컴퓨터 구조 Project4 보고서

202011125 유선우

1. 컴파일/실행 방법, 환경

코딩과 테스트는 docker를 이용한 ubuntu 20.04에서 진행하였다.

해당 환경에서 g++의 버전은 9.4.0을 사용하였고,

다음 command를 통해서 컴파일과 실행을 할 수 있었다.

$ g++ cache.cpp -o runfile

$ ./runfile <-c capacity> <-a associativity> <-b block_size> <-lru or -random> <trace file>

2. Cache Structure

Cache의 구조로는 cache class를 생성하여 사용하였다.

캐시 class에서 index에 대한 tag, valid, dirty 정보를 map구조를 사용하여 사용하였다.

```
map<long long int, vector<pair<long long int, tuple<int, int>>>> cache_list;
```

이와 같이 사용하여, 하나의 index에 대해서도 여러 개의 tag, valid, dirty set이 vector의 element로 들어올 수 있도록 하였고, associativity level을 바탕으로 실행 중에 vector 길이가 associativity level을 넘지 못하도록 하였다.

또한 lru정책을 사용하기 위해 가장 최근에 방문한 cache에 대해 valid bit가 1이 되도록 하였다. 이후로 매 instruction마다 모든 valid bit가 1씩 증가하도록 하여서 lru가 적용되었을 때, valid bit가 가장 큰 cache 값이 먼저 제거되도록 하였다.

또한 valid bit의 목적으로 처음 생성하였지만 vector에 특성상 valid bit가 0인 상황은 생성하지 않고 vector에서 push, pop하는 방식으로 valid한 값들만 남도록 handling하였다.

3. 코드 작동 Flow

Pipelining의 동작을 하기 위해서는

1) Input command parsing
2) Trace File execution

의 순서를 거쳐야 한다.

(해당 코드를 작동시키면 fileName_capacity(L2)_associativity(L2)_blockSize.out의 이름의 파일이 생성되어 결과가 저장된다.)

1) Input command parsing

[-c capacity] 파라미터는 cache의 capacity를 의미한다. **[-a associativity]** 파라미터는 cache의 associativity level을 의미한다. **[-b block_size]** 파라미터는 offset으로 사용되는 block의 size를 의미한다. **<-lru or -random>** 파라미터는 replacement policy를 random으로 할지 lru로 할지를 의미한다.

2) Trace File execution

Trace file이 너무 크기 때문에 한번에 파일을 읽고, 실행하는 것은 시간이 너무 오래걸려 매 line을 읽으며 해당 line에 대한 cache 실행을 하였다.

Bit operation을 이용하여 각 L1, L2의 index, tag값을 구하였다.

해당 값들에 대하여 Read의 경우와 Write의 경우를 나누어 수행하였다.

2경우에서 기본적인 실행은 동일하였다. 해당 index가 L1에서 있는지 확인하고, 있으면 L1, L2 hit를 적용하였으며, L2에만 있을경우 L2 hit와 L1 update, L2에도 없는 경우는, Cache update를 해주었다.

하지만 Write의 경우에 dirty bit를 수정해주었으며, 해당 작업으로 dirty bit가 1인 값들은 eviction될 때, dirty eviction이 되도록 구현하였다.

3) Output

Trace file의 이름이 다양하기에, 각 파일의 이름을 다음과 같이 설정한다.

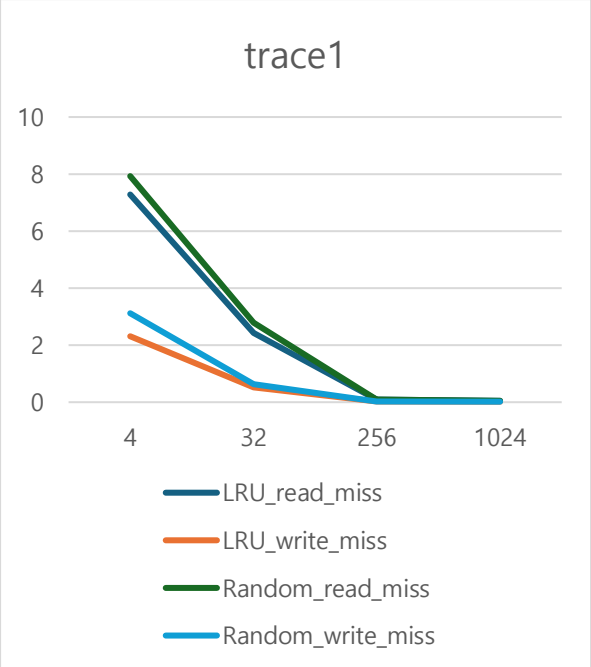
1. trace1 : 400_perlbench.out
2. trace2 : 450_soplex.out
3. trace3 : 453_povray.out
4. trace4 : 462_libquantum.out
5. trace5 : 472_astar.out
6. trace6 : 483_xalancbmk.out

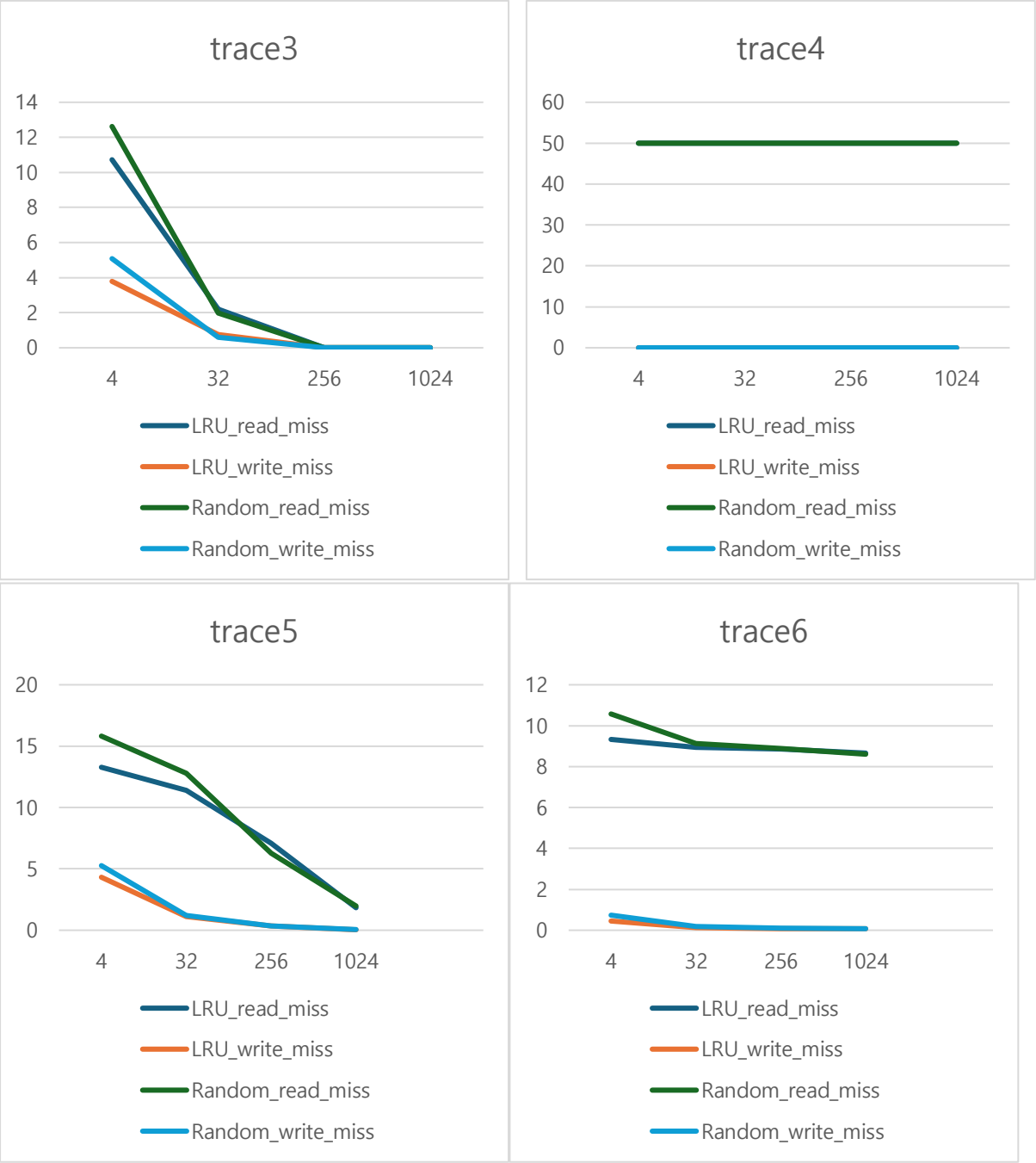
먼저 첫 실험은 다음과 같다.

Cache의 capacity에 따른 read, write miss를 구하였다.

L2 cache에 대하여 Associativity는 4, block_size는 32로 설정하였으며, capacity를 4, 32, 256, 1024KB에 대해 LRU, Random 각 경우를 실험 해보았다.

x축이 capacity(KB), y축이 miss rate(%)이라 볼 수 있다.





이와 같은 결과가 나온다는 사실을 알 수 있다.

여기서 일반적인 사실을 확인해보면 LRU policy를 사용했을 때, random policy보다 더 낮은 miss rate를 갖는다는 점을 알 수 있다.

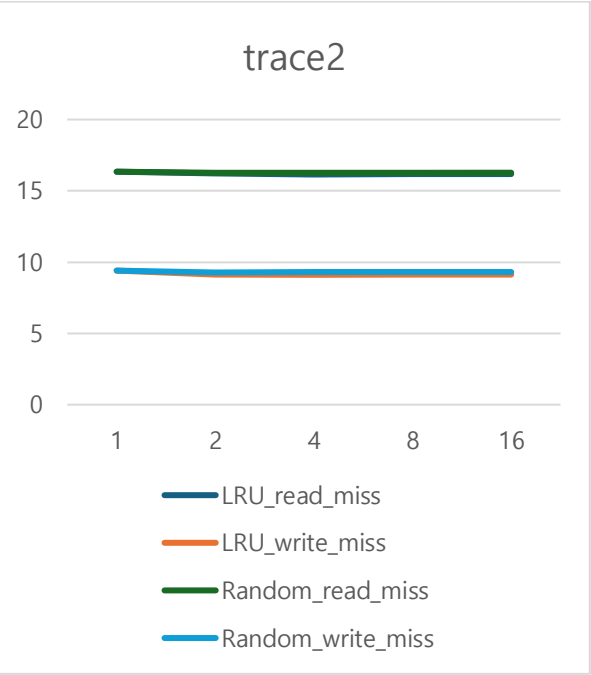
또한 수업에서 배운 내용을 보면 cache capacity가 커지면 miss rate이 낮아지고, access time이 길어져 hit time이 길어진다. 여기서 trace2, 4, 6에서는 큰 차이가 나지 않지만 일반적으로 cache capacity가 커지면 miss rate가 감소하는 결과를 확

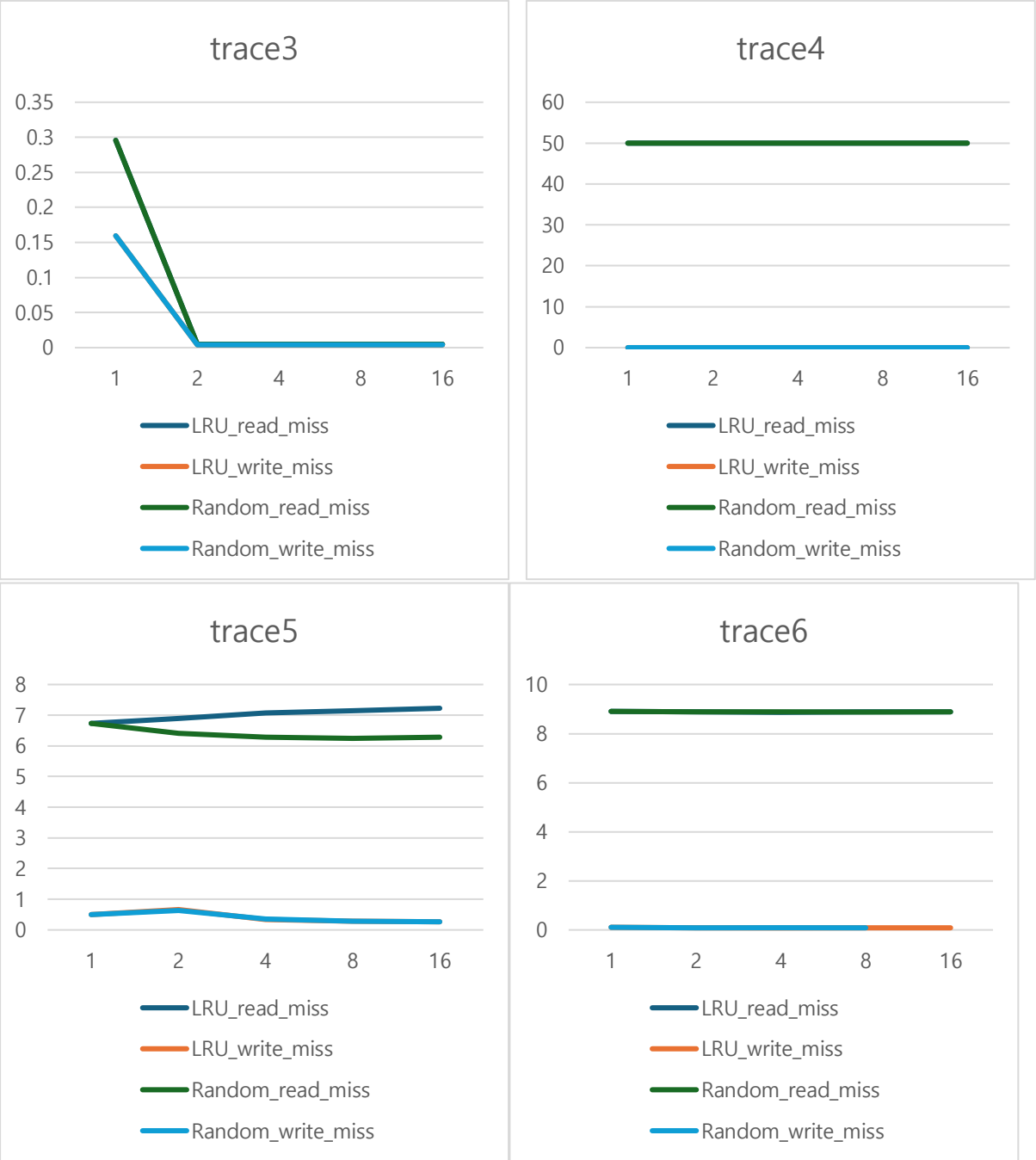
인할 수 있다. 이 결과는 block_size에 따른 miss rate을 확인해본 뒤의 실험에서 재해석할 수 있으며, spatial locality에서 stride가 커서 miss rate이 크게 변하지 않는다고 생각할 수 있다.

이 다음 실험으로는 associativity에 따른 read, write miss rate을 살펴본다.

여기서 L2 cache에 대하여 capacity는 256KB, block_size = 32로 설정하여 associativity 1, 4, 8, 16에 대하여 LRU, Random의 각 경우에 대하여 살펴본다.

x축이 associativity, y축이 miss rate(%)이라 볼 수 있다.





이와 같은 결과가 나온다는 사실을 알 수 있다.

여기서 trace 1, 3을 제외한 trace에 대해서는 miss rate이 크게 변하지 않았다. 하지만 trace 1, 3에 대해서는 associativity가 증가할수록 miss rate이 감소하는 경향을 확인할 수 있었다.

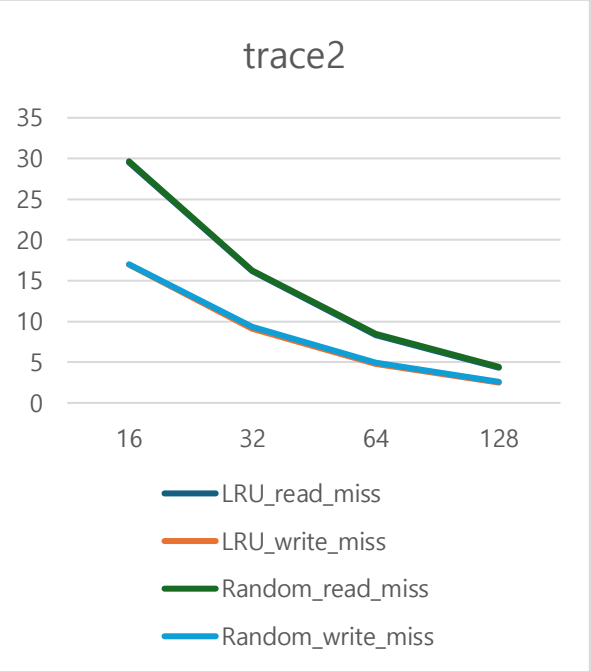
Trace 1, 3을 제외한 trace에 대해서 miss rate이 잘 변하지 않는 결과에 대해서 spatial locality 입장에서 stride가 너무 크거나, associativity 입장에서 서로 다른

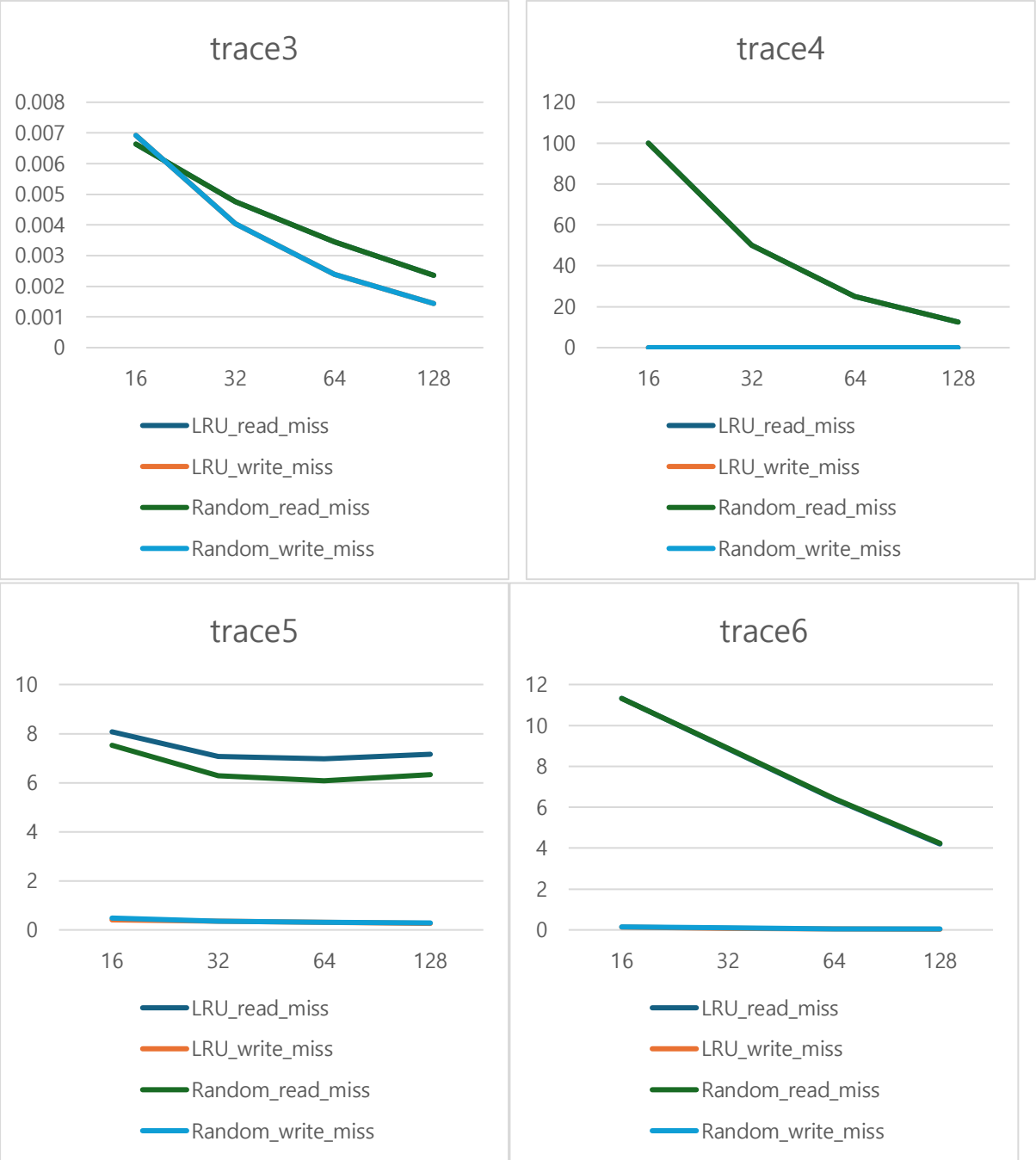
set에 있는 data에 더 많이 접근해 miss rate가 잘 변하지 않는다고 해석해볼 수 있다.

이 다음 실험으로는 associativity에 따른 read, write miss rate을 살펴본다.

여기서 L2 cache에 대하여 capacity는 256KB, associativity = 4로 설정하여 block_size 16, 32, 64, 128에 대하여 LRU, Random의 각 경우에 대하여 살펴본다.

x축이 block_size(Byte), y축이 miss rate(%)이라 볼 수 있다.





이와 같은 결과가 나온다는 사실을 알 수 있다.

해당 결과에 대해서 이전에 잘 변하지 않았던 trace 2, 4, 6의 miss rate이 감소한다는 사실을 알 수 있다.

따라서 이전에 고려해봤던 가설처럼 trace 2, 4, 6의 spatial locality의 stride가 커서 miss rate이 잘 변하지 않았고, 이번 실험에서 block_size를 바꿔가며 실행하여, block_size가 커졌을 때, 높은 stride에 대해 더 많은 hit가 발생하게 되어 miss

rate의 감소를 확인할 수 있었다.

또한 수업시간에 배웠던 내용에 기반하여 살펴보면 trace 1, 5에서 miss rate이 증가하는 현상은 block size가 너무 커졌기 때문에 block의 수가 줄어 miss rate이 증가하였다고 해석할 수 있다.