Project 4: Multi—level Cache model and Performance Analysis

201911039 김태연

1. Modeling a Two-Level cache

A. 간략한 설명

이 캐시는 다음과 같이 동작한다.

1) 읽기 혹은 쓰기 명령이 물리 주소와 함께 들어오면, L1 캐시에서 해당 물리 주소를 담은 block이 있는지 탐색한다.

2) 있다면 hit이 발생하고, 없다면 l1 miss가 발생한 후 l2 캐시에서 해당 물리 주소를 담은 block이 있는지 탐색한다.

3) l2에 해당 물리 주소 block 이 있다면 l2 hit이 발생하고, l1 캐시에 block을 업로드해 준다. 이 때 해당 물리 주소에 대해 l1 캐시의 set이 꽉 차 있다면 eviction이 발생한다.

4) 없다면 l2 miss가 발생한 후 block을 생성하여 l2와 l1에 모두 업로드해 준다. 이 때 l2캐시와 l1캐시에서 eviction이 발생한다. 만일 l2 캐시에서 evict된 block 이 l1 cache에도 있다면, 이 또한 evict 해 준다.

캐시는 block으로 상징되는 unsigned long int 타입의 변수를 담은 list로 구현하였다. block은 다음과 같이 구성되어 있다.

Tag	Index (capacity/(blocksize*associativity))bits	Dirty(1bit)

B. 트레이스 파일 및 패러미터에 따른 결과 분석

모든 tracefile에 대해 실행할 필요성이 없다고 생각되어, 400_perlbench.out에 대해서 실행하고 분석하였다. tracefile 간의 비교는 분석 4)에서 확인할 수 있다. 각각의 그래프는 read miss rate와 write miss rate의 변화를 담은 것이며, LRU와 random간의 차이는 miss rate 수의 차이로 확인하였다.

모두 L1 cahce에 대한 miss rate이다.

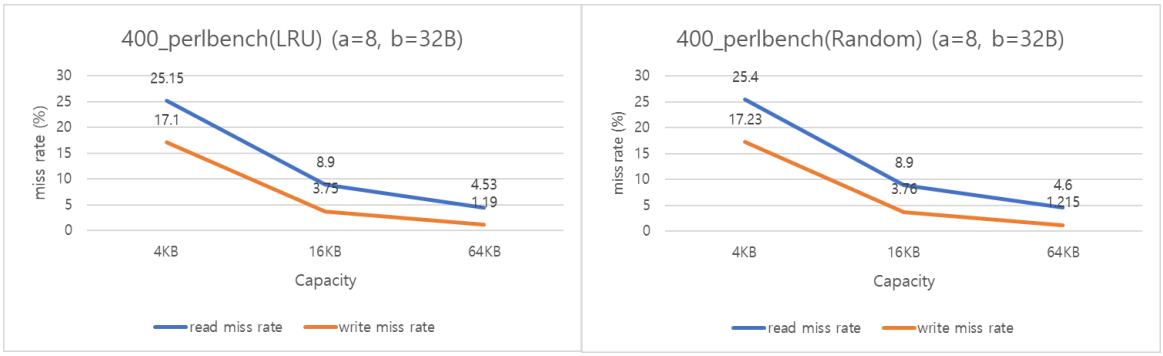
capacity, associativity, block size변화에 따라 miss rate의 변화를 분석하였으며, 각각을 비교 분석할 때 나머지 parameter는 임의의 값으로 설정하여 고정시켰다. 각 parameter가 커짐에 따라, miss rate가 감소할 것으로 예상했다.

1) capacity 변화에 따른 miss rate 분석 (associativity: 8, block_size: 32)

여기서 Capacity란 option으로 주어진 l2의 capacity를 의미하며, l1의 capacity는 그것의 1/4 크

기임을 밝힌다.

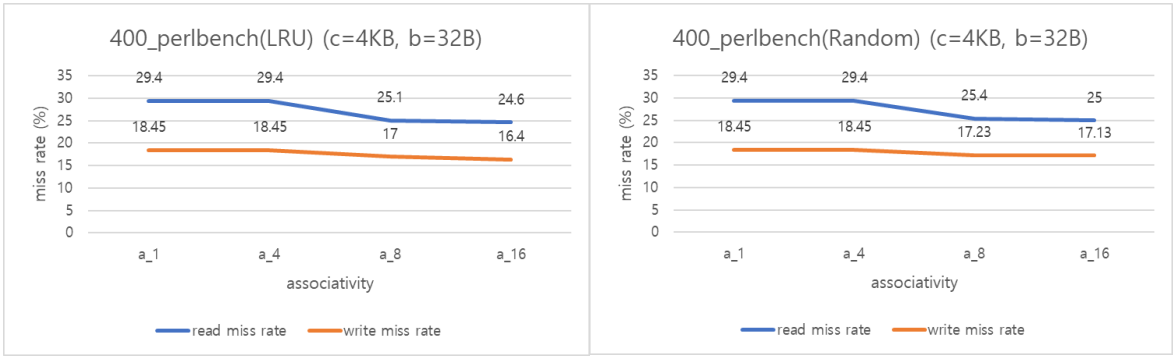
capacity를 4, 16, 64KB로 바꿔 가며 측정하였다. capacity가 클 경우 연산시간이 과도하게 오래 소요되어 분석에서는 제외하였다.



두 가지 교체정책 모두에서 캐퍼시티가 증가할수록 miss rate이 감소하는 것을 확인할 수 있었다. 이는 캐시 용량이 증가할수록 올라갈 수 있는 block의 수가 증가하기 때문이다. LRU가 Random에 비해 조금 더 낮은 miss rate를 보였다.

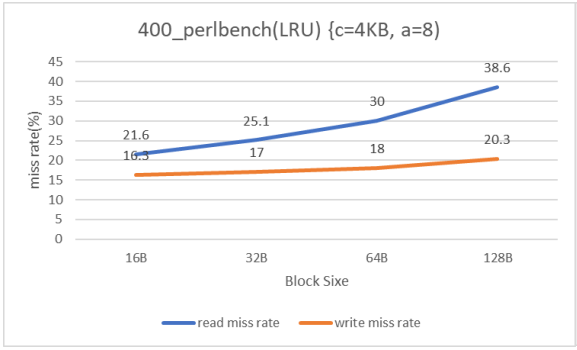
2) associativity 변화에 따른 miss rate 분석(capacity 4KB, block_size: 32B)

여기서 associativity란 option으로 주어진 I2의 associativity를 의미하며, I1의 associativity는 그것의 1/4 크기임을 밝힌다.

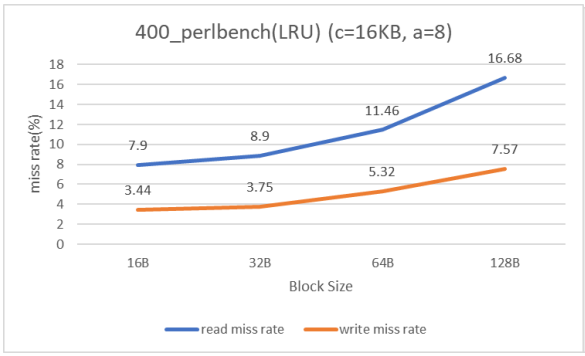


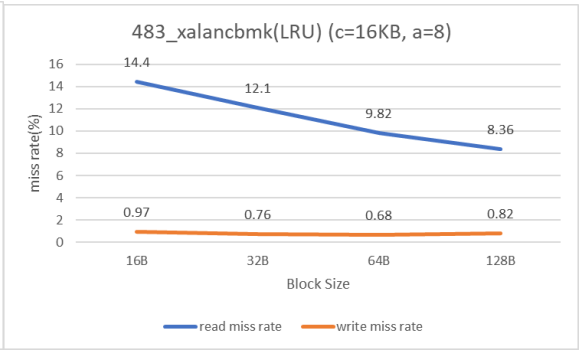
associativity가 증가함에 따라 miss rate가 감소함을 알 수 있다. 이는 예상과 부합하는 결과이다. way가 증가할수록 set(index)에 대해 유연성이 늘어나므로, conflict miss의 수를 줄일 수 있기 때문이다. associativity가 1인 경우에서 4인경우로 바뀔 때 변화가 없는 것은, associativity option이 I2에 대해 주어지기 때문이다. 이 분석 결과는 I1에 대한 것이므로, cache design 가정상 I1의 associativity에 변화는 없다. Random과 LRU의 두 가지 경우에 크게 차이는 없으나, LRU정책을 사용했을 때 miss rate이 조금 더 작은 것을 확인할 수 있다.

3) block size 변화에 따른 miss rate 분석(capacity 4KB, associativity: 8)



예상과 다르게, block size가 증가할수록 miss rate이 증가했다. 이는 block offset이 증가하면 index bit가 감소하게 되는데, 이 경우에 capacity를 너무 작게 설정하여 set의 감소에 따른 miss 발생률이 늘어난 것으로 예상된다. Random교체정책에 비해 LRU 교체정책이 더 낮은 miss rate를 보였다.

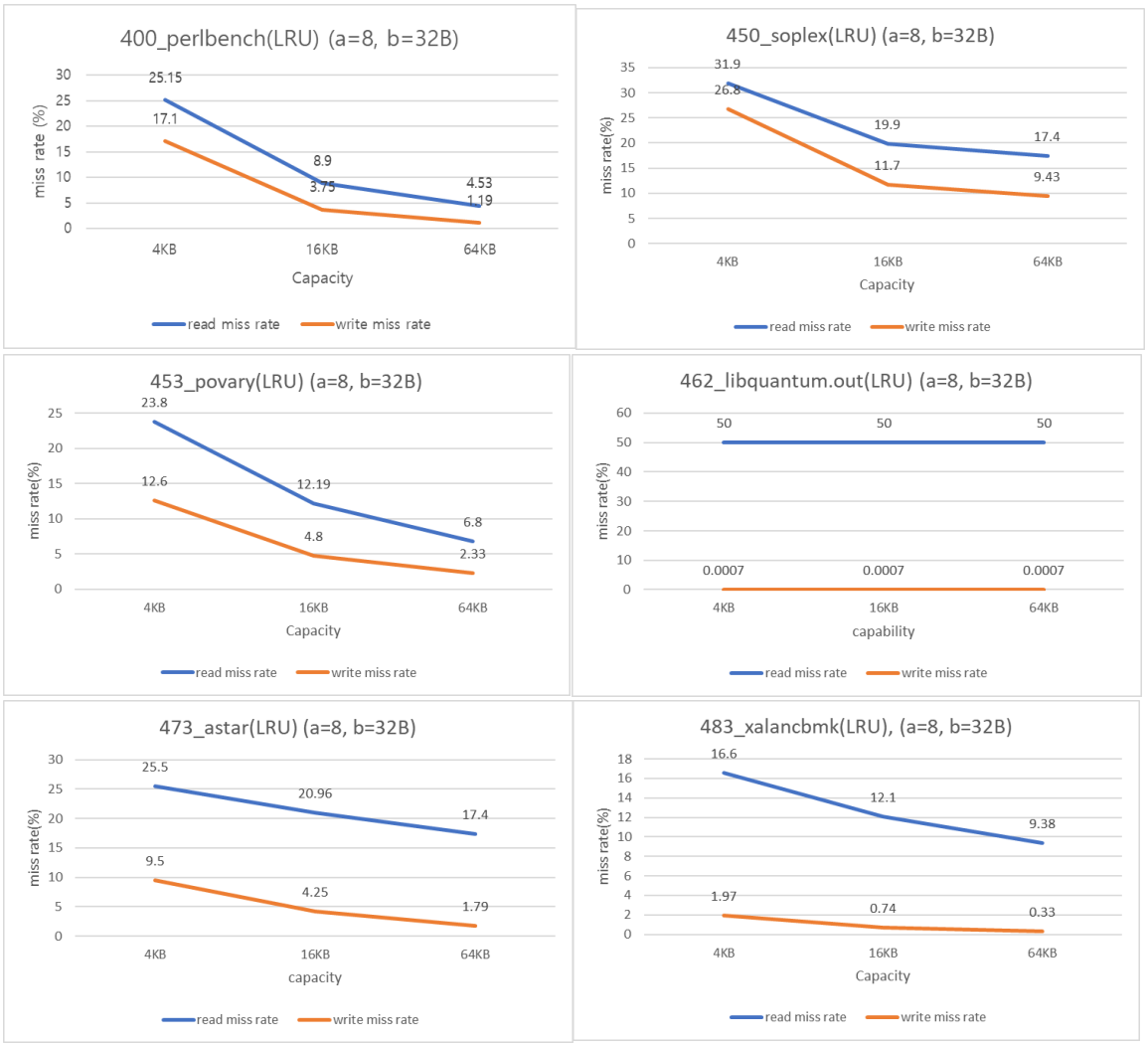




예상이 맞는지 확인하기 위해, capacity를 16KB로 늘리고 LRU 교체정책을 사용하여 위의 perlbench file과 또 하나의 tracefile인 xalancbmk file에 대해 Block Size의 증가에 따른 miss rate 변화를 조사하였다. perlbench file에서는 capacity를 늘리면 Block size의 증가에 따라 감소할 것이라는 예상과 달리, 여전히 증가하는 양상을 보였다. 반면 xalancbmk의 경우 예상대로 감소하는 것을 확인할 수 있었다. 그래프로 기재하지는 않았으나 다른 tracefile(soplex.out)에서도 block size의 증가에 따라 miss rate의 감소 양상을 보였다. 따라서 이 결과는 각 tracefile이 접근하는 주소 패턴상의 차이에서 오는 것으로 예상된다. 주소 패턴상 만약 접근하고자 하는 데이터들이 매우 멀리 산포해 있다면, Block size가 큰 것으로 인한 이점보다 set 개수가 작아져 얻는 불이익이 더 클 것이기 때문이다. 이는 block이 크더라도 block 안에 함께 담길 만큼 데이터 주소의 거리가 가깝지 않다면, 오히려 set의 개수가 작아 set내에서 conflict miss가 발생할 확률이 높아지기 때문으로 예상된다.

4) tracefile 간의 capacity 변화에 따른 miss rate 분석(associativity: 8, block_size: 32B)

여기서 Capacity란 option으로 주어진 I2의 capacity를 의미하며, I1의 capacity는 그것의 1/4 크기임을 밝힌다.



Tracefile에 따라 read miss와 write miss가 서로 다른 비율로 나타나며, capacity 변화에 따른 그 변화 양상도 다르게 나타남을 확인할 수 있었다.

C. 과제 실행 방법 및 실행 환경

Ubuntu 20.04 환경에서 **g++ 버전 9.3.0**을 이용하여 다음 명령어로 컴파일 하였다.

```
g++ main.cpp cache.cpp
```

위의 컴파일 명령으로 실행 파일을 생성하여 다음과 같은 명령어로 실행하였다.

```
./a.out -c 4 -a 8 -b 32 -random 450_soplex.out
```

위 실행예시의 경우는 옵션을 capacity 4, associativity 8, blocksize 32로 준 경우이다. 위와 같은 형태로 옵션을 달리하여 실행하였다.

2. Integrating the multi-level cache to MIPS processor

해당 과제에 대하여는 -atp 옵션을 줄 시 발생하는 코드상 오류를 잡지 못해 정상적인 결과를 도출해내지 못했다. procssor를 디자인한 아이디어와 -antp옵션을 사용하는 경우에 대한 최선의 분석을 수행했다.

A. 간략한 설명

캐시의 사용에 따른 CPI의 증가를 조사하기 위해, 실제 캐시의 동작을 모사하지는 않고 캐시 접근 동작만 모사하여 추가하였다. 즉, Instruction fetch나 memory load/write가 일어날 때 캐시 리스트를 lookup하는 과정이 함께 수행되며 이 때 miss가 발생할 시에, 1.과 동일하게 캐시에 block을 업데이트 하게 된다. Data write 발생 시에는 마지막 block에 대해 dirty bit를 setting하게끔 하였다. 파이프라인에서 일어나는 일은 이전과 동일하다. 다만, stall 만큼의 cycle이 cycle count에 더해지게 된다.

1.의 two-level cache와 논리적으로 동일하다. project 3의 pipeline에 IF-stage에서는 L1-I에 대한 접근이, MEM-stage에서는 L1-D에 대한 접근이 추가되었다. 각각의 highest cache에 접근하는 것이 실패할 경우에, L2 캐시에 접근하도록 했다.

L1 캐시 접근 실패 시, 9 cycle의 stall 이 발생한다.

L2 캐시 접근 실패 시, 99 cycle의 stall 이 발생한다.

B. 샘플 파일 및 패러미터에 따른 CPI 및 캐시 미스율 결과 분석

-atp옵션을 줄 경우 locality.o file에서 프로그램의 수행이 과도하게 오래 걸린다. 무한 루프에 빠졌거나 불필요한 연산을 계속해서 수행하는 것으로 예상된다. 따라서 다음 결과 분석은 모두 -antp를 옵션으로 설정하고 이루어졌다.

1) Capacity 변화에 따른 차이

캐시용량 1024B, 4096B에 대하여 교체정책을 LRU로 고정한 뒤 수행하여 비교하였다.

a) -antp, -lru, -c 1024

```
Pipeline Stats:
Completion Instructions: 345
CPI: 12.453757
Cache States:
Total accesses: 683
Instruction read accesses: 355
Data read accesses: 296
Write accesses: 32
L1_I Read misses: 89
L1_D Read misses: 7
L2 Read misses: 33
L1_D Write misses: 3
L2 Write misses: 1
L1_I Read miss rate: 25.070423%
L1_D Read miss rate: 2.364865%
L2 Read miss rate: 27.049181%
L1_D Write miss rate: 9.375000%
L2 Write miss rate: 33.333336%
L1-D clean eviction: 91
L2 clean eviction: 91
L1-D dirty eviction: 4
L2 dirty eviction: 0
```

locality.o, completion_cycle: 4309

```
Pipeline Stats:
Completion Instructions: 35992
CPI: 5.878032
Cache States:
Total accesses: 43998
Instruction read accesses: 39998
Data read accesses: 2
Write accesses: 3998
L1_I Read misses: 8002
L1_D Read misses: 1
L2 Read misses: 6
L1_D Write misses: 1000
L2 Write misses: 1000
L1_I Read miss rate: 20.006001%
L1_D Read miss rate: 50.000000%
L2 Read miss rate: 0.074925%
L1_D Write miss rate: 25.012505%
L2 Write miss rate: 100.000000%
L1-D clean eviction: 8000
L2 clean eviction: 8000
L1-D dirty eviction: 999
L2 dirty eviction: 0
```

sequential.o, completion_cycle: 211568

b) -antp, -lru, -c 4096

```
Pipeline Stats:
Completion Instructions: 345
CPI: 27.072254
Cache States:
Total accesses: 683
Instruction read accesses: 355
Data read accesses: 296
Write accesses: 32
L1_I Read misses: 88
L1_D Read misses: 3
L2 Read misses: 91
L1_D Write misses: 0
L2 Write misses: 0
L1_I Read miss rate: 24.788733%
L1_D Read miss rate: 1.013513%
L2 Read miss rate: 50.837986%
L1_D Write miss rate: 0.000000%
L2 Write miss rate: -nan%
L1-D clean eviction: 85
L2 clean eviction: 85
L1-D dirty eviction: 0
L2 dirty eviction: 0
```

locality.o, completion_cycle: 9367

```
Pipeline Stats:
Completion Instructions: 35992
CPI: 3.878393
Cache States:
Total accesses: 43998
Instruction read accesses: 39998
Data read accesses: 2
Write accesses: 3998
L1_I Read misses: 5
L1_D Read misses: 1
L2 Read misses: 6
L1_D Write misses: 1000
L2 Write misses: 1000
L1_I Read miss rate: 0.012501%
L1_D Read miss rate: 50.000000%
L2 Read miss rate: 54.545456%
L1_D Write miss rate: 25.012505%
L2 Write miss rate: 100.000000%
L1-D clean eviction: 997
L2 clean eviction: 997
L1-D dirty eviction: 985
L2 dirty eviction: 0
```

sequential.o, completion_cycle: 139595

locality 샘플 파일의 경우에는, L1 캐시의 miss rate가 전체적으로 줄었다. L1-D에서 write miss가 발생하지 않았기 때문에 L2 캐시의 write miss는 zero division으로 인한 nan값이 확인되었다. L1의 read miss는 감소한데 반해 L2의 read miss는 증가하였는데, 이는 L1 miss가 발생할 때 L2

에도 해당 블록이 없는 경우가 많았기 때문으로 보인다. 이는 block을 처음으로 올려둘 때(L2 miss가 발생할 때) L1까지 업데이트 되는데, 용량이 충분히 커 L1에서 eviction이 잘 일어나지 않아 그렇게 된 것으로 보인다. 즉, L1의 용량이 커서 L2가 본 기능을 제대로 수행하지 못했음을 알 수 있었다. CPI는 증가했다.

sequential의 경우 캐시 용량이 커짐에 따라 CPI가 감소하였고, L1-I miss rate가 크게 줄었음을 확인할 수 있었다. instruction의 경우 그 수가 정해져 있기 때문에, cache용량이 커졌을 때 거의 모든 instruction을 hold할 수 있기 때문으로 예상된다. L1-D에 대한 miss rate는 크게 변화가 없는 데 반해, L2 read의 miss rate가 크게 증가하였는데 이는 L1-I의 miss rate 감소와 연관이 있을 것으로 생각된다. locality 샘플의 경우와 마찬가지로, L1-I 캐시의 크기가 충분히 커지면서 L2가 담고 있는 element에 대해 대부분 hold하고 있었을 가능성이 높다. 반면, L2의 경우 data section과 instruction section이 나누어져 있지 않기 때문에, LRU를 채택하여 오래된 instruction block은 evict 하게 되었을 것이다. 결국 L2와 L1-D사이의 유사성이 높아지면서, L2 Read miss를 증가시키게 되었을 것이다. Sequential 샘플의 경우에는 추가적으로, L2의 write miss rate가 100%인 것으로 나타나는데, 이는 읽기 작업을 위해 cache에 업로드되었던 block중 시간적으로 먼 데이터에 대해 write를 하려고 접근하는 경우가 자주 발생해, 해당 block이 L2에서도 evicted되었을 가능성을 생각해 볼 수 있다.

2) 교체 정책 변화에 따른 차이

a) locality, capacity: 4096, antp

```
Pipeline Stats:
Completion Instructions: 345
CPI: 27.072254
Cache States:
Total accesses: 683
Instruction read accesses: 355
Data read accesses: 296
Write accesses: 32
L1_I Read misses: 88
L1_D Read misses: 3
L2 Read misses: 91
L1_D Write misses: 0
L2 Write misses: 0
L1_I Read miss rate: 24.788733%
L1_D Read miss rate: 1.013513%
L2 Read miss rate: 50.837986%
L1_D Write miss rate: 0.000000%
L2 Write miss rate: -nan%
L1-D clean eviction: 85
L2 clean eviction: 85
L1-D dirty eviction: 0
L2 dirty eviction: 0
```

lru, cycle 9367

```
Pipeline Stats:
Completion Instructions: 345
CPI: 26.500000
Cache States:
Total accesses: 683
Instruction read accesses: 355
Data read accesses: 296
Write accesses: 32
L1_I Read misses: 85
L1_D Read misses: 4
L2 Read misses: 89
L1_D Write misses: 0
L2 Write misses: 0
L1_I Read miss rate: 23.943663%
L1_D Read miss rate: 1.351351%
L2 Read miss rate: 51.149429%
L1_D Write miss rate: 0.000000%
L2 Write miss rate: -nan%
L1-D clean eviction: 81
L2 clean eviction: 81
L1-D dirty eviction: 0
L2 dirty eviction: 0
```

random, cycle 9169

b) sequential, capacity: 4096, antp

```
Pipeline Stats:
Completion Instructions: 35992
CPI: 3.878393
Cache States:
Total accesses: 43998
Instruction read accesses: 39998
Data read accesses: 2
Write accesses: 3998
L1_I Read misses: 5
L1_D Read misses: 1
L2 Read misses: 6
L1_D Write misses: 1000
L2 Write misses: 1000
L1_I Read miss rate: 0.012501%
L1_D Read miss rate: 50.000000%
L2 Read miss rate: 54.545456%
L1_D Write miss rate: 25.012505%
L2 Write miss rate: 100.000000%
L1-D clean eviction: 997
L2 clean eviction: 997
L1-D dirty eviction: 985
L2 dirty eviction: 0
```

lru, cycle 139595

```
Pipeline Stats:
Completion Instructions: 35992
CPI: 3.883894
Cache States:
Total accesses: 43998
Instruction read accesses: 39998
Data read accesses: 2
Write accesses: 3998
L1_I Read misses: 7
L1_D Read misses: 1
L2 Read misses: 8
L1_D Write misses: 1000
L2 Write misses: 1000
L1_I Read miss rate: 0.017501%
L1_D Read miss rate: 50.000000%
L2 Read miss rate: 53.333336%
L1_D Write miss rate: 25.012505%
L2 Write miss rate: 100.000000%
L1-D clean eviction: 985
L2 clean eviction: 985
L1-D dirty eviction: 985
L2 dirty eviction: 0
```

random, cycle: 139793

locality.o의 경우 교체정책 random을 사용하는 것이 더 낮은 CPI를 보였다. 이는 random을 사용할 때 L1-I 캐시에 대한 miss rate가 낮게 나타났기 때문으로 보인다.

반면, sequential.o의 경우 교체정책 lru를 사용하는 것이 조금 더 낮은 CPI를 보여주었다. 이는 lru를 사용할 때 L1-I 캐시에 대한 miss rate가 낮게 나타났기 때문으로 보인다.

이처럼 입력으로 주어지는 sample file들의 데이터 접근 패턴이 다르기 때문에, 최적의 교체정책은 프로그램마다 다를 수 있으며 때로는 random 교체정책이 LRU보다 더 좋은 성능을 보이기도 한다.

C. 과제 실행 방법 및 실행 환경

Ubuntu 20.04 환경에서 **g++ 버전 9.3.0**을 이용하여 다음 명령어로 컴파일 하였다.

```
g++ main.cpp pipeFunc.cpp memory.cpp cache.cpp
```

위의 컴파일 명령으로 실행 파일을 생성하여 다음과 같은 명령어로 실행하였다.

```
./a.out -antp -c 4096 -lru sequential.o
```

위 실행예시의 경우는 필수 실행 옵션을 준 경우이다.

3. 소스코드의 제출

proj4-1과 proj4-2에서, proj4-2를 수행하는 과정에서 cache.cpp의 함수를 일부 수정했기 때문에, 그리고 main.cpp의 이름이 겹치는 관계로 다른 폴더에 나누어 제출합니다.