Project 4: Multi-level Cache Model and Performance Analysis

Submitter: Jiwon Jang (202211167)

Contents Table

---

1. Compile and Execution

언어는 C++를 사용하였고, 컴파일러는 g++ 9.5.0을 이용하였다. 다음 명령어를 치면 컴파일할 수 있다.

```
g++ -o homework4 homework4.cpp
```

이후 파일 실행은 다음과 같이 진행했다. `./homework4 -c -a -b TraceFile.out` 순서로 입력하면, 프로그램이 Trace File을 읽어와 결과를 출력을 해준다. 우리가 입력해야 하는 옵션은 다음과 같다.

`-c`: Cache 용량 (단위는 KB이다)

`-a`: Way 개수

`-b`: Block size (단위는 B이다)

`-lru`: Replacement 정책으로 LRU 사용

`-random`: Replacement 정책으로 Random 사용

2. Definition of Class and Function

이번 과제에서 구현한 클래스와 함수에 대한 설명이다. 코드의 흐름을 설명하기 위한 배경 설명이다.

<library>

총 5개의 library를 이용하였다.

iostream: 표준 입출력을 지원한다.

fstream: 파일 생성 및 read/write를 지원한다.

string: 문자열 형식을 지원한다.

vector: vector 자료형을 지원한다.

cstring: 다양한 문자열 연산을 지원한다.

<class>

class는 Cache와 Cache의 Block을 관리하기 위해서 사용하였다.

1) BLOCK class definition

private:

int index: 몇 개의 index로 Cache가 이루어져 있는지 알고 있다.

int way: 몇 개의 way(num of associativity)로 Cache가 이루어져 있는지 알고 있다.

vector<vector<vector<int>>> block: Block vector이다. [Index][Way][Info]의 3-dim 형태의 vector 이다.

public:

BLOCK(int index, int way): Block Instructor이다. Cache에서 받은 정보를 바탕으로 Block을 resizing한다.

void Set_Tag(int tag, int now_index, int now_way): Index와 Way 정보를 받아와서 Tag를 설정하는 함수이다.

void Set_Dirty(int dirty, int now_index, int now_way): Index와 Way 정보를 받아와서 Dirty bit를 설정하는 함수이다.

void Get_Tag(int tag, int now_index, int now_way): Index와 Way 정보를 받아와서 Tag를 알아오는 함수이다.

void Get_Dirty(int dirty, int now_index, int now_way): Index와 Way 정보를 받아와서 Dirty bit를 알아오는 함수이다.

1) CACHE class definition

private:

int index: 몇 개의 index로 Cache가 이루어져 있는지 알고 있다.

int way: 몇 개의 way(num of associativity)로 Cache가 이루어져 있는지 알고 있다.

vector<vector<vector<int>>> block: Block vector이다. [Index][Way][Info]의 3-dim 형태의 vector 이다.

int capacity: Cache의 용량 정보를 담고 있다.

int associativity: Cache의 Way(Associativity) 정보를 담고 있다.

int block_size: Cache Block size 정보를 담고 있다.

string replacement_policy: Repacement 정책 정보를 담고 있다.

vector<deque<int>> lru_list: lru replacement policy를 사용할 때 활용할 List이다. deque이다.

public:

BLOCK B*: Cache에서 사용할 Block class이다.

void Set_Capacity(int capacity): Cache 용량을 설정하는 함수이다.

void Set_Associativity(int associativity): Cache way을 설정하는 함수이다.

void Set_Block_Size(int block_size): Cache block size을 설정하는 함수이다.

void Set_Replacement_Policy(string policy): Cache replacement 정책을 설정하는 함수이다.

int Get_capacity(): Cache 용량을 알아오는 함수이다.

int Get_associativity(): Cache way을 알아오는 함수이다.

int Get_Index(): Cache index을 설정하는 함수이다.

void Initialize_Blocks(): Cache Block을 초기화하는 함수이다.

void Initialize_LRU(): LRU 리스트를 초기화하는 함수이다.

int Find_data(int addr, string W): Cache에 데이터를 찾는 함수이다.

void Set_Tag(int addr): Cache에 데이터를 집어넣는 함수이다.

int Get_LRU_Victim(int data_index): LRU일 때 victim을 결정하는 함수이다.

int Get_Random_Victim(): Random일 때 victim을 결정하는 함수이다.

void Update_LRU(int index, int way): LRU 리스트를 업데이트 하는 함수이다.

void Update_Eviction(int eviction_type): Eviction 정보를 업데이트 하는 함수이다.

int Get_dirty_eviction(): Dirty eviction을 가져오는 함수이다.

int Get_clean_eviction(): Clean eviction을 가져오는 함수이다.

<other functions>

다른 함수로는 Caching 함수가 있다. 이 함수에서는 input 정보를 바탕으로 L1, L2 Cache를 생성해 준다. 이후 CACHE class에 정의되어 있는 *Find_data*함수로 Cache miss와 Cache hit을 판단하고, 각 상황에 따라 올바른 행동을 취한다(*i.e. Cache Block update, Write Back, Write allocate*). 밑은 Caching 함수의 일부 코드이다. 밑과 같은 방식으로 Cache miss, Cache hit을 판단한다.

```
if (W == "R") {
            read_access++;
            if (L1.Find_data(addr, "R") == 1) { // L1.hit L2.hit
            } else {
                L1_read_miss++;
                if (L2.Find_data(addr, "R") == 1) { // L1.miss, L2.hit
                    L1.Set_Tag(addr);
                } else {
                    L2_read_miss++;
                    // L1.miss, L2.miss
                    L1.Set_Tag(addr);
                    L2.Set_Tag(addr);
                }
            }
```

```
        }
```

이후 모든 동작이 완료되면 TraceFile_C_A_B.out 파일을 출력해 준다.

<main function>

main 함수는 input 값을 해석하여, 해석 결과를 Caching 함수에 전달하는 역할을 한다

3. Code idea

assignment에 사용한 몇 가지 코딩 아이디어이다.

1. LRU replacement's victim

밑의 코드와 같이 가장 최근에 접근 한 tag는 뒤 Index에 넣고, 가장 접근한지 오래된 tag는 앞 index에 넣으므로써 LRU replacement policy를 실현하였다.

```
int CACHE::Get_LRU_Victim(int data_index) {
    int victim = lru_list[data_index].front(); // 가장 오랫동안 사용되지 않은 데이터가 맨 앞에 위치
    lru_list[data_index].pop_front();
    return victim;
}
void CACHE::Update_Eviction(int eviction_type) {
    if (eviction_type == 1) {
        this->dirty_eviction++;
    } else { this->clean_eviction++; }
}
```

2. addr's index bit

Addr에서의 index bit를 얻어오기 위해 다음과 같이 masking을 하였다.

```
 int block_offset = log2(block_size);
    int index_bit = log2(Get_Index());
    int tag = addr >> (block_offset + index_bit);
    int data_index = (unsigned)(addr << (32 - block_offset - index_bit)) >> (32 - index_bit);
```

4. Code flow

코드는

1) main 함수를 통한 파일 read 및 input 해석

2) Cache 함수를 통한 File 실행 및 Cache 관리

3) Cache 함수를 통해 얻은 데이터를 바탕으로 한 OutputFile 출력

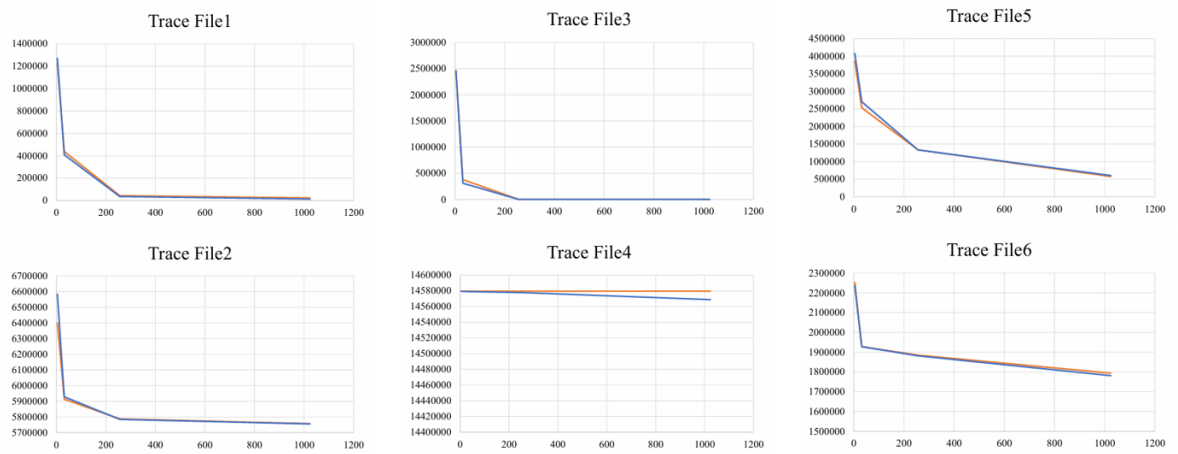
으로 이루어 진다.

5. Results Analysis

1. Cache Capacity에 따른 결과 분석

: Capacity를 4, 32, 256, 1024로 설정하고, L2 cache의 capacity에 대한 L2 misses를 분석한 결과는 다음과 같다. 이 때 Trace File은 400_perlbench, 450_soplex, 453_povlay, 462_libquantom, 473_astar, 483_xalancbmk 순서이다. 주황 선은 random replacement policy를 하였을 때의 결과이고, 파란 선은 LRU replacement policy를 하였을 때의 결과이다.

(이 때 associativity = 2, block size = 16으로 설정하였다)

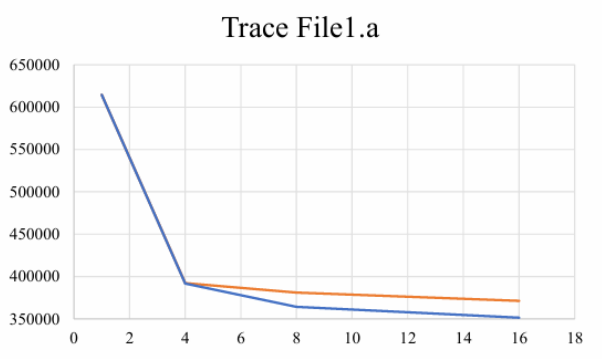


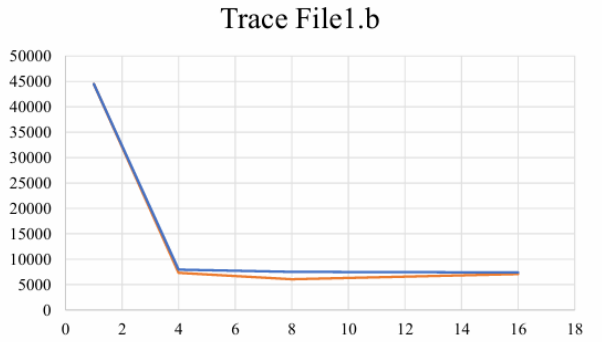
결과를 확인해 보면 대체로 Cache 용량이 커질수록 misses가 낮아짐을 확인할 수 있다. 또한 많은 차이는 가지지 않지만, 대체로 LRU replacement policy가 random replacement policy보다 좋은 성능을 유도함을 알 수도 있다.

2. Cache Associativity에 따른 결과 분석

: Associativity를 4, 8, 12, 16로 설정하고, L2 cache의 capacity에 대한 L2 misses를 분석한 결과는 다음과 같다. Trace File 1에 대해서만 분석을 진행하였다.

(이 때 각각 capacity = 32, block size = 16, capacity = 1024, block size = 16으로 설정하였다)





결과를 확인해 보면 way-1일 때 특히 많은 miss율을 보임을 확인할 수 있다. miss 차이를 보면 index 개수가 적을수록(*capacity가 적을수록*) way-1 vs way-2,3,4 격차가 심함을 확인할 수 있다. 그리고 이는 index가 적으면 같은 index에 접근할 가능성이 높아지기 때문이다.