Cache Lab: Understanding Cache Memories Documentation

Wu Jialong 2018013418

June 19, 2020

1 Part A: Writing a Cache Simulator

There are only three functions in `csim.c`: `printHelp`, `main` and `access`. There are also some important global variables: `clk` represents clock, which is used to perform LRU replacement policy.

`printHelp` is just a function printing help information, like `csim-ref`.

`main` function read command line arguments, allocate memory for `int **tag,**last` and initialize them with empty: `tag` represents tags stored in the cache, and `last` represents the last time of access. Then main function processes trace file line by line. For line starts with:

- I: it ignores it.
- S or L: it calls `access`
- M: it calls `access` twice.

`access` simulates read or write operation on the cache. First of all, calculate target group and tag according to the address. And go through the group to find the tag. If success, it's a hit. Otherwise, it's a miss. Go through the group again to find if there is an empty line. if not, there will be a eviction and we perform LRU replacement policy according to `last`. Remember for all the cases, we need to update `last` with current `clk`.

2 Part B: Optimizing Matrix Transpose

In this part, $s = 5, E = 1, b = 5$, the size of the cache is

$$2^5 \times 2^5 = 256 \times 4 \text{ Bytes} = 256 \text{ int}$$

And we observed that the lowest b bits of `A[0][0]` and `B[0][0]`'s address are always 0, and the lowest $s + b$ bits of them are always identical. Thus, every 8 elements of the array lie in the same group, and `A[i][j]` and `B[i][j]` would cause a conflict.

2.1 M = 32, N = 32

Here is the code for this case, and we will explain it below.

```c
// misses=259 = 3+4*4*16
for (i=0; i<N; i+=8)
    for (j=0; j<M; j+=8){
        for (r=0; r<8; r++){
            a=A[i+r][j]; b=A[i+r][j+1]; c=A[i+r][j+2]; d=A[i+r][j+3];
            e=A[i+r][j+4]; f=A[i+r][j+5]; g=A[i+r][j+6]; h=A[i+r][j+7];

            B[j+r][i]=a; B[j+r][i+1]=b; B[j+r][i+2]=c; B[j+r][i+3]=d;
            B[j+r][i+4]=e; B[j+r][i+5]=f; B[j+r][i+6]=g; B[j+r][i+7]=h;
        }
        for (r=0; r<8; r++)
            for (c=0; c<r; c++){
                d=B[j+r][i+c]; B[j+r][i+c]=B[j+c][i+r]; B[j+c][i+r]=d;
            }
    }
```

In this case, every 8 lines of the matrix fills the cache, so we block the matrix with 8×8 block. Denote $A = [A_{ij}]_{4\times4}, B = [B_{ij}]_{4\times4}$, where A_{ij} and B_{ij} are 8×8 block. The target is $B_{ji} = A_{ij}^T$.

The idea is that we copy A_{ij} to B_{ji} line by line and transpose it in-place. Copying a block results in 16 cold misses, 8 for A_{ij} and 8 for B_{ji}. Then since B_{ji} is totally in the cache, it consumes no misses to transpose B_{ji} in-place.

Total number of misses is approximately equal to $4 \times 4 \times 16 = 256$.

2.2 M = 64, N = 64

Here is the code for this case, and we will explain it below.

```c
// misses=1259 = 3+24*8+19*56
for (i=0; i<N; i+=8)
    for (j=0; j<M; j+=8){
        if (i!=j){
            e=A[i][j+4]; f=A[i][j+5]; g=A[i][j+6]; h=A[i][j+7];
            for (r=i; r<i+8; r++) {
                a=A[r][j]; b=A[r][j+1]; c=A[r][j+2]; d=A[r][j+3];
                B[j][r]=a; B[j+1][r]=b; B[j+2][r]=c; B[j+3][r]=d;
            }
            for (r=i+7; r>i; r--) {
                a=A[r][j+4]; b=A[r][j+5]; c=A[r][j+6]; d=A[r][j+7];
                B[j+4][r]=a; B[j+5][r]=b; B[j+6][r]=c; B[j+7][r]=d;
            }
            B[j+4][i]=e; B[j+5][i]=f; B[j+6][i]=g; B[j+7][i]=h;
        }else{
            // Block = [[X,Y],[Z,W]], totally miss+=24
            // move X and Y: miss+=8
            for (r=0; r<4; r++){
                a=A[i+r][j]; b=A[i+r][j+1]; c=A[i+r][j+2]; d=A[i+r][j+3];
                e=A[i+r][j+4]; f=A[i+r][j+5]; g=A[i+r][j+6]; h=A[i+r][j+7];

                B[j+r][i]=a; B[j+r][i+1]=b; B[j+r][i+2]=c; B[j+r][i+3]=d;
                B[j+r][i+4]=e; B[j+r][i+5]=f; B[j+r][i+6]=g; B[j+r][i+7]=h;
            }
            // transpose X: miss+=0
            for (r=0; r<4; r++)
```

```
                for (c=0; c<r; c++){
                    a=B[j+r][i+c]; B[j+r][i+c]=B[j+c][i+r]; B[j+c][i+r]=a;
                }
            // transpose Y: miss+=0
            for (r=0; r<4; r++)
                for (c=0;c<r;c++){
                    a=B[j+r][i+4+c]; B[j+r][i+4+c]=B[j+c][i+4+r]; B[j+c][i+4+r]=a;
                }

            // move Z and W: miss+=8
            for (r=4; r<8; r++){
                a=A[i+r][j]; b=A[i+r][j+1]; c=A[i+r][j+2]; d=A[i+r][j+3];
                e=A[i+r][j+4]; f=A[i+r][j+5]; g=A[i+r][j+6]; h=A[i+r][j+7];

                B[j+r][i]=a; B[j+r][i+1]=b; B[j+r][i+2]=c; B[j+r][i+3]=d;
                B[j+r][i+4]=e; B[j+r][i+5]=f; B[j+r][i+6]=g; B[j+r][i+7]=h;
            }
            // transpose Z: miss+=0
            for (r=0; r<4; r++)
                for (c=0; c<r; c++){
                    a=B[j+4+r][i+c]; B[j+4+r][i+c]=B[j+4+c][i+r]; B[j+4+c][i+r]=a;
                }
            // transpose W: miss+=0
            for (r=0; r<4; r++)
                for (c=0; c<r; c++){
                    a=B[j+4+r][i+4+c]; B[j+4+r][i+4+c]=B[j+4+c][i+4+r]; B[j+4+c][i+4+r]=a;
                }

            // swap Y^T and Z^T
            for (c=0; c<4; c++){
                a=B[j+4][i+c]; B[j+4][i+c]=B[j+5][i+c]; B[j+5][i+c]=a;
            } // miss+=0
            a=B[j+5][i]; b=B[j+5][i+1]; c=B[j+5][i+2]; d=B[j+5][i+3]; // miss+=0
            e=B[j][i+4]; f=B[j][i+5]; g=B[j][i+6]; h=B[j][i+7]; //miss+=1
            B[j+5][i]=e; B[j+5][i+1]=f; B[j+5][i+2]=g; B[j+5][i+3]=h; // miss+=0
            B[j][i+4]=a; B[j][i+5]=b; B[j][i+6]=c; B[j][i+7]=d; // miss+=0

            a=B[j+4][i]; b=B[j+4][i+1]; c=B[j+4][i+2]; d=B[j+4][i+3]; // miss+=1
            e=B[j+1][i+4]; f=B[j+1][i+5]; g=B[j+1][i+6]; h=B[j+1][i+7]; // miss+=1
            B[j+4][i]=e; B[j+4][i+1]=f; B[j+4][i+2]=g; B[j+4][i+3]=h; // miss+=0
            B[j+1][i+4]=a; B[j+1][i+5]=b; B[j+1][i+6]=c; B[j+1][i+7]=d; // miss+=0
            for (c=0; c<4; c++){
                a=B[j+4][i+c]; B[j+4][i+c]=B[j+5][i+c]; B[j+5][i+c]=a;
            } // miss+=1

            for (c=0; c<4; c++){
                a=B[j+6][i+c]; B[j+6][i+c]=B[j+7][i+c]; B[j+7][i+c]=a;
            } // miss+=0
            a=B[j+7][i]; b=B[j+7][i+1]; c=B[j+7][i+2]; d=B[j+7][i+3]; // miss+=0
            e=B[j+2][i+4]; f=B[j+2][i+5]; g=B[j+2][i+6]; h=B[j+2][i+7]; //miss+=1
            B[j+7][i]=e; B[j+7][i+1]=f; B[j+7][i+2]=g; B[j+7][i+3]=h; // miss+=0
            B[j+2][i+4]=a; B[j+2][i+5]=b; B[j+2][i+6]=c; B[j+2][i+7]=d; // miss+=0

            a=B[j+6][i]; b=B[j+6][i+1]; c=B[j+6][i+2]; d=B[j+6][i+3]; // miss+=1
            e=B[j+3][i+4]; f=B[j+3][i+5]; g=B[j+3][i+6]; h=B[j+3][i+7]; // miss+=1
            B[j+6][i]=e; B[j+6][i+1]=f; B[j+6][i+2]=g; B[j+6][i+3]=h; // miss+=0
```

```
83                B[j+3][i+4]=a; B[j+3][i+5]=b; B[j+3][i+6]=c; B[j+3][i+7]=d; // miss+=0
84                for (c=0; c<4; c++){
85                    a=B[j+6][i+c]; B[j+6][i+c]=B[j+7][i+c]; B[j+7][i+c]=a;
86                } // miss+=1
87            }
88        }
```

In this case, every 4 lines of the matrix fills the cache. We block the matrices with 8×8 block again, that is $A = [A_{ij}]_{8\times 8}, B = [B_{ij}]_{8\times 8}$, where A_{ij} and B_{ij} are 8×8 submatrices. We respectively process diagonal blocks and off-diagonal blocks.

2.2.1 Off-diagonal Blocks

When it comes to off-diagonal blocks, A_{ij} and B_{ji} $(i \neq j)$ don't share any groups in the cache. Conflicts only comes from the first 4 lines and the last 4 lines of A_{ij} or B_{ji}, for example, $A_{ij}[0]$ and $A_{ij}[4]$ are mapped to the same group, $A_{ij}[1]$ and $A_{ij}[5]$ are mapped to the same group, $B_{ji}[0]$ and $B_{ji}[4]$ are mapped to the same group, and the like.

Line 5 to 14 of the code process off-diagonal blocks. The idea is that we first copy $A_{ij}[0..7][0..3]$ to $B_{ji}[0..3][0..7]$ and then copy $A_{ij}[0..7][4..7]$ to $B_{ji}[4..7][0..7]$.

Misses and hits are plot in Table 1. There are totally 19 misses. Note that the hit in $A_{ij}[0][4]$ is because of temporary varibles e, f, g, h. This trick can save one miss per block.

m	h	h	h	**h**	h	h	h
m	h	h	h	m	h	h	h
m	h	h	h	m	h	h	h
m	h	h	h	m	h	h	h
m	h	h	h	h	h	h	h
m	h	h	h	h	h	h	h
m	h	h	h	h	h	h	h
m	h	h	h	h	h	h	h

m	h	h	h	h	h	h	h
m	h	h	h	h	h	h	h
m	h	h	h	h	h	h	h
m	h	h	h	h	h	h	h
m	h	h	h	h	h	h	h
m	h	h	h	h	h	h	h
m	h	h	h	h	h	h	h
m	h	h	h	h	h	h	h

Table 1: $i \neq j$: (Left) misses and hits on A_{ij}; (Right) misses and hits on B_{ji}.

2.2.2 Diagonal Blocks

It is kind of complicated for diagonal blocks. Line 16 to 86 process them. For simplicity, we will denote the blocks A_{ij} and B_{ji} by A and B in this section. And we further block A into four part

$$A = \begin{bmatrix} X & Y \\ Z & W \end{bmatrix}$$

where X, Y, Z, W are 4×4 submatrices. Just like what we have done in the case of 32×32, we copy each block to B and transpose them in-place, resulting in $B = \begin{bmatrix} X^T & Y^T \\ Z^T & W^T \end{bmatrix}$. This causes $8 + 8 = 16$ misses.

Then we need to swap Z^T and Y^T in B. To avoid thrashing between $Y^T[k]$ and $Z^T[k]$, which are mapped to the same group, we do such process which causes 4 misses:

$$B = \left[\begin{array}{c|c} & Y^T[0] \\ & Y^T[1] \\ & Y^T[2] \\ & Y^T[3] \\ \hline Z^T[0] & \\ Z^T[1] & \\ Z^T[2] & \\ Z^T[3] & \end{array}\right] \Rightarrow \left[\begin{array}{c|c} & Y^T[0] \\ & Y^T[1] \\ & Y^T[2] \\ & Y^T[3] \\ \hline Z^T[1] & \\ Z^T[0] & \\ Z^T[2] & \\ Z^T[3] & \end{array}\right] \Rightarrow \left[\begin{array}{c|c} & Z^T[0] \\ & Z^T[1] \\ & Y^T[2] \\ & Y^T[3] \\ \hline Y^T[1] & \\ Y^T[0] & \\ Z^T[2] & \\ Z^T[3] & \end{array}\right] \Rightarrow \left[\begin{array}{c|c} & Z^T[0] \\ & Z^T[1] \\ & Y^T[2] \\ & Y^T[3] \\ \hline Y^T[0] & \\ Y^T[1] & \\ Z^T[2] & \\ Z^T[3] & \end{array}\right]$$

Same things are done for $Y^T[2], Y^T[3]$, which also causes 4 misses. Details about misses can be found in the code comments.

For diagonal blocks, there are $8 + 8 + 4 + 4 = 24$ misses per block. To transpose the whole 64×64 matrix, it consumes about $24 \times 8 + 19 \times 56 = 1256$ misses.

2.3 M = 61, N = 67

In this case, N and M are not a multiple of 8, so we hypothesize that conflict misses are not as frequent as in the cases above. Actually, when we directly use the blocking technique introduced in http://csapp.cs.cmu.edu/public/ waside/waside-blocking.pdf, the number of misses is equal to 1931, which is enough to get the score.

```
for (i=0;i<N;i+=8)
    for (j=0;j<M;j+=8)
        for (c=j;c<j+8 && c<M;c++)
            for (r=i;r<i+8 && r<N;r++)
                B[c][r] = A[r][c];
```

Then we can make use of local variables to further reduce potential conficts, which results in 1755 misses.

```
for (i=0;i<N;i+=8)
    for (j=0;j<M;j+=8){
        // misses=1755
        for (r=j;r<j+8 && r<M;r++){
            a=A[i][r]; b=A[i+1][r]; c=A[i+2][r];
            if (i<64){
                d=A[i+3][r]; e=A[i+4][r]; f=A[i+5][r]; g=A[i+6][r]; h=A[i+7][r];
            }

            B[r][i]=a; B[r][i+1]=b; B[r][i+2]=c;
            if (i<64){
                B[r][i+3]=d; B[r][i+4]=e; B[r][i+5]=f; B[r][i+6]=g; B[r][i+7]=h;
            }
        }
    }
```

2.4 M = 48, N = 48

In this case, we directly copy the code from case 32×32。

```c
// misses=595 \approx 6*6*16
for (i=0; i<N; i+=8)
    for (j=0; j<M; j+=8){
        for (r=0; r<8; r++){
            a=A[i+r][j]; b=A[i+r][j+1]; c=A[i+r][j+2]; d=A[i+r][j+3];
            e=A[i+r][j+4]; f=A[i+r][j+5]; g=A[i+r][j+6]; h=A[i+r][j+7];

            B[j+r][i]=a; B[j+r][i+1]=b; B[j+r][i+2]=c; B[j+r][i+3]=d;
            B[j+r][i+4]=e; B[j+r][i+5]=f; B[j+r][i+6]=g; B[j+r][i+7]=h;
        }
        for (r=0; r<8; r++)
            for (c=0; c<r; c++){
                d=B[j+r][i+c]; B[j+r][i+c]=B[j+c][i+r]; B[j+c][i+r]=d;
            }
    }
```

There are basically $6 \times 6 \times 16 = 576$ cold misses. But there are also some conflict misses. For example, 8 lines in block A_{02} are mapped into group `2,8,14,20,26,0,6,12` and those in block B_{20} are mapped into group `0,6,12,18,24,30,4,10`. When we load $A_{02}[5..7]$, it will evict $B_{20}[0..2]$, resulting in 3 additional misses when transpose B_{20} in-place.

Fortunately, this kind of conflict misses rarely happen. After test, we find that our code only have 595 misses, just a little bit more than 576.