

CSED211: CacheLab Report

20240832 / 채승현

1. Part A – Cache Simulator (20240832_csim.c)

먼저 편의를 위해 여러 함수에서 사용되는 s, E, b를 전역으로 선언해주었다. s, b는 2의 지수로 들어갈 것이고, E 또한 아주 커지지 않을 것이기 때문에(int의 최댓값을 가질 경우 약 21억 * 32 바이트가 필요함) 전부 int로 선언해도 충분하다. trace할 파일의 경로를 나타내는 trace_file은 char형 포인터로 선언하여 입력된 문자열을 가리키도록 하였다. verbose는 1인 경우(정확히는 0이 아닌 경우) hit, miss, eviction이 일어날 때마다 메시지를 출력하도록 하는 flag 역할을 한다. 기본값은 0이다. 또한 hit, miss, eviciton이 몇 번 일어났는지 저장할 hit_count, miss_count, eviction_count도 int형으로 선언하였다. 이는 test하는 모든 파일의 hit, miss, eviction이 int형을 넘지 않는 것을 ./test-csim을 통해 확인하였기 때문이다. 마지막으로 unsigned int형 global_timer는 LRU를 구현하기 위해 두었다.

```
typedef struct {
    int valid;
    unsigned long long tag;
    unsigned int last_access;
} CacheLine;

typedef struct {
    CacheLine* lines;
} CacheSet;

typedef struct {
    CacheSet* sets;
} Cache;

Cache cache;
```

이후 CacheLine, CacheSet, Cache 순으로 구조체를 선언하였다. CacheLine에는 int형 valid (0: 유효하지 않음, 1: 유효함), unsigned long long형 tag (입력으로 들어올 address가 64 byte), unsigned int형 last_access (global_timer와 같도록, LRU 구현에 사용)을 넣었다. 그리고 CacheSet에는 CacheBlock형 포인터 lines를 넣어 malloc을 통해 원하는 크기로 만들 수 있도록 했다. Cache도 CacheSet형 포인터 sets를 넣어 malloc을 통해 원하는 크기로 만들 수 있도록 했다. 마지막으로 전역으로 Cache형 cache를 선언해주었다.

이제부터 함수들을 살펴보겠다.

먼저 allocateCache이다. 이 함수는 이름과 같이 Cache를 할당하는 함수이다. Cache에 2^s 개 만큼의 CacheSet을 malloc으로 할당한다. 또한 각 CacheSet에 E가 만큼의 CacheLine을 malloc으로 할당한다. 각 line의 valid, tag, last_access는 전부 0으로 초기화 한다.

두 번째는 freeCache이다. 이 함수는 이름과 같이 Cache의 메모리를 해제하는 함수이다. 우선 CacheSet을 돌면서 CacheLine들을 전부 메모리 해제한 후, CacheSet도 메모리 해제한다. 1차원 배열의 경우 배열의 시작 주소만 free에 넣어도 배열의 모든 원소가 메모리 해제된다.

세 번째는 accessCache이다. 이 함수는 Cache Hit / Miss / Eviction을 처리하는 함수라고 할 수 있다. 먼저 파라미터로 주어진 64 byte인 address와 s, b를 이용해 set_index와 tag를 계산한다. set_index는 맨 오른쪽 b (block offset)개의 비트를 없애고 맨 오른쪽에서부터 s개의 비트를 넣어서 구한다. tag는 맨 오른쪽 s + b개의 비트를 없애서 담는다. 즉, address의 상위 비트들을 구한 것이다. 이후 cache의 set_index 번째 CacheSet의 주소를 CacheSet 포인터 set에 넣어서 간단히 사용할 수 있도록 했다. 또한 global_timer를 1만큼 올리고, int형 변수 empty를 -1, lru를 0으로 선언하였다. empty는 처음으로 valid하지 않은 CacheLine의 index를 저장하는 용도이며, lru는 말 그대로 LRU의 index를 저장하는 용도이다. 그리고 unsinged int형 min_time을 -1로 초기화하여 LRU를 구할 때 사용할 수 있도록 하였다. unsigned int에서 -1은 제일 큰 값이기 때문에, 순회를 도는 처음부터 lru 변수가 갱신되도록 하는 것이다.

```

for (int i = 0; i < E; i++) {
    if (set->lines[i].valid) {
        if (set->lines[i].tag == tag) { // Hit
            hit_count++;
            set->lines[i].last_access = global_timer;
            if (verbose) printf(" hit");
            return;
        }
        if (set->lines[i].last_access < min_time) { // LRU 찾기
            min_time = set->lines[i].last_access;
            lru = i;
        }
    } else if (empty == -1) {
        empty = i; // 빈 공간 찾기
    }
}

```

이후 set에서 같은 tag를 가진 CacheLine이 있는지 순차적으로 확인한다. 확인하는 CacheLine이 valid하다면 그 CacheLine의 tag를 우리가 이 함수의 처음에 구했던 tag를 비교한다. 두 tag가 같다면 Cache Hit이므로, hit_count를 1만큼 올리고, 해당 CacheLine의 last_access를 global_timer로 변경한다. 또한 verbose가 1이면 hit를 출력하고 함수를 바로 종료한다. 만약 두 tag가 다른 경우, 해당 CacheLine의 last_access가 min_time보다 작다면 변수 lru를 해당 CacheLine의 index로 바꾸고, min_time을 해당 CacheLine의 last_access로 바꾼다. 이 과정을 통해 set에서의 LRU를 찾을 수 있다.

```

miss_count++; // Miss
if (verbose) printf(" miss");

int target = (empty != -1) ? empty : lru;
if (empty == -1) {
    eviction_count++; // Eviction
    if (verbose) printf(" eviction");
}

```

tag가 일치하는 CacheLine이 set에 없다면 Cache Miss이므로, miss_count를 1만큼 올리고, verbose가 1이라면 miss를 출력하도록 했다. 이 경우, 캐시에 해당 tag를 넣어야 하므로 이를 넣을 CacheLine의 index, 즉 target을 정해야 한다. 여기서 empty가 -1이 아니라면 빈 CacheLine이 있는 것이므로 empty값을 target으로 정한다. 그렇지 않다면 변수 lru의 값을 target으로 한다. 이는 해당 set에 빈 CacheLine이 있다면 그 곳에 tag를 넣고, 그렇지 않다면 LRU에 tag를 넣겠다는 말이다. 그래서 empty가 -1이라면 Cache Eviction이므로, eviction_count를 1만큼 올리고 verbose가 1이면 eviction을 출력한다.

마지막으로 set의 target번째 CacheLine의 valid에는 1, tag에는 함수의 처음에 구한 tag, last_access에는 global_timer를 넣으면 accessCache 구현이 모두 완료된다.

```
[cotmdgus@programming2 cachelab-handout]$ ./csim-ref -h
Usage: ./csim-ref [-hv] -s <num> -E <num> -b <num> -t <file>
Options:
  -h      Print this help message.
  -v      Optional verbose flag.
  -s <num> Number of set index bits.
  -E <num> Number of lines per set.
  -b <num> Number of block offset bits.
  -t <file> Trace file.

Examples:
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
linux> ./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace
```

```
char opt;
while ((opt = getopt(argc, argv, "hvs:E:b:t")) != -1) {
    switch (opt) {
    case 'h': help(); exit(1); break;
    case 'v': verbose = 1; break;
    case 's': s = atoi(optarg); break;
    case 'E': E = atoi(optarg); break;
    case 'b': b = atoi(optarg); break;
    case 't': trace_file = optarg; break;
    default: exit(1); // 잘못된 인자 들어오면 그냥 종료
    }
}

if (s == 0 || E == 0 || b == 0 || trace_file == NULL) {
    printf("Missing arguments\n"); // 필수 인자 없으면 메시지 띄우고 종료
    exit(1);
}
```

네 번째는 help이다. 이 함수는 ./csim -h이 입력되었을 때 호출되어 csim의 사용법을 출력하는 함수이다. 이 함수의 출력 결과는 원쪽의 ./csim-ref -h를 했을 때 나오는 출력 결과처럼 나오도록 했다.

이제 main 함수를 살펴보자. [CSED211] Cache Lab.pdf 24p를 참고하여 csim-ref와 같은 입력 형식을 구현하였다. -h가 있는 경우엔 help를 호출한 후 종료시켰다. 또한 -v가 있는 경우 verbose를 1로 바꾸었다. s, E, b에는 atoi로 입력된 문자를 int형으로 바꾸어 넣어줬다. 또한 trace_file이 마지막에 입력된 파일의 경로를 가리키도록 했다. 만약 이상한 값이 입력되거나, 인자 값이 이상하다면 종료하도록 하였다.

입력 값에 문제가 없다면, allocateCache를 통해 주어진 입력 값에 맞도록 Cache를 할당한다. 그리고 trace_file 경로에 있는 파일을 fopen으로 불러와서 fscanf를 이용해 operation (op), address (addr), size를 while을 통해 순차적으로 읽어온다. (fscanf의 반환값은 읽어온 값의 개수) Writeup의 설명에 따라 operation이 1인 경우는 바로 continue를 하고, 그게 아닌 경우 verbose가 1이라면 operation address,size를 출력하도록 했다. 이후 accessCache에 address를 넣어 호출한다. 여기서 operation이 M (Modify)인 경우 Load와 Store로 2번 접근하기 때문에 한 번 더 accessCache를 호출하였다.

마지막으로 fclose로 파일을 닫고, freeCache를 통해 Cache의 메모리를 해제한 후 printSummary를 통해 hit_count, miss_count, eviction_count가 출력되도록 했다.

```
[cotmdgus@programming2 cachelab-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/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

이렇게 구현된 csim을 컴파일하고, ./test-csim을 통해 확인해본 결과 27점으로 만점을 얻을 수 있었다.

2. Part B – Optimized matrix transpose (20240832_trans.c)

Part B에서 주어진 Cache는 $s = 5$, $E = 1$, $b = 5$ 로 cache block 크기가 32 byte이다.

먼저 지역 변수를 12개까지 사용할 수 있으므로 반복문에 사용할 i, j, k, l 그리고 임시로 데이터를 저장하는 데 사용할 $v0, v1, v2, v3, v4, v5, v6, v7$ 을 모두 int형으로 선언해주었다. writeup에서는 주어지는 입력이 $32 * 32, 64 * 64, 61 * 67$ 밖에 없고, 이 특정한 케이스에 각각에 대해서 miss를 최소화하는 작업을 수행하여도 된다고 하였기에 이 세 경우를 각각 나누어 구현하였다.

```
if (M == 32 && N == 32) {
    for (i = 0; i < N; i += 8) {
        for (j = 0; j < M; j += 8) {
            for (k = i; k < i + 8; k++) {
                v0 = A[k][j];
                v1 = A[k][j + 1];
                v2 = A[k][j + 2];
                v3 = A[k][j + 3];
                v4 = A[k][j + 4];
                v5 = A[k][j + 5];
                v6 = A[k][j + 6];
                v7 = A[k][j + 7];

                B[j][k] = v0;
                B[j + 1][k] = v1;
                B[j + 2][k] = v2;
                B[j + 3][k] = v3;
                B[j + 4][k] = v4;
                B[j + 5][k] = v5;
                B[j + 6][k] = v6;
                B[j + 7][k] = v7;
            }
        }
    }
}
```

첫 번째로 $32 * 32$ 의 경우를 살펴보자. 가장 기본적인 방법인 blocking을 사용하여 miss를 줄여보자 하였다. 여기서 cache block의 크기가 32 byte이고, int가 4 byte이기 때문에 block의 크기를 $8 * 8$ 로 잡아 왼쪽과 같이 코드를 작성했다. 세 번째 for문 안에서 cache miss를 최소화하기 위해서 $A[k][j]$ 를 불러온 이후 $A[k][j+7]$ 까지 총 8개의 값을 불러와 $v0 \sim v7$ 에 임시로 저장하였다. 여기까지 cache miss는 1번 일어난다. 이후 $B[j][k] \sim B[j+7][k]$ 에 $v0 \sim v7$ 값을 넣어 transpose가 된다. 여기서 세 번째 for문의 첫 번째 실행의 경우 B 에 값을 넣는 부분에서 cache miss가 전부 발생한다. 하지만 이것을 다시 7번 더 반복할 때는 전부 cache hit가 될 것이라 예상했다.

```
[cotmdgus@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
```

로 300보다 작아 8 point를 받을 수 있음을 확인하였다.

```
else if(M == 61 && N == 67){
    for (i = 0; i < N; i += 16) {
        for (j = 0; j < M; j += 16) {
            for (k = i; k < i + 16 && k < N; k++) {
                for (l = j; l < j + 16 && l < M; l++) {
                    B[l][k] = A[k][l];
                }
            }
        }
    }
}
```

A와 B가 연달아 선언되었으며, 크기가 $2^n, 2^n$ 형태이기에 같은 set index를 사용하여 많은 miss가 나는 문제가 발생할 수 있지만, 일단 이 코드로 $32 * 32$ 의 경우를 테스트해보니 miss가 287

두 번째로는 $61 * 67$ 의 경우를 먼저 살펴보자. 61과 67은 소수이기 때문에 같은 set index를 사용해서 많은 miss가 나는 문제가 발생하지 않을 확률이 매우 높았다. 이에 따라 기본적인 방법인 blocking만 사용하여 miss를 줄여보

```
[cotmdgus@programming2 cachelab-handout]$ ./test-trans -M 61 -N 67
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:6187, misses:1992, evictions:1960

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=1992
TEST_TRANS_RESULTS=1:1992
```

고자 하였다. 여기서 주어진 cache가 충분히 크기 때문에 block size를 조금 커 보이는 16으로 잡아서 테스트해보았다. 61 * 67의 경우를 테스트해보니 miss가 1992로 2000보다 작아 10 point를 받을 수 있음을 확인하였다.

```
else if (M == 64 && N == 64) {
    for (i = 0; i < N; i += 8) {
        for (j = 0; j < M; j += 8) {
            for (k = i; k < i + 4; k++) {
                v0 = A[k][j];
                v1 = A[k][j + 1];
                v2 = A[k][j + 2];
                v3 = A[k][j + 3];
                v4 = A[k][j + 4];
                v5 = A[k][j + 5];
                v6 = A[k][j + 6];
                v7 = A[k][j + 7];

                B[j][k] = v0;
                B[j + 1][k] = v1;
                B[j + 2][k] = v2;
                B[j + 3][k] = v3;

                B[j][k + 4] = v4;
                B[j + 1][k + 4] = v5;
                B[j + 2][k + 4] = v6;
                B[j + 3][k + 4] = v7;
            }

            for (k = j; k < j + 4; k++) {
                v4 = A[i + 4][k];
                v5 = A[i + 5][k];
                v6 = A[i + 6][k];
                v7 = A[i + 7][k];

                v0 = B[k][i + 4];
                v1 = B[k][i + 5];
                v2 = B[k][i + 6];
                v3 = B[k][i + 7];

                B[k][i + 4] = v4;
                B[k][i + 5] = v5;
                B[k][i + 6] = v6;
                B[k][i + 7] = v7;

                B[k + 4][i] = v0;
                B[k + 4][i + 1] = v1;
                B[k + 4][i + 2] = v2;
                B[k + 4][i + 3] = v3;
            }

            for (k = i + 4; k < i + 8; k++) {
                v4 = A[k][j + 4];
                v5 = A[k][j + 5];
                v6 = A[k][j + 6];
                v7 = A[k][j + 7];

                B[j + 4][k] = v4;
                B[j + 5][k] = v5;
                B[j + 6][k] = v6;
                B[j + 7][k] = v7;
            }
        }
    }
}
```

마지막으로는 64 * 64의 경우이다. 이 경우는 앞선 32 * 32과는 달리 한 행이 256 byte여서 4행마다 같은 set index를 공유하고, 이 때문에 Cache Miss가 매우 빈번하게 발생하게 된다. 나는 앞선 경우들과 똑같이 단순히 Block size를 4, 8과 같이 조절해보면서 여러 번 시도해보았지만, Miss 수가 기준치를 초과하여 점수를 전부 받지 못했다.

이에 대한 해결책으로 8 * 8 Blocking을 기반으로 하고, 그 블록에서도 4 * 4 단위로 나누어 데이터를 이동시키는 방법을 사용하였다. 반복문 안에서 A의 위쪽 4개 행을 읽어 v0 ~ v7에 저장하고 B의 위쪽 행에 값을 대입하는데, 나중에 B의 아래쪽 행으로 이동해야 할 값들을 B의 위쪽 행(현재 Cache에 올라와 있는 부분)에 임시로 저장해두도록 했다. 이는 B 행렬 자체를 임시 버퍼로 활용하여 B 내부의 Conflict Miss를 최소화한 것이다. 이후 A의 아래쪽 행들을 처리할 때, 미리 B의 위쪽에 저장해둔 값들을 꺼내어 제자리(B의 아래쪽)로 옮기고, A의 나머지 값들을 대입하였다.

```
[cotmdgus@programming2 cachelab-handout]$ ./test-trans -M 64 -N 64
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:9066, misses:1179, evictions:1147

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=1179
TEST_TRANS_RESULTS=1:1179
```

이 방법을 코드로 구현하여 64 * 64의 경우를 테스트해보니 miss가 1179로 1300보다 작아 8 point를 받을 수 있음을 확인하였다.