Cache Lab: Understanding Cache Memories Report

20220127 임유진

1. 개요

이번 Lab 은 Cache 와 같은 동작을 할 수 있는 Cache simulator 를 구현하는 Part A 와 Cache 메모리를 최적으로 활용하여 matrix 를 transpose 하는 함수를 작성해보는 Part B 으로 이루어져 있으며, 이를 통해 Cache 메모리의 동작 방식을 이해하는 것을 목표로 한다.

2. 이론적 배경

- Memory Hierarchy

현대 컴퓨터 시스템에서는 상위 레벨로 갈수 빠르지만 작고 비싼 저장장치, 하위 레벨로 갈수록 느리지만 크고 싼 저장 장치로 이루어진 메모리 계층 구조를 통해 메모리 시스템을 조직한다.

메모리 계층 구조에서 Cache 는 보다 느리고 큰 저장장치에 저장된 데이터에 대해 준비 영역으로 사용되는 보다 작고 빠른 저장장치를 의미하며, 메모리 계층 구조에서 레벨 k 에 있는 보다 빠른 저장장치는 레벨 k+1 에 있는 보다 느린 저장장치를 위한 Cache 서비스를 제공한다.

- Cache Memory

Cache Memory 는 CPU 와 메인 메모리의 성능 격차가 커짐에 따라 메모리 계층구조에서 CPU 와 메인 메모리 사이에 위치하게 된 저장장치이며, 현대의 많은 시스템에서는 CPU 와 메인 메모리 사이에 L1, L2, L3 캐시를 위치시킨다.

메모리 주소가 m 비트로 이루어진 컴퓨터 시스템에 대한 Cache 는 E 개의 Cache line 으로 이루어진 집합 S($=2^s$)개로 이루어져 있으며, 각 line 은 B($=2^b$) 바이트의 데이터 블록과 valid bit, 메모리 주소의 일부분으로 Cache 의 line matching 에 이용되는 tag bit 로 이루어져 있다. 이때 Cache 의 용량은 tag bit 와 valid bit 는 포함하지 않아 C = S * E * B 로 나타내어진다.

- Cache Hit/Miss/Eviction

프로그램이 Block B 에 위치하는 데이터를 필요로 하는 상황에서, Cache 에 Block B 가 저장되어 있는 경우를 Hit, 저장되어 있지 않은 경우를 Miss, Cache Miss 가 일어나 프로그램이 요청한 Block 을 집합의 line 중 하나에 저장해야 하는 상황에서 집합이 유효한 Cache line 으로 꽉 차 있어 기존 line 중 하나가 축출되어야 하는 경우가 Eviction 이 된다.

3. 풀이 과정

Part A: Writing a Cache Simulator

- 구조체 선언

```
typedef struct line {
    char valid;
    int count;
    unsigned long tag;
} Line;
```

Cache 를 구현하기 앞서 Cache 의 line 을 나타내는 구조체를 선언하고, typedef 를 통해 Line 이라는 별칭을 붙여주었다. 본래 Cache 의 line 은 1 valid bits, t tag bits, B cache blocks 으로 이루어져 있지만, 이번 Lab 에서는 data 까지는 다루지 않고 line matching 만으로 충분하기에 Line 구조체에 데이터를 저장하는 블록은 할당해주지 않았다. 또한 이번 Lab 에서는 Cache 의 block 을 대체해야 할 때 Cache Replacement Policy 로 가장 오랫동안 참조되지 않은 Cache line 을 교체하는 LRU(Least Recently Used) 방식을 사용하기 때문에, 최근에 접근한 line 을 추적할 수 있도록 counter 의 사용이 필요하다. 따라서 Line 구조체에 해당 Line 이 접근된 시기를 알 수 있게 하는 count 변수를 추가해주었다. 또한 이때 memory alignment 를 고려해 변수를 위와 같은 순서로 위치시켰고, 결과적으로 위와 같은 구조체를 선언해주었다.

- 변수 선언

사용한 변수와 그 의미는 다음과 같다.

변수	의미
int hit_count, miss_count, eviction_count	총 hit, miss, eviction 의 횟수
int count	LRU 를 위한 count
int i, j	For 문의 index
int s, E, b	인자로 받은 set bit 의 수, 한 set 당 line 의 수, block offset bit 의 수
int opt	command line argument 를 받기 위해 사용하는 getopt 함수의 반환값
int data_size	operation 에 의해 접근되는 바이트의 수
int set_num	총 set 의 개수
char* trace_file	인자로 받은 파일의 이름
FILE* fp	파일 입출력을 위한 파일 포인터

char op	파일로부터 읽은 operation
unsigned long mask	주소로부터 tag, set index, block offset 을 얻기 위해 사용하는 mask
unsigned long address	파일로부터 읽은, 프로그램이 접근하고자 하는 virtual memory address
unsigned long tag, set, block_offset	주소로부터 얻은 tag, set index, block offset
Line** cache_line	cache 메모리를 나타내는 2 차원 배열을 가리키는 포인터

- Command line argument parsing

```
while((opt = getopt(argc, argv, "hvs:E:b:t:")) != -1) {
    switch(opt) {
        case 'h':
            break;
        case 'v':
            break;
        case 's':
            s = string_to_num(optarg);
            break;
        case 'E':
            E = string_to_num(optarg);
            break;
        case 'b':
            b = string_to_num(optarg);
            break;
        case 't':
            trace_file = optarg;
            break;
    }
}
```

Cache simulator 에서는 command line argument 를 통해 임의의 s, E, b 를 입력 받아 Cache 를 구성하게 된다. 이를 위해 main 함수에 프로그램이 실행될 때 전달되는 인자의 개수를 저장하는 int argc 와 프로그램에 전달한 인자가 차례대로 저장되는 char *argv[]를 인자로 설정해주었다. 또한 이렇게 받은 command line option 을 parsing 하는 것이 필요하므로 getopt.h 헤더파일을 포함하여 getopt 함수와 optarg 를 이용하였다.

Cache simulator 에서는 -h, -v, -s, -E, -b, -f 의 옵션이 존재하며, -s, -E, -b, -f 의 경우 옵션 뒤에 파라미터 값이 함께 오기 때문에 option string 에서 콜론을 함께 써주어야 한다. 결과적으로 getopt 함수의 option string 은 “hvs:E:b:t:”이 된다. getopt.h 의 헤더파일에

선언되어 있는 전역 변수에는 -s, -E, -b, -f 와 같이 옵션 뒤에 파라미터 값을 가지는 옵션의 경우 그 파라미터 값을 문자열로 저장하는 optarg 라는 변수가 존재한다. 즉, 이를 이용해 각 옵션의 파라미터 값을 받아와야 한다. 이때 simulator 에서 필요한 -s, -E, -b 옵션의 경우 파라미터를 숫자로 받아야 하는데, optarg 에는 파라미터가 문자열로 저장되어 있으므로 문자열을 숫자로 변환시켜주는 string_to_num 이라는 이름의 사용자 정의 함수를 정의하여 사용하였다.

모든 command line option 이 parsing 되면 getopt()함수는 -1 을 반환하기 때문에 while 반복문을 통해 getopt()가 -1 을 반환할 때까지 반복하여 command line option 을 parsing 하였다. 종합적으로, 위와 같은 코드를 통해 option 을 parsing 해낼 수 있었다.

- Memory allocation & Cache initialization

```
    cache_line = (Line**)malloc(sizeof(Line*) * set_num);
    for(i = 0; i < set_num; i++) {
        cache_line[i] = (Line*)malloc(sizeof(Line) * E);
    }
```

〈Memory Allocation〉

Cache simulator 는 command line argument 로 받은 임의의 s, E, b 에 대해 작동할 수 있어야 한다. 따라서 Cache 의 크기가 argument 에 따라 유동적이게 되므로, 메모리를 동적으로 할당할 수 있는 malloc function 을 사용하며, 이를 위해 stdlib.h 헤더파일을 포함시켜주었다. 이때 Cache Memory 는 하나의 set 당 E 개의 line 을 가지는 2^s 개의 set 으로 이루어진다. 따라서 하나의 row 에 E 개의 Line 으로 이루어진 2^s 개의 row 형태의 2 차원 배열로 Cache Memory 를 할당해주었다. 이때 2^s 를 계산하기 위해, power 라는 이름의 함수를 만들어 사용하였다.

```
    for(i = 0; i < set_num; i++) {
        for(j = 0; j < E; j++) {
            cache_line[i][j].valid = 0;
            cache_line[i][j].tag = 0;
            cache_line[i][j].count = 0;
        }
    }
```

〈Cache initialization〉

또한 Cache 의 초기화를 위해 2 차원 배열을 구성하고 있는 각 Line 의 멤버를 모두 0 으로 초기화해주었다.

- File Read & Cache line match

입력을 받는 파일의 내용은 (operation address,size)의 형식을 가지고 있는데, 이때 instruction load 를 의미하는 "I"를 제외한 "L"(data load), "S"(data store), "M"(data modify)의 경우 operation 의 앞에 하나의 공백이 존재한다. fscanf 를 통해 파일의 내용을 읽어올 때 %c를 통해 변수 op에 operation을, %lx,%d를 통해 변수 address와 data_size에 각각 address 와 size 를 각각 받아오고자 하였는데 이때 %c 앞에 공백을 주어 operation 앞에 공백이 존재 유무와 상관없이 operation 이 %c 로 받아질 수 있게 하였다.

```
        mask = 0xffffffffffffffff << (b + s);
        tag = address & mask;
        mask = ~(0xffffffffffffffff << b);
        block_offset = address & mask;
        mask = tag | block_offset;
        set = (address ^ mask) >> b;
        tag >>= (b + s);
```

mask 라는 unsigned long 타입의 변수를 이용해, 구한 address 에 대해 마스킹을 해주었다. 이를 통해 프로그램에서 m개의 비트가 주소를 나타낸다고 할 때, tag는 address의 상위 m – (s + b) 개의 비트의 값을, set 은 중간의 s 개의 비트의 값을, block_offset 은 하위 b 개의 비트의 값을 가지도록 계산해주었다.

```
void cache_line_match(Line* cache, int E, int count, unsigned long tag, int*
hit_count, int* miss_count, int* eviction_count) {
    int min = cache[0].count, min_idx = 0;
    int full = 1;
    int i;
    for (i = 0; i < E; i++) {
        if(cache[i].valid && (cache[i].tag == tag)) { //match
            *hit_count += 1;
            cache[i].count = count;
            return;
        }
        if (!cache[i].valid) {
            full = 0;
        }
        if (min > cache[i].count) {
            min = cache[i].count;
            min_idx = i;
        }
    }

    //no match
    *miss_count += 1;
    if (full) {
```

```
        cache[min_idx].valid = 1;
        cache[min_idx].tag = tag;
        cache[min_idx].count = count;
        *eviction_count += 1;
    } else {
        for (i = 0; i < E; i++) {
            if(!cache[i].valid) {
                cache[i].count = count;
                cache[i].valid = 1;
                cache[i].tag = tag;
                break;
            }
        }
    }
    return;
}
```

Cache 가 메모리 요청을 수행하는 과정은 (1) 집합 선택, (2) 라인 매칭, (3) 워드 추출로 이루어지는데, 이번 Lab 에서는 라인 매칭까지의 수행으로 충분하므로 라인 매칭을 위한 cache_line_match 라는 이름의 사용자 정의 함수를 만들어주었다. 또한 address 를 통해 집합을 계산하였으므로, 집합 선택을 위해 Cache Memory 를 나타내는 2 차원 배열인 cache_line 에서 계산한 집합에 해당하는 row, 즉 cache_line[set]을 인자로 넘겨주었다. 또한 0 부터 시작하여 반복마다 1 씩 증가하는 count 의 값을 접근한 Cache Memory line 의 count 로 설정해줌으로써, count 를 통해 Cache 의 집합이 꽉 차 있을 경우 축출할 line 이 결정될 수 있도록 하였다.

cache_line_match 함수에서는 입력 받은 주소에 해당하는 집합에서 입력 받은 주소에서 얻은 tag 와 일치하는 tag 를 가지면서, valid bit 가 1 로 유효한 경우 match 이므로 hit_count 를 1 증가시키고, count 를 갱신한 후 return 을 통해 돌아간다. 집합을 순회하는 과정에서 valid bit 가 0 인 line 이 존재하는지 여부를 확인해 집합이 유효한 line 으로 꽉 차 있는지 확인하여 꽉 찬 경우 변수 full 의 값이 1, 아닌 경우 0 이 되도록 한다. 또한 replacement policy 인 LRU 를 위해 해당 집합에서 count 의 값이 가장 작은 라인을 찾아 놓는다. match 되지 못한 경우, 즉 Miss 가 난 경우에는 우선 miss_count 를 1 증가시키고, 만약 해당 집합이 유효한 line 으로 꽉 차 있다면 가장 오래전에 참조되었던, 즉 count 의 값이 가장 작은 line 을 대체해주도록 하며 eviction_count 를 1 증가시켜준다. 해당 집합이 유효한 line 으로 꽉 차있지 않은 경우에는 유효하지 않은 line 에 저장해주도록 한다.

이후 switch case 문을 통해 operation 에 따라 Cache memory 접근을 진행하도록 했는데, 이번 Lab 에서는 instruction load 는 다루지 않으므로 op 가 “I”인 경우에는 continue 를 통해

해당 반복이 건너뛰어지도록 하였다. op 가 “S”, “L”인 경우에는 Cache Memory 의 접근이 한 번 일어나지만, “M”인 경우 같은 주소에 대해 load 후 store 을 하는 operation 이기 때문에 Cache Memory 에 대한 접근이 2 번 이루어지게 된다.

따라서 op 가 “S”, “L”인 경우 cache_line_match 를 한 번, “M”인 경우 cache_line_match 를 두 번 호출하도록 해주었다.

- Memory Deallocation

```
    for (i = 0; i < set_num; i++) {
        free(cache_line[i]);
    }
    free(cache_line);

    printSummary(hit_count, miss_count, eviction_count);
    fclose(fp);
    return 0;
```

동적할당해주었던 cache_line 을 free 를 통해 할당 해제해주며, 전 과정에서 기록된 hit, miss, eviction 의 횟수를 printSummary 함수를 통해 출력해준 후 읽어주었던 파일을 닫으면서 종료된다.

- Result

```
[limyoojin@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
```

위와 같이 구현한 Cache simulator 를 시뮬레이션한 결과 Reference simulator 와 같은 결과를 얻을 수 있었다.

Part B: Optimizing Matrix Transpose

Part B 는 int 형 2 차원 배열 A 와 B 에 대해 B[M][N]이 A[N][M]의 Transpose Matrix 가 되도록 Matrix Transpose 를 수행하는 과정에서 Cache 에서 발생하는 Miss 를 최대한 줄이도록 최적화하는 것이다. 또한, Part B 에서 가정되는 Cache 는 s = 5, E = 1, b = 5, 즉 집합이 32 개, cache block 의 크기가 32 bytes 인 direct mapping Cache 이다.

- 32 x 32 (M = 32, N = 32)

2 차원 배열은 1 차원 배열들의 배열로 생각할 수 있으며, row major order 로 연속된 메모리 주소를 가진다. Cache block 의 크기가 32 bytes 이므로, int 형 데이터를 8(=32/4)개 저장할 수 있으므로, 2 차원 배열을 8 x 8 의 작은 block 들로 쪼개서 Transpose 해볼 수 있다. 이때 Cache 에 집합이 32 개 존재하므로 8 x 8 의 하나의 block 안에서는 각 행에 해당하는 집합이 겹치지 않게 된다. Cache block 이 int 형 데이터를 8 개 저장할 수 있으므로, A 의 데이터에 접근하는 과정에서 Miss 를 최대한 줄이기 위해서는 A 의 8 x 8 block 에서 하나의 행이 Cache 에 저장되었을 때 해당 행이 Cache 에서 내려가기 전 저장되어 있는 8 개의 int 형 데이터를 최대한 이용해야 한다.

따라서, 이를 위해 지역 변수 tmp0 ~ tmp7 을 이용하였다. A 의 8 x 8 block 안에서 각 행이 Cache 에 저장되었을 때, 해당 행의 8 개의 int 데이터를 tmp0 ~ tmp7 의 지역 변수에 저장했다. 이후 해당 A 의 8 x 8 block 의 transpose 한 위치에 해당하는 B 의 8 x 8 block 에서 tmp0 ~ tmp7 에 저장된 값을 이용해 A 의 행의 Transpose 한 위치에 해당하는 B 의 열을 바꿔주었다. 모든 8 x 8 block 에 대해 위와 같은 방식을 적용하여, miss 를 287 번까지 줄일 수 있었다.

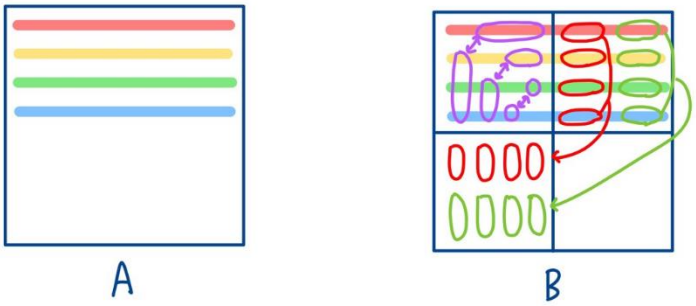
```
[limyoojin@programming2 cachelab-handout]$ ./test-trans -M 32 -N 32

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
```

miss 를 더욱 줄일 수 있는 방안을 생각해보면, ./test-trans -M 32 -N 32 를 실행시켰을 때 생성되는 trace.f1 을 통해 확인해보면 2 차원 배열 A 와 B 는 시작 주소에 해당하는 Cache 의 집합이 같아 같은 위치에 존재하는 8 x 8 block 에 대해 집합이 겹치게 된다. 따라서, A 와 B 에서 Transpose 해도 Block 의 위치가 같게 되는, 대각선에 위치하는 block 에 대해서 conflict miss 가 나타나 같은 집합에 해당하는 A 의 행과 B 의 행이 번갈아가면서 Cache 에 올라갈 가능성이 크게 된다.

따라서 대각선에 위치하는 block 의 경우 다른 block 들에 비해 miss 가 많이 발생하게 된다. 이런 miss 를 줄이기 위해 대각선의 block 에 대해서는 다른 방법을 적용하여 conflict miss 를

줄이고자 하였다. 이를 위해 B 의 공간을 이용하는 방식을 사용하였는데, B 의 block 상단에 A 의 block 상단을 옮겨 놓은 후, B 의 block 내부에서 왼쪽 상단의 4 x 4 부분에 대해 지역변수와 저장되어 있는 값을 활용하여 바꾸어 주고 왼쪽 하단의 4 x 4 부분에 대해 오른쪽 상단의 4 x 4 부분에 저장되어 있는 값과 지역 변수를 활용해 바꾸어 주었다. 과정은 아래 그림과 같다.



그리고 B 의 block 하단부에 대해서는 다른 block 들과 마찬가지로 지역 변수 8 개를 이용해 A의 block의 한 행을 저장해두고, B의 block의 한 열을 바꿔주는 방식을 적용했다. 이와 같이 대각선에 위치하는 block 에 대해서는 다른 방법을 적용시킨 경우 275 번까지 miss 를 줄이는 결과를 얻을 수 있었다.

결과적으로 Cache에 대해 최적화되지 않은 단순한 row-wise scan transpose에서 1183이던 miss의 횟수를 아래와 같이 275번까지 줄이는 결과를 얻을 수 있었다.

```
[limyoojin@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:2002, misses:275, evictions:243
```

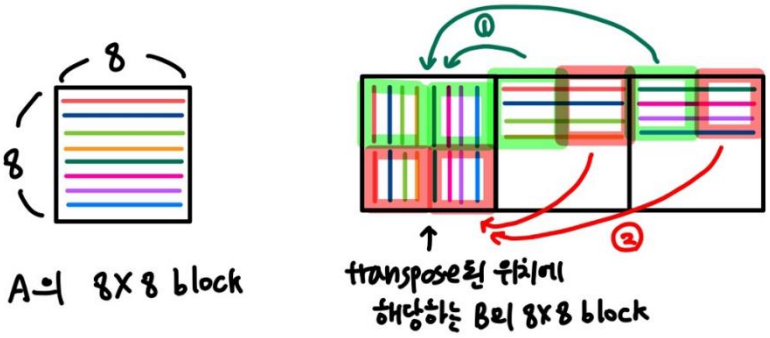
- 64 x 64 (M = 64, N = 64)

M = 32, N = 32 일 때와 마찬가지로 trace.f1 을 통해 확인해보면 2 차원 배열 A 와 B 는 시작 주소에 해당하는 Cache 의 집합이 같다는 것을 알 수 있다. 또한, M 과 N 이 모두 64 인 경우 M 과 N 이 모두 32 였던 경우와는 다르게 2 차원 배열을 8 x 8 block 들로 나누었을 때 하나의 Block 안에서 상단과 하단에서 Cache 의 집합이 겹치게 된다. 하지만 A 의 데이터를 읽어올 때 miss 를 최소한으로 할 수 있는 방안은 A 에서 32 bytes, 즉 int 형 데이터 8 개가 Cache 에 올라갔을 때, 올라간 8 개의 int 형 데이터가 전부 이용된 후에 Cache 에서 내려가 해당 32 bytes 가 다시 Cache 에 올라가는 일이 없도록 하는 것이라고 생각하여 32 x 32 에서와 같이 8 x 8 block 을 기준으로 Transpose 를 진행하기로 하였다.

8 x 8 block 에서 상하단의 집합이 겹침에 따라 하나의 block 을 transpose 하는 과정에서 conflict miss 가 자주 발생하는 것을 살펴볼 수 있었다. Part B 의 Programming Rule 에서 배열 A 은 수정해서는 안 되지만 배열 B 의 값에 대해서는 무엇이든 해도 된다고 하였으므로, 이러한 conflict miss 를 줄이기 위해 배열 B 에서 transpose 을 진행하는 8 x 8 block 외의 공간을 이용하기로 하였다.

우선 A 의 8 x 8 block 에서 지역변수 tmp0~tmp7 을 이용하여 상단에 해당하는 데이터를 transpose 해야하는 B 의 8 x 8 block 바로 오른쪽의 block 상단에, 하단에 해당하는 데이터를 그 다음 오른쪽 block 상단에 저장해 둔다. 이후 transpose 된 위치에 해당하는 B 의 8 x 8 block 의 상단부터 오른쪽 block 에 저장된 값을 활용해 변경하고, 하단의 값도 그 다음 오른쪽 block 에 저장된 값을 이용해 변경해준다.

또한 Transpose 의 과정에서는 A 와 B 둘 중 하나에 대해서는 row major 로, 다른 하나는 column major 로 접근이 이루어지게 된다. 이때 A 의 8 x 8 block 들에 대해서 column major, B 의 8 x 8 block 들에 대해서 row major 의 순서로 접근하면서, 위와 같은 방법을 사용하는 경우 transpose 가 진행되는 block 의 오른쪽과 그 다음 오른쪽 block 의 상단이 Cache 에 이미 올라가 있게 되어 다음 block 에 대해 transpose 를 진행할 때 conflict miss 와 cold miss 가 적게 일어날 수 있다. 그림으로 나타내면 아래와 같다.



가장 오른쪽 2 열에 해당하는 block 들에 대해서는 이와 같이 오른쪽과 그 다음 오른쪽의 block 을 이용하는 방법이 불가능하므로, tmp0 ~ tmp7 의 지역 변수 8 개를 최대한 활용하고자 했다. 이때 32 x 32 에서와는 다르게 64 x 64 에서는 하나의 8 x 8 block 안에서 상단부와 하단부의 집합이 겹치기 때문에, A 의 8 x 8 block 에서 Cache 에 올라갔던 행이 다시 올라가는 일이 최소화되도록 최적화를 진행했으며, A 의 8x8 block 에서 상단 부분을 먼저 옮긴 후 하단 부분을 옮기도록 하였다. 또한 이때 A 를 옮기는 과정에서도 B 의 상단부가 먼저 변경된 후, 하단부가 변경될 수 있도록 했다.

위와 같은 방식을 통해 캐시에 최적화되지 않은 Simple row-wise scan transpose 방식에서 4723 번 발생하던 miss 의 개수를 아래와 같이 1256 번까지 줄일 수 있었다.

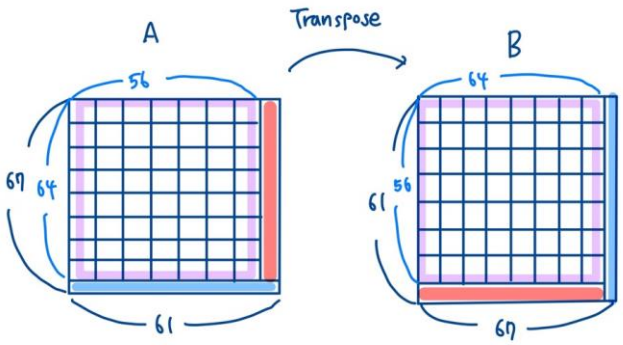
```
[limyoojin@programming2 cachelab-handout]$ ./test-trans -M 64 -N 64

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:13085, misses:1256, evictions:1224
```

- 61 x 67 (M = 61, N = 67)

앞서 32 x 32, 64 x 64 matrix 에서는 열의 개수가 Cache block 에 저장되는 int 형 데이터의 개수인 8 의 배수였기 때문에 matrix 를 8 x 8 의 작은 block 들로 나누었을 때 각 block 안에서 한 행의 데이터는 같은 Cache block 으로 매핑되었다. 하지만 61 x 67, 67 x 61 matrix 의 경우 열의 개수가 8 의 배수가 아니기 때문에 matrix 를 8 x 8 의 작은 block 들로 나누었을 때 각 block 안에서 한 행의 데이터가 1 ~ 2 개의 Cache block 으로 매핑된다. 그럼에도 불구하고 앞선 경우들과 마찬가지로 하나의 8 x 8 block 안에서 하나의 행이 Cache 에 올라가게 되었을 때 지역변수를 이용하여 그 행의 데이터들을 저장해 두었다가 B 을 수정한다면 conflict miss 를 감소시킬 수 있으므로, Simple row-wise scan transpose 방식에 비해서는 miss 를 줄일 수 있을 것이다. 그렇지만 61 x 67, 67 x 61 matrix 의 경우 행과 열의 개수가 모두 8 로 나누어 떨어지지 않기 때문에 matrix 전체를 8 x 8 의 block 으로 나누는 것은 불가능하다. 따라서 61 x 67, 67 x 61 matrix 안에서 각각 56 x 64, 64 x 56 만큼은 8 x 8 block 으로 나누어 지역변수를 활용하여 transpose 하기로 하였으며, tmp0 ~ tmp7 의 지역 변수 8 개를 활용해 A 의 8 x 8 block 의 상단 부분을 옮긴 후 하단 부분을 옮겼다.

나머지 부분에 대해서는 block 으로 나누지는 않고 지역변수를 활용하여 transpose 하기로 하였다. 먼저 A 의 64 ~ 66 행에 대하여 0~59 열을 지역 변수 6 개를 이용해 2 열씩 B 로 옮기고 60 열은 지역 변수 3 개를 이용해 B 로 옮겼다. 그 다음으로 A 의 56~60 열에 대해 0~63 행을 지역 변수 5 개를 이용해 1 행씩 B 로 옮겼다. 결과적으로 아래 그림과 같은 방식으로 최적화를 진행하게 되었다.



이와 같은 최적화 방식을 통해 Simple row-wise scan transpose 방식을 이용하는 경우 4423 번 발생하던 miss 의 개수를 아래처럼 1979 번까지 줄일 수 있었다.

```
[limyoojin@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:6200, misses:1979, evictions:1947
```

```
[limyoojin@programming2 cachelab-handout]$ 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: 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         275
Trans perf 64x64           8.0         8        1256
Trans perf 61x67          10.0        10        1979
          Total points    53.0        53
```

4. 결론

이번 Lab 에서 Cache simulator 를 직접 구현해보면서 Cache Memory 의 작동 원리에 대해 공부하고, 현대 컴퓨터 시스템을 이루는 Memory Hierarchy 개념에 대한 이해도를 높일 수 있었다. 또한 Matrix Transpose 를 최적화해보면서 수업 시간에 배웠던 Cache 친화적 코드를 작성하는 방법을 직접 적용해보면서 익힐 수 있었다.