understanding Cache Memories

Zhang, @sjtu.edu.cn

May 28, 2023

1 Introduction

In this project, I need to do two parts of work.

In Part A, I need to write a cache simulator in `csim.c` and it simulates the hit/miss behavior of a cache memory with the trace input. I need to output the total number of hits, misses, and evictions. I should write this simulator with the LRU replacement policy when choosing a cache line to evict.

In Part A, I reviewed how the cache works carefully, thought about the structure of the cache, and finally wrote a cache simulator in `csim.c` which received full grades.

In Part B, I need to optimize matrix transpose. I need to write a transpose function in `trans.c` to try my best to reduce the misses during the matrix's transposition.

First, I read through the related PDF provided by CMU's teacher and understood how the misses happen.

Then, when coding, for a 32x32 matrix's transposition, I use an 8x8 blocking technique and special diagonal handling.

For a 64x64 matrix's transposition, I use an 8x8 blocking and apply a special handling method, which will be described in detail in the following report.

For a 61x67 matrix, I use a 1x8 blocking to accomplish this part. Eventually, I completed it successfully and received full grades.

2 Experiments

2.1 Part A

2.1.1 Analysis

In Part A, I need to write a cache simulator in `csim.c` that takes a `valgrind` memory trace as input. It simulates the hit/miss behavior of a cache memory based on the trace input. I need to use function `printSummary()` to output the number of hits, misses, and evictions. This simulator should be implemented with the LRU replacement policy when selecting a cache line to evict. I have

been provided with a standard cache simulator called `csim-ref`, which provides the expected output. Our simulator's output should match the expected output when our simulator is functioning correctly.

To understand the complete working principle of the cache, thorough comprehension is crucial; otherwise, I won't be able to simulate its behavior.That's where the difficulty point lies. To achieve this, I reviewed the class PoIrPoint slides and searched for additional reference information on the Internet. Additionally, I revisited the relevant chapters in CSAPP for a deeper understanding of cache operations.

The key challenge lies in designing a cache simulator that calculates the number of hits, misses, and evictions without performing actual store/load operations. It is important to program this simulator correctly. To accomplish this, I divided the process of writing my program into three parts. Firstly, I used `getopt()` to retrieve parameters and values from `argv`. Secondly, I built the cache based on the input parameters and started reading commands from the input file. Lastly, I implemented a function called `cache_access()` to simulate the core cache operations that determine hits, misses, or evictions. Additionally, I made sure to free the allocated memory space once the simulation is completed.

2.1.2 Code

Only the main part of codes are shown here.

```
/*
 * name:
 * ID:
*/
int hit = 0,miss = 0,evict = 0; //record of hit,miss,evict times

int verbose_flag = 0;           //-v parameter
int time = 0;                   //time stamp to record the time for LRU
struct cache_line {             //cache line struct
    int valid_bit;
    ull tag;
    int time_stamp;
};
struct cache_line **cache;

int s,b,E,S,B;

void cache_access(ull address_but_b_bits,int modify_flag) {
    // (tag + s) bits which remove loIr b_bits

    ull set_index = address_but_b_bits - ((address_but_b_bits >> s) << s);
//get the set number
```

```
    ull tag = address_but_b_bits >> s;
    //hit code...
    //miss and no evict code...
    //miss and evict
    int mn_time_stamp = 0x7f7f7f7f;      //record the minimum time stamp
    for(int i = 0; i < E; ++i) {
        mn_time_stamp = min(mn_time_stamp, cache[set_index][i].time_stamp);
    }
    for(int i = 0; i < E; ++i) {
        if(mn_time_stamp == cache[set_index][i].time_stamp) {
            evict += 1;
            miss += 1;
            if(modify_flag == 1) hit += 1;
            cache[set_index][i].tag = tag;
            cache[set_index][i].time_stamp = time;
            cache[set_index][i].valid_bit = 1;
            if(verbose_flag) {
                if(modify_flag)
                    printf(" miss eviction hit\n");
                else
                    printf(" miss eviction\n");
            }
        }
    }
}
int main(int argc,char **argv)
{
    int opt;
    char *name;
    name = malloc(NAME_MAX_LENGTH * sizeof(char));
    while(-1 != (opt = getopt(argc, argv, "vs:E:b:t:"))) {
        switch(opt) {       //parse the input
            case 'v':
            case 's':       //set number...
            case 'E':       //set associativity...
            case 'b':       //block offset...
            case 't':
        }
    }
    //get 2^s 2^b
    S = 1,B = 1;
    for(int i = 0; i < s; ++i) S *= 2;
    for(int i = 0; i < b; ++i) B *= 2;
    //malloc cache[S][E]
    cache = (struct cache_line **) malloc (sizeof(struct cache_line *) * S);
    //initial cache ...
```

```
    FILE *pFile = fopen(name,"r");
    if(pFile == NULL) {
        printf("open error %d\n",E);
    }
    char identifier;
    ull address;
    int size;
    while(fscanf(pFile, " %c %llx,%d", &identifier, &address, &size) > 0) {
        if(identifier == 'I') continue;
        if(verbose_flag)
            printf("%c %llx,%d",identifier,address,size);

        ull start_address = address;
        ull start_E = start_address >> b;    //remove lower b bits
        if(identifier == 'L') {              //load
            time += 1;
            cache_access(start_E,0);
        } else if(identifier == 'S') {       //store
            time += 1;
            cache_access(start_E,0);
        } else if(identifier == 'M') {       //modify
            time += 1;
            cache_access(start_E,1);
        }
    }
    printSummary(hit, miss, evict);
    //free malloc space...
    return 0;
}
```

2.1.3 Evaluation

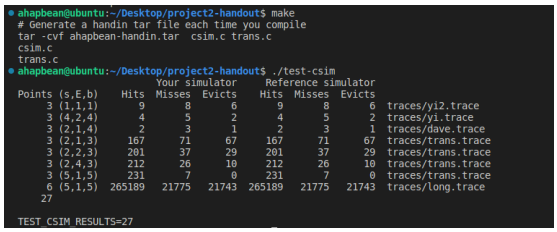

Figure 1: the evaluation result of csim.c

Eventually, see Figure 1. I passed the yi2.trace, yi.trace, dave.trace, trans.trace, and long.trace,testing with different values of s, E, and b. The output of my

cache simulator matched the results of the reference simulator, and I received 27 points (full grades).

It's noteworthy that I added the `verbose` mode to my cache simulator, which helps to output detailed cache trace information.

2.2 Part B

2.2.1 Analysis

In this part, I need to optimize matrix transpose with three kinds of matrix with size of 32x32/64x64/61x67, namely writing a transpose function in 'trans.c' that causes as few cache misses as possible. Actually, miss means performance loss. So when given the cache structure and the cache line replacement policy, it's important for a programmer to be aware of the existence of cache misses and know how to reduce the misses when programming. This part is trying to lead us to think from this perspective and let us be familiar with cache miss optimization techniques which are beneficial to us.

Back to the task itself, this part can be divided into three parts: first, 32x32 transposition optimization; second, 64x64 transposition optimization; third, 61x67 transposition optimization. Because their blocking methods are different, I need to analyze them one by one.

The difficult point lies in knowing where and why the miss happens and combining miss calculation to calculate the lowest miss bound when using different blocking techniques. If the lowest bound is not achieved, I need to use the trace file to figure out where the additional misses come from.

Given the parameters s=5, E=1, b=5, which means I have 32 sets in the cache, one cache line is of 32 bytes (4 int), and E=1 means the cache is direct-mapped.

The core technique here is to determine the size of the blocking matrix and, after properly blocking, analyze the eviction relationship between A and B. By calculation, careful analysis, and conducting numerous experiments, finally, for a 32x32 matrix, I use an 8x8 blocking and apply a special treatment to the diagonal elements. For a 64x64 matrix, I still use an 8x8 blocking, but I employ specific operations to minimize the misses. For a 61x67 matrix, I use an 8x blocking to transform this problem into first transposing a 61x64 matrix and then transposing a 61x3 matrix.

2.2.2 Code

The core code shown here is divided into three parts including 32x32 ,64x64,61x67 matrix operation.Attention that ... means code omission.

- 32x32 matrix transpose optimization code.

```
if(M == 32 && N == 32) {
    for(i = 0; i < N; i += 8) {        //8x8 blocking
```

```
            for(j = 0; j < M; j += 8) {
                if(i == j) {              //diagonal handle
                    for(x = i; x < i + 8; ++x) {
                        if(x != i)        //avoid race in diagonal
                            B[x - 1][x - 1] = _1;
                        for(y = j; y < j + 8; ++y) {
                            if(x == y) {
                                _1 = A[x][y];
                            } else {
                                _2 = A[x][y];
                                B[y][x] = _2;
                            }
                        }
                        if(x == i + 7)
                            B[x][x] = _1;
                    }
                } else for(x = i; x < i + 8; ++x) {
                    for(y = j; y < j + 8; ++y) {
                        _1 = A[x][y];
                        B[y][x] = _1;
                    }
                }
            }
        }
    }
```

- 64x64 matrix transpose optimization code.

```
    if(M == 64 && N == 64) {
        for(i = 0; i < N; i += 8) {
            for(j = 0; j < M; j += 8) {
                for(x = 0; x < 4; ++x) {//copy top 4x8 matrix first
                    _1 = A[i + x][j];    //...
                    _8 = A[i + x][j + 7];

                    B[j + x][i] = _1;    //...
                    B[j + x][i + 7] = _8;
                }
                //transpose top left
                for(x = 0; x < 4; ++x) {
                    for(y = x + 1; y < 4; ++y) {
                        _1 = B[j + x][i + y];
                        B[j + x][i + y] = B[j + y][i + x];
                        B[j + y][i + x] = _1;
                    }
                }
```

```
                //transpose top right
                for(x = 0; x < 4; ++x) {//base j,i + 4
                    for(y = x + 1; y < 4; ++y) {
                        _1 = B[j + x][i + 4 + y];
                        B[j + x][i + 4 + y] = B[j + y][i + 4 + x];
                        B[j + y][i + 4 + x] = _1;
                    }
                }
                //reduce evction times between B's 8 lines
                for(x = 0; x < 4; ++x) {
                    _1 = B[j + x][i + 4];    //...
                    _4 = B[j + x][i + 4 + 3];
                    //base i+4,y
                    B[j + x][i + 4] = A[i + 4][j + x];
                    B[j + x][i + 4 + 1] = A[i + 4 + 1][j + x];
                    B[j + x][i + 4 + 2] = A[i + 4 + 2][j + x];
                    B[j + x][i + 4 + 3] = A[i + 4 + 3][j + x];
                    //B left bottom
                    B[j + 4 + x][i] = _1;    //...
                    B[j + 4 + x][i + 3] = _4;
                }
                //right bottom copy and transpose
                for(x = 0; x < 4; ++x) {
                    for(y = 0; y < 4; ++y) {
                        _1 = A[i + 4 + x][j + 4 + y];
                        B[j + 4 + y][i + 4 + x] = _1;
                    }
                }
            }
        }
    }
```

- 61x67 matrix optimization code.

```
    if(M == 61) {
        //N = 67 M = 61
        //first part cope 64x61 matrix
        for(i = 0; i + 8 < N; i += 8) {
            for(j = 0; j < M; ++j) {
                _1 = A[i][j];    //...
                _8 = A[i + 7][j];

                B[j][i] = _1;    //...
                B[j][i + 7] = _8;
            }
        }
```

```
        //second part copy the remainder area 4x61
        for(x = i; x < N; ++x) {
            for(y = 0; y < M; ++y) {
                B[y][x] = A[x][y];
            }
        }
    }
```

2.2.3 Evaluation

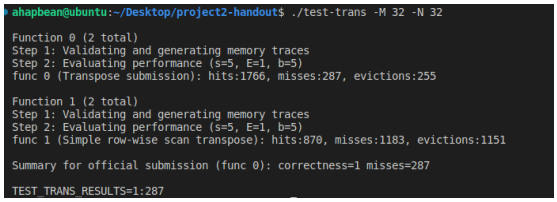

Figure 2: 32x32 transpose performance

The figure 2 shows that the misses my transpose of 32x32 matrix are 287 which is under 300.

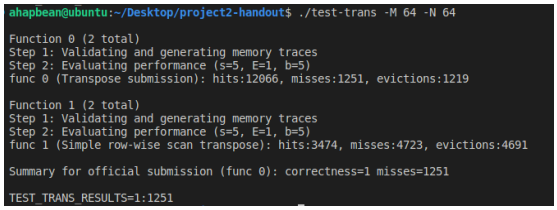

Figure 3: 64x64 transpose performance

The figure 3 shows that the misses my transpose of 64x64 matrix are 1251 which is under 1300.

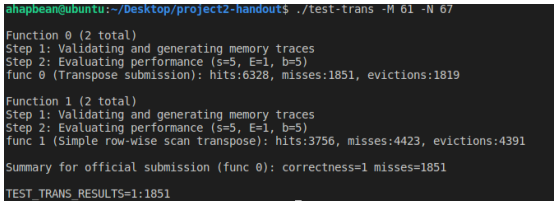

Figure 4: 61x67 transpose performance

The figure 4 shows that the misses my transpose of 61x67 matrix are 1851 which is under 2000.

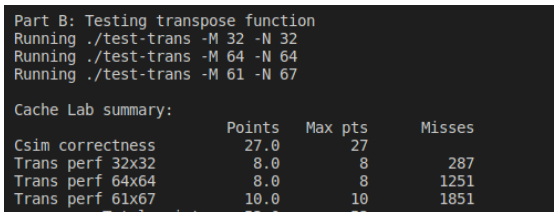

Figure 5: 61x67 transpose performance

The figure 5 shows that by running `make` and `./driver.py`,I get 27 points(full grades) in this part.

Besides,I keep a good coding style while coding for I don't malloc any array and I meet the demand that only 12 local variables of type *int* can be defined.I only defined 12 *int* in the following form.

```
int i, j, x, y;
int _1, _2, _3, _4, _5, _6, _7, _8;
```

2.2.4 Improvement Process

- For the 32x32 matrix, I use an 8x8 blocking technique and a special diagonal handling to achieve 287 misses.

  At first, with $b = 5$, I know that one cache line can store 8 ints and with $s = 5$, I know there are 32 cache lines. Therefore, 8 lines in matrix A can be stored in the cache without eviction. So, my initial approach involves using 8x8 blocking. However, this results in 343 misses, which is close to 300.

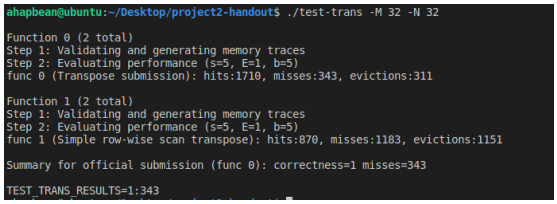

Figure 6: Result of 343 misses

After analyzing the situation, I calculate the loIr bound for my method as follows:

$$\text{loIr bound misses} = 8 \times 4 \times 4 \times 2 = 256$$

Since $343 > 256$, something unexpected happens during the transpose process. By using the '$./csim - ref$' to analyze my trace file, I discover that my 8x8 blocking technique is fine, but when transposing the diagonal

8x8 block, a race condition occurs between A and B. For example, when transposing A(0-7)x(0-7), I load A[1][1] and put it in B[1][1]. This eviction of A[1][0-7] by B[1][0-7] results in three misses when loading A[1][2] to B[2][1].

To overcome this issue, I use a technique called "delayed putting of diagonal elements." Here's how it works: when traversing A[1][0-7] to transpose, I don't put B[1][1] right away; instead, I record its value. When traversing A[2][0-7], I put the recorded value into B[1][1]. This avoids eviction between A and B, reducing the misses from three to one for every diagonal element. However, for A[7][7], this technique doesn't work. Therefore, the misses for every 8x8 block theoretically reduce by 7*2 times. The overall misses will be:

$$\text{real misses} = 343 - 7 \times 2 \times 4 = 287$$

According to the results shown by the auto-grader, my code perfectly matches my calculation. So far, I have successfully achieved the desired results for the 32x32 matrix, obtaining full grades in this part (the result of 287 misses is shown in the evaluation part).

- For the 64x64 matrix, I use an 8x8 blocking technique with a special handling method, resulting in 1251 misses.

  Initially, through analysis, I determine that with $s = 5$, $E = 1$, and $b = 5$, only a 4x64 submatrix can be stored in the cache at once, and line 4 will evict line 0. So, I attempt to transpose the 64x64 matrix using a 4x4 blocking technique. However, this approach results in 1699 misses.

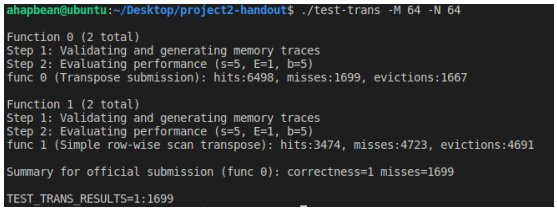

Figure 7: Result of 1699 misses

Upon further calculation, I find that using a 4x4 blocking technique yields a loIr bound of misses as follows:

$$\text{misses} = 4 \times 8 \times 16 + 8 \times 8 \times 16 = 1536$$

So it is evident that using a 4x4 blocking technique, even if I remove all evictions between A and B, it is impossible to achieve full grades. Therefore, I consider using an 8x8 blocking technique instead, with a theoretical loIr bound of misses:

$$16 \times (8 \times 8) = 1024$$

However, as mentioned earlier, line i + 4 will evict line i directly with 8x8 blocking, resulting in poor performance due to numerous evictions within the block. To address this, I introduce a special handling technique to reduce the misses to only 8 for one 8x8 block transpose, or rather 16 misses for every 8x8 block.

Here's how the special handling technique works: Taking A[0-7][0-7] to B[0-7][0-7] as an example, I first put [0-3][0-7] from A into B without transposing, incurring 8 misses. Then, I directly perform the transpose on B[0-3][0-3] and B[0-3][4-7] with 0 misses. Next, I record the values of B[0][4-7] and A[4-7][0], and put the recorded value in B[0][4-7]. Finally, I put the recorded value in B[4][0-3], incurring 2 misses. I repeat this process four times, resulting in 8 misses. Lastly, I perform the transpose on A[4-7][4-7] to B[4-7][4-7]. Since the [0-7] 8 ints have been put in the cache before, there are no misses.

In summary, for one 8x8 block transpose, it incurs 16 misses. However, due to diagonal evictions, the actual number of misses is higher than the theoretical loIr bound. Nonetheless, it still achieves full grades with 1251 misses.

- For the 61x67 matrix, I tried different blocking techniques and special handling methods, ultimately achieving full grades.

  Initially, I attempted the same method as used for the 64x64 matrix. However, due to the different matrix size, the previous handling method did not work well for the 61x67 matrix, resulting in 2132 misses.

  Since I couldn't find any improvement with that configuration, I tried using a 1x8 blocking technique. Specifically, with $N = 67$ and $M = 61$, I recorded A[0-7][0] and put the recorded values into B[0][0-7]. With this approach, I managed to handle the transpose for the 61x64 submatrix. As for the remaining 61x3 matrix, I used a simple load and store approach.

  Surprisingly, by using this simple method, I achieved 1851 misses, meeting the requirements for full grades.

3 Conclusion

3.1 Problems

- When completing the first version of my `csim.c`, I got 21 points, but I didn't know what was wrong with my code for a long time. However, I added verbose mode to my code and used its output to compare with the output of `csim.ref`. I finally found that the problem lay in the cache line's timestamp's design. After I fixed that, it worked and I got 26 points.

- When doing the 64x64 transpose, I doubted whether an 8x8 block's transpose could reduce its misses to 16 misses because I thought there must be an eviction in one 8x8 block. But by thinking about this carefully, I finally found a way to completely avoid eviction in one block.
- In the 61x67 matrix transpose optimization, I had about 2100 misses at first and I didn't know how to continue with optimization because it seemed I couldn't get any optimization from the previous transpose policy. To overcome this difficulty, I tried many methods and finally determined the best one, which is the one I mentioned in section 2.2.4.

3.2 Achievements

- I keep improving my performance in part B and get full grades finally. Sometimes, it's really hard to figure out whether a solution would work perfectly, but with my grit, I try many times and calculate a lot. Finally, I achieve it.
- I successfully program csim.c with about 160 lines and it works correctly. By carefully designing the cache before I code, I gain a relatively clear perspective on the cache simulator and I translate my thoughts into my code in a quick and clear manner. I'm just so proud of it.
- I code in a relatively clear manner. Whether in csim.c or in trans.c, I comment clearly in the code to express my thought process.
- I have a comprehensive understanding of cache and cache miss optimization. After completing part A and part B by myself, I have learned a lot of relevant knowledge, overcome numerous difficulties, and finally succeeded. I believe that the improved understanding of cache ultimately led to my success.