Computer Architecture Project 3 : Simulating Pipelined Execution

201911106 유제이

1. Files

소스파일은 아래와 같다.

1) cache.cpp

: cache.cpp는 main function이 있는 소스파일로, 프로그램 작동을 위해서는 해당 파일을 컴파일 및 실행해야 한다.

2) Contents.h

: Contents.h는 content class를 담고있는 헤더파일로, 각 trace file 내의 physical memory address, 해당 memory의 tag, validity, 그리고 dirty bit에 대한 정보를 담고있다.

2. Compile and Execution

해당 코드는 Window hyper-V를 이용한 가상 컴퓨터 내 Ubuntu 20.04 환경에서 g++ 9.4.0 ver을 사용하여 컴파일 후 실행하였다. 컴파일 및 실행 방법은 아래와 같다.

1) Compile

cache.cpp, 400_perlbench.out 등 실행 및 읽기에 관련된 파일이 있는 Directory에서 Open in terminal -> `g++ -o cache cache.cpp`을 입력한다.

2) Execution

Complie 후 다음과 같은 옵션을 조합하여 실행할 수 있다. -c, -a, -b, -lru/random 이외의 flag가 입력되면 "unknown flag inserted"를 출력하고 프로그램을 종료한다.

`./cache <-c capacity> <-a associativity> <-b block_size> <-lru 또는 -random> <trace file>`

1) `-c` : kB 단위로 L2 cache size를 설정한다. (4-1024, 2의 거듭제곱)

2) `-a` : L2 cache의 associativity를 설정한다. (1-16, 2의 거듭제곱)

3) `-b` : Byte 단위로 cache block size를 설정한다. (16-128, 2의 거듭제곱)

-c, -a, -b의 option 값이 설정된 범위 내에 해당하지 않거나 2의 거듭제곱 꼴이 아닌 경우 error message와 함께 프로그램이 종료된다.

4-a) `-lru` : -Cache replacement 정책으로 LRU를 사용한다.

4-b) `-random` : Cache replacement 정책으로 Random을 사용한다.

5) `trace file`: 파일 이름이 주어지지 않았을 때 "There's no file provided"를 출력하고 프로그램을 종료한다.

3. Flow

전체적인 플로우는 다음과 같다.

① 입력된 값들을 통해 출력 형식 결정 및 참조 파일 불러들이기

② 파일 내 정보를 읽어들이고 각 instruction의 모드 (W / R) 및 address 읽기

③ address값을 통해 offset bit, index bit, tag bit 추출, index 확인 후 각 모드에 따라 동작 수행

Offset bit : block size (B)를 2의 거듭제곱 꼴로 표현하여, 그 지수 만큼의 bit를 뒤에서 count하여 설정

Index bit : cache entry (cache size / (block size * way))를 2의 거듭제곱 꼴로 표현하여, 그 지수 만큼의 bit를 전체에서 offset bit를 제외한 bit의 뒤에서 count하여 설정

Tag bit : offset bit와 index bit를 제외한 나머지 bit들로 설정

④ 각 동작 별 hit / miss 에 따른 동작은 아래와 같다.

a) '**R**' (읽기 모드)

a-1) Read hit : read hit 횟수 count.

a-2) Read miss : read miss 횟수 count.

b) '**W**' (쓰기 모드)

a-3) Write hit : write hit 횟수 count. Dirty bit를 1로 set, Write back 구현

a-4) Write miss : write miss 횟수 count. Write allocate 방식, memory로부터 값을 가져온다. Cache에 값을 새로 update 해준다.

L2 cache는 Inclusive Cache로 구성하였다. L1, L2 모두 miss가 난 경우는 두 level 모두로 블록을 로드하였다.

⑥ Replacement

: cache index에서 해당 block이 다 찼을 경우, evict할 후보를 골라 값을 replace하는 방식은 아래와 같이 구현되었다.

1) LRU : 접근된 메모리와 같은 index의 다른 way에 있는 address들의 priority를 계산하여 evict될 후보 (LRU replacement 사용 시)를 계산한다.

2) Random : LRU와 상관 없이 random으로 evict할 way index를 고른다.

이때, evict되는 후보의 dirty bit를 보고 dirty = 1인 경우는 dirty eviction count를 +1 해주고, dirty = 0인 경우는 clean eviction count를 +1 해주었다.

⑦ 파일 저장 : 출력 파일명은 workloadName_capacity_associativity_bockSize.out 의 형식을 갖춘다.

4. Cache Performance Characterization

각 tracefile에 따라 같은 조건 실험 시 성능도 다 달랐다. 이는 컴퓨터의 성능을 실험하고자 할 때 같은 환경, 같은 instruction의 조건에서 실험을 해야 한다는 사실을 뒷받침해준다.

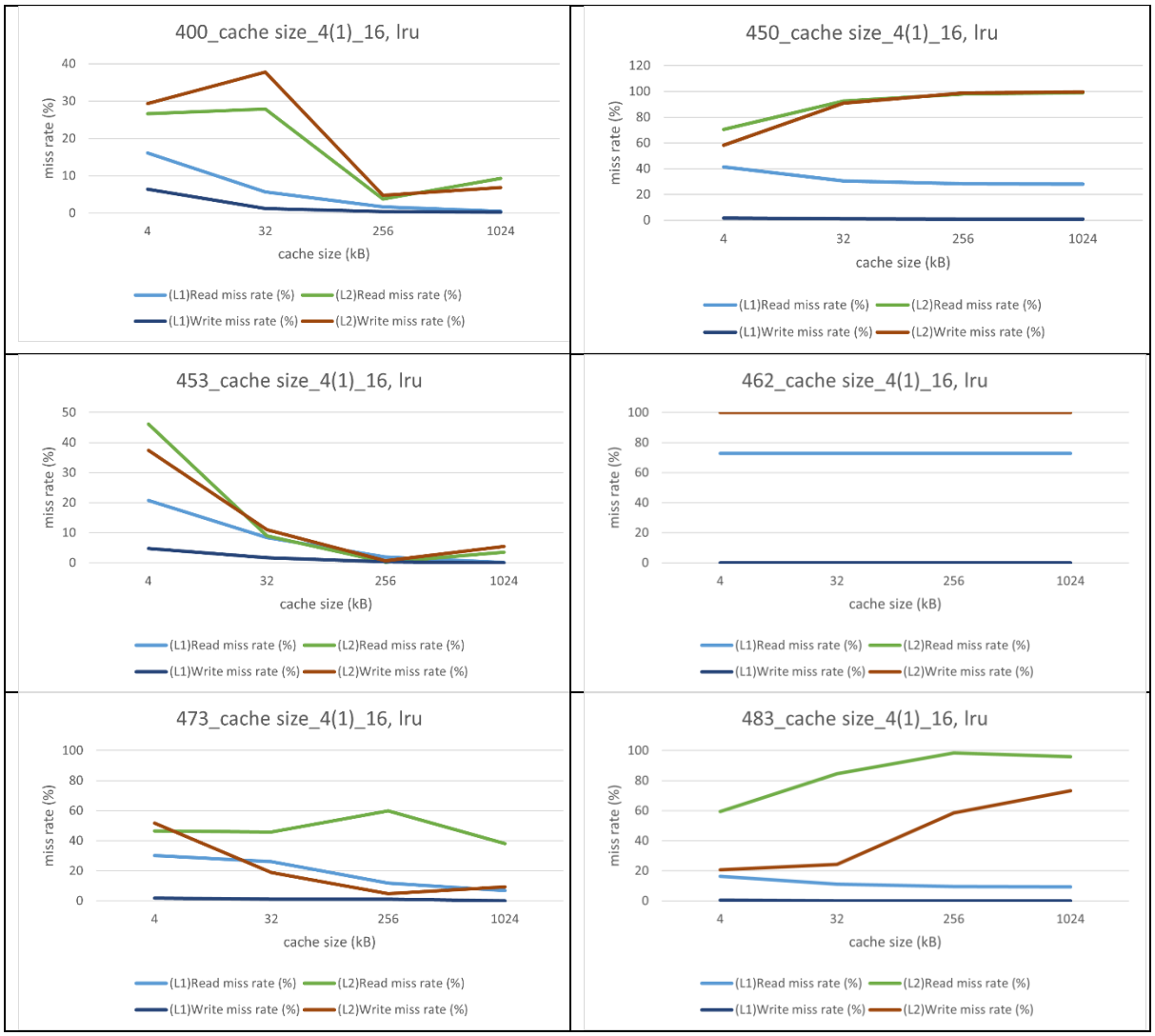
① Cache size variation : L2 associativity 4, Block size 16 B로 설정 후 각각 lru, random replacement를 적용하여 6개의 trace file에 따른 miss rate 변화를 아래와 같이 나타내었다.

Lru와 random 방식을 비교해 보았을 때, random에 의한 miss rate가 약간 더 높은 부분을 확인할 수 있었다. 그러나 전반적인 그래프의 개형과 miss rate의 대략적인 퍼센티지는 유사하다. Lru 방식에서의 miss rate가 더 낮은 모습을 보여주는 경우, 해당 trace file내의 data 사용이 temporal locality를 따르는 것으로 이해할 수 있다.

462 에서는 cache size의 변화에 의한 성능 변화가 없는 것으로 확인되었다. 이는 해당 trace file에서 새롭게 받는 data가 자주 등장한다는 것으로 예상해 볼 수 있다.

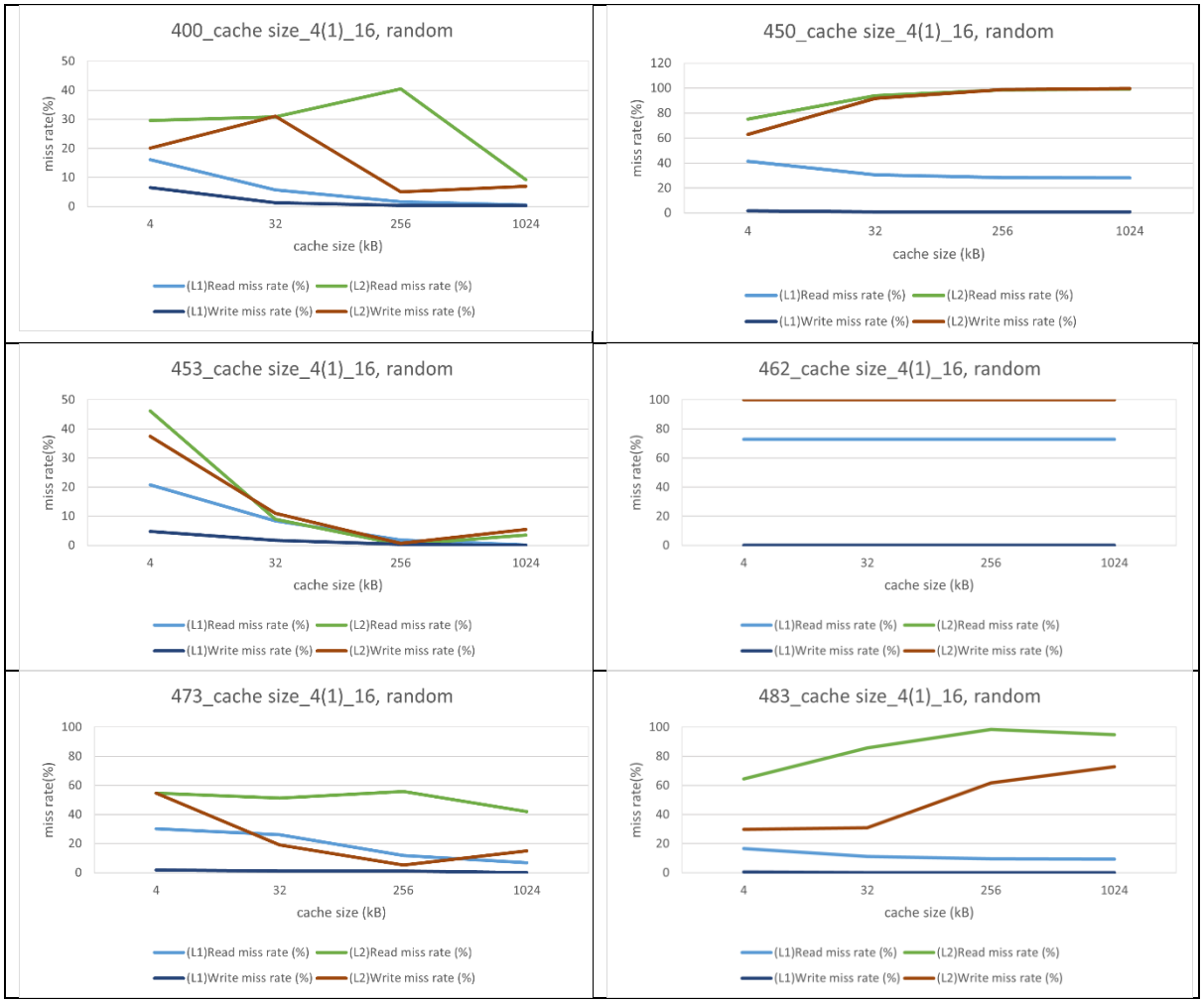
대체적으로 cache size가 증가함에 따라 cache 내 block의 개수가 증가하므로 miss rate는 감소하는 경향을 보인다. 그러나 다른 경향을 보이는 경우도 존재하는데, 이는 data access pattern, spatial locality에 따라 cache behavior가 다를 수 있음을 암시한다.

a. -lru



*462는 L1 Write miss rate, L2의 Read miss rate가 모두 0에 가까워 그래프가 겹쳐져있다.

b. -random



*462는 L1 Write miss rate, L2의 Read miss rate가 모두 0에 가까워 그래프가 겹쳐져있다.

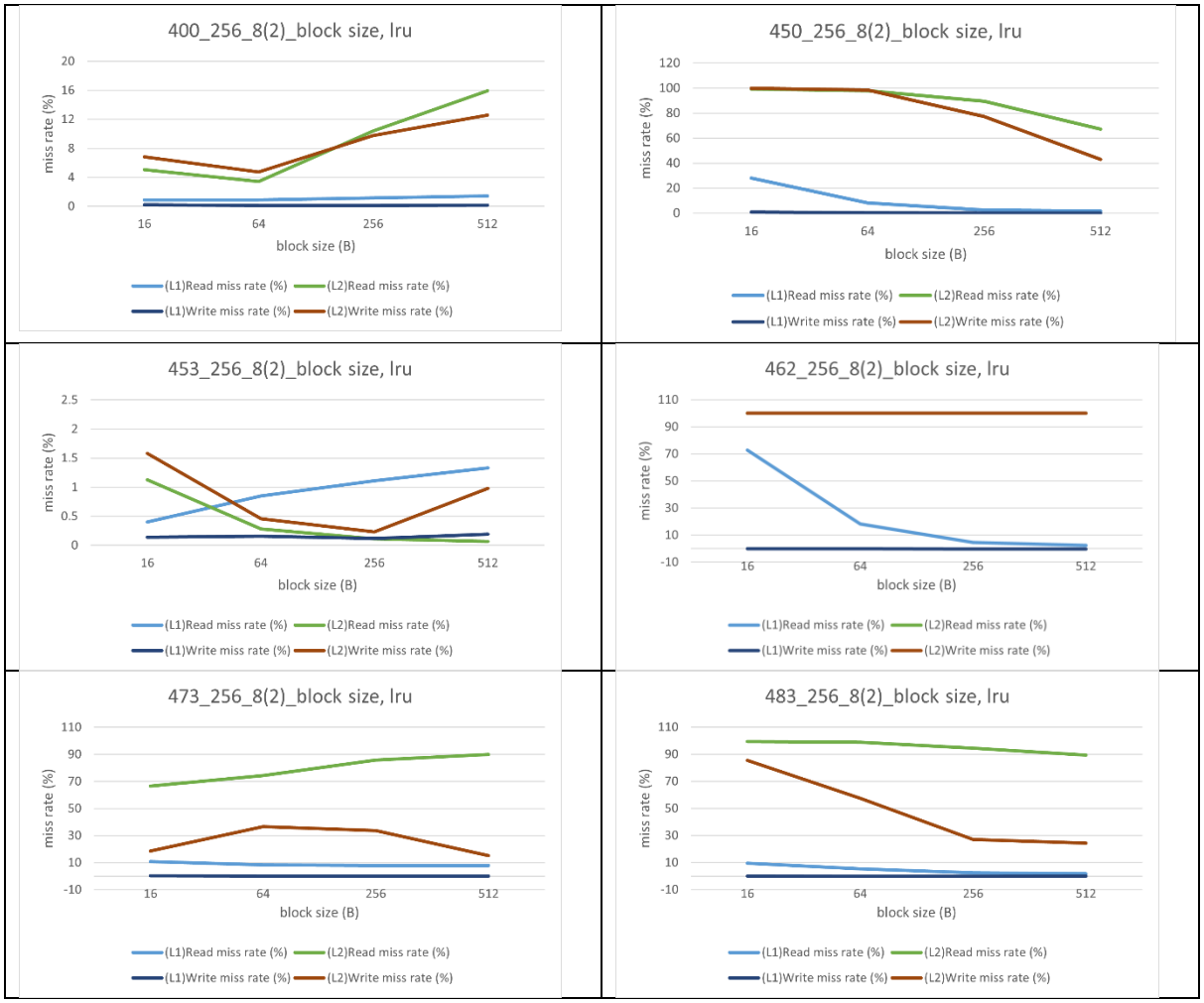
② Block size variation : Capacity 256 (kB), L2 Associativity 8 (L1 : 2)로 설정 후 block size를 각기 달리 설정하여 성능을 비교해보았다.

Lru와 random 방식을 비교해 보았을 때, lru에 의한 miss rate가 약간 더 높은 부분을 확인할 수 있었다. 이를 통해 lru 방식이 무조건적으로 miss rate를 낮추는 것은 아니라는 점을 알 수 있다. Lru는 가장 최근에 접근한 데이터를 쓸 확률이 높다는 가정 하에 구현된 것인데, 실제 data 사용은 다를 수 있기 때문이다. 전반적인 그래프의 개형과 miss rate의 대략적인 퍼센티지는 유사하다.

대체적으로 block size가 증가함에 따라 spatial locality가 증가하게 되므로 miss rate가 감소하는 추세를 확인할 수 있다. 그러나 400의 L2 Read/Write miss rate, 453의 L1 Read miss rate, 453의 256, 512 B block 조건의 L2 Write miss rate (in lru) 473의 L2 Read miss rate의 경우는 오히려 miss rate가 증가하는 모습을 보인다.

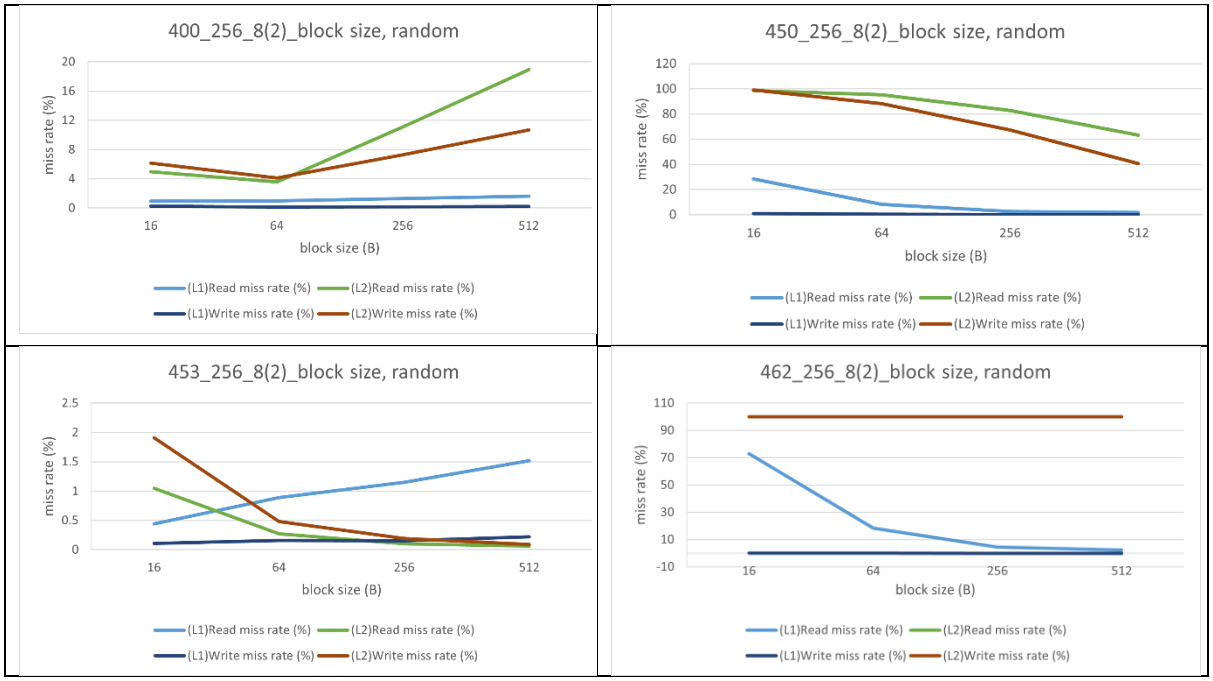
이는 Cache size에 비해 block size가 너무 크기 때문으로, cache 내에서 hold할 수 있는 block의 개수가 줄어듦에 따라 miss rate가 증가하는 것으로 생각해 볼 수 있다.

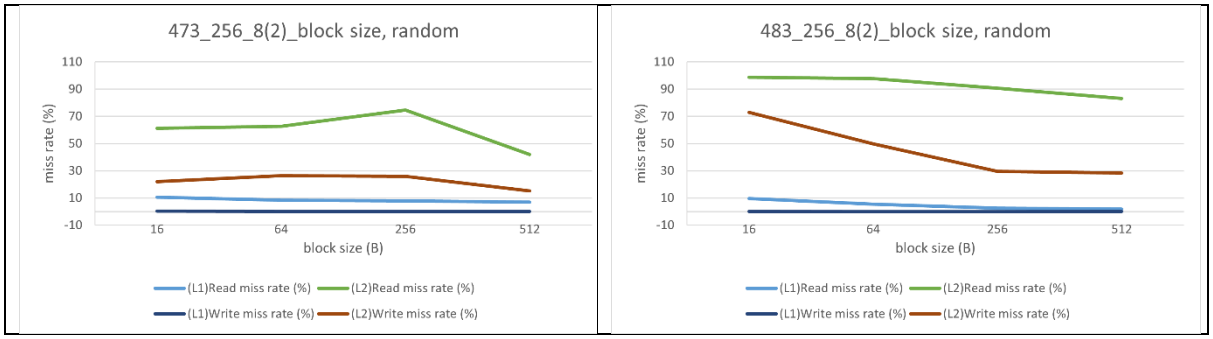
a. -lru



*462는 L1 Write miss rate, L2의 Read miss rate가 모두 0에 가까워 그래프가 겹쳐져있다.

b. -random





*462는 L1 Write miss rate, L2의 Read miss rate가 모두 0에 가까워 그래프가 겹쳐져있다.

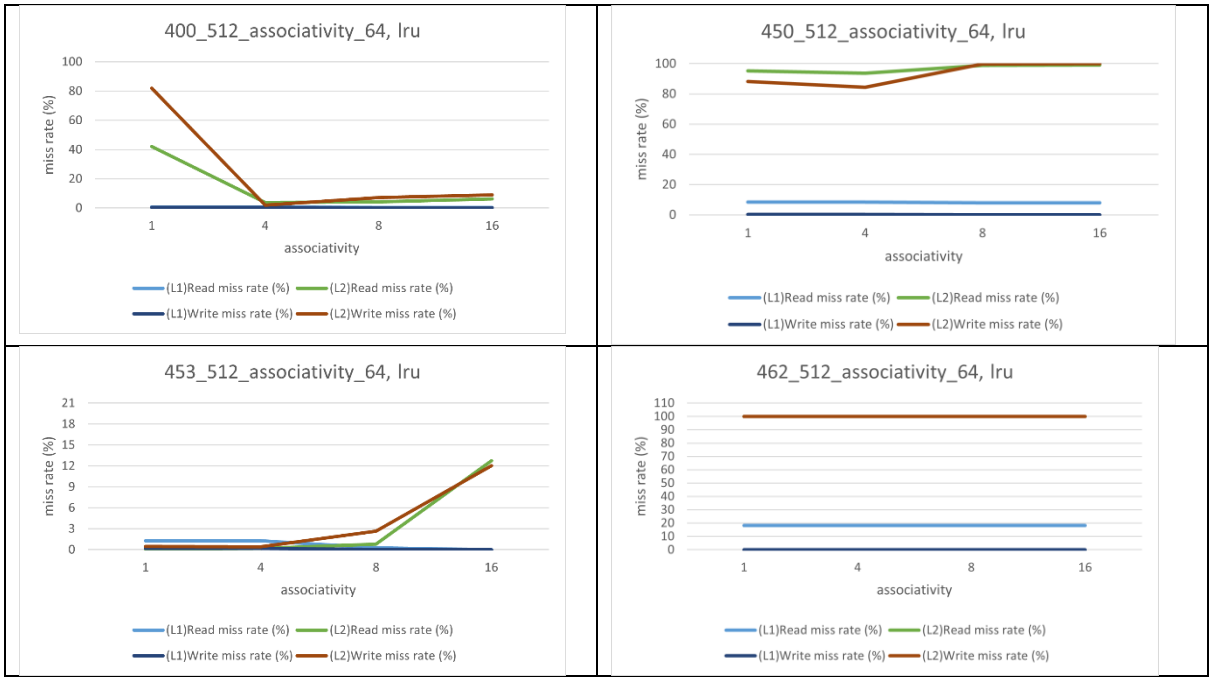
③ Associativity variation : Capacity 512 (kB), Block size 64 (B)로 고정한 뒤 associativity를 각기 달리하여 성능을 비교하였다.

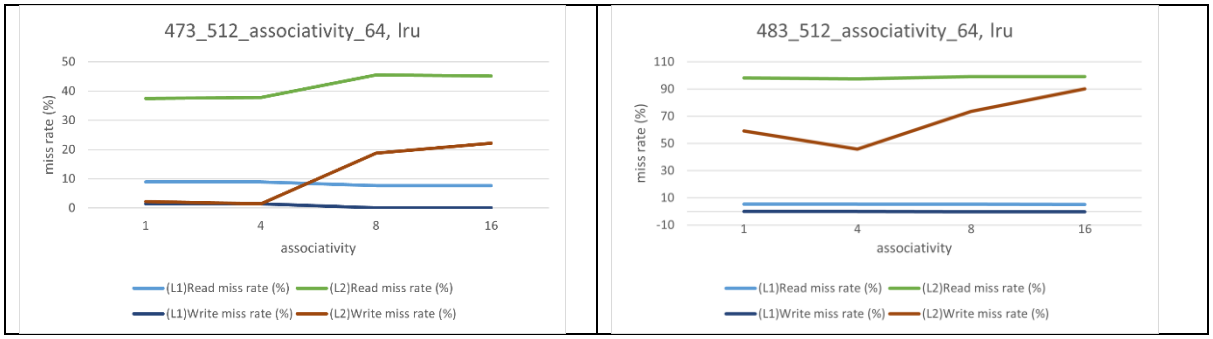
Lru와 random 방식을 비교해 보았을 때, 일부에서 random에 의한 miss rate가 약간 더 높은 부분을 (473), 일부에서는 lru에 의한 miss rate가 약간 더 높은 부분을 (453) 확인할 수 있었다. 그러나 전반적인 그래프의 개형과 miss rate의 대략적인 퍼센티지는 유사하다. Replacement 정책에 따른 miss rate의 차이에 관하여서는 위에서 기술하였다.

462 에서는 cache size의 변화에 의한 성능 변화가 없는 것으로 확인되었다. 이는 해당 trace file에서 새롭게 받는 data가 자주 등장한다는 것으로 예상해 볼 수 있다.

일반적으로 associativity의 증가는 spatial locality가 있는 경우 miss rate를 줄여준다는 점을 확인할 수 있다. 그러나 spatial locality가 좋지 못한 경우 associativity가 증가해도 miss rate는 그렇지 않다.

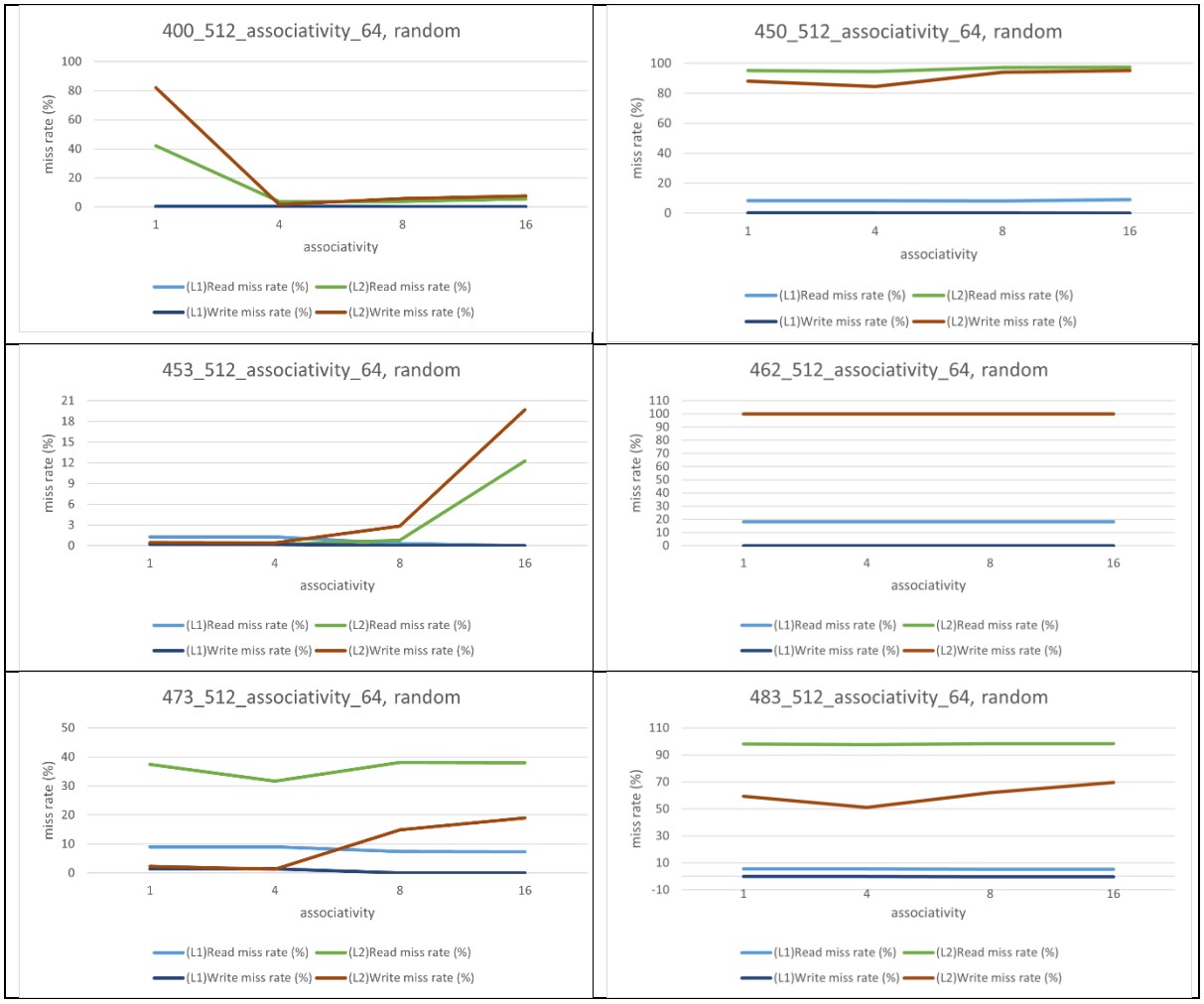
a. -lru





*462는 L1 Write miss rate, L2의 Read miss rate가 모두 0에 가까워 그래프가 겹쳐져있다.

b. -random



*462는 L1 Write miss rate, L2의 Read miss rate가 모두 0에 가까워 그래프가 겹쳐져있다.

각 workload의 characteristic과 cache behavior에 관한 결론은 아래와 같다.

① **perlbench**

: 정규식, text processing, 연산, 파일 입출력 등 여러 data access pattern을 가지고 있다. Spatial locality와 temporal locality는 보통정도이며, data access pattern에 따라 cache behavior가 다르다. Cache size가 증가함

에 따라 많은 data set을 처리하는데에 도움이 되기 때문에 miss rate가 감소할 수 있다.

② **450 soplex**

: array operation을 하기 때문에 spatial locality가 좋다. Temporal locality도 좋다. 따라서 cache size, associativity 증가에 따라 miss rate을 낮출 수 있다.

③ **453 povray**

: cache size를 늘림에 따라 miss rate가 줄어들 수 있다. 그러나 spatial, temporal locality가 좋다고 하기에는 mixed 되어있기에 cache behavior를 예측하기 어렵다.

④ **462 libquantum**

: quantum computation을 하기 때문에 계속해서 새로운 data가 잦게 등장한다. 따라서 temporal locality가 좋지 않다. 또한 quantum state와 gate에서의 operation이 독립적으로 동작하기 때문에 spatial locality 또한 좋지 않다. 따라서 cache size를 늘려도 miss rate의 개선에 도움이 되지 않는다.

⑤ **473 astar**

: path finding problem 등에 이용되므로 traversal 동안 graph 내 node 간 연관관계에 의해 spatial locality가 좋다. 따라서 Cache size의 증가와 associativity 증가에 따라 miss rate이 감소할 수 있다.

⑥ **483 xalancbmk**

: input, output file에 따라 spatial locality는 달라질 수 있으며, XML과 output file에 따라 다른 process를 갖기 때문에 temporal locality가 좋지 않다. 따라서 일반적으로 cache size 증가에 따라 miss rate를 줄일 수 있으나 spatial, temporal locality가 상황별로 다를 수 있기 때문에 예측이 어렵다.