컴퓨터 SW시스템 개론 Lab 8 & 9 Report

20220923 이윤혁

1. 개요

Lab 8 & 9에서는 cache의 구조를 이해하고 C를 이용해 cache의 작동을 구현한다.

2. 구현

2.1. csim.c

csim.c은 operation, address, size를 포함하고 있는 trace file을 읽어 cache access를 시뮬레이션하는 코드이다.

2.1.1. 옵션 입력

writeup lab에 명시된 대로 getopt.h를 이용해 cache의 meta data인 s, E, b를 입력받았다. verbose 모드 플래그인 v와 trace파일의 파일명도 옵션으로 받는다. 다음 코드는 옵션을 입력받는다.

```
while ((option = getopt(argc, argv, "s:E:b:t:hv")) != -1){
        if (option == 'v')
            is_display_trace = 1;
        else if (option == 's')
            s = atoi(optarg);
        else if (option == 'E')
            E = atoi(optarg);
        else if (option == 'b')
            b = atoi(optarg);
        else if (option == 't')
            t = optarg;
        else {
            print_help();
            return 1;
        }
}
```

2.1.2. cache 구축

입력받은 s, E, b를 토대로 cache를 구축한다. 다음 코드는 cache를 구성하는 struct이다.

```
// Define Cache components
typedef struct {
	int valid; // vaild bit
	unsigned long long tag; // tag
	unsigned long long lru;
} Line;

typedef struct {
	Line* line; // line
} Set;

typedef struct {
	Set* set; // set
	// Meta data
	int s;
	int E;
	int b;
} Cache;
```

다음 코드는 cache를 동적할당하여 구축하는 코드이다.

```
Cache* construct_cache(Cache* cache, int s, int E, int b){

	// Allocate cache
	cache = malloc(sizeof(Cache));

	// Record metadata
	cache->s = s;
	cache->E = E;
	cache->b = b;

	// Allocate sets
	cache->set = calloc(1 << s, sizeof(Set));

	// Allocate lines
	for (int i = 0; i < (1 << s); i++)
		cache->set[i].line = calloc(E, sizeof(Line));

	return cache;
}
```

2.1.3. address parsing

address로부터 tag, set index, block offset를 얻기 위해 bitwise operation을 이용했다. 다음 코드는 tag, set index와 block offset을 얻는다.

```
// Create mask of set index
mask_of_set_index = (~0) << (64 - cache->s - cache->b);
mask_of_set_index = mask_of_set_index >> (64 - cache->s);
mask_of_set_index = mask_of_set_index << (cache->b);

// Create mask of block offset
mask_of_block_offset = (~0) << (64 - cache->b);
mask_of_block_offset = mask_of_block_offset >> (64 - cache->b);
```

```
// Parse address
tag = (address) >> (cache->s + cache->b);
set_index = (address & mask_of_set_index) >> (cache->b);
block_offset = address & mask_of_block_offset;
```

2.1.4. trace simulation

다음 코드는 trace를 읽어 cache access를 시뮬레이션하는 함수이다.

```
void simulate_trace(FILE* fp, Cache* cache, int is_display_trace, int*
hit_count, int* miss_count, int* eviction_count){
    // Trace variable
    char operation;
    unsigned long long address;
    unsigned int size;
    int result;
    unsigned int global_lru = 0;
    unsigned long long tag, set_index, block_offset;
    unsigned long long mask_of_set_index = 0, mask_of_block_offset = 0;

    // Create mask of set index
    mask_of_set_index = (~0) << (64 - cache->s - cache->b);
    mask_of_set_index = mask_of_set_index >> (64 - cache->s);
    mask_of_set_index = mask_of_set_index << (cache->b);
```

```
// Create mask of block offset
mask_of_block_offset = (~0) << (64 - cache->b);
mask_of_block_offset = mask_of_block_offset >> (64 - cache->b);

// Read trace file
while (fscanf(fp, " %c %llx, %u" , &operation, &address, &size) != EOF) {

    // Parse address
    tag = (address) >> (cache->s + cache->b);
    set_index = (address & mask_of_set_index) >> (cache->b);
    block_offset = address & mask_of_block_offset;

    if (operation == 'I')
        continue;

    else if (operation == 'L') {
        result = access(cache, tag, set_index, block_offset, &global_lru);
        update_count(result, hit_count, miss_count, eviction_count);
        if (is_display_trace){
            printf("%c %llx, %u ", operation, address, size);
            display_trace(result);
            printf("₩n");
        }
    }

    else if (operation == 'M') {
        result = access(cache, tag, set_index, block_offset, &global_lru);
        update_count(result, hit_count, miss_count, eviction_count);
        if (is_display_trace) {
            printf("%c %llx, %u ", operation, address, size);
            display_trace(result);
        }

        result = access(cache, tag, set_index, block_offset, &global_lru);
        update_count(result, hit_count, miss_count, eviction_count);
        if (is_display_trace) {
            display_trace(result);
            printf("₩n");
```

```
            }
        }

        else if (operation == 'S') {
            result = access(cache, tag, set_index, block_offset, &global_lru);
            update_count(result, hit_count, miss_count, eviction_count);
            if (is_display_trace) {
                printf("%c %llx, %u ", operation, address, size);
                display_trace(result);
                printf("₩n");
            }
        }
    }

    return;
}
```

trace 파일을 열어 한 줄씩 operation, address, size를 읽고, address를 parse하여 tag, set index, block offset을 구한다. access함수로 tag와 set index에 부합하는 cache를 찾고, hit, miss, eviction miss를 판별한다.

다음 코드는 cache의 접근을 시뮬레이션하는 access 함수이다.

```
int access(Cache* cache, unsigned long long tag, unsigned long long
set_index, unsigned long long block_offset, unsigned int* global_lru) {
    unsigned int min_lru = 4294967295;
    int target_index = 0;

    for (int i = 0; i < cache->E; i++){
        if (cache->set[set_index].line[i].valid == 1){
            if (cache->set[set_index].line[i].tag == tag){
                cache->set[set_index].line[i].lru = *global_lru;
                (*global_lru)++;
                return 0;
            }
        }
    }
    // If the function did not return, it is cold miss.
    for (int i = 0; i < cache->E; i++){
        if (cache->set[set_index].line[i].valid == 0){
            cache->set[set_index].line[i].valid = 1;
```

```
                cache->set[set_index].line[i].tag = tag;
                cache->set[set_index].line[i].lru = *global_lru;
                (*global_lru)++;
                return 1;
            }
        }

        // If the function did not return, it is eviction miss
        for (int i = 0; i < cache->E; i++){
            if (cache->set[set_index].line[i].lru < min_lru){
                min_lru = cache->set[set_index].line[i].lru;
                target_index = i;
            }
        }
        cache->set[set_index].line[target_index].tag = tag;
        cache->set[set_index].line[target_index].lru = *global_lru;
        (*global_lru)++;

        return 2;
}
```

line마다 LRU 값을 저장하여, set에서 가장 접근한지 오래된 line을 찾는다.

return value로 hit, miss, eviction miss를 찾는다.

다음 코드는 hit, miss, eviction miss를 세는 함수이다.

```
void update_count(int result, int* hit_count, int* miss_count, int*
eviction_count){
    if (result == 0)
        (*hit_count)++;
    else if (result == 1){
        (*miss_count)++;
    }
    else if (result == 2){
        (*miss_count)++;
        (*eviction_count)++;
    }

    return;
}
```

2.1.5. 프로그램 종료

cache access 시뮬레이션이 끝난 후, 프로그램을 종료한다. 종료시 printSummery()함수를 이용해 hit, miss, eviction miss를 출력한다.

그 다음 cache를 할당해제하고, 프로그램을 종료한다.

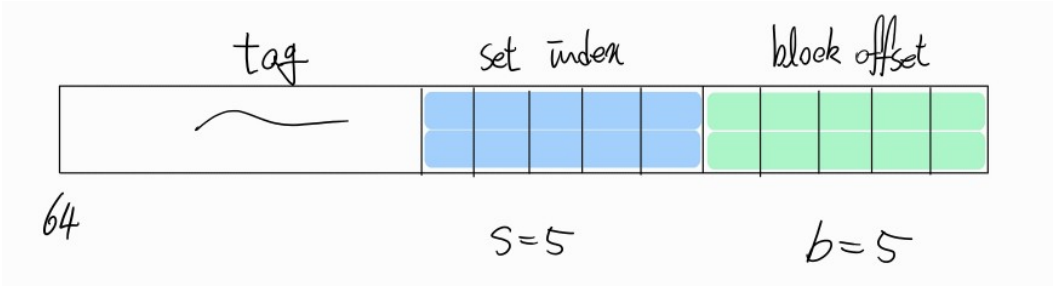
다음 코드는 cache를 할당해제한다.

```
void free_cache(Cache* cache){

    // Free lines
    for (int i = 0; i < cache->s; i++)
        free(cache->set[i].line);

    // Free sets
    free(cache->set);

    // Free cache
    free(cache);

    return;
}
```

2.2. trace.c

trace.c에서는 32 x 32, 64 x 64, 67 x 61 matrix를 transpose한다. 이때 사용하는 cache는 S = 32, E = 1, B = 32의 속성을 가진다. 따라서 cache는 총 32개의 set을 가지고, 각 set은 1개의 cache line, cache line은 각각 32 byte를 저장할 수 있다.

address의 구조를 분석하면 다음과 같다.



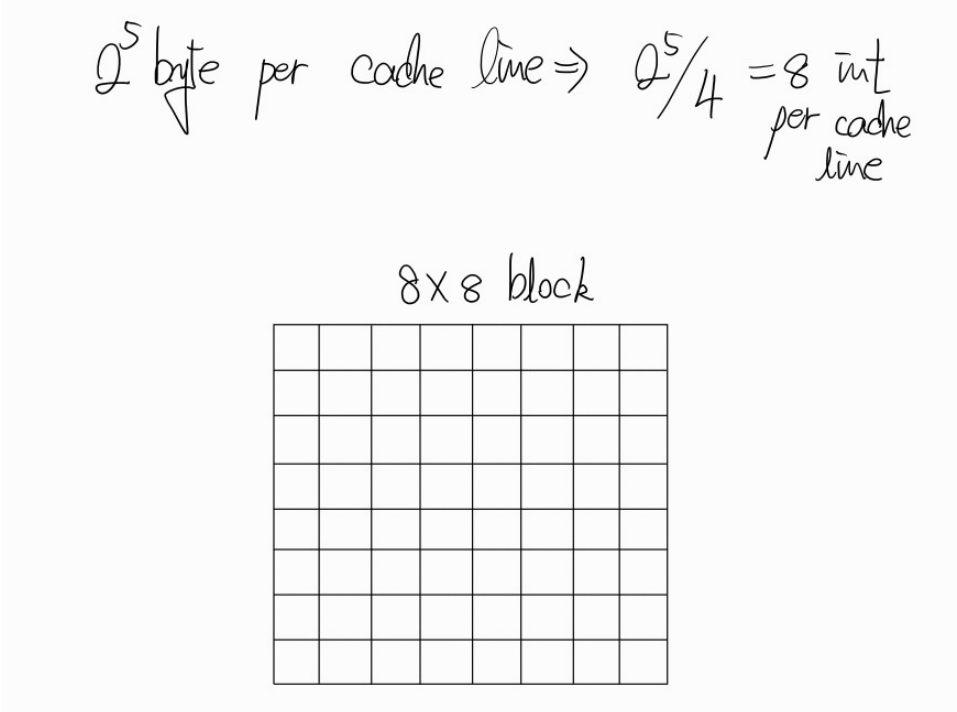
2.2.1. void transpose_submit(int M, int N, int A[N][M], int B[M][N])

각각 32 x 32, 64 x 64, 61 x 67 matrix에 대해 transpose를 수행하는 함수를 만들고, 조건문으로 기능을 분리했다. 다음은 함수의 코드이다.

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    if (M == N && M == 32) // 32 x 32 matrix
        transpose_3232(M, N, A, B);
    else if (M == N && M == 64) // 64 x 64 matrix
        transpose_6464(M, N, A, B);
    else // 61 x 67 matrix
        transpose_6167(M, N, A, B);

    return;
}
```

2.2.2. void transpose_3232(int M, int N, int A[N][M], int B[M][N])

이 함수는 32 x 32 matrix를 transpose한다.



cache line 당 32byte를 저장하므로, 한 cache line당 int는 총 32 / 4 = 8개 저

장할 수 있다. blocking method를 위해 8 x 8 block 단위로 transpose를 한다.

B의 address는 A의 맨 마지막 element address 직후에 위치하므로, A와 B의 주소는 다음과 같은 관계를 가진다.

```
B = A + row x col
row = (int 크기) 0x4 x (array height) 0x20 = 0x80
col = (array width) 0x20 = 0x20

    ➔ B = A + 0x1000 (0b100 00000 00000)
```

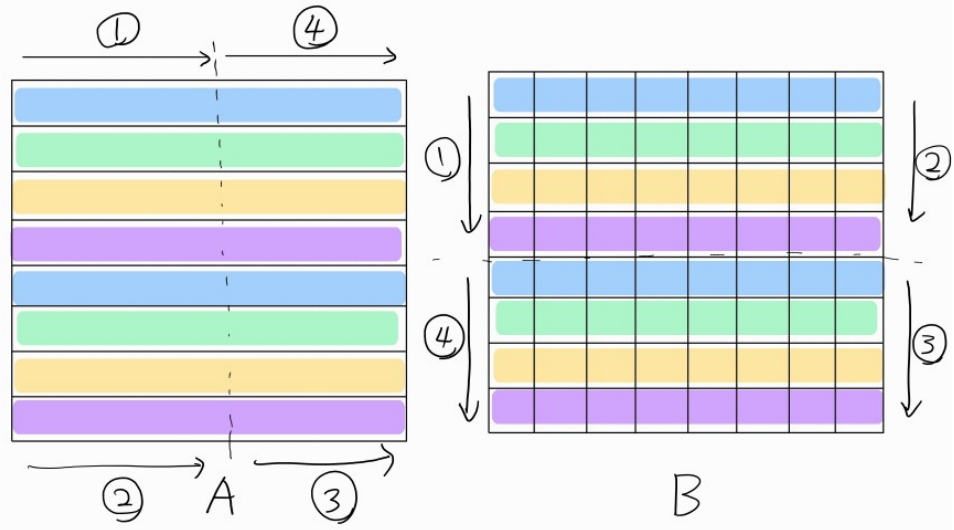
배열의 index를 i, j라고 할 때, B = A + 0x1000인 특성 때문에, A[i][j]와 B[j][i]는 i = j일 때 tag가 다르나 set index와 block offset이 같다. 따라서 transpose를 하는 반복문을 실행시, i = j인 케이스는 transpose를 실행하지 않고, block 내에서 i = j 를 제외한 모든 케이스가 transpose 된 후 실행해야 eviction miss를 최소화 할 수 있다. 이 원리를 적용해 코드를 짜면 다음과 같다.

```
void transpose_3232(int M, int N, int A[N][M], int B[M][N]){
    // Transpose 32 x 32 matrix A to 32 x 32 matrix B
    int i, j, i_, j_; // Index variables

    for(i = 0; i < N; i += 8){ // 8 -> 1 cache-line
        for (j = 0; j < M; j += 8) {

            // In-block operation
            for (i_ = 0; i_ < 8; i_++) {
                for (j_ = 0; j_ < 8; j_++) {
                    if (i + i_== j + j_) {
                        continue;
                    }
                    B[j + j_][i + i_] = A[i + i_][j + j_];
                }
                if (i == j){
                    B[j + i_][i + i_] = A[i + i_][j + i_];
                }
            }
        }
    }

    return;
}
```

2.2.3. void transpose_6464(int M, int N, int A[N][M], int B[M][N])

이 함수는 64 x 64 matrix를 transpose한다.

64 x 64 matrix는 column이 길어 8 x 8 block 내 address가 32 x 32 matrix와 다르다. set index와 block offset을 분석해보면, 한 8 x 8 내 set index와 block offset이 겹치는 element가 4행 주기에 걸쳐 나타난다. set index와 block offset을 색을 칠해 visualize하면 아래 그림과 같다.



eviction miss를 최소화하기 위해, 8 x 8 matrix를 4분할하여 transpose를 수행해야한다. 분할한 component를 각각 top left, top right, bottom left, bottom right로 명명하면, A의 component는 다음과 top left(①) bottom left(②), bottom right(③), top left(④) 순으로 접근해야 cache 내 데이터를 최대한 재사용할 수 있다.

32 x 32 matrix와 동일하게 i = j일 때 eviction miss가 난다. 단 4행 주기로 set index 및 block offset이 같아지는 element들이 있으므로, i – j = 4이거나 j – I = 4일 때도 block내 transpose 처리 완료 후 따로 처리해야한다.

모든 block 내 operation을 위와 같이 수행하는 코드는 다음과 같다.

```
void transpose_6464(int M, int N, int A[N][M], int B[M][N]){
    // Transpose 64 x 64 matrix A to 64 x 64 matrix B
    int i, j, i_, j_; // Index variables

    for (i = 0; i < N; i += 8) { // 8 -> 1 cache-line
        for (j = 0; j < N; j += 8) {
```

```
// In-block operation
for (i_ = 0; i_ < 8 / 2; i_++) {
    for (j_ = 0; j_ < 8 / 2; j_++) {
        if (i + i_== j + j_) {
            continue;
        }
        B[j + j_][i + i_] = A[i + i_][j + j_];
    }
    if (i == j) {
        B[j + i_][i + i_] = A[i + i_][j + i_];
    }
}
// In-block operation
for (i_ = 8 / 2; i_ < 8; i_++) {
    for (j_ = 0; j_ < 8 / 2; j_++) {
        if (i == j && i_ - j_ == 8 / 2) {
            continue;
        }
        B[j + j_][i + i_] = A[i + i_][j + j_];
    }
    if (i == j) {.
        B[j + i_ - 8 / 2][i + i_] = A[i + i_][j + i_ - 8 / 2];
    }
}
// In-block operation
for (i_ = 8 / 2; i_ < 8; i_++) {
    for (j_ = 8 / 2; j_ < 8; j_++) {
        if (i + i_== j + j_) {
            continue;
        }
        B[j + j_][i + i_] = A[i + i_][j + j_];
    }
    if (i == j) {
        B[j + i_][i + i_] = A[i + i_][j + i_];
    }
}
// In-block operation
for (i_ = 0; i_ < 8 / 2; i_++) {
```

```
                    for (j_ = 8 / 2; j_ < 8; j_++) {
                        if (i == j && j_ - i_ == 8 / 2) {
                            continue;
                        }
                        B[j + j_][i + i_] = A[i + i_][j + j_];
                    }
                    if (i == j) {
                        B[j + i_ + 8 / 2][i + i_] = A[i + i_][j + i_ + 8 / 2];
                    }
                }

            }
        }

        return;
}
```

2.2.4. void transpose_6167(int M, int N, int A[N][M], int B[M][N])

이 함수는 67 x 61 matrix를 transpose한다.

67 x 61 matrix는 64 x 64 matrix와 다르게 squere matrix가 아니고, width 및 height가 8의 배수가 아니므로 8n x 8n 꼴 block 사용시 set index 및 block offset이 행을 주기로 맞아떨어지지 않는다. 따라서 이 matrix를 transpose할 때 8 x 8 block 또는 16 x 16 block을 사용할 수 있다. 직관적으로, 8 x 8 block 내에서 transpose 진행 시, 16 x 16 block에 비해 set index와 block offset이 같은 A, B의 element끼리 transpose될 확률이 높다. 또한 caching을 여러 번 해야하므로, 16 x 16 block을 사용한다. set_index와 block offset이 당연히 같은 경우는 i = j = 0일 때 밖에 없으므로, 그 부분을 block 내 transpose가 진행된 이후 따로 처리한다. 이 원리를 적용해 코드를 짜면 다음과 같다.

```
void transpose_6167(int M, int N, int A[N][M], int B[M][N]){
    // Transpose 67 x 61 matrix A to 61 x 67 matrix B
    int i, j, i_, j_; // Index variable

    for(i = 0; i < N; i += 16){ // 16 -> 2 cache-line
        for(j = 0; j < M; j += 16){

            // In-block operation
```

```
                for(i_ = 0; i_ < 16 && i + i_ < N; i_++){
                    for(j_ = 0; j_ < 16 && j + j_ < M; j_++){
                        if (i + i_ == j + j_ && i + i_ == 0){
                            continue;
                        }
                        B[j + j_][i + i_] = A[i + i_][j + j_];
                    }
                    if (i == j && i + i_ == 0){
                        B[0][0] = A[0][0];
                    }
                }
            }
        }

        return;
}
```

2.3. 결과

2.3.1. csim.c 결과

```
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
     3 (4,2,4)        4       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
    27


TEST_CSIM_RESULTS=27
```

test-csim 파일을 이용한 결과, 구현이 올바르게 이루어졌음을 확인할 수 있다.

2.3.2. trans.c 결과

다음은 ./test-trans -M 32 -N 32로 실행한 결과이다.

```
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
```

correctness=1, total miss가 287로, writeup lab에 제시된 만점 기준을 통과한다.

다음은 ./test-trans -M 64 -N 64로 실행한 결과이다.

```
        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:6770, misses:1427,
        evictions:1395

        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:3474, misses:4723,
        evictions:4691

        Summary for official submission (func 0): correctness=1
        misses=1427

TEST_TRANS_RESULTS=1:1427
```

correctness=1, total miss가 1427로, writeup lab에 제시된 만점 기준을 통과하지는 못하지만, 부분 점수기준에 부합한다.

다음은 ./test-trans -M 61 -N 67로 실행한 결과이다.

```
        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:6189, misses:1990,
evictions:1958

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:3756, misses:4423,
evictions:4391

Summary for official submission (func 0): correctness=1
misses=1990

TEST_TRANS_RESULTS=1:1990
```

correctness=1, total miss가 1990로, writeup lab에 제시된 만점 기준을 통과한다.

trans.c도 구현이 어느정도 올바르게 되었음을 확인할 수 있다.

이번 랩을 통해 cache의 구조, address의 구조, hit, miss, eviction, cache blocking, cache-friendly optimization을 익힐 수 있었다.

2.4. 참고한 자료

matrix transpose의 blocking에 대한 아이디어 및 이해를 얻기 위해 다음 링크를 참고했습니다.

https://www.youtube.com/watch?v=huz6hJPl_cU