

● 자신이 작성한 과제에 대한 간략한 설명

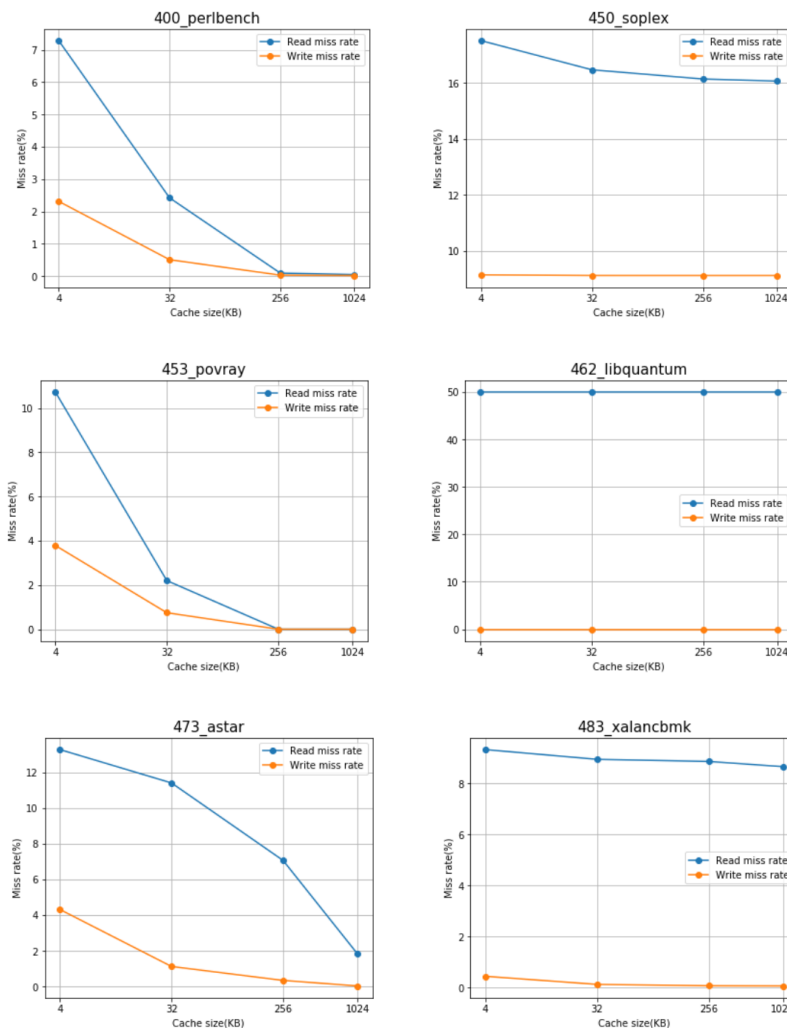
- 실행 순서: main함수에서 c, a, b 옵션이 잘 들어왔는지 확인하고 하나라도 누락된 옵션이 있다면 에러를 발생시키고 종료된다. 잘 들어왔다면, 입력 트레이스 파일의 내용을 한 줄씩 읽어오며, 현재 block을 W(write)하는지, R(read)하는지 확인하며 출력해야 할 변수들의 값들을 저장하며 동작한다. 끝으로 .out파일을 출력하고 종료된다.
- 구체적인 구현: single level 캐시를 효율적으로 구현하기 위해, Map[index][tag]=실행순서, Map[tag+index]=dirty bit (1 or 0)와 같은 두개의 map 자료구조를 활용했다. if문을 통해 트레이스 파일 내용의 한 줄을 읽어올 때마다 R,W를 확인하고 Map[index].find(tag)를 통해 해당 물리주소의 tag가 저장되어있는지 확인하고(해당 block이 cache에 caching되어 있는지 확인), tag가 저장되어 있지 않은 상태라면, miss이기 때문에 새롭게 tag를 추가하는 방식으로 block을 들고오는 것을 구현하였다. 이때, set별로 정해진 associativity 즉, way수를 넘어가는지 먼저 확인하여서 이미 해당 set의 모든 way에 block이 꽂 차 있는 상태면, LRU 방식을 이용해서 가장 오래전에 접근된 시각을 가진 block을 제거하고(즉, tag를 제거) 새로운 block을 추가(tag 추가)하였다. 또한, W(write)의 경우 해당 block의 dirty bit를 1로 설정하고, R의 경우 새로운 block을 추가할 때는 block의 dirty bit를 0으로 들고오고, 기존에 caching되어 있는 block을 읽을 때는 dirty bit를 건드리지 않고 기존 block의 접근 실행시각(실행순서)만 변경하도록 하였다. 위의 모든 실행과정에서 두개의 map 자료구조를 같이 수정하면서 cache 동작을 구현하였다. 그리고 과정 중에 전체 접근, 읽기 접근, 쓰기 접근, 읽기 실패, 쓰기 실패 횟수를 기록하였다. 기존의 블록을 교체할 때도 dirty bit가 0으로 설정된 블록을 교체하는 경우에는 clean eviction 횟수를 증가시키고 dirty bit가 1로 설정된 블록을 교체할 때는 dirty eviction 횟수를 증가시켰다. 마지막 파일 출력은 과제 안내 pdf의 양식과 checksum pseudo code를 그대로 따랐다.

### ● 트레이스 파일 및 패러미터에 따른결과 분석

3개의 패러미터(Capacity, associativity, block size)의 변화에 따른 miss rate를 관찰하기 위해 아래의 3가지 경우를 나눠서 결과를 분석하였다.

(1) Capacity 변경(연관 정도, 한 블록의 크기는 고정):

아래 그래프는 과제안내 pdf와 같이, Associativity를 4, block size를 32B로 고정하고, 캐시 용량을 4, 32, 256, 1,024 KB로 변경해가며 6개의 트레이스 파일을 실행하여 퍼센트 단위로 읽기 실패율(파란색)과 쓰기 실패율(주황색)을 기록한 결과이다. (Cache size =Capacity)

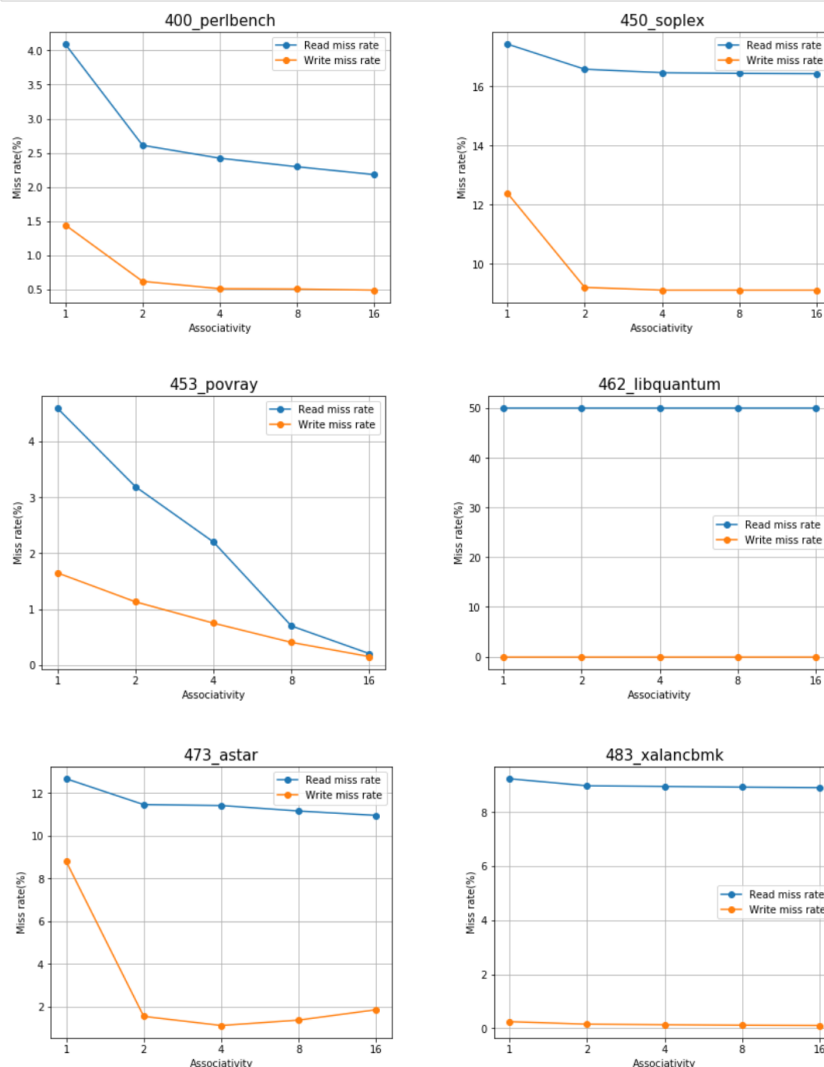


X 축은 Cache size(KB)를 의미하며, Y 축은 Miss rate(%)로, 파란선이 Read miss rate를 나타내고 주황선이 Write miss rate를 나타낸다. Cache size가 증가함에 따라 miss rate가 감소하는 경향이 있음을 알 수 있다. Cache size가 크면 더 많은 데이터를 caching하여 cache로 들고올 수 있기에 locality를 더 잘 활용하여 전체적인 miss rate를 줄일 수 있음을 확인할 수 있다. 462\_libquantum의 경우, 일정한 miss rate를 보이는데, cache size가 증가함에 따라 read miss rate가 50%로 일정하게 유지되면서도 write miss rate는 거의 0%이다. Write 하는 block들은 cache size에 상관없이 이

미 cache에 block이 존재하는 데이터들을 write하는 경우가 많음을 의미한다. Read miss가 cache size에 상관없이 50%로 일정한데, 이는 block을 read하는 경우, 내가 읽으려는 block이 없는 경우가 자주 발생함을 의미한다. 이는 한 set내에서 자주 block의 교체가 일어나기 때문에 발생할 수 있다. 실제 파일의 물리주소들을 확인해보면 random하지 않고 sequential하고 index bit(set bit)가 같은 물리주소들을 많이 발견할 수 있다.

## (2) Associativity 변경:

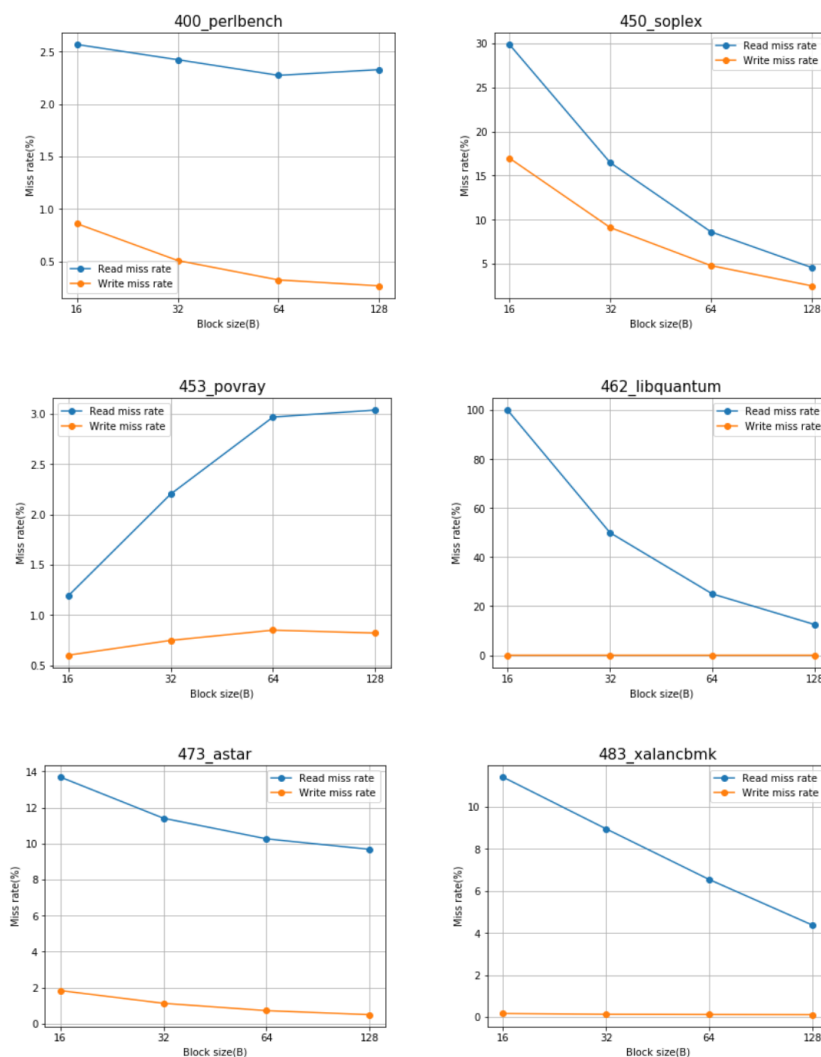
Capacity를 32KB로, block size를 32B로 고정한 후, Associativity를 1,2,4,8,16으로 변화시키며 아래의 그래프와 같은 결과를 도출하였다. (capacity를 16KB로 block size를 16B로 고정시켜서도 진행했는데, miss rate경향이 전체적으로 작은 용량에도 영향을 받았기에, 적절한 고정크기로 capacity 32KB와 block size 32B를 설정하였다.)



X 축은 Associativity를 의미하며, Y 축은 Miss rate(%)로, 파란선이 Read miss rate를 나타내고 주황선이 Write miss rate를 나타낸다. Associativity가 증가함에 따라 miss rate가 전체적으로 감소함을 알 수 있다. Associativity를 증가시키면, 그만큼 cache내의 공간을 유연하게 사용함을 의미한다.

Associativity를 증가시키면, Set 안에서 block이 들어갈 수 있는 way의 수가 증가하고, set 내에서도 주소에 상관없이 way로 갈 수 있기 때문에 공간을 유연하게 사용할 수 있다. 따라서 이러한 공간 유연성 덕분에 그래프의 결과와 같이 전체적으로 miss rate가 감소한다고 설명할 수 있다. 473\_astar의 경우 write miss rate가 associativity 4에서부터는 증가한 것을 알 수 있는데, 이는 동일한 cache size내에서 associativity가 증가함에 따라 set의 수가 감소하였기 때문에 cache내에 주소로 인해 결정되는 set (index bit)의 충돌이 많아졌기 때문으로 설명할 수 있다. (다른 파일에서는 괜찮았지만, 473\_astar는 증가된 associativity로 인해 감소한 index bit 수 때문에, index bit가 같지만 tag비트가 다른 물리주소들의 write횟수가 많기 때문이라고 설명할 수 있다.) 483\_xalancbmk의 경우, 일정해 보이지만 associativity가 증가함에 따라 조금의 감소 경향이 보이며, 462\_libquantum의 경우 (1) 경우의 설명과 같이, 파일내에 물리주소들이 sequential하기 때문에 발생한 케이스라고 말할 수 있다.

(3) Block size 변경: capacity는 32KB로 Associativity는 4로 고정하고, Block size를 16,32,64,128B로 변화시켰을 때의 결과를 살펴보았으며 결과는 아래의 그래프와 같다.



X 축은 Block Size(B)를 의미하며, Y 축은 Miss rate(%)로, 파란선이 Read miss rate를 나타내고 주황선이 Write miss rate를 나타낸다. 453\_provray를 제외하고 전체적으로 block size가 증가함에 따라 miss rate가 감소한다는 것을 알 수 있다. 이는 Block size를 크게 하면, 물리주소로 접근한 데이터 근처의 더 많은 word를 한 block안에 넣어서 들고올 수 있기 때문에 spatial locality를 더 잘 활용할 수 있기에 miss rate가 감소되는 것으로 설명할 수 있다. 453\_provray의 경우, spatial locality를 활용하지 못해 block size 너머로 데이터를 접근하는 즉, random한 접근을 miss rate가 증가한 원인으로 꼽을 수 있다. 462\_libquantum의 경우, (1)과 (2)의 결과와 달리 현저히 miss rate가 낮아지는 것을 알 수 있는데, 이는 해당 파일의 R과 W가 물리주소들을 sequential하게 접근하는 경우인 만큼, spatial locality가 강하기 때문에 block size를 증가시킴으로써 miss rate를 해결할 수 있음을 다시한번 확인시켜 준다. 이외에도 일반적으로 Block size의 경우, 증가한다고해서 무조건적으로 miss rate가 줄어들지 않는다. 동일한 capacity내에서 block의 size를 늘리면 block의 개수 자체는 줄어들어 적은 수의 block을 유지하기 때문에 miss rate의 수가 많아지는 경우가 생길 수 있기 때문이다.

● (C, C++) 과제의 컴파일 방법 및 컴파일 환경(사용한 OS, 컴파일러 버전).

C++ 로 코드를 작성하였으며 VMware workstation 환경에서 g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0 버전을 이용하여 다음 명령어로 컴파일 하였다.

```
g++ -o main main.cpp
```

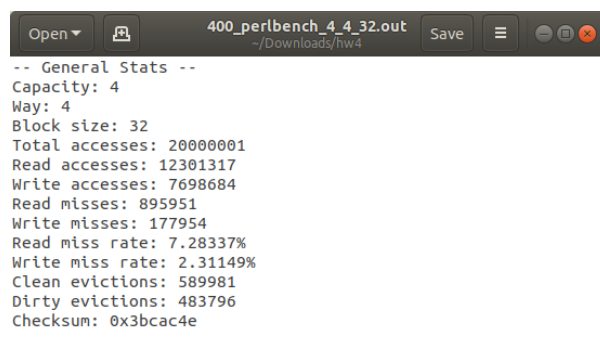
● 과제의 실행 방법 및 실행 환경

과제 안내 pdf예제 명령어들과 유사하게 다음과 같은 명령어들로 실행하였으며, 입력 파일이 들어있는 파일경로에서 아래의 명령어를 이용하여 실행하였다.

예시 출력옵션과 출력파일의 내용은 다음과 같다.

Ex1) ./main -c 4 -a 4 -b 32 400\_perlbench.out

출력파일:



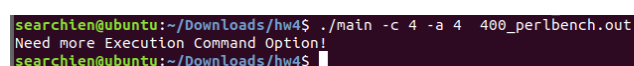
```

400_perlbench_4_4_32.out
~/Downloads/hw4
-- General Stats --
Capacity: 4
Way: 4
Block size: 32
Total accesses: 20000001
Read accesses: 12301317
Write accesses: 7698684
Read misses: 895951
Write misses: 177954
Read miss rate: 7.28337%
Write miss rate: 2.31149%
Clean evictions: 589981
Dirty evictions: 483796
Checksum: 0x3bcac4e

```

EX1) ./main -c 4 -a 4 400\_perlbench.out

옵션 중 하나라도 없으면 에러가 발생하는데, 에러발생의 예시는 아래와 같다.



```

searchtien@ubuntu:~/Downloads/hw4$ ./main -c 4 -a 4 400_perlbench.out
Need more Execution Command Option!
searchtien@ubuntu:~/Downloads/hw4$

```