Project 2: Understanding Cache Memories

Zhu Fanyue, lokawa0.0@sjtu.edu.cn

June 2, 2024

1 Introduction

In this project, we are to complete two tasks about cache, which help in understanding the impact it can have on the performance of C programs. The project consists of two parts:

1. Part A: Implement a cache simulator in `csim.c` that takes `valgrind` trace as input, simulates the cache hit and miss behavior and outputs the total number of hits, misses and evictions.
2. Part B: Optimize the matrix transpose function in `trans.c` to minimize its cache misses by understanding the cache's structure and replacement strategy clearly.

2 Experiments

2.1 Part A

2.1.1 Analysis

From the input parameters, we can figure out that the cache structure, which the cache has 2^s sets, each set has E rows, and the size of each row is 2^b. And according to the given test examples, we know that the in a trace file there are three instructions, whose form is like **Operation, Address, Size**.(Since instruction I needs no consideration, we don't introduce it here.)

1. L: Load **Size** bytes of data from **Address**.
2. S: Store **Size** bytes from cache to **Address**.
3. M: Load **Size** bytes of data from **Address**, modify the cache, and then store them back to **Address**

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

2.1.2 Code

First we define the cache data structure `cacheLine`, then define `cacheSet` and `cache` with it.

```
typedef struct {
    int valid;
    int tag;
    int time;
} cacheLine, *cacheSet, **cache;
cache cacheSim=NULL;
```

Then we have the cache initialization and free function.

```
void initCache()
{
    int i,j;
    cacheSim=(cache)malloc(S*sizeof(cacheSet));
    for(i=0;i<S;i++)
    {
        cacheSim[i]=(cacheSet)malloc(E*sizeof(cacheLine));
        for(j=0;j<E;j++)
        {
         //   cacheSim[i][j]=(cacheLine)malloc(sizeof(cacheLine));
            cacheSim[i][j].valid=0;
            cacheSim[i][j].tag=0;
            cacheSim[i][j].time=0;
        }
    }
}

void freeCache()
{
    for(int i=0;i<S;i++)
    {
        free(cacheSim[i]);
    }
    free(cacheSim);
}
```

After init cache, we calculate hits, misses and evictions when update cache.

```
void updateCache(int addr)
{
    int set=(addr>>b)&((1<<s)-1);
    int tag=addr>>(s+b);
    //hit
    for (int i=0;i<E;i++)
    {
        if (cacheSim[set][i].valid==1 && cacheSim[set][i].tag==tag)
        {
            hits++;
            cacheSim[set][i].time=0;
            if (v==1)
            {
                printf(" hit");
            }
            return;
        }
```

```
    }
    //miss
    if (v==1)
    {
        printf(" miss");
    }
    for (int i=0;i<E;i++)
    {
        if (cacheSim[set][i].valid==0)
        {
            misses++;
            cacheSim[set][i].valid=1;
            cacheSim[set][i].tag=tag;
            cacheSim[set][i].time=0;
            return;
        }
    }
    //eviction
    evictions++;
    misses++;
    int max=0;
    for (int i=0;i<E;i++)
    {
        if (cacheSim[set][i].time>cacheSim[set][max].time)
        {
            max=i;
        }
    }
    cacheSim[set][max].tag=tag;
    cacheSim[set][max].time=0;
    if (v==1)
    {
        printf(" eviction");
    }
}
```

Use `getopt()` to parse the commands and get the parameters, analyze the input instruction and update cache accordingly, then update the time stamp of each elements of cache.

```
while (fscanf(fp," %c %x,%d",&op,&addr,&size)>0)
{
    if (v==1)
    {
        printf(" %c %x,%d",op,addr,size);
    }
    switch (op)
    {
    case 'L':
        updateCache(addr);
        break;
    case 'S':
        updateCache(addr);
        break;
    case 'M':
        updateCache(addr);
        updateCache(addr);// L+S Modify should update twice
        break;
```

```
    default:
        break;
    }
    if (v==1) printf("\n");
    for (int i=0;i<S;i++)
    {
        for (int j=0;j<E;j++)
        {
            if (cacheSim[i][j].valid==1)
            {
                cacheSim[i][j].time++;
            }
        }
    }       //update time stamp
}
```

2.1.3 Evaluation

In Figure 1, we can see that the simulator's output consistent with expected output produced by `csim-ref`, and we got the full score.

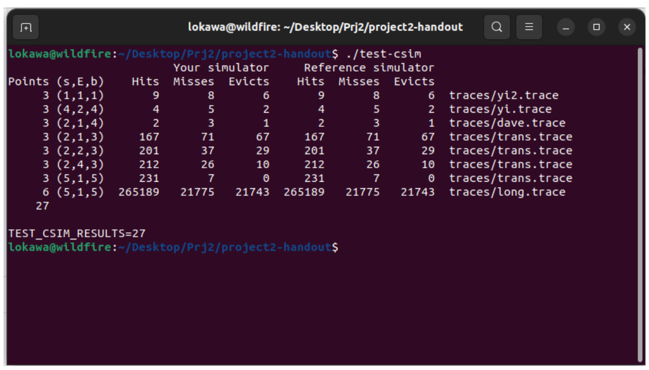

Figure 1: Evaluation of Part A's cache simulator

2.2 Part B

2.2.1 Analysis

In part B, we are asked to minimize the cache miss during the matrix transpose process by optimizing its algorithm. The techniques I uesd are listed below.

1. 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. An example of the optimization that dividing the 32*32 matrix into 4, 9, 16, 25 blocks have on the program is shown as below.

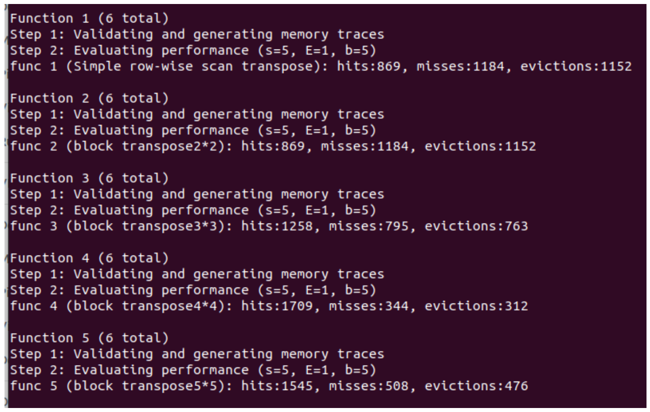

Figure 2: Optimization that dividing the 32*32 matrix into 4, 9, 16, 25 blocks have on the program

We can see from the output that dividing the matrix into blocks reduces the cache misses sharply, and further experiments indicate that 9-25 blocks perform best.

2. Temporary store: Due to the limit of number of local variables, I declare 8 variables names v_1 to v_8 to store the data read from A first, then copy it to B.

In `test-trans`, the parameters are set as $s = 5, E = 1, b = 5$.

In order to minimize the cache miss, I actually wrote 3 help functions for each matrix size, and my `transpose_submit()` is look like below.

```
    if (M==61 && N==67)
    {
        transpose_61(M,N,A,B);
    }
    else if (M==64 && N==64)
    {
        transpose_64(M,N,A,B);
    }
    else if (M==32 && N==32)
    {
        transpose_32(M,N,A,B);
    }
```

2.2.2 Code

1. **32*32 matrix:**

   As the output shown in Figure2, 4*4 blocking performs best in 32*32 matrix transposing. But to reduce cache miss even further, the temporary store is implemented, and the block become a 8*1 block.

```
void transpose_32(int M, int N, int A[N][M], int B[M][N])
{
    //288 FOR 32*32
    //11 VARIABLES
    int v1,v2,v3,v4,v5,v6,v7,v8;
    for (int i=0;i<N;i+=8)
    {
        for (int j=0;j<M;j+=8)
        {
            for (int k=j;k<j+8;k++)
            {
                v1=A[i][k];
                v2=A[i+1][k];
                v3=A[i+2][k];
                v4=A[i+3][k];
                v5=A[i+4][k];
                v6=A[i+5][k];
                v7=A[i+6][k];
                v8=A[i+7][k]; //copy first
                B[k][i]=v1;
                B[k][i+1]=v2;
                B[k][i+2]=v3;
                B[k][i+3]=v4;
                B[k][i+4]=v5;
                B[k][i+5]=v6;
                B[k][i+6]=v7;
                B[k][i+7]=v8; //then transpose
            }

        }
    }
}

```

This version gets 288 cache misses in test.

2. **64*64 matrix**: Similar to 32*32, blocking and temporary store are implemented. But the same function didn't do well in size of 64*64. I figure out that it may be because the size is 4 times larger, and data in different row has greater spacing between their address, thus lead to more cache miss.

   So I experimented some other shape of blocks and figured out the 2*4 block get the best result.

```
void transpose_64(int M, int N, int A[N][M], int B[M][N])
{
    //1636 FOR 64*64
    //11 VARIABLES
    int v1,v2,v3,v4,v5,v6,v7,v8;
    int i,j,k;
    for (i=0;i<N;i+=8)
    {
        for (j=0;j<M;j+=8)
        {
            for (k=0;k<8;k+=2)
            {
```

```
                v1=A[i+k][j];
                v2=A[i+k][j+1];
                v3=A[i+k][j+2];
                v4=A[i+k][j+3];
                v5=A[i+k+1][j];
                v6=A[i+k+1][j+1];
                v7=A[i+k+1][j+2];
                v8=A[i+k+1][j+3];
                B[j][i+k]=v1;
                B[j+1][i+k]=v2;
                B[j+2][i+k]=v3;
                B[j+3][i+k]=v4;
                B[j][i+k+1]=v5;
                B[j+1][i+k+1]=v6;
                B[j+2][i+k+1]=v7;
                B[j+3][i+k+1]=v8;
            }  //left half
            for (int k=0;k<8;k+=2)
            {
                v1=A[i+k][j+4];
                v2=A[i+k][j+5];
                v3=A[i+k][j+6];
                v4=A[i+k][j+7];
                v5=A[i+k+1][j+4];
                v6=A[i+k+1][j+5];
                v7=A[i+k+1][j+6];
                v8=A[i+k+1][j+7];
                B[j+4][i+k]=v1;
                B[j+5][i+k]=v2;
                B[j+6][i+k]=v3;
                B[j+7][i+k]=v4;
                B[j+4][i+k+1]=v5;
                B[j+5][i+k+1]=v6;
                B[j+6][i+k+1]=v7;
                B[j+7][i+k+1]=v8;
            } //right half
        }
    }
}

```

This version gets 1636 cache misses in test.

3. **61*67 matrix**: This test case's biggest difference between the former two is that its shape is not square. So when transposing in blocks we should check the shape left first. 8*1 block performs well in 61*67 matrix.

```
void transpose_61(int M, int N, int A[N][M], int B[M][N])
{
    //1793 FOR 61*67
    //12 VARIABLES
    int v1,v2,v3,v4,v5,v6,v7,v8;
    for (int i=0;i<N;i+=8)
    {
        for (int j=0;j<M;j+=8)
        {
            if (i+8<N && j+8<M)
```

```c
            {
                for (int k=j;k<j+8;k++)
                {
                    v1=A[i][k];
                    v2=A[i+1][k];
                    v3=A[i+2][k];
                    v4=A[i+3][k];
                    v5=A[i+4][k];
                    v6=A[i+5][k];
                    v7=A[i+6][k];
                    v8=A[i+7][k];
                    B[k][i]=v1;
                    B[k][i+1]=v2;
                    B[k][i+2]=v3;
                    B[k][i+3]=v4;
                    B[k][i+4]=v5;
                    B[k][i+5]=v6;
                    B[k][i+6]=v7;
                    B[k][i+7]=v8;
                } //square part
            }
            else
            {
                for (int k=0;k<8 && i+k<N;k++)
                {
                    for (int l=0;l<8 && j+l<M;l++)
                    {
                        B[j+l][i+k]=A[i+k][j+l];
                    }
                } //rest part
            }
        }
    }
}

```

This version gets 1793 cache misses in test.

2.2.3 Evaluation

In Figure 3 we can see that all the task are all correct, and the 32*32 matrix along with the 61*67 one get full marks, while the 64*64 only get 4.2.

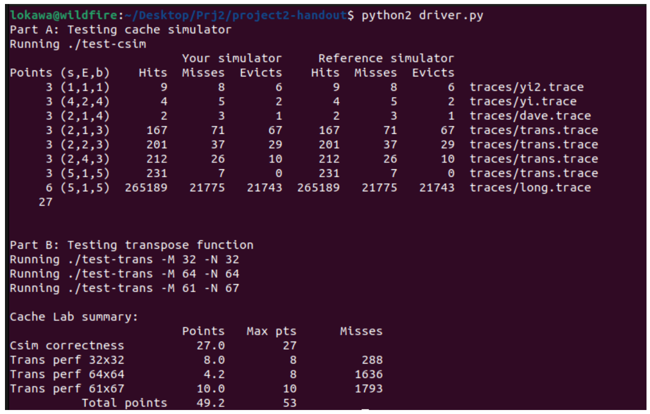

Figure 3: Final evaluation results

3 Conclusion

3.1 Problems

1. The second test case in Part B didn't get full mark, so there must be some other optimizing method to improve the performance, but I've not figured out yet.

2. All the optimization in Part B is not the best result.

3.2 Achievements

1. In Part A, I successfully implemented a cache simulator using LRU, deepened understanding of cache structure and replacement strategy.

2. In Part B, I realized the great impact cache can have on performance of C programs, and tried my best to minimize cache misses in matrix transposing. For the three different test cases, I analyzed and tried different methods accordingly to optimize their performance.