컴퓨터 SW시스템개론 Lab5 보고서

20210207 이지현

Ⅰ. 소개

part A에서는 Cache의 구조와 원리를 이해하고, address의 bit가 어떤 식으로 구성이 되어있는지, 각각의 역할과 hit, miss, eviction를 직접 구현한다. part B에서는 locality 성질을 이용하여 miss rate를 줄여 cache 친화적인 c코드를 구현하는 것을 목표로 한다.

Ⅱ. Lab 내용 요약 (8,9)

-Cache organization에 대해서 배웠다. memory address는 t bits와 s bits와 b bits로 이루어져 있으며, 각각 tag, set index, block offset이라고 불린다.

- Cache들은 S개의 set으로 이루어져 있고, E개의 line으로 이루어져 있다. E=1일 때를 directed mapped cache라고 부른다. 특정 address에 접근을 하고자 하면 cache에 들어가 특정 set을 확인하고 해당되는 tag가 있으면 B 위치를 통해 data를 가지고 온다.

-Cache hit : 프로그램이 memory block에 접근하려고 할 때, 그 memory block이 캐시 안에 있을 때를 말한다. / Cache miss ; 그 block이 cache 안에 없을 때, 가져와서 cache안에 넣는다. / Cache eviction ; 캐시가 가득차 있을 때 miss가 일어나면 한 line을 내보내야 한다.

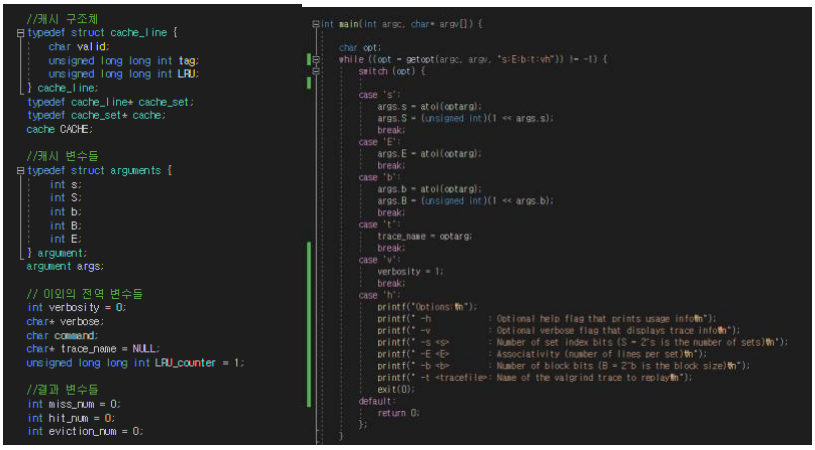
- 이 eviction에서 자주 사용하는 어떤 라인을 내보낼 지 결정하는 알고리즘을 LRU라고 하며, 제일 최근에 사용되지 않았던 라인을 내보낸다.

-row major order를 이용하면 column major order 보다 더 hit ratio가 좋다. 이는 locality를 이용하기 때문에 eviction이 적게 일어나기 때문이다.

-element별로 접근하는 것이 아니라 한 블락에 있는 데이터들을 전부 캐시로 옮기는 방식으로, 끝나면 다시 eviction하는 방식을 사용하면, 블락의 크기가 충분히 커지면 miss rate이 훨씬 적다.

Ⅲ. 코드 설명

(1) PART A



```c
//캐시 구조체
typedef struct cache_line {
    char valid;
    unsigned long long int tag;
    unsigned long long int LRU;
} cache_line;
typedef cache_line* cache_set;
typedef cache_set* cache;
cache CACHE;

//캐시 변수들
typedef struct arguments {
    int s;
    int S;
    int b;
    int B;
    int E;
} argument;
argument args;

// 이외의 전역 변수들
int verbosity = 0;
char* verbose;
char command;
char* trace_name = NULL;
unsigned long long int LRU_counter = 1;

//결과 변수들
int miss_num = 0;
int hit_num = 0;
int eviction_num = 0;
```

```c
int main(int argc, char* argv[]) {

    char opt;
    while ((opt = getopt(argc, argv, "s:E:b:t:vh")) != -1) {
        switch (opt) {

        case 's':
            args.s = atoi(optarg);
            args.S = (unsigned int)(1 << args.s);
            break;
        case 'E':
            args.E = atoi(optarg);
            break;
        case 'b':
            args.b = atoi(optarg);
            args.B = (unsigned int)(1 << args.b);
            break;
        case 't':
            trace_name = optarg;
            break;
        case 'v':
            verbosity = 1;
            break;
        case 'h':
            printf("Options:\n");
            printf("  -h            : Optional help flag that prints usage info\n");
            printf("  -v            : Optional verbose flag that displays trace info\n");
            printf("  -s <s>        : Number of set index bits (S = 2^s is the number of sets)\n");
            printf("  -E <E>        : Associativity (number of lines per set)\n");
            printf("  -b <b>        : Number of block bits (B = 2^b is the block size)\n");
            printf("  -t <tracefile>: Name of the valgrind trace to replay\n");
            exit(0);
        default:
            return 0;
        }
    }
```

우선 구조체를 이용해서 cache line에 있는 valid와 tag 비트를 구현했고, eviction의 LRU 알고리즘 구현을 위해 LRU 변수도 추가해 주었다. 이후 cache line을 cache set, cache가 차례로 갖는 순서로 이차원 배열 형태를 만들 수 있었다. 이후 S,B,E 비트를 arguments라는 구조체에 묶어 보기 쉽게 구현하였고, v 옵션을 위해 필요한 변수들, 함수 이름을 받는데 필요한 변수들, 결과 count를 저장하기 필요한 변수들을 전역변수로 선언하였다. h 옵션 case에는 문제에서 요구한 문장들을 print하고 exit하는 내용을 구성하였다.

이후 main 함수에서, pdf에 나와있었던 <getopt.h> <unistd.h> <stdlib.h> 함수를 include하고, main의 인수를 받아 argument s, E, b, t에 분배하는 과정을 switch문을 통해 구현할 수 있었다.

```c
int i, j;
CACHE = (cache_set*)malloc(sizeof(cache_set) * args.S);
for (i = 0; i < args.S; i++) {
    CACHE[i] = (cache_line*)malloc(sizeof(cache_line) * args.E);
    for (j = 0; j < args.E; j++) {
        CACHE[i][j].valid = 0;
        CACHE[i][j].tag = 0;
        CACHE[i][j].LRU = 0;
    }
}
```

```c
FILE* trace = fopen(trace_name, "r");
unsigned long long int address;
int size;

while (fscanf(trace, " %c %llx,%d", &command, &address, &size)!=EOF) {
    switch (command) {
    case 'L': accessCache(address); break;
    case 'S': accessCache(address); break;
    case 'M': accessCache(address); accessCache(address); break;
    default: break;
    }
    //만약 v 옵션이 주어지면 자세한 창을 PRINT하기
    if (verbosity != 0) {
        printf("%c %llx,%d %s\n", command, address, size, verbose);
    }
}
```

또한, 위에서 구현한 CACHE 이차원 배열을 각각 S와 E에 대하여 malloc을 통해 할당해 줄 수 있었고, trace 파일에 담겨 있는 값들을 command, address, size로 나누어서 파일 끝까지 읽어올 수 있었다. 이후 command에 따라 switch 문으로 cache에 access하는 함수를 불러 올 수 있었다 (accessCache). 이후 verbosity라는 v 옵션이 있다면 위에서 받은 command, address, size, verbose가 모두 출력되도록 설정해놓았다.

```
//HIT, MISS, EVICTION 된 NUM을 PRINT한다.
    printSummary(hit_num, miss_num, eviction_num);

//CACHE 동적 할당 해제
    for (i = 0; i < args.S; i++) {
        free(CACHE[i]);
    }
    free(CACHE);

//TRACE 파일 닫기
    fclose(trace);
    return 0;
}
```

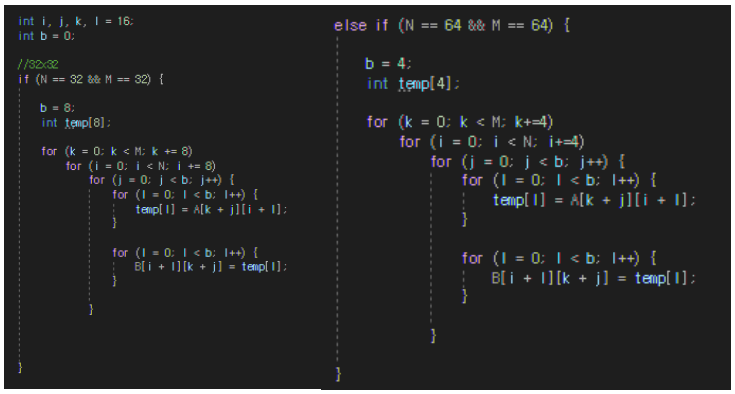
```
int i;
unsigned long long int set = (0x7fffffff >> (31 - args.s)) & (address >> args.b);
unsigned long long int tag = 0x7fffffff & (address >> args.s >> args.b);

cache_set CACHE_SET = CACHE[set];

int local_hit = 0;
int local_eviction = 0;


//hit
for (i = 0; i < args.E; i++) {
    if (CACHE_SET[i].tag == tag && CACHE_SET[i].valid == 1) {
        CACHE_SET[i].LRU = LRU_counter++;
        hit_num++;
        local_hit = 1;
        return;
    }
}

//miss
miss_num++;
```

위에서 가져온 hit, miss, eviction 개수를 프린트하는 함수를 호출하고, CACHE를 동적할당을 해제한다. accessCache 함수에 대해 자세히 알아보면, 0x7fffffff 함수는 01111..111의 32비트를 31-s 비트만큼 right shift시킨 값 (즉, 00000..111111 = mask)과 address를 b만큼 right shift 시킨값의 &비트 연산이다. 이를 통해 set 중간값을 구할 수 있다. tag값도 마찬가지로 mask & (address의 right shift of s, b)로 구할 수 있다. 이후 CACHE_SET에 CACHE의 SET INDEX값을 넣은 후, 그 SET의 block값 내에서 hit, miss, eviction 여부를 찾도록 하였다. 만약 tag가 동일하고 valid가 1인 block이 존재하면, LRU를 증가시키고 HIT시킨다. 만약 찾지 못했다면 MISS_num이 올라가도록 설정했다.

```
//LRU 만족하는 가장 최근에 사용되지 않았던 block 찾기
unsigned long long int eviction_LRU = ULONG_MAX;
unsigned int eviction_line = 0;

for (i = 0; i < args.E; ++i) {
    if (eviction_LRU > CACHE_SET[i].LRU) {
        eviction_line = i;
        eviction_LRU = CACHE_SET[i].LRU;
    }
}


//eviction
if (CACHE_SET[eviction_line].valid) {
    eviction_num++;
    local_eviction = 1;
}

CACHE_SET[eviction_line].valid = 1;
CACHE_SET[eviction_line].tag = tag;
CACHE_SET[eviction_line].LRU = LRU_counter++;

//v 옵션일 때 verbose 설정
if (command == 'L' || command == 'S') {

    if (local_hit) verbose = "hit";
    else if (local_eviction) verbose = "miss evction";
    else verbose = "miss";

}
else if (command == 'M') {

    if (local_hit) verbose = "hit hit";
    else if (local_eviction) verbose = "miss evction hit";
    else verbose = "miss hit";

}
```

eviction을 처리하기 위하여 eviction_LRU 값을 해당 범위에서 가장 큰 값으로 잡고, 특정 BLOCK의 LRU값이 더 작으면 그 값으로 eviction_LRU를 바꾸는 식으로, LRU에 들어가는 LRU_counter의 값은 계속 올라가므로 가장 낮은 LRU의 값을 가진 line이 가장 오래된 blocck일 것이라는 사실을 이용하여 그 block에 치환하고 valid, tag값을 변환하고, LRU 값도 업데이트 한다. 이후 V가 옵션일 때, hit와 eviction, v 여부를 따져서 설정할 수 있다.

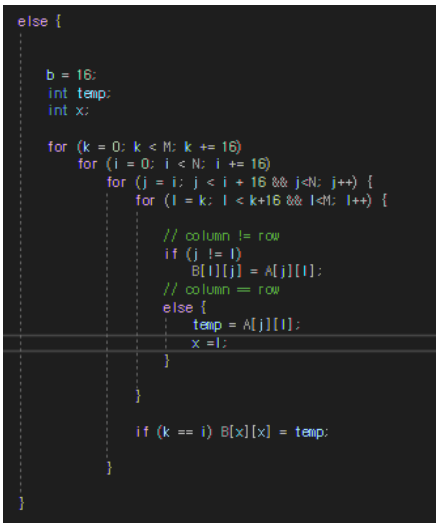
(2) PART B



```
int i, j, k, l = 16;
int b = 0;

//32x32
if (N == 32 && M == 32) {

    b = 8;
    int temp[8];

    for (k = 0; k < M; k += 8)
        for (i = 0; i < N; i += 8)
            for (j = 0; j < b; j++) {
                for (l = 0; l < b; l++) {
                    temp[l] = A[k + j][i + l];
                }

                for (l = 0; l < b; l++) {
                    B[i + l][k + j] = temp[l];
                }
            }

}
else if (N == 64 && M == 64) {

    b = 4;
    int temp[4];

    for (k = 0; k < M; k+=4)
        for (i = 0; i < N; i+=4)
            for (j = 0; j < b; j++) {
                for (l = 0; l < b; l++) {
                    temp[l] = A[k + j][i + l];
                }

                for (l = 0; l < b; l++) {
                    B[i + l][k + j] = temp[l];
                }

            }
}
```

32x32의 경우는 block size를 8로 설정해 blocking을 시도했다. row-major access를 위해 k -> i -> j -> l 접근을 통해 A[k+j][i+l] <-> B[i+l][k+j]를 8개 단위로 swap할 수 있었다.

64x64의 경우는 block size를 4로 설정해 32x32와 똑 같은 코드로 blocking을 시도했다. 32x32 코드와 다를 건 없었다. 다만 이경우에 miss rate가 생각보다 줄지 않아 만점이 나오진 않았다.



```
else {


    b = 16;
    int temp;
    int x;

    for (k = 0; k < M; k += 16)
        for (i = 0; i < N; i += 16)
            for (j = i; j < i + 16 && j<N; j++) {
                for (l = k; l < k+16 && l<M; l++) {

                    // column != row
                    if (j != l)
                        B[l][j] = A[j][l];
                    // column == row
                    else {
                        temp = A[j][l];
                        x =l;
                    }

                }

                if (k == i) B[x][x] = temp;

            }
}
```

61x67의 경우 block size를 16으로 설정해 blocking을 시도했다. 단, square matrix가 아니기에 범위 설정이 달라졌는데, 비슷한 코드를 구현해도 correctness가 1이 나오지 않아서, column이 row랑 같을때에는 swap을 하지 않는 방식으로 과정을 줄였더니 correctness 1을 도출할 수 있었다.

VI. 실행 결과

```
Running ./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
```

```
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         287
Trans perf 64x64           3.4         8        1699
Trans perf 61x67          10.0        10        1809
          Total points    48.4        53
```

V.결론 및 고찰

-Cache의 구조와 bit 역할들을 이해하고 shift 연산자 및 mask를 통해 구현할 수 있었다.

-hit / miss/ eviction이 일어나는 조건과 원리를 이해하고 구현할 수 있었다.

-blocking을 통해 miss rate을 줄일 수 있음을 이해할 수 있었고, locality를 활용하는 방법을 배웠다.