**CSED211: Cache Lab Report**

20200952 / Chaeyeon Jang

## 1. Overview

Cache Lab 은 크게 cache memory simulator 를 구현하는 part A 와, matrix transpose function 을 최대한 적은 cache miss 횟수로 구현하는 part B 로 나누어져 있다. Part A 에서는 csim.c 파일에 cache simulator 를 작성하는데, valgrind memory trace 를 input 으로 받아 cache memory 의 hit/miss behavior 를 simulate 한 후 hit, miss, eviction 횟수를 출력하는 코드를 작성해야 한다. Part B 에서는 32 x 32, 64 x 64, 61 x 67 matrix 를 input 으로 받아 최대한 적은 miss 횟수로 transpose 를 수행하는 function 코드를 작성해야 한다.

This section describes how your work will be evaluated. The full score for this lab is 53 points:

- Part A: 27 Points
- Part B: 26 Points

## 2. Part A

우선, 아래 코드와 같이 c 언어의 struct type 을 이용하여 cache line 을 구현하였다.

```
typedef struct{
    char valid;
    unsigned long tag;
    int used;
} line;
```

Cache line 구조체는 valid bit, tag bit, 그리고 LRU replacement policy 를 위해 used 된 순서를 저장하기 위한 변수인 used 변수로 구성되어 있다.

Main 함수에서는 먼저 getopt 함수를 이용하여 입력된 명령어로부터 s, E, b, t 에 대한 option 값을 parsing 해야 한다. 입력되는 명령어의 형태는 아래와 같다.

```
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
linux> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/long.trace
```

이를 바탕으로 getopt 함수로 명령어를 parsing 받는 코드를 작성하면 아래와 같다.

```
while((opt = getopt(argc, argv, "hvs:E:b:t:")) != -1){
        switch(opt){
            case 'h': break;
```

```
            case 'v': break;
            case 's':
                s = atoi(optarg);
                break;
            case 'E':
                E = atoi(optarg);
                break;
            case 'b':
                b = atoi(optarg);
                break;
            case 't':
                fname = optarg;
                break;
        }

    }
```

Atoi 함수로 s, E, b 옵션에 해당하는 글자를 int 자료형으로 변환 후 각각 s, E, b 변수에 저장하였고, t 옵션에 해당하는 글자는 추후 fopen 함수로 open 할 file 의 이름이므로 파일 이름을 저장하는 fname 변수에 저장해주었다. 이때 s, E, b 는 각각 set index bit 수, associativity 수, block offset bit 수이다.
다음으로, 전달받은 s, E, b 값을 바탕으로 cache 를 생성하고 initialize 해주어야 한다.

```
S = pow(2, s);

//cache create & initialize
cache_line = (line **)malloc(sizeof(line*) * S);
for(i = 0; i < S; i++){
    cache_line[i] = (line *)malloc(sizeof(line) * E);
    for(j = 0; j < E; j++){
        cache_line[i][j].tag = 0;
        cache_line[i][j].valid = 0;
        cache_line[i][j].used = 0;
    }
}
```

Pow 함수를 통해 cache set 개수인 S = $2^s$를 구해주고, malloc 동적할당을 통해 line ** 형태인 cache_line 을 생성해주었다. 그리고, 각 set 에 해당하는 cache_line pointer 마다 cache_line 을 할당해주고, 이 구조체의 tag, valid, used 변수를 0 으로 초기화해주었다.

이제 file I/O api 를 사용하여 명령어로 입력받은 trace file 을 읽으며 cache simulation 을 수행하는 코드를 구현하자.

```
```

```
    file = fopen(fname, "r"); //declare file pointer

    //file read
    while(fscanf(file, " %c %lx,%d", &operation, &address, &size) != EOF){
        mask = (0xffffffffffffffff << b);
        block = ~mask & address;
        mask = 0xffffffffffffffff << (b + s);
        tag = mask & address;

        set = (address & (~(tag | block))) >> b;
        tag = tag >> (b + s);

        // printf("set: %ld tag: %ld block: %ld", set, tag, block);

        if(operation == 'I') continue;
        else if(operation == 'L'){
            cache_op(tag, set, used, E, &num_hit, &num_miss, &num_evict, cache_line);
        }
        else if(operation == 'S'){
            cache_op(tag, set, used, E, &num_hit, &num_miss, &num_evict, cache_line);
        }
        else if(operation == 'M'){
            cache_op(tag, set, used, E, &num_hit, &num_miss, &num_evict, cache_line);
            cache_op(tag, set, used, E, &num_hit, &num_miss, &num_evict, cache_line);
        }
        used++;

    }
```

Fopen 으로 trace file 을 read mode 로 열고, 이를 FILE* 포인터에 저장해준다. 그리고, valgrind trace 파일에서 I instructioin 을 제외한 L, S, M instruction 은 아래와 같은 형태를 지니고 있으므로,

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

fscanf 함수로 file 의 입력을 받을 때 %c %lx,%d" 형태로 입력을 받도록 명시해준다. Instruction 을 한 줄 씩 읽으며, 우선 address 주소로부터 tag bit, set bit, block bit 를 extract 해주어야 한다. >>, << operator 및 mask 를 사용하여 set index bit, tag bit, block offset bit 를 추출한 후 set, tag, block 변수에 저장해주었다. 그 후, operation 종류에 따라 cache hit/miss simulator 함수인 cache_op 함수를 호출해주었는데, operation 이 I 일 경우 pass, L 혹은 S 일

경우 cache_op 함수를 한 번 호출, M 일 경우 cache_op 함수를 두 번 호출해주었다. 다음은 cache_op 함수 코드이다.

```
void cache_op(unsigned long tag, unsigned long set, int used, int E, int *num_hit, int
*num_miss, int *num_evict, line** cache_line){
    int i; //for iteration
    line* cache_line_for_index = cache_line[set];
    int min_used = cache_line_for_index[0].used;
    int min_used_idx = 0;
    int is_full = 1;

    for(i = 0 ; i < E ; i++){
        if(min_used > cache_line_for_index[i].used){
            min_used_idx = i;
            min_used = cache_line_for_index[i].used;
        }
        if(cache_line_for_index[i].valid == 0){
            is_full = 0;
        }
        if(cache_line_for_index[i].valid && (cache_line_for_index[i].tag == tag)){
            *num_hit += 1;
            cache_line_for_index[i].used  = used;
            return;
        }
    }

    if(is_full == 0){
        //cache line is not full; use empty cache block
        *num_miss += 1;
        for(i = 0; i < E; i++){
            if(cache_line_for_index[i].valid == 0){
                cache_line_for_index[i].valid = 1;
                cache_line_for_index[i].tag = tag;
                cache_line_for_index[i].used = used;
                break;
            }
        }
    }else{
        //cache line is full; evict with LRU policy
        *num_miss += 1;
        *num_evict += 1;
        cache_line_for_index[min_used_idx].valid = 1;
        cache_line_for_index[min_used_idx].tag = tag;
        cache_line_for_index[min_used_idx].used = used;
    }

    return;
```

```
}
```

Set 의 block 들을 for iteration 으로 순회하며 만약 valid bit 가 1 이고, tag 가 address 의 tag bit 와 같은 block 이 존재한다면 num_hit 수를 1 만큼 증가시키고, used 변수를 함수로 전달받은 현재 used 값으로 바꾼 후 함수를 return 한다. 이때 block 을 순회할 때, 처음 설정한 min_used 값보다 작은 used 값을 가지고 있는 block 이 존재한다면 min_used_idx 를 이 block 의 idx 로 바꾸고, min_used 값을 이 block 의 used 값으로 바꾸며 min_used 값을 최소 값으로 update 해준다.
동시에, valid bit 가 0 인 block 이 존재한다면, 즉, 아직 empty 상태인 block 이 존재한다면, cache set 가 full 인지 여부를 저장하는 is_full 변수에 0(false)을 저장한다.
만약 is_full 이 1 이라면, 즉 cache_set 가 가득 차있는 상태라면, LRU replacement policy 에 따라 min_used_idx 의 block 을 evict 하고, miss 횟수와 evict 횟수를 각각 1 만큼 증가시킨다.
만약 is_full 이 0 이라면, 즉 cache_set 에 empty block 이 존재한다면, 가장 처음의 empty block 에 requested 값을 저장해주고 miss 횟수를 1 만큼 증가시킨다.
마지막으로, 아래와 같이 file pointer 로 파일을 close 해주고 할당되어 있던 메모리를 해제해준다.

```
fclose(file);

    //free
    for(i = 0; i < S; i++){
        free(cache_line[i]);
    }
    free(cache_line);

    printSummary(num_hit, num_miss, num_evict); //hit, miss, eviction
```

위와 같이 cache simulator 를 구현한 결과, 아래와 같이 CSIM_RESULTS 값이 26 으로 도출되었다.

```
                        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
      2 (4,2,4)     259       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
     26

 TEST_CSIM_RESULTS=26
```

## 3. Part B

다음은 기본적으로 제공되는 transpose 함수이다.

```
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, tmp;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            tmp = A[i][j];
            B[j][i] = tmp;
        }
    }

}
```

위 함수로 transpose 를 구현할 수 있지만, cache miss 가 많이 발생하기 때문에 최대한 miss 횟수를 줄일 수 있도록 각 input matrix size 에 맞는 함수를 구현해야 한다.
우선, 아래와 같이 transpose_submit 함수를 구현하여 M 과 N 의 값에 따라 원하는 함수가 호출되도록 하였다.

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    if(M == 32 && N == 32) transpose_32(M, N, A, B);
    else if(M == 64 && N == 64) transpose_64(M, N, A, B);
    else if(M == 61 && N == 67) transpose_61(M, N, A, B);
    return;

}
```

### (1) 32 x 32 matrix

32x32 matrix 는 이를 8 x 8 의 block 씩 잘라서 matrix 를 접근해준다면 cache miss rate 를 감소시킬 수 있다고 생각하였다. 따라서 아래와 같이 8 개의 temp local variable 을 선언하여 8x8 block 으로 matrix 에 접근하는 코드를 작성하였다.

```
void transpose_32(int M, int N, int A[N][M], int B[M][N]){
    int i, j, _i; //variable for iteration
    int tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, tmp8;
    for(i = 0; i < M; i += 8){
```

```
        for(j = 0; j < N; j += 8){
            for(_i = i ; _i < i + 8; _i++){
                tmp1 = A[_i][j];
                tmp2 = A[_i][j+1];
                tmp3 = A[_i][j+2];
                tmp4 = A[_i][j+3];
                tmp5 = A[_i][j+4];
                tmp6 = A[_i][j+5];
                tmp7 = A[_i][j+6];
                tmp8 = A[_i][j+7];

                B[j][_i] = tmp1;
                B[j+1][_i] = tmp2;
                B[j+2][_i] = tmp3;
                B[j+3][_i] = tmp4;
                B[j+4][_i] = tmp5;
                B[j+5][_i] = tmp6;
                B[j+6][_i] = tmp7;
                B[j+7][_i] = tmp8;
            }
        }
    }
    return;
}
```

A matrix 에서 _i 행의 j ~ j + 7 까지의 원소를 차례로 tmp1, tmp2, …, tmp7 에 저장하고, 이를 B matrix 의 _i 열, j ~ j + 7 까지의 행에 저장해주었다. 이렇게 8 x 8 blocking 방식으로 transpose 를 구현한 결과 아래와 같이 miss 횟수가 287 회로 도출되었다.

```
[jcy2749@programming2 cachelab-handout]$ ./test-trans -M 32 -N 32

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255

Function 1 (2 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:870, misses:1183, evictions:1151

Summary for official submission (func 0): correctness=1 misses=287

TEST_TRANS_RESULTS=1:287
```

**(2) 64 x 64 matrix**

64x64 matrix 도 32x32 matrix 와 같이 8x8 blocking 을 사용하여 matrix 에 접근해보려고 하였다. 그 결과 cache miss 횟수가 4611 회로 굉장히 큼을 확인할 수 있었고, 32x32 matrix 와는 다른 방식으로 접근해야 함을 확인하였다.

그래서 우선, 64 x 64 matrix 를 8x8 block 으로 쪼개서 접근하되, 8x8 block 안에서 다시 4x4 block 4 개로 쪼개서 matrix 에 접근해보았다. 아래 코드와 같이, 4x4 block 을 접근하는 순서는 i ~ i+3 번째 행, j ~ j + 3 번째 열의 block, i+4 ~ i+7 번째 행, j ~ j+3 번째 열의 block, i ~ i+3 번째 행, j+4 ~ j + 7 번째 열의 block, , i+4 ~ i+7 번째 행, j+4 ~ j+7 번째 열의 block 순서로 접근하였다.

```
void transpose_64(int M, int N, int A[N][M], int B[M][N]){
    int i, j, _i; //variable for iteration
    int tmp1, tmp2, tmp3, tmp4;
    for(i = 0; i < M; i += 8){
        for(j = 0; j < N; j += 8){
            for(_i = i ; _i < i + 4; _i++){
                tmp1 = A[_i][j];
                tmp2 = A[_i][j+1];
                tmp3 = A[_i][j+2];
                tmp4 = A[_i][j+3];

                B[j][_i] = tmp1;
                B[j+1][_i] = tmp2;
                B[j+2][_i] = tmp3;
                B[j+3][_i] = tmp4;
            }
            for(_i = i + 4; _i < i + 8; _i++){
                tmp1 = A[_i][j];
                tmp2 = A[_i][j+1];
                tmp3 = A[_i][j+2];
                tmp4 = A[_i][j+3];

                B[j][_i] = tmp1;
                B[j+1][_i] = tmp2;
                B[j+2][_i] = tmp3;
                B[j+3][_i] = tmp4;
            }
            for(_i = i ; _i < i + 4; _i++){
                tmp1 = A[_i][j+4];
                tmp2 = A[_i][j+5];
                tmp3 = A[_i][j+6];
                tmp4 = A[_i][j+7];

                B[j+4][_i] = tmp1;
                B[j+5][_i] = tmp2;
                B[j+6][_i] = tmp3;
                B[j+7][_i] = tmp4;
            }
            for(_i = i + 4; _i < i + 8; _i++){
                tmp1 = A[_i][j+4];
                tmp2 = A[_i][j+5];
                tmp3 = A[_i][j+6];
```

```
                tmp4 = A[_i][j+7];

                B[j+4][_i] = tmp1;
                B[j+5][_i] = tmp2;
                B[j+6][_i] = tmp3;
                B[j+7][_i] = tmp4;
            }

        }
    }
    return;
}
```

위와 같은 방식으로 64 x 64 matrix transpose 를 구현한 결과 아래와 같이 miss 횟수가 1651 회로 도출되었다.

```
[jcy2749@programming2 cachelab-handout]$ ./test-trans -M 64 -N 64

Function 0 (2 total)
Step 1: Validating and generating memory traces
sdb1: warning, user block quota exceeded.
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6546, misses:1651, evictions:1619

Function 1 (2 total)
Step 1: Validating and generating memory traces
sdb1: warning, user block quota exceeded.
sdb1: write failed, user block limit reached.
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:0, misses:0, evictions:0

Summary for official submission (func 0): correctness=1 misses=1651

TEST_TRANS_RESULTS=1:1651
```

**(3) 61 x 67 matrix**

61x67 matrix 도 8x8 blocking 을 통해 접근을 해보았지만, 아래와 같이 user block limit reached 오류가 발생하였다.

```
[jcy2749@programming2 cachelab-handout]$ ./test-trans -M 61 -N 67

Function 0 (2 total)
Step 1: Validating and generating memory traces
sdb1: warning, user block quota exceeded.
sdb1: write failed, user block limit reached.
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:0, misses:0, evictions:0
```