Cache Lab

20230024 문요준

[Part A]

<code>

```
typedef struct {
    int is_valid;    // 유효성 여부
    int tag;         // 태그
    int last_used;   // 최근 사용된 순서 (LRU 관리)
} CacheLine;

CacheLine** cache_system; // 캐시 2차원 배열

// 전역 변수
int misses        = 0;  // 캐시 미스 카운트
int hits          = 0;  // 캐시 히트 카운트
int evictions     = 0;  // Eviction 카운트

int current_order = 0;  // LRU를 위한 최근 접근 순서

// 커맨드 옵션 변수 초기화
int set_bits      = 0;  // s
int num_sets      = 0;  // S
int assoc         = 0;  // E
int block_bits    = 0;  // b
int verbose_flag  = 0;

char* trace_file_path;

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

구조체로 Cache Line을 구현한다. 구조체는 is_valid, tag, last_used로 구성되어있고 각각 해당 Line이 유효한 데이터를 포함하고 있는지, Line의 tag, 마지막으로 사용된 순서를 저장한다. 그리고 캐시 전체를 나타내기 위해서 2차원 배열로 캐시를 구현한다. 다음으로, 전체 Miss, Hit, Eviction의 횟수를 세기 위해서 각각의 횟수를 저장하는 변수를 전역변수로 선언하고, 명령어 옵션을 저장하기 위한 변수들을 선언한다. trace_file_path는 Trace파일의 경로를 저장하는 문자열 변수를 선언한 것이며, void로 선언된 cache_simulator 함수는 캐시의 동작을 구현하기 위한 함수를 선언한 것이다.

<code>

```
char option;

// 명령어에서 파싱하여 커맨드 옵션 변수에 값 대입
while ((option = getopt(argc, argv, "s:E:b:t:hv")) != -1)
{
    switch (option)
    {
        case 'h':
            printf("Usage: ./csim [-hv] -s <s> -E <E> -b <b> -t <tracefile>\n");
            return 0;
        case 'v':
            verbose_flag = 1; // Verbose 출력 ON
            break;
        case 's':
            set_bits = atoi(optarg); // optarg(str) -> optarg(int)
            break;
        case 'E':
            assoc = atoi(optarg);
            break;
        case 'b':
            block_bits = atoi(optarg);
            break;
        case 't':
            trace_file_path = optarg;
            break;
        default:
            return 0;
    }
}

num_sets = 1 << set_bits; // 2^s = 세트 개수
```

main 함수 내에서 getopt 함수를 사용해 명령줄 인자를 순서대로 읽어들여 명령어를 입력받아 처리한다. "s:E:b:t:hv" (옵션 정의 문자열)을 이용해서 추가 값이 필요한 옵션(s, E, b, t)의 경우 값(숫자 또는 주소)를 option에 저장하고 h와 v의 경우 이 문자 자체를 option에 저장한다. 그리고 switch문을 사용해 option에 들어간 문자에 따라서 코드를 실행한다. h의 경우 도움말을 출력, v의 경우 전역변수 verbose_flag에 1을 저장해 출력 상태로 바꾸고, 나머지 문자에는 명령어와 함께 캐시 설정을 위해 입력받는 값(숫자 또는 주소)을 앞에서 선언했던 변수들에 저장한다.

<code>

```
num_sets = 1 << set_bits; // 2^s = 세트 개수

// 캐시 전체(2차원 배열) 동적 할당 – 캐시의 세트 수(num_sets)만큼 메모리를 할당하고 각 세트를 가리키는 포인터를 저장할 배열을 생성
// Ex) 각 세트당 assoc(E)개의 캐시 라인이 있고 캐시 전체에 num_sets만큼의 세트가 있는 경우, CacheLine*은 E개의 CacheLine 배열을 가리키고, CacheLine**은 S개의 포인터 배열을 가리킴
cache_system = (CacheLine**)malloc(num_sets * sizeof(CacheLine*));

// 캐시 전체 내에서 세트 순회
for (int i = 0; i < num_sets; i++)
{
    cache_system[i] = (CacheLine*)malloc(assoc * sizeof(CacheLine)); // 동적 메모리 할당으로 i 세트에 포함할 assoc(E) 개의 라인을 저장
    // 세트 내부의 라인 순회하면서 초기화
    for (int j = 0; j < assoc; j++)
    {
        cache_system[i][j].is_valid  = 0;
        cache_system[i][j].tag       = 0;
        cache_system[i][j].last_used = -1;
    }
}
```

입력받는 s의 값을 사용해 우측 비트 시프트로 S=2^s를 계산, 전체 세트의 개수를 계산한다. 그리고 전체 캐시를 선언하고 초기화하는데, malloc을 사용해 전체 세트의 용량만큼 동적 메모리 할당을 수행한 다음 캐시 내에서 세트를 순회하면서 세트 당 라인의 개수(E, 여기선 assoc) 메모리만큼 각각의 배열에 할당하고 라인 구조체의 값들을 초기화한다.

<code>

```
void cache_simulator(unsigned long long address)
{
    // 메모리 주소 형태: [ Tag | Set Index | Block Offset ]
    int tag       = address >> (set_bits + block_bits);       // (set_bits + block_bits) 만큼 우측 시프트해서 태그 부분 외 하위 비트를 제거, 남은 상위 비트는 태그
    int set_index = (address >> block_bits) & (num_sets - 1); // (block_bits)만큼 우측 시프트해서 Block Offset 제거, (num_sets - 1)를 세트 인덱스를 추출하는데 필요한 비트마스크로 사용, set_bits만 남김

    current_order++;                                          // 최근 접근 순서 증가
```

다음으로 캐시의 동작을 구현하는 cache_simulator 함수이다. 주소를 입력받은 다음 tag와 set_index를 구한다. tag는 (set_bits + block_bits) 만큼의 비트 수 만큼 우측 시프트해서 태그 부분 외 하위 비트를 제거함으로써 구하고, set_index는 (block_bits)만큼 우측 시프트서 Block Offset을 제거한 뒤, (num_sets - 1)를 세트 인덱스를 추출하는데 필요한 bit mask로 사용해서 set_bits만 남긴다. 그리고 LRU를 위해서 current_order(최근 접근 순서) 값을 업데이트(1 증가)한다.

<code>

```
// 캐시 히트 검사, set_index 세트 내에서 라인 순회
for (int i = 0; i < assoc; i++)
{
    if (cache_system[set_index][i].is_valid && cache_system[set_index][i].tag == tag) // 유효비트가 1이고 address의 tag와 라인의 tag가 일치할 때 --> hit!
    {
        hits++;                                                                       // 캐시 히트 카운트 증가
        cache_system[set_index][i].last_used = current_order;                         // LRU를 위한 가장 최근에 사용된 순서를 기록

        if (verbose_flag) printf(" hit");                                             // 캐시 히트 시 결과 출력
        return;
    }
}
```

캐시 히트 처리 부분이다. 앞에서 구한 set_index로 캐시에서 해당 세트의 라인을 순회하면서 hit를 검사한다. 유효비트가 1이고, 입력받은 주소의 tag와 캐시 내 라인의 tag가 일치할 때 hit이 발생한다. hit이 발생하면 hits(히트 카운트를 위한 전역변수)를 1 증가시키고 해당 라인의 최근 사용 순서를 업데이트한다. 그리고 verbose_flag가 1로 설정되어 있을 때 결과를 출력한다.

<code>

```
// 캐시 미스 처리
misses++;

// set_index 세트 내에서 라인 순회
for (int i = 0; i < assoc; i++)
{
    if (cache_system[set_index][i].is_valid == 0)                 // 유효비트가 0인 경우 (라인이 비어있을 경우)
    {
        cache_system[set_index][i].is_valid  = 1;                 // 해당 라인을 유효한 상태로 변환
        cache_system[set_index][i].tag       = tag;               // 새 데이터의 태그 저장
        cache_system[set_index][i].last_used = current_order;     // LRU를 위한 가장 최근에 사용된 순서를 기록

        if (verbose_flag) printf(" miss");                        // 캐시 미스 시 결과 출력
        return;
    }
}
```

캐시 미스 처리 부분이다. 상단에서 히트가 발생하지 않으면 miss이므로 misses(미스 카운트를 위한 전역변수)를 1 증가시키고 set_index에 해당하는 세트의 라인을 순회한다. 만약 라인이 비어있으면(유효비트가 0이면) 해당 라인을 유효한 상태로 바꾸고, 새 데이터의 태그를 저장, 최근 사용 순서를 업데이트한다. 그리고 verbose_flag가 1로 설정되어 있을 때 결과를 출력한다.

<code>

```
// Eviction 처리, 캐시의 특정 세트가 다 차있을 때(캐시 미스 처리 for문에서 모든 is_valid가 1일 때) 데이터 교체를 위한 LRU 구현
evictions++;

int lru_index = 0;                                      // LRU(가장 오래 사용되지 않은) 라인의 인덱스를 저장하는 변수, 세트의 첫 번째 라인으로 초기화
int min_order = cache_system[set_index][0].last_used;   // 각 캐시 라인의 last_used 값을 비교해 LRU 데이터 판단을 위한 변수, 첫 번째 라인의 last_used 값으로 초기화

// set_index 세트 내에서 라인 순회
for (int i = 1; i < assoc; i++)
{
    if (cache_system[set_index][i].last_used < min_order) // 각 라인의 last_used 값을 min_order와 비교한 뒤 더 작으면
    {
        min_order = cache_system[set_index][i].last_used; // 현재 가장 오래된 라인(LRU, last_used가 가장 작은 라인)으로 min_order 값을 업데이트
        lru_index = i;                                    // lru_index 값을 LRU 데이터의 인덱스로 업데이트
    }
}

// 교체
cache_system[set_index][lru_index].tag       = tag;            // LRU 데이터의 tag를 새로운 데이터의 tag로 업데이트
cache_system[set_index][lru_index].last_used = current_order;  // 최근 사용 순서 갱신
if (verbose_flag) printf(" miss eviction");                    // miss eviction 시 결과 출력
```

캐시 교체 처리 부분이다. 상단에서 set_index 세트의 모든 라인이 전부 차있는 경우 가장 오래 사용되지 않은 라인에서 교체하기 위해 실행되고, evictions(교체 카운트를 위한 전역변수)를 1 증가시키고 set_index에 해당하는 세트의 라인을 순회한다. LRU(Last Recently Used) 규칙에 따라 교체할 라인의 인덱스를 저장할 변수 lru_index를 0으로 초기화하고, 교체할 라인의 last_used를 저장할 변수 min_order를 첫 번째 라인의 last_used 값으로 초기화한다. 라인을 순회하면서 라인의 last_used가 min_order보다 작을 경우 min_order(가장 오래된 순서)를 해당 라인의 last_used로 갱신하고 lru_dex를 해당 라인의 인덱스로 갱신한다. 모든 순회가 끝난 후 갱신된 lru_index에 해당하는 라인의 값(tag, last_used)를 새로운 값으로 교체한다. 그리고 verbose_flag가 1로 설정되어있을 때 결과를 출력한다.

<code>

```
// Trace 파일 열기
FILE* trace_file = fopen(trace_file_path, "r");
if (!trace_file) {
    printf("File is not valid");
    return 0;
}

char operation;              // Trace 파일의 명령어를 저장할 변수
unsigned long long address;  // 메모리 주소를 저장할 변수
int size;                    // 메모리 접근 시 데이터 크기(byte)를 저장할 변수

// Trace 파일에서 한 줄씩 데이터를 읽음 --> %c: 명령어 / %llx: 16진수 메모리 주소 / %d: 데이터 크기 / EOF(파일 끝)에 도달할 때까지 루프
while (fscanf(trace_file, " %c %llx,%d", &operation, &address, &size) != EOF)
{
    switch (operation)
    {

        case 'I':                    // 명령어를 메모리에서 읽기 -> 캐시에 영향 X
            break;
        case 'L':                    // Load, 메모리 읽기
            cache_simulator(address);
            break;
        case 'S':                    // Store, 메모리 쓰기
            cache_simulator(address);
            break;
        case 'M':                    // 읽기와 쓰기 둘 다 포함 -> simulator 두 번 호출
            cache_simulator(address);
            cache_simulator(address);
            break;
    }
}
fclose(trace_file); // Trace 파일 닫기
```

다시 main 함수로 돌아와서, trace_file_path를 이용해 Trace File의 내용을 읽은 뒤, 선언된 operation, address, size에 각각 Trace 파일의 명령어, 메모리 주소, 데이터 크기를 저장한다. 그리고 case문을 사용해 각각의 operation에 해당하는 동작을 실행한다.

<code>

```
// 메모리 해제
for (int i = 0; i < num_sets; i++)
{
    free(cache_system[i]);
}
free(cache_system);

// 결과 출력
printSummary(hits, misses, evictions);

return 0;
```

cache_simulator 함수의 동작이 모두 종료되면 캐시의 메모리를 전부 해제하고 결과를 출력한 뒤 main 함수를 종료한다.

```
[yojun313@programming2 5_CacheLab]$ python2 driver.py
Part A: Testing cache simulator
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]

Part B에서는 (s=5, E=1, b=5)인 캐시를 사용하여 miss 개수의 제한 범위를 지키며 세 가지 행렬에 대해 transpose 연산을 수행해야한다. miss를 최소화하는 방향으로 최적화하기 위해서 blocking을 사용할 것이다.

<code>

```
void transpose_MxN(int M, int N, int A[N][M], int B[M][N], int block_size)
{
    int i, j;                                   // i, j: 현재 처리 중인 블록 시작점,
    int row, col;                               // row, col: 블록 내 인덱스
    int diag_value, diag_index;                 // 대각선 원소 처리용 변수

    // i, j 순회 --> 한 번의 반복마다 block_size x block_size 블록 처리
    for (i = 0; i < N; i += block_size)
    {
        for (j = 0; j < M; j += block_size)
        {
            // row, col 순회 --> 블록 내부 데이터 순회
            for (row = 0; row < block_size && i + row < N; row++)
            {
                for (col = 0; col < block_size && j + col < M; col++)
                {
                    if (i + row == j + col)                         // 대각선 원소일 때 (열과 행 인덱스가 같을 때)
                    {                                               // 대각선 원소는 전치 시 동일한 위치에 저장됨 --> 캐시 미스 발생 가능
                        diag_value = A[i + row][j + col];           // diag_value에 대각선 원소의 값 저장
                        diag_index = i + row;                       // diag_index에 대각선 원소의 위치 저장
                    }
                    else
                    {
                        B[j + col][i + row] = A[i + row][j + col];  // 전치된 위치(B 행렬)로 값 복사
                    }
                }

                if (i + row == j + row)                             // 블록 행 내에서 열 순회가 끝나고 대각선 원소가 존재하면
                {
                    B[diag_index][diag_index] = diag_value;         // 해당 위치에 대각선 원소의 값 저장
                }
            }
        }
    }
}
```

transpose_MxN의 이름으로 M, N (행렬의 가로 세로 크기), block_size를 입력받는 함수를 정의했다. 32x32 행렬에 대해서 transpose 실행 결과, blocking만으로는 과제에서 요구하는 misses의 개수를 만족하지 않음을 알 수 있었다. eviction이 일어나는 지점을 더 찾아본 결과, 열과 행 인덱스가 같은 값 (전체 행렬에서 대각선에 위치한 값)을 transpose해서 값을 저장할 때 eviction이 일어남을 알 수 있었다. 즉 A[i][i]의 값을 읽는 순간 B[i][i]에 저장한다고 하면, 같은 캐시 공간을 참조해 eviction이 발생할 수 있는 것이다. 따라서 if문으로 해당 원소가 대각선 원소 일 때 값과 인덱스를 로컬 변수에 저장해두고 블록 행 내에서 열 순회가 끝나면 B 행렬에 값을 저장하는 방식으로 로직을 구현했다.

한 라인의 크기는 2^5=32비트이고 한 라인 안에 8개의 정수가 들어갈 수 있으므로 32x32 행렬에 대해서 block_size를 8로 설정하고 transpose_MxN을 실행했으며 misses가 기준치 범위 안으로 들어왔다.

마찬가지로 (M, N) = (64, 64)에 대해서 transponse_MxN 함수를 실행한 결과 misses가 기준치를 많이 초과함을 알 수 있었다. 64x64 행렬에서 한 행을 저장하는데 4*64 = 256바이트가 필요하며 한 세트(direct cache이므로 라인의 크기 = 세트의 크기)의 크기가 32바이트이므로 총 8개의 세트가 필요하다. 총 세트의 수는 2^s = 2^5 = 32개이므로, 4줄 주기로 같은 세트를 공유하게 된다. 즉 블록으로 쪼갰을 때 블록의 상단 (4x8)과 하단 (4x8)이 같은 set를 공유하기 때문에, 더욱 최적화(캐시 충돌 방지)를 위해 상단부 처리가 끝난 후 하단부 처리를 수행하는 방법으로 transpose 과정을 수행할 것이다. 64x64 행렬에 대한 함수를 따로 transpose_64x64로 정의했다.

<code>

```
void transpose_64x64(int A[64][64], int B[64][64])
{
    int i, j;                                                   // 현재 처리 중인 블록의 행 및 열 시작점
    int row, col;                                               // 블록 내 행과 열 인덱스
    int temp0, temp1, temp2, temp3, temp4, temp5, temp6, temp7; // 임시 저장 변수

    // i, j 순회 --> 한 번의 반복마다 8x8 블록 처리
    for (i = 0; i < 64; i += 8)
    {
        for (j = 0; j < 64; j += 8)
        {

            // 상단 4x8 영역 처리
            for (row = 0; row < 4; row++)
            {
                // 행렬 A에서 현재 블록의 상단 4행 데이터를 tmp 배열에 저장
                // tmp[0~3]: A 블록의 좌상단(4x4) 데이터
                // tmp[4~7]: A 블록의 우상단(4x4) 데이터
                temp0 = A[i + row][j + 0];
                temp1 = A[i + row][j + 1];
                temp2 = A[i + row][j + 2];
                temp3 = A[i + row][j + 3];
                temp4 = A[i + row][j + 4];
                temp5 = A[i + row][j + 5];
                temp6 = A[i + row][j + 6];
                temp7 = A[i + row][j + 7];

                // tmp 배열의 데이터를 행렬 B에 전치된 위치로 복사
                B[j + 0][i + row] = temp0;                      // A 블록의 좌상단(4x4) 데이터를 전치해서 B 블록의 좌상단에 저장 --> A의 좌상단(4x4) 부분 전치 완료
                B[j + 1][i + row] = temp1;
                B[j + 2][i + row] = temp2;
                B[j + 3][i + row] = temp3;

                B[j + 3][i + row + 4] = temp4;                  // A 블록의 우상단(4x4) 데이터를 전치해서 B 블록의 우상단에 저장
                B[j + 2][i + row + 4] = temp5;
                B[j + 1][i + row + 4] = temp6;
                B[j + 0][i + row + 4] = temp7;
            }
```

```
            // 하단 4x8 영역 처리
            for (col = 0; col < 4; col++)
            {
                // A 블록의 하단 4x8 영역 데이터를 tmp 배열에 저장
                // temp0 ~ temp3: A 블록의 좌하단(4x4) 데이터
                // temp4 ~ temp7: A 블록의 우하단(4x4) 데이터
                temp0 = A[i + 4][j + 3 - col];
                temp1 = A[i + 5][j + 3 - col];
                temp2 = A[i + 6][j + 3 - col];
                temp3 = A[i + 7][j + 3 - col];
                temp4 = A[i + 4][j + 4 + col];
                temp5 = A[i + 5][j + 4 + col];
                temp6 = A[i + 6][j + 4 + col];
                temp7 = A[i + 7][j + 4 + col];

                // B 블록의 우상단(4x4) 영역 데이터를 좌하단(4x4) 영역으로 재배치
                // B[j + 4 + col][i + row] = B[j + 3 - col][i + 4 + row]
                for (row = 0; row < 4; row++) {
                    B[j + 4 + col][i + row] = B[j + 3 - col][i + 4 + row];
                }

                // temp0 ~ temp3에 저장된 A 블록의 좌하단(4x4) 데이터를
                // B 블록의 우상단(4x4) 영역에 전치된 형태로 복사
                B[j + 3 - col][i + 4] = temp0;
                B[j + 3 - col][i + 5] = temp1;
                B[j + 3 - col][i + 6] = temp2;
                B[j + 3 - col][i + 7] = temp3;

                // temp4 ~ temp7에 저장된 A 블록의 우하단(4x4) 데이터를
                // B 블록의 우하단(4x4) 영역에 전치된 형태로 복사
                B[j + 4 + col][i + 4] = temp4;
                B[j + 4 + col][i + 5] = temp5;
                B[j + 4 + col][i + 6] = temp6;
                B[j + 4 + col][i + 7] = temp7;
            }
        }
    }
}
```

블록의 사이즈를 8x8로 설정하고 블록 내에서 4x4 사이즈로 4개로 쪼개 transpose 작업을 수행한다. 우선 A 블록의 상단 4x8 영역을 처리한다. temp 지역 변수들을 선언하고 temp0~3에 A 블록의 좌상단 데이터를, temp4~7에 우상단 데이터를 담는다. 그리고 저장했던 변수를 B 행렬에 복사함으로써 A 블록의 좌상단 데이터는 전치해서 B 블록의 좌상단에 담는다. A 블록의 우상단 데이터는 전치해서 임시로 우상단에 담는데, 나중에 최종적인 transpose를 위해 B 블록의 좌하단으로 위치 이동을 해야하므로 임시로 담아둔다.

다음으로 A 블록의 하단 4x8 영역을 처리한다. temp0~3에 A블록의 좌하단 데이터를, temp4~7에 우하단 데이터를 담는다. 그리고 임시로 저장해두었던 B 블록의 우상단 데이터를 좌하단 영역으로 재배치해 A 블록의 우상단 영역 transpose를 완료한다. 마지막으로 저장했던 변수를 B 행렬에 복사함으로써 A 블록의 좌하단 데이터는 전치해서 B 블록의 우상단에, A 블록의 우하단 데이터는 전치해서 B 블록의 우하단에 저장함으로써 transpose를 마무리한다.

61x67 행렬에 대해서는 block_size를 늘려 transpose_MxN 함수를 실행해 transpose를 수행한다. 기존처럼 block_size를 8로 설정하고 함수를 실행했을 때 misses가 기준치를 넘는다는 것을 알 수 있었다. block_size가 기존(64x64)처럼 8일 때 블록의 크기는 256바이트이므로 8개의 캐시 세트를 차지하게 된다. 이때 행렬의 61줄과 67줄이 캐시 세트 매핑에서 불균형을 일으켜서 한 번의 블록 처리가 끝날 때마다 세트 충돌이 발생할 확률이 증가하게 되는 것이다. 따라서 block_size를 16으로 블록의 크기를 1024바이트로 설정, 캐시 세트(32바이트) 32개를 모두 사용하도록 설정했다. 한 번에 더 큰 블록 단위로 데이터를 처리하기 때문에 블록 내부의 데이터 재사용률이 증가, 행과 열 비대칭성이 block_size가 8일 때보다 덜 영향을 끼치게 되어 misses를 기준치 범위 안에 들도록 했다.

driver.py로 작성한 코드파일을 검사한 결과이다.

```
Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           8.0         8        1243
Trans perf 61x67          10.0        10        1985
          Total points    53.0        53
```