CSE305 Project 3: Multi-level Cache Model and Performance Analysis

201811011 곽지혁

- Ubuntu 20.04.5 LTS 환경에서 gcc 9.4.0 버전을 이용하여 다음 명령어로 컴파일하였다.

  gcc -o <runfile> project4.c -lm

- 실행 명령어는 다음과 같다.

  $ ./runfile <-c capacity> <-a associativity> <-b block_size> <-lru 또는 -random> <tracefile>

- 명령어 Option에 관한 설명은 다음과 같다.

  -c capacity: L2 캐시의 전체 용량을 KB 단위의 capacity(4-1024)로 지정한다.
  -a associativity: L2 캐시의 연관 정도를 associativity(1-16)로 지정한다.
  -b block_size: 캐시의 블록 크기를 바이트 단위의 block_size(16-128)로 지정한 다.
  -lru: Cache replacement 정책으로 LRU를 사용한다.
  -random: Cache replacement 정책으로 Random을 사용한다.

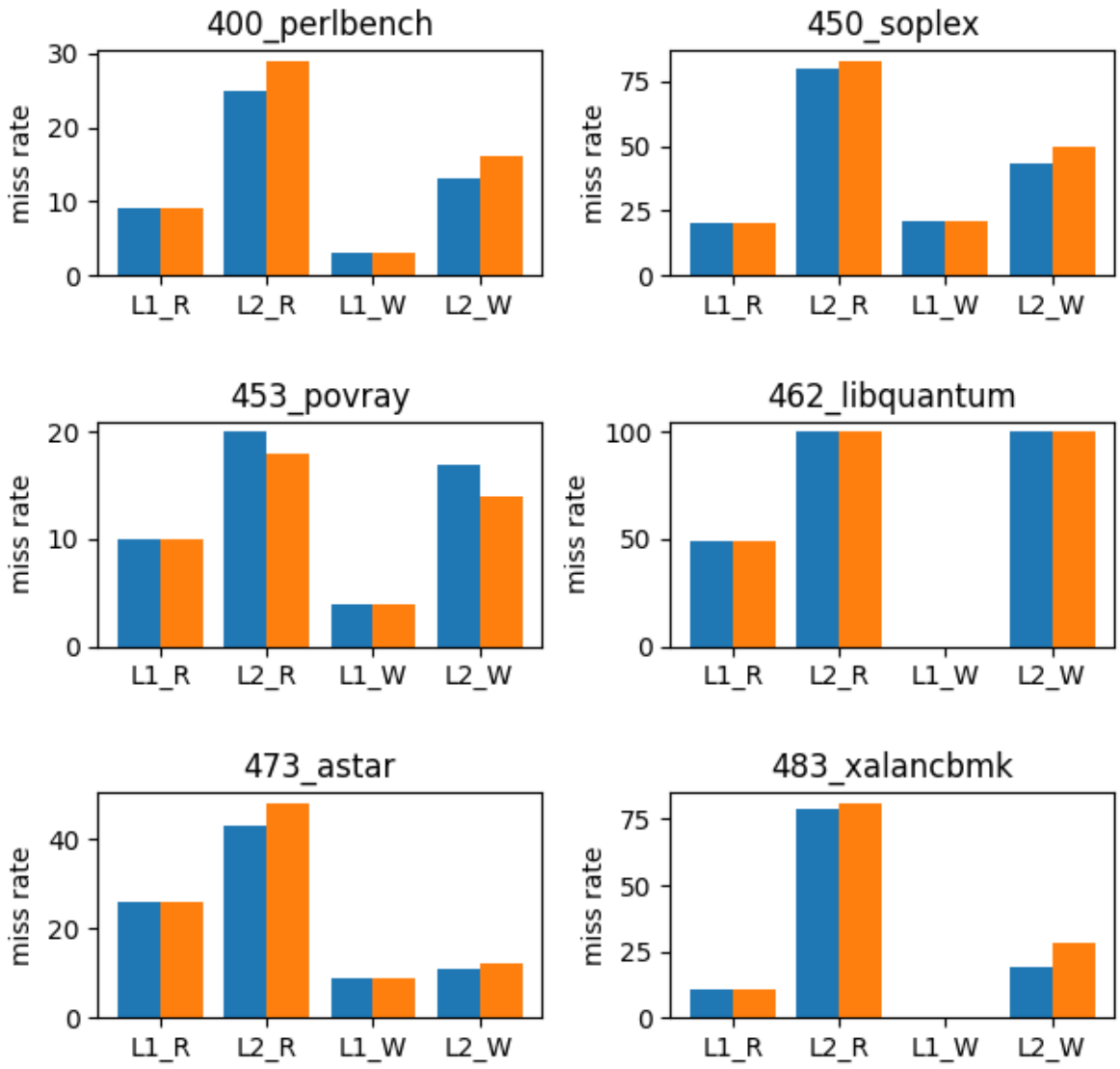
  - L1 cache의 capacity와 associativity는 각각 L2 cache의 capacity와 associativity를 4로 나눈 값과 같다. L2의 associativity가 2이하인 경우 L1과 L2의 associativity가 같다고 가정한다. (i.e., 2일경우 L1, L2의 associativity가 동일하 게 2가 된다). L1의 block size의 경우 L2의 block size와 동일하다.

- 전체적인 Flow는 다음과 같다.

1. 인자로 받은 옵션들을 저장하고 인자의 크기에 맞추어 L1, L2를 구현한다. 이때, 해당 캐시와 관련한 값들 capacity의 크기, way, block size, index_bit 사이즈 등은 CacheData라는 구조체를 이용해 저장한다. Cache block 구조체는 valid, dirty, access(시간을 체크하기 위해), tag 등으로 구성되어 있다.
2. 인자로 받은 파일을 열고 while문을 통해 한 줄씩 읽어온다.
3. addr에서 L1의 index와 tag값을 이용해 L1에서 해당 값을 찾는다. 만일 값이 있다면 hit이므로 while 반복문을 계속 시행한다.
4. L1 miss가 났다면, 해당 캐시 블락을 L1에 추가해 준다. 빈 블락이 없다면, lru, random을 이용해 제거한 후에 추가한다.
5. addr에서 L2의 index와 tag값을 이용해 L2에서 해당 값을 찾는다. 만일 값이 있다면 hit이므로 while 반복문을 계속 시행한다.
6. miss가 났다면, 해당 캐시블락을 L2에 추가해 준다. 빈 블락이 없다면, lru, random을 이용해 제거한 후에 추가한다.
7. 제거된 블락은 L1에서도 제거 되어야하므로 L1에서 해당 블락을 찾아 제거한다.
8. 모든 명령어를 수행했다면, 결과를 out_file에 기록한다.
9. 종료.

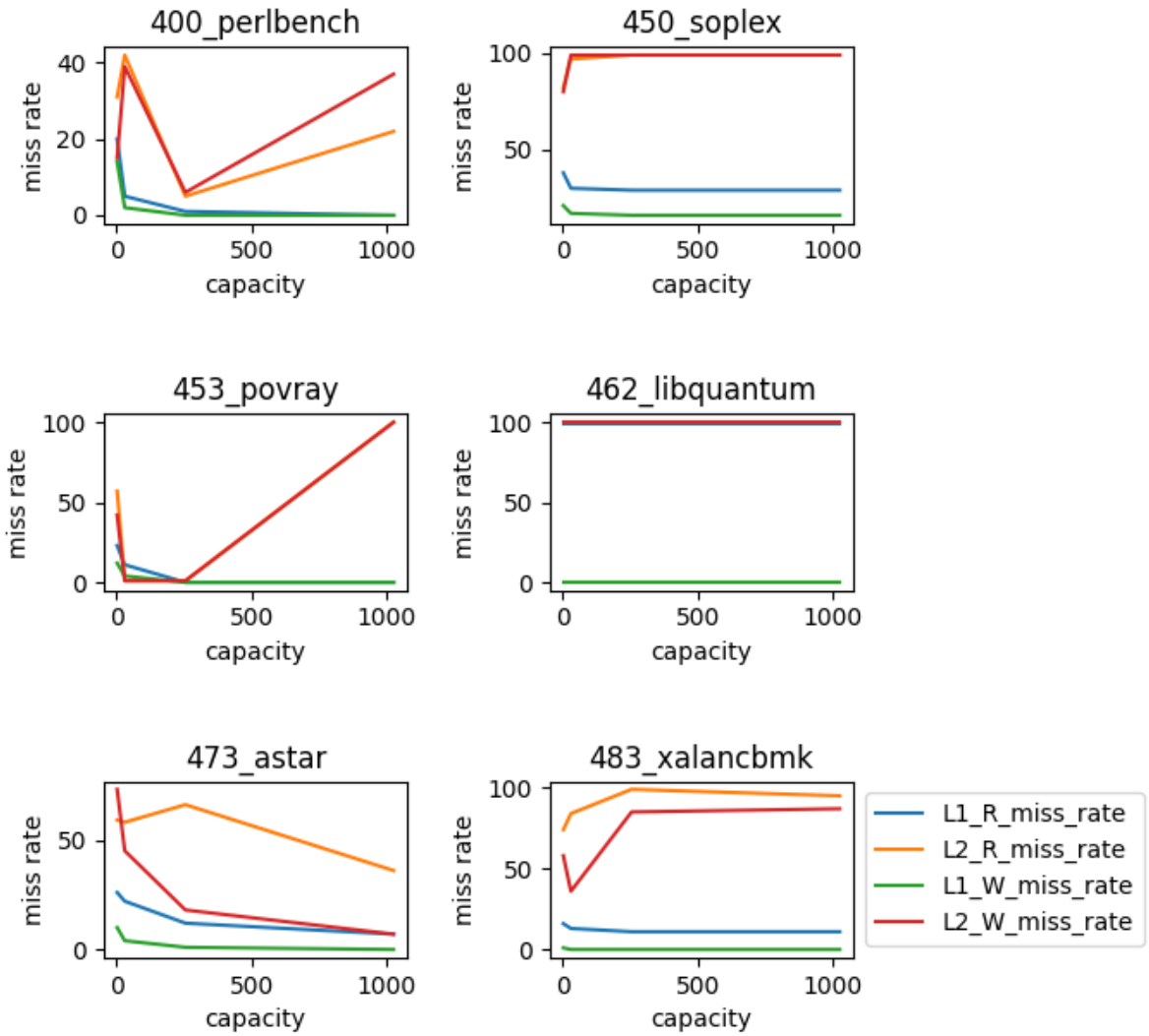
<결과 분석>

- 다음은 capacity 32, associativity 4, block_size 32로 고정한 후에 lru, random 옵션을 주어 trace별로 miss rate를 비교해본 결과이다. (파란색: lru, 주황색: random)



출력결과, 400_perlbench, 450_soplex, 473_astar, 483_xalancbmk의 경우 lru를 이용한 방식의 miss rate가 더 적었고, 453_povray의 경우 random이, 462_libquantum는 서로 같음을 알 수 있다. 해당 결과를 통해 대부분의 파일은 Temporal Locality에 의해 lru를 이용한 방식이 miss rate를 줄이는 데 효과적이라는 것을 알 수 있다. 또한, 453_povray, 462_libquantum처럼 Temporal Locality가 적용되지 않는 경우도 있음을 알 수 있다.

- 다음은 associativity 8, block_size 16, lru option을 고정으로 제공하고 capacity를 4, 32, 256, 1024로 여러 값을 주며 trace별로 miss rate를 비교해본 결과이다.



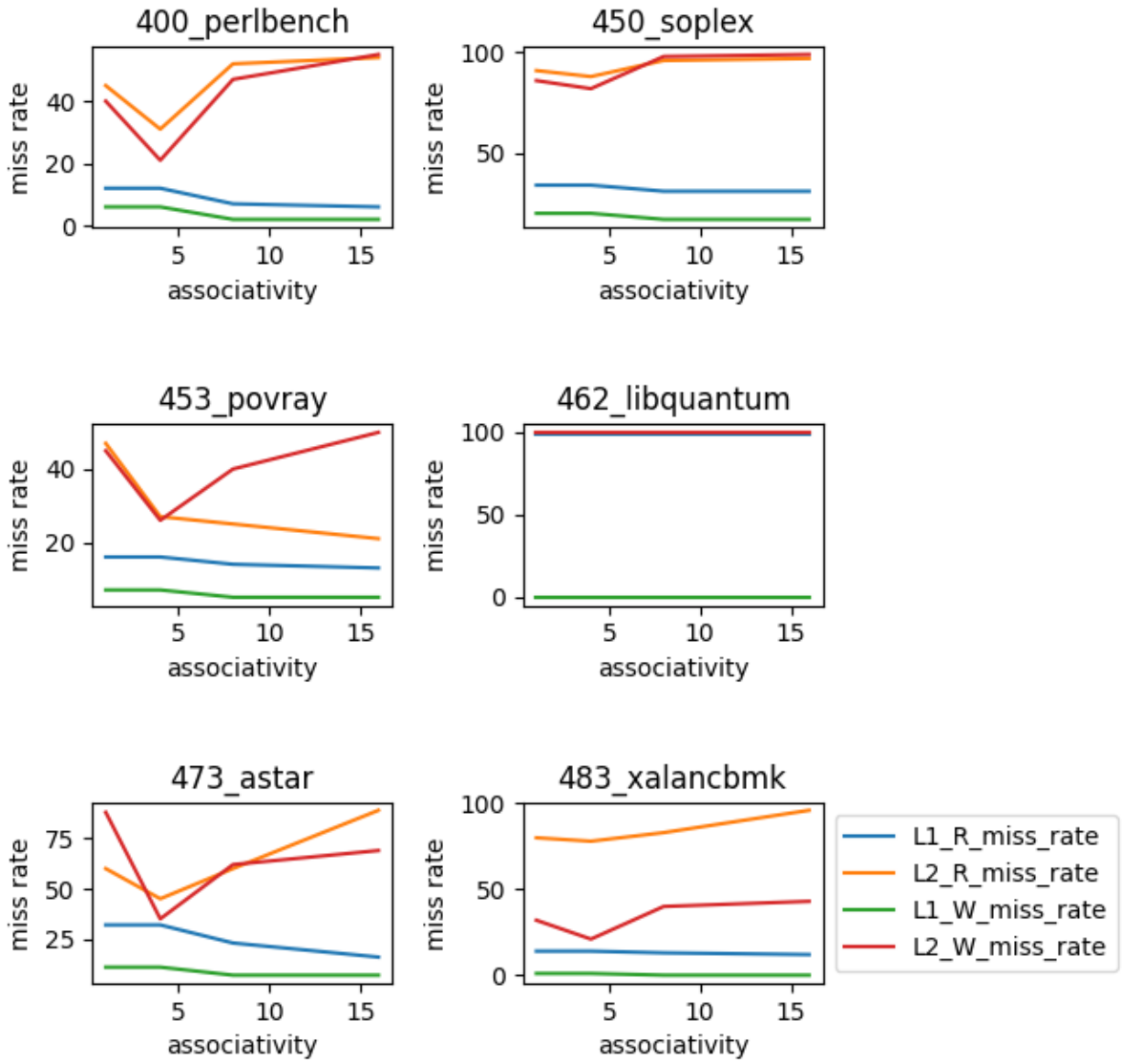
출력결과, L1의 miss rate들이 capacity가 증가함에 따라 줄어든 모습을 확인할 수 있었다. 따라서 cache의 사이즈가 커짐에 따라 많은 block들을 저장할 수 있기에 miss rate가 줄어든다는 것을 알 수 있다. L2의 miss rate의 경우 global miss rate가 아닌 local miss rate이다. 대부분의 L2 miss rate가 상승한 것은 L1에서 miss 난 것이 L2에서도 그대로 miss가 발생했다는 의미이다. 즉 capacity가 증가함에 따라 L1에서 locality를 가진 명령어들을 대부분 hit처리 했고, L2에서는 locality가 없는 명령어들만 처리하게 되어 miss rate가 높아진 것으로 예상된다. 1024의 크기에도 점점 줄어들고 있는 473_astar 같은 경우 cache 사이즈가 더 커진다면 더 많은 miss를 처리할 수 있다는 가능성을 내포하고 있다.

462_libquantum 의 경우 L2 R miss rate 가 0 을 가지며 그 외에는 100% 값을 가졌다. 해당 조건에서 capacity 256 을 가질 때를 결과를 출력해 보니 아래의 결과가 도출되었다. 이를 해석해 보면, L1 의 read 의 경우 L1 에서 대부분 miss 가 발생했으며, L2 에서도 그대로 miss 가 발생했다. Write 의 경우 miss 가 대부분 발생하지 않았으며 L1 의 miss 가 L2 에서 그대로 발생함을 알 수 있었다.

이를 통해 462_libquantum의 read 명령어는 대부분 이전에 불리지 않은 주소의 명령어가 수행되어 temporary locality가 낮으며, write 명령어는 매번 같은 주소의 명령어가 수행되어 locality가 매우 높다는 것을 알 수 있다.

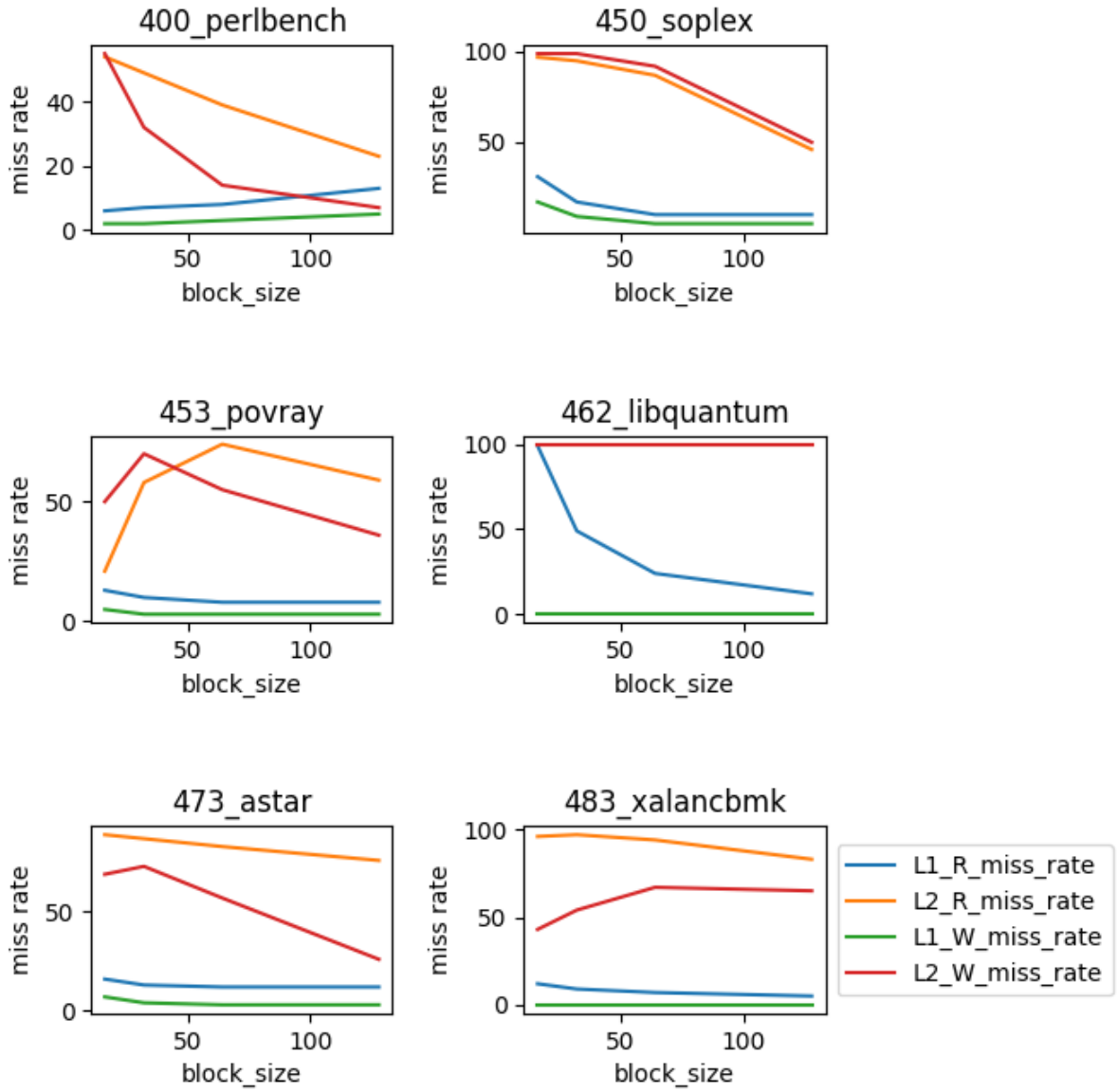
```
L1 Capacity: 64
L1 Way: 2
L2 Capacity: 256
L2 Way: 8
Block Size: 16
Total accesses: 20000001
Read accesses: 14579599
Write accesses: 5420402
L1 Read misses: 14579488
L2 Read misses: 14579488
L1 Write misses: 61
L2 Write misses: 61
L1 Read miss rate: 99%
L2 Read miss rate: 100%
L1 Write miss rate: 0%
L2 Write miss rate: 100%
L1 clean eviction: 9159238
L2 clean eviction: 9159238
L1 dirty eviction: 5416215
L2 dirty eviction: 5403927
```

- 다음은 capacity 16, block_size 16, lru option 을 고정으로 제공하고 associativity 를 1, 4, 8, 16 으로 여러 값을 주며 trace 별로 miss rate 를 비교해본 결과이다.



실행결과, L1의 miss rate들이 associativity가 증가함에 따라 줄어든 모습을 확인할 수 있었다. 따라서 cache의 사이즈가 커짐에 따라 많은 block들을 저장할 수 있기에 miss rate가 줄어든 다는 것을 알 수 있다. L2의 miss rate의 경우 local miss rate이다. associativity가 증가함에 대부분의 locality를 가진 명령어가 L1에서 hit 처리되기에, L2에는 점점 locality가 없는 값들만 access된다. 따라서 asoociaivity에 따른 miss rate가 일정 수준까지는 감소하다가 그 이상부터는 증가함을 알 수 있다.

- 다음은 capacity 16, associativity 16, lru option을 고정으로 제공하고 block size를 16, 32, 64 ,128으로 여러 값을 주며 trace별로 miss rate를 비교해본 결과이다.



Block size는 spatial locality와 연관되어 있다. block size가 커질수록 연관된 주위의 값들을 같이 저장하기에 spatial locality가 증가하게 된다. 따라서 해당 그래프에서 대다수의 L1은 miss rate가 줄어드는 것이 확인된다. 하지만 400_perlbench의 경우 오히려 증가하는 모습을 보이는 데, 이는 block size를 키우면 한 블락에 저장되는 양이 많아져 spatial locality이 증가하지만, 반대로 블락의 수는 줄어들어, temporal locality은 덜 반영될 수도 있다. 즉 400_perlbench은 spatial locality보다 temporl locality가 더 높다는 것을 확인할 수 있다. 반대로 462_libquantum은 miss rate가 확연히 줄어드는 것으로 보아 spatial locality가 높다는 것을 확인할 수 있다.