

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.

```
1 typedef struct {
2     int valid;
3     int tag;
4     int time;
5 } cacheLine, *cacheSet, **cache;
6 cache cacheSim=NULL;
```

Then we have the cache initialization and free function.

```
1 void initCache()
2 {
3     int i,j;
4     cacheSim=(cache)malloc(S*sizeof(cacheSet));
5     for(i=0;i<S;i++)
6     {
7         cacheSim[i]=(cacheSet)malloc(E*sizeof(cacheLine));
8         for(j=0;j<E;j++)
9         {
10             // cacheSim[i][j]=(cacheLine)malloc(sizeof(cacheLine));
11             cacheSim[i][j].valid=0;
12             cacheSim[i][j].tag=0;
13             cacheSim[i][j].time=0;
14         }
15     }
16 }
17
18 void freeCache()
19 {
20     for(int i=0;i<S;i++)
21     {
22         free(cacheSim[i]);
23     }
24     free(cacheSim);
25 }
```

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

```
1 void updateCache(int addr)
2 {
3     int set=(addr>>b)&((1<<s)-1);
4     int tag=addr>>(s+b);
5     //hit
6     for (int i=0;i<E;i++)
7     {
8         if (cacheSim[set][i].valid==1 && cacheSim[set][i].tag==tag)
9         {
10             hits++;
11             cacheSim[set][i].time=0;
12             if (v==1)
13             {
14                 printf(" hit");
15             }
16             return;
17         }
18     }
```

```

18     }
19     //miss
20     if (v==1)
21     {
22         printf(" miss");
23     }
24     for (int i=0;i<E;i++)
25     {
26         if (cacheSim[set][i].valid==0)
27         {
28             misses++;
29             cacheSim[set][i].valid=1;
30             cacheSim[set][i].tag=tag;
31             cacheSim[set][i].time=0;
32             return;
33         }
34     }
35     //eviction
36     evictions++;
37     misses++;
38     int max=0;
39     for (int i=0;i<E;i++)
40     {
41         if (cacheSim[set][i].time>cacheSim[set][max].time)
42         {
43             max=i;
44         }
45     }
46     cacheSim[set][max].tag=tag;
47     cacheSim[set][max].time=0;
48     if (v==1)
49     {
50         printf(" eviction");
51     }
52 }

```

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.

```

1 while (fscanf(fp," %c %x,%d",&op,&addr,&size)>0)
2 {
3     if (v==1)
4     {
5         printf(" %c %x,%d",op,addr,size);
6     }
7     switch (op)
8     {
9         case 'L':
10            updateCache(addr);
11            break;
12         case 'S':
13            updateCache(addr);
14            break;
15         case 'M':
16            updateCache(addr);
17            updateCache(addr);// L+S Modify should update twice
18            break;

```

```

19     default:
20         break;
21     }
22     if (v==1) printf("\n");
23     for (int i=0;i<S;i++)
24     {
25         for (int j=0;j<E;j++)
26         {
27             if (cacheSim[i][j].valid==1)
28             {
29                 cacheSim[i][j].time++;
30             }
31         }
32     } //update time stamp
33 }

```

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.

```

lokawa@wildfire: ~/Desktop/Prj2/project2-handout
lokawa@wildfire:~/Desktop/Prj2/project2-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/yl2.trace
3 (4,2,4) 4 5 2 4 5 2 traces/yl.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) 291 37 29 291 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
lokawa@wildfire:~/Desktop/Prj2/project2-handout$

```

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 used 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.

```

Function 1 (6 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:869, misses:1184, evictions:1152

Function 2 (6 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 2 (block transpose2*2): hits:869, misses:1184, evictions:1152

Function 3 (6 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 3 (block transpose3*3): hits:1258, misses:795, evictions:763

Function 4 (6 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 4 (block transpose4*4): hits:1709, misses:344, evictions:312

Function 5 (6 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 5 (block transpose5*5): hits:1545, misses:508, evictions:476

```

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.

```

1  if (M==61 && N==67)
2  {
3      transpose_61(M,N,A,B);
4  }
5  else if (M==64 && N==64)
6  {
7      transpose_64(M,N,A,B);
8  }
9  else if (M==32 && N==32)
10 {
11     transpose_32(M,N,A,B);
12 }

```

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.

```

1 void transpose_32(int M, int N, int A[N][M], int B[M][N])
2 {
3     //288 FOR 32*32
4     //11 VARIABLES
5     int v1,v2,v3,v4,v5,v6,v7,v8;
6     for (int i=0;i<N;i+=8)
7     {
8         for (int j=0;j<M;j+=8)
9         {
10             for (int k=j;k<j+8;k++)
11             {
12                 v1=A[i][k];
13                 v2=A[i+1][k];
14                 v3=A[i+2][k];
15                 v4=A[i+3][k];
16                 v5=A[i+4][k];
17                 v6=A[i+5][k];
18                 v7=A[i+6][k];
19                 v8=A[i+7][k]; //copy first
20                 B[k][i]=v1;
21                 B[k][i+1]=v2;
22                 B[k][i+2]=v3;
23                 B[k][i+3]=v4;
24                 B[k][i+4]=v5;
25                 B[k][i+5]=v6;
26                 B[k][i+6]=v7;
27                 B[k][i+7]=v8; //then transpose
28             }
29         }
30     }
31 }
32 }
33

```

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.

```

1 void transpose_64(int M, int N, int A[N][M], int B[M][N])
2 {
3     //1636 FOR 64*64
4     //11 VARIABLES
5     int v1,v2,v3,v4,v5,v6,v7,v8;
6     int i,j,k;
7     for (i=0;i<N;i+=8)
8     {
9         for (j=0;j<M;j+=8)
10        {
11            for (k=0;k<8;k+=2)
12            {

```

```

13         v1=A[i+k][j];
14         v2=A[i+k][j+1];
15         v3=A[i+k][j+2];
16         v4=A[i+k][j+3];
17         v5=A[i+k+1][j];
18         v6=A[i+k+1][j+1];
19         v7=A[i+k+1][j+2];
20         v8=A[i+k+1][j+3];
21         B[j][i+k]=v1;
22         B[j+1][i+k]=v2;
23         B[j+2][i+k]=v3;
24         B[j+3][i+k]=v4;
25         B[j][i+k+1]=v5;
26         B[j+1][i+k+1]=v6;
27         B[j+2][i+k+1]=v7;
28         B[j+3][i+k+1]=v8;
29     } //left half
30     for (int k=0;k<8;k+=2)
31     {
32         v1=A[i+k][j+4];
33         v2=A[i+k][j+5];
34         v3=A[i+k][j+6];
35         v4=A[i+k][j+7];
36         v5=A[i+k+1][j+4];
37         v6=A[i+k+1][j+5];
38         v7=A[i+k+1][j+6];
39         v8=A[i+k+1][j+7];
40         B[j+4][i+k]=v1;
41         B[j+5][i+k]=v2;
42         B[j+6][i+k]=v3;
43         B[j+7][i+k]=v4;
44         B[j+4][i+k+1]=v5;
45         B[j+5][i+k+1]=v6;
46         B[j+6][i+k+1]=v7;
47         B[j+7][i+k+1]=v8;
48     } //right half
49     }
50 }
51 }
52

```

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.

```

1 void transpose_61(int M, int N, int A[N][M], int B[M][N])
2 {
3     //1793 FOR 61*67
4     //12 VARIABLES
5     int v1,v2,v3,v4,v5,v6,v7,v8;
6     for (int i=0;i<N;i+=8)
7     {
8         for (int j=0;j<M;j+=8)
9         {
10             if (i+8<N && j+8<M)

```

```

11      {
12          for (int k=j;k<j+8;k++)
13          {
14              v1=A[i][k];
15              v2=A[i+1][k];
16              v3=A[i+2][k];
17              v4=A[i+3][k];
18              v5=A[i+4][k];
19              v6=A[i+5][k];
20              v7=A[i+6][k];
21              v8=A[i+7][k];
22              B[k][i]=v1;
23              B[k][i+1]=v2;
24              B[k][i+2]=v3;
25              B[k][i+3]=v4;
26              B[k][i+4]=v5;
27              B[k][i+5]=v6;
28              B[k][i+6]=v7;
29              B[k][i+7]=v8;
30          } //square part
31      }
32      else
33      {
34          for (int k=0;k<8 && i+k<N;k++)
35          {
36              for (int l=0;l<8 && j+l<M;l++)
37              {
38                  B[j+l][i+k]=A[i+k][j+l];
39              }
40          } //rest part
41      }
42  }
43 }
44 }
45

```

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.


```

lokawa@wildfire:~/Desktop/Prj2/project2-handout$ python2 driver.py
Part A: Testing cache simulator
Running ./test-csim

Points (s,E,b)   Hits   Your simulator   Reference simulator
                  Misses  Evicts         Hits   Misses   Evicts
3 (1,1,1)         9       8       6         9       8       6   traces/y12.trace
3 (4,2,4)         4       5       2         4       5       2   traces/y1.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	288
Trans perf 64x64	4.2	8	1636
Trans perf 61x67	10.0	10	1793
Total points	49.2	53	

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.