CSE305
Project 4. Multi-level Cache Model and Performance

201611057 김준우

1. Introduction

이 프로그램은 2-level 구조의 캐시를 시뮬레이션 할 수 있는 프로그램이다.

2. Environment

VMWare Workstation 16을 사용하여 Ubuntu 20.04.4를 구동하였으며, Ubuntu 내부에서는 VSCode로 코딩을 한 후, 터미널을 통해 구동을 확인하였다. 사용한 언어는 C++이며 컴파일 환경은 c++ 9.4.0 version이다. 프로그램은 Ubuntu의 Terminal 콘솔창에서 **c++ -o runfile project4.cpp** 명령어로 컴파일 후 실행이 가능하다.

3. Explanation

해당 프로그램은 우선 L1_cache와 L2_cache, 2가지의 클래스로 구성되어있다. 클래스는 각각 address, index, tag, vaild, lru, dirty_bit과 같은 인자들을 가지고 있으며, 2차원 배열 구조의 동적할당을 통해서 캐쉬의 구조를 구현했다.

```
class L1_cache {
    public:
        int lru=0;
        int vaild=0;
        int dirty_bit=0;
        long long int address=0;
        long long int index=0;
        long long int tag=0;
        L1_cache(){};
        L1_cache(long long int add, long long int ind, long long int tg){address=add; index=ind; tag=tg;};
};
```

그리고 나머지 아래의 함수로 구성이 되었다.

int cache_hit_check(string level,L1_cache l1_new, L2_cache l2_new)

void L1_cache_operator(string str)

void L2_cache_operator(string str)

첫 번째의 cache_hit_check는 접근하고자 하는 block의 cache hit 여부를 판별해준다. 그리고 나머지 operator함수는 각각의 레벨에 맞는 operation을 진행하는 구조로 구현하였으며, L2_cache_operator의 경우, L1에서 miss가 났을 때에만 작동이 되도록 L1 operator안에 조건문과 함께 들어가있다.

또, lru 정책을 구현하기 위해서 각 class 마다 lru 변수를 포함하고 있으며, 이 변수는 최근에 쓰여진 값일수록 큰 값을 가지게 된다. 그리고 우선적으로 배열의 앞쪽에 위치한 블락주소들이 먼저 캐쉬에 쓰여진 값이다. 그러므로 lru와 이 순서를 비교하여 lru policy를 구현했다. 또, random library를 활용하여 random policy도 구현하였다.

프로그램은 과제의 조건에서 주어진대로 "./runfile -c 256 -a 8 -b 128 -lru 473_astar.out" 와 같은 구조의 명령어로 실행이 가능하다. (이때 input파일은 runfile과 같은 위치에 있어야한다.) 더불어, input값이 충분히 주어지지 않은 상황에 따라서 조건문을 만들어 실행되지 않도록 구성했다.

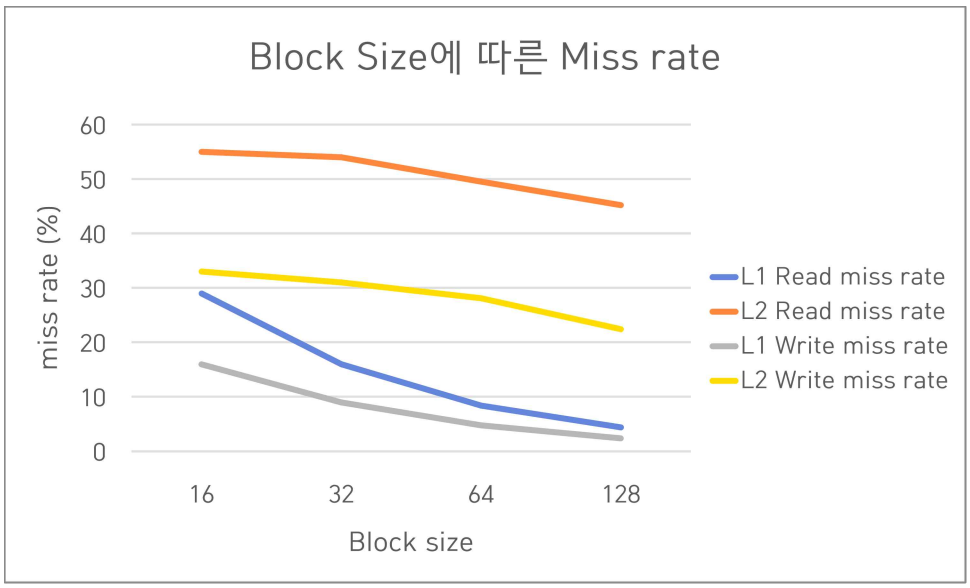
4. Results

A. Trace File에 따른 결과 (256KB, 4associativity, 32Block)

400_perlbench	450_soplex
L1 Read miss rate: 0.109639% L2 Read miss rate: 4.05522% L1 Write miss rate: 0.0335902% L2 Write miss rate: 3.51125%	L1 Read miss rate: 16.2557% L2 Read miss rate: 54.4021% L1 Write miss rate: 9.11065% L2 Write miss rate: 31.666%
453_povray	462_libquantum
L1 Read miss rate: 0.00475143% L2 Read miss rate: 0.154296% L1 Write miss rate: 0.00404362% L2 Write miss rate: 0.405728%	L1 Read miss rate: 49.9998% L2 Read miss rate: 61.4272% L1 Write miss rate: 0.000719504% L2 Write miss rate: 0.00143901%
473_astar	483_xalancbmk
L1 Read miss rate: 6.95097% L2 Read miss rate: 43.5398% L1 Write miss rate: 0.381188% L2 Write miss rate: 1.9645%	L1 Read miss rate: 8.88024% L2 Read miss rate: 71.0813% L1 Write miss rate: 0.083471% L2 Write miss rate: 34.3476%

trace파일의 구성에 따라서, 다른 miss rate의 결과가 나왔다. 최근 접근한 block에 얼마나 자주 접근하는지, 또는 locality에 따른 차이의 결과로 보인다.

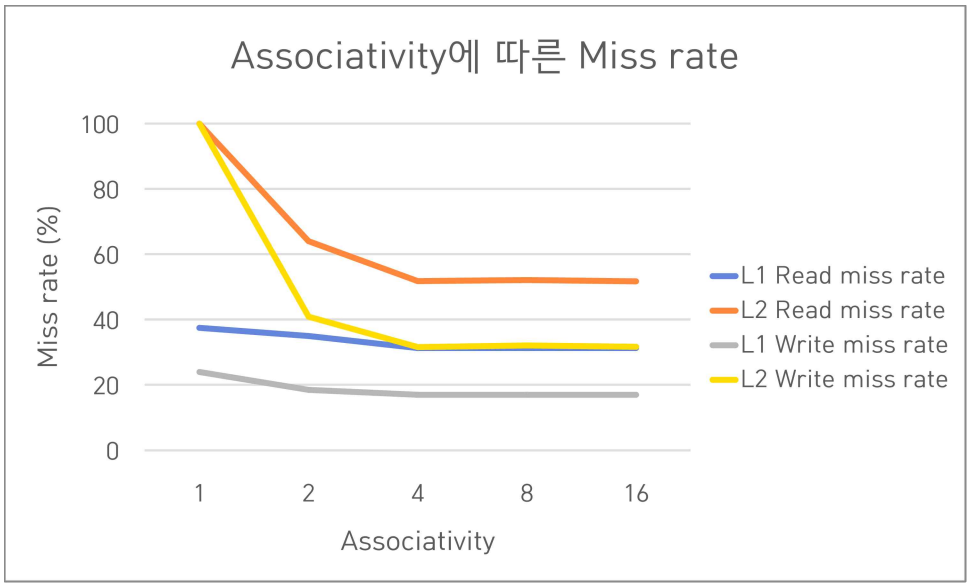
B. Block Size에 따른 Miss Rate



Block의 크기에 따른 Miss rate의 비교 그래프이다. trace file은 450_soplex.out을 사용했으며, 16 associativity와 256KB의 조건으로 시뮬레이션했다.

Block의 사이즈가 클수록 Miss rate의 비율이 줄어드는 것을 확인 할 수 있었으며, Block의 크기가 크면 여러 가지 주소들이 캐시에 함께 쓰여지고, Spatial Locality의 영향 덕분에 miss rate가 줄어든 것으로 볼 수 있다. 다만, 이 시뮬레이션에서 시간은 확인할 수 없지만, block size가 클수록 miss penalty가 크기 때문에, 적절한 block size를 선정하는 것이 중요하다.

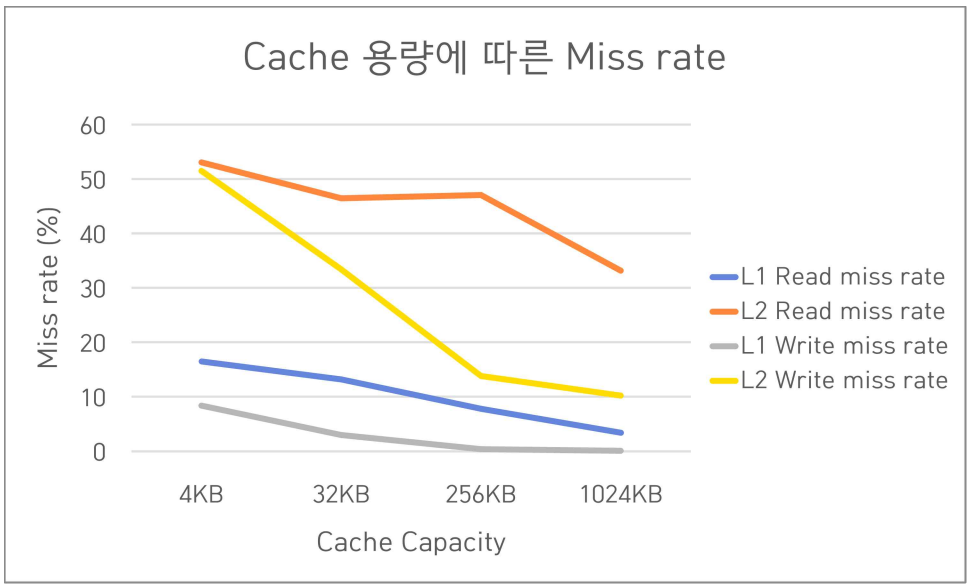
C. Associativity에 따른 Miss Rate



Associativity에 따른 Miss rate의 비교 그래프이다. trace file은 450_soplex.out을 사용했으며, 16 associativity와 8KB의 조건으로 시뮬레이션했다.

Associativity가 작을수록 Miss rate가 큰 것을 확인 할 수 있었고, 클수록 Miss rate가 줄어드는 것을 볼 수 있었다. Associativity가 클수록 동일한 index에서 Block Replacement를 더 유연하게 할 수 있기 때문에 miss rate가 줄어든 것으로 볼 수 있다.

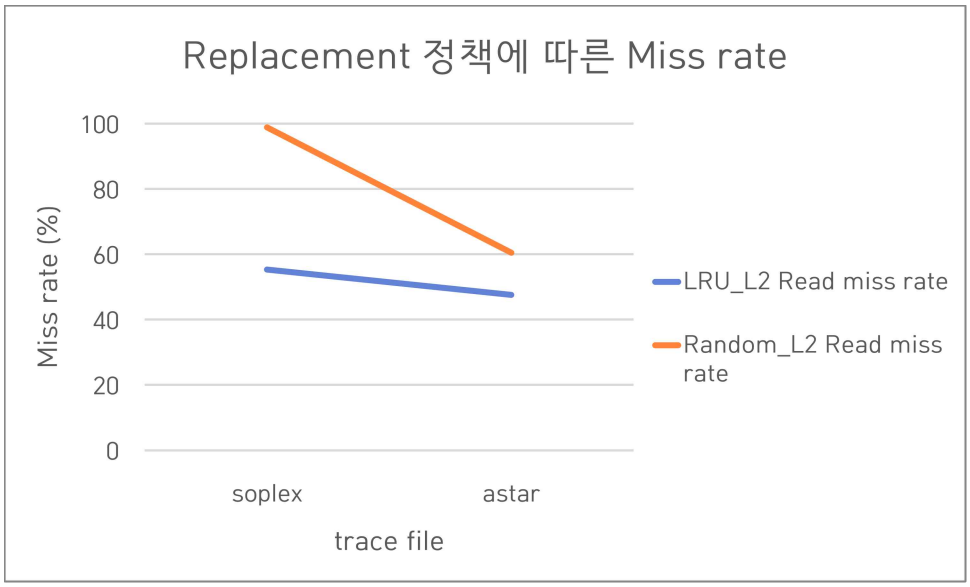
D. Cache Capacity에 따른 Miss Rate



Cache의 용량에 따른 Miss rate의 비교 그래프이다. trace file은 473_astar.out을 사용했으며, 16 associativity와 16Block의 조건으로 시뮬레이션했다.

캐시의 크기가 클수록, 캐시에 저장할 수 있는 데이터의 양이 늘어나므로, miss rate가 줄어드는 것을 볼 수 있었다. 다만, 이 시뮬레이션에서 시간에 대한 고려는 하지 못하였는데, cache의 용량이 클수록 읽기/쓰기의 시간 지연이 오래걸리므로 적절한 cache 용량을 선택하는 것이 중요하다.

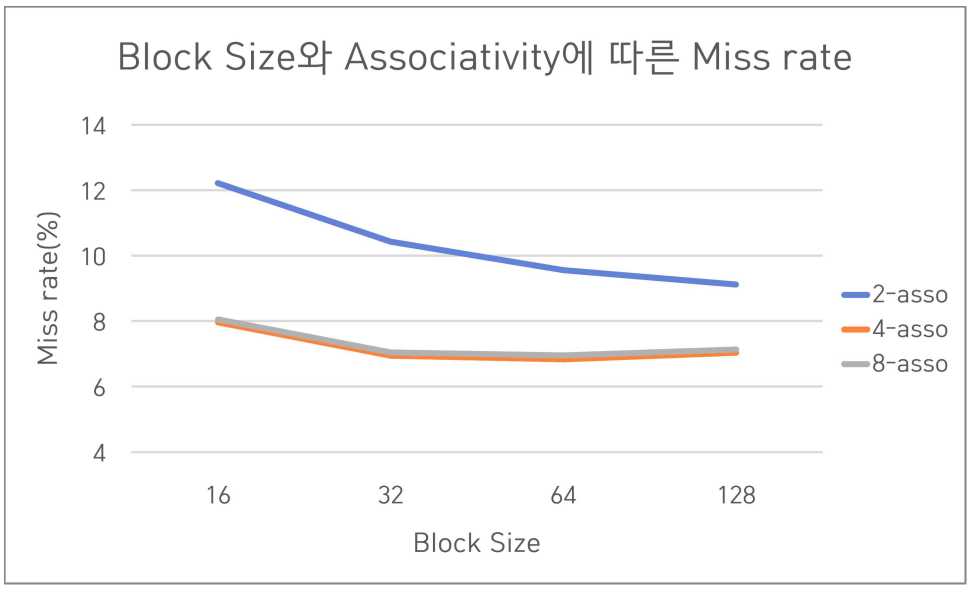
E. Replacement Policy에 따른 Miss Rate



Replacement Policy에 따른 Miss rate의 비교 그래프이다. trace file은 두 가지를 사용했으며, 16 associativity와 16Block, 256KB의 조건으로 시뮬레이션했다.

LRU정책이 위 두 개의 trace에서 더 나은 miss rate를 보여주었다. LRU정책은 Locality를 고려한 정책이므로, 이것이 random정책 보다 더욱 효과적이다.

F. Block Size와 Associativity에 따른 Miss Rate



Block의 크기에 따른 Miss rate의 비교 그래프이다. trace file은 473_astar.out을 사용했으며, 값은 L1_Cache_Miss_rate이다.

Block의 사이즈와 Associativity가 클수록 Miss rate는 상대적으로 줄어드는 것을 볼 수 있지만, 그 두 개의 값이 중간으로 적절할 때에 Miss rate가 가장 줄어듬을 확인할 수 있었고, 수업자료에서 보았던 그래프와 굉장히 유사한 결과를 볼 수 있었다.