CSED211 LAB8&9 REPORT

20220778 표승현

Part a.

```
// 캐시 구조
typedef struct{
    int valid;
    int tag;
    int order;
}line;

line** cache;

// miss, hit, evict 수
int miss_count=0;
int hit_count=0;
int evict_count=0;

int recent_order = 0;  // 전역변수

// 커맨드 변수
int s=0;
int s_=0;
int E=0;
int b=0;
int b_=0;
int check_v=0;

//trace 정보
char* trace;

void caching(unsigned long long address);
```

캐시를 구현하기 위해 cache line 을 구조체로 구현한다. 유효비트와 태그 정보, 그리고 LRU eviction 에 활용한 order 를 포함한다. 이후 line** 타입의 이차원 배열을 선언하여 cache 를 구현한다. 다음은 각각 miss, hit, eviction 의 빈도를 계산하기 위한 전역변수들이 선언되어 있고, LRU 를 구현하기 위한 변수인 recent_order 가 선언되어 있다. 다음은 커맨드 변수를 저장하고 활용하기 위한 변수들이 선언되어 있고, 추가적으로 trace 의 파일 이름을 저장할 변수가 마지막으로 저장되어 있다.

caching 함수는 접근하고자 하는 주소를 매개변수로 하여 cache 에 주소를 저장하고 evict 하는 기능을 구현한다. 이 함수 내에서 hit, miss, eviction 빈도도 계산된다.

```
char opt;
while ((opt = getopt(argc, argv, "s:E:b:t: ")) != -1)
{
switch (opt)
{
case 'h':
    printf("Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>\n");
    return 0;
case 'v':
    check_v=1;
    break;
case 's':
    s = atoi(optarg);
    break;
case 'E':
    E = atoi(optarg);
    break;
case 'b':
    b = atoi(optarg);
    break;
case 't':
    trace = optarg;
    break;
default:
    return 0;
    }
}
```

getopt 함수를 이용해 커맨드 변수를 전달받는다. opt 값이 각각 h 이면 가이드를 출력하고, v 이면 이후 캐싱 현황을 출력한다. 이후 s, E, b, t 에는 각각 세트비트 수, 한 세트당 라인 수, 블록 비트 수, 태그비트를 저장한다.

```
//initialize cache
s_ = 1<<s;
cache = (line**)malloc(sizeof(line*) * s_);
for (int i = 0; i < s_; i++){
    cache[i] = (line*)malloc(sizeof(line) * E);
}

for(int i=0;i<s_;i++){
    for(int j=0;j<E;j++){
        cache[i][j].valid=0;
        cache[i][j].tag=0;
        cache[i][j].order=s_*E;
    }
}
```

malloc 을 활용하여 line 의 이차원 배열 cache 에 적절한 공간을 할당한다. 동시에 값 초기화도 진행하는데 order 에는 최소값 판단 알고리즘을 위해 가질 수 있는 가장 큰 값을 저장한다.

```
//read trace
FILE* trace_p= fopen(trace, "r");
char cmd;
unsigned long long address;
int size;

if(trace_p!=NULL){
while (fscanf(trace_p, " %c %llx %d", &cmd, &address, &size) != EOF)
  {
    switch(cmd){
        case 'I':
            continue;
        case 'L':
            caching(address);
            break;
        case 'S':
            caching(address);
            break;
        case 'M':
            caching(address);
            caching(address);
            break;
        default:
            break;
    }
  }
}
else{
    return 0;
}
fclose(trace_p);
```

trace 파일을 연다. 이후 trace 파일의 명령을 차례대로 읽으면서 캐싱 시뮬레이션을 수행한다. I 는 무시하고 L 과 S 의 경우는 주소 접근이 한 번 일어나기 때문에 캐싱 함수를 한 번 실행한다. 하지만 M 은 수정하는 명령이기 때문에 접근이 두 번 발생한다. 따라서 캐싱 함수를 두 번 실행한다. 캐싱 함수는 아래와 같은 방법으로 구현되어 있다.

```
int t_bit = address >> (s + b);
int s_index = (address >> b) & (s_ - 1);

recent_order ++; // Update the most recently used line
```

먼저 비트 연산을 통해 tag 비트와 set index 를 추출한다. 그리고 매 실행때마다 order 를 증가시켜 라인이 고유한 순서 값을 지니게 한다.

```
for (int i = 0; i < E; i++)
{
    if (cache[s_index][i].valid)
    {
        if (cache[s_index][i].tag == t_bit) // Hit
        {
            hit_count++;
            cache[s_index][i].order = recent_order;
            if (check_v == 1)
            {
                printf(" hit");
            }
            return;  // Return early on a hit
        }
    }
}

// Miss
miss_count++;

for (int i = 0; i < E; i++)
{
    if (!cache[s_index][i].valid)
    {
        // Set is not full, insert the data
        cache[s_index][i].valid = 1;
        cache[s_index][i].tag = t_bit;
        cache[s_index][i].order=recent_order;
        if (check_v == 1)
        {
            printf(" miss");
        }
        return;
    }
}
```

유효비트가 1 이고 태그비트가 일치하는 캐시 라인이 존재하는 경우 hit 이다. hit 빈도수와 해당 캐시라인의 order 를 업데이트하고 함수를 종료한다.

만약 반복문을 다 돌았는데 함수가 끝나지 않았다면 캐시에 목표 주소가 존재하지 않았다는 뜻이며 miss 에 해당한다. miss 빈도를 증가시키고 캐시에 주소를 저장할 공간이 있는지 탐색한다. 그러나 빈 공간이 없다면 eviction 을 수행해야 한다.

```
// Eviction
evict_count++;
int min_order=recent_order;
int position=0;
for (int i = 0; i < E; i++)
{
    if (cache[s_index][i].order<min_order)
    {
        min_order=cache[s_index][i].order;
        position=i;
    }
}
cache[s_index][position].tag = t_bit;  // Replace the least recently used line
cache[s_index][position].order=recent_order;

if (check_v == 1)
{
    printf(" miss eviction");
}
```

캐싱 함수를 실행할 때마다 recent oreder 를 증가시켰으므로 가장 마지막에 사용한 캐시 라인에는 가장 작은 order 값이 저장되어 있다. 따라서 이를 찾아 eviction 해줌으로써 LRU 를 구현한다.

Part b.

```
int i, j, k, l;
int v0, v1, v2, v3, v4, v5, v6, v7;
v0 = 0;
v1=0;
```

다음과 같이 활용할 변수를 선언 및 초기화했다.

1) 32*32 matrix

```
// 32x32 matrix
if(M==32&&N==32){
    for(i=0;i<M;i += 8){
        for(j=0;j<N;j += 8){
            for(k=0;k<8;k++){
                for(l=0;l<8;l++){
                    if(j+l==i+k) {
                        v0 = A[i+k][i+k]; // v0: 대각성분 값
                        v2 = i+k; // v2: 대각성분 좌표
                        v1=1;
                        continue;
                    };
                    B[j+l][i+k]=A[i+k][j+l];
                }

                if(v1==1){
                    v1=0;
                    B[v2][v2] = v0;
                }
            }
        }
    }
}
```

b 의 비트 수가 5 이기 때문에 2^5=32 바이트의 데이터를 저장할 수 있다. int 는 4 바이트이므로 하나의 캐시라인에는 총 8 개의 int 가 저장될 수 있다. 이에 따라 8*8 블록을 통해 최적화를 진행해본다. 하지만 블록을 이용하는 것만으로는 만점에 해당하는 수치를 얻을 수 없었다.

행과 열의 값이 같은 값을 전치하는 경우 eviction 이 일어난다. 따라서 v 값에 일시적으로 대각성분을 저장한 후 따로 이동하는 과정을 통해 이를 피한다.

2) 64*64 matrix

```
else if (M==64 && N==64) // 64x64 matrix
{
    i=0;
    j=0;
    while (i < 64)
    {
    j=0;
        while(j < 64)
        {
            for (k = 0; k < 4; k+=1)
            {
                v0 = A[i+k][j+0];
                v1 = A[i+k][j+1];
                v2 = A[i+k][j+2];
                v3 = A[i+k][j+3];
                v4 = A[i+k][j+4];
                v5 = A[i+k][j+5];
                v6 = A[i+k][j+6];
                v7 = A[i+k][j+7];

                B[j+0][i+k] = v0;
                B[j+1][i+k] = v1;
                B[j+2][i+k] = v2;
                B[j+3][i+k] = v3;
                B[j+3][i+k+4] = v4;
                B[j+2][i+k+4] = v5;
                B[j+1][i+k+4] = v6;
                B[j+0][i+k+4] = v7;
            }
            for (k = 0; k < 4; k+=1)
            {
                v0 = A[i+4][j-k+3];
                v1 = A[i+5][j-k+3];
                v2 = A[i+6][j-k+3];
                v3 = A[i+7][j-k+3];
                v4 = A[i+4][j+k+4];
                v5 = A[i+5][j+k+4];
                v6 = A[i+6][j+k+4];
                v7 = A[i+7][j+k+4];
```

```
            for (k = 0; k < 4; k+=1)
            {
                v0 = A[i+4][j-k+3];
                v1 = A[i+5][j-k+3];
                v2 = A[i+6][j-k+3];
                v3 = A[i+7][j-k+3];
                v4 = A[i+4][j+k+4];
                v5 = A[i+5][j+k+4];
                v6 = A[i+6][j+k+4];
                v7 = A[i+7][j+k+4];
                for(l=0;l<4;l++){
                    B[j+k+4][i+1] = B[j-k+3][i+4+1];
                }
                B[j-k+3][i+4]=v0;
                B[j-k+3][i+5]=v1;
                B[j-k+3][i+6]=v2;
                B[j-k+3][i+7]=v3;
                B[j+k+4][i+4]=v4;
                B[j+k+4][i+5]=v5;
                B[j+k+4][i+6]=v6;
                B[j+k+4][i+7]=v7;
            }
        j=j+8;
        }
    i=i+8;
    }
}
```

64*64 matrix 의 경우에도 8*8 블록을 이용하여 최적화한다. 다만 int 가 4 바이트이고 64*64 matrix 에서 한 줄에는 64 개의 정수가 들어가므로 64 행렬의 한 줄을 저장하는데 필요한 바이트는 4*64=2^8 바이트, 필요한 set 의 수는 2^8 / 32 = 2^3 = 8 개이다. 활용할 수 있는 set 의 수는 총 32 개이므로 블록의 입장에서 4 줄 주기로 같은 set 를 공유하게 된다는 뜻이다. 이 때문에 8*8 블록을 이용하여 최적화를 수행하되 블록을 다시 위 아래로 나누어 set 를 독립적으로 활용해야 한다.

첫 번째는 A 의 위 절반을 옮기는 과정이다. 그러나 이를 수행하면 B 블록의 오른쪽 위 16 개 위치에 잘못된 값이 저장된다. 따라서 A 의 나머지 절반을 옮기기 전, A 의 잘못 옮겨진 데이터들을 제자리로 옮겨주는 작업을 수행한다.

3) 61*67 matrix

```
else if(M==61 && N==67){ // 61x67 matrix
    for(i=0;i<N;i += 16){
        for(j=0;j<M;j += 16){
            for(k=0;k+i<N&&k<16;k++){
                for(l=0;l+j<M&&l<16;l++){
                if(j+l==i+k) {
                        v0 = A[i+k][i+k]; // v0: 대각성분 값
                        v2 = i+k; // v2: 대각성분 좌표
                        v1=1;
                        continue;
                };
                    B[j+l][i+k]=A[i+k][j+l];
                }

                if(v1==1){
                    v1=0;
                    B[v2][v2] = v0;
                }
            }
        }
    }
}
```

1)에서 했던 것처럼 블록을 이용하여 최적화를 진행한다. 그러나 이번에는 8 단위 블록보다 16 단위 블록을 수행한 결과가 더 좋았기 때문에 16 을 단위로 하여 최적화를 진행했다. 이번에는 정사각형 행렬이 아니라 loop 속에서 16 의 배수가 아닌 좌표가 걸리는 부분에서는 추가적인 조건문을 통해 처리해주었다.