Project 4 : Multi-level Cache Model and Performance Analysis Report

추성재 (201911185)

1. 개발 환경

이번 프로젝트에서 사용한 언어는 C++이다. Ubuntu 20.04에서 g++ 9.4.0을 이용해 컴파일 했고, make를 이용해 빌드했다.

2. 컴파일 방법

압축파일 속 "realProjects" 폴더 속에 Makefile이 있다. 다음과 같은 명령어를 통해 컴파일하면 된다.

```
$make clean; make
```

컴파일이 완료되면, "runfile"이라는 이름의 실행파일이 나온다.

3. 실행 방법

실행파일 뒤에 과제 안내에 명시된 인자를 다음과 같이 입력 후 실행하면 된다. 인자들의 순서나 목적 파일을 제외한 인자들의 유무는 상관없다.

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

4. 코드의 flow

A. 00main.cpp

이 코드는 실행할 때 argument들을 받아 변수들을 initialization 한다. 또한 L1 Cache와 L2 Cache를 argument를 이용해 초기화한다. 이후 tracefile의 내용을 받아서 한 줄씩 L1 Cache에 전달한다. 모든 과정이 마무리되면 Cache들에 저장되어 있는 cleaneviction과 같은 함수들을 "workloadName_capacity_associativity_blockSize.out과 같은 형태로 파일 출력을 한다.

B. 01cache.cpp

캐시의 전반적인 동작을 수행하는 Class들이 정의되어 있다.

- class Block

Cache 속에 들어가는 block들을 정의한 class이다.

dirty_bit	Block에 write를 할 때 dirty_bit가 1로 변한다. 이는 write-back, write-allocate를 지원하기 위함이다.
valid_bit	valid_bit가 1이면 Block에 내용물이 있다는 의미이고, valid_bit가 0이면 Block에 내용물이 없다는 의미이다.
block_start_addr	Block의 시작 주소를 bit 형태의 string으로 저장한다.
accessedTime	lru를 지원하기 위해 Block이 access 된 timestep을 저장하는 변수이다.
isEmpty	빈 Block을 표기하기 위한 bool 변수이다.
Block : constructor 1	일반적인 Block을 생성하기 위한 constructor이다.
Block : constructor 2	빈 Block을 생성하기 위한 constructor이다.

- class Cache

  Cache의 전반적인 동작을 지원하기 위한 parent class이다. L1Cache와 L2Cache가 이를 상속받는다.

readHit	Cache에서 일어난 read hit을 기록하는 변수이다.
readMiss	Cache에서 일어난 read miss를 기록하는 변수이다.
writeHit	Cache에서 일어난 write hit을 기록하는 변수이다.
writeMiss	Cache에서 일어난 write miss를 기록하는 변수이다.
block_size	Cache에 들어가는 Block의 size를 기록하는 변수이다.
cleanEviction	Cache에서 일어난 clean eviction을 기록하는 변수이다.
dirtyEviction	Cache에서 일어난 dirty eviction을 기록하는 변수이다.
cache_size	Cache의 크기를 기록하는 변수이다
total_block_num	Cache 안에 몇 개의 Block이 있는지를 기록하는 변수이다.
asso_way	Cache의 associativity를 나타내는 변수이다.
set_num	Cache 속 set의 개수를 저장하는 변수이다.
is_lru	Cache가 evict policy로 lru를 쓰는지 여부를 저장하는 변수이다. true이면 lru, false이면 random을 택한다.
cache	실제 Block들을 담고 있는 vector이다.
whatCache	이 Cache가 L1인지 L2인지를 저장하는 변수이다.
emptyBlock	빈 Block을 나타내는 Block pointer이다.
power_2	2의 거듭제곱을 받으면 몇 제곱지를 반환하는 함수이다.
get_tag	입력된 Block의 block_start_addr를 바탕으로 Cache에서 사용할 수 있는 tag와 index를 반환한다.
access_addr	Cache에 읽기, 쓰기 여부와 접속하고자 하는 주소를 담고 있는 meminfo를 받은 후 해당하는 Block이 어딘지를 반환하는 함수이다.

- class L1Cache

  L1 Cache의 역할을 담당하는 Cache의 child class이다.

L2	L2 Cache의 위치를 담고 있는 pointer이다.
L1Cache : constructor	capacity, associativity, block size, lru 여부, emptyBlock의 주소를 받아서 L1Cache의 member variable과 cache vector를 초기화하는 역할을 한다.
~L1Cache : deconstructor	L1Cache를 만들면서 동적할당 된 요소들을 동적할당 해제한다.
init_cache_pointer	L2 Cache의 위치를 저장하는 member function이다.
miss_handler	miss가 발생했을 시 L2에서 해당하는 Block을 받아오는 member function이다.
add_block	새로운 Block을 Cache에 저장하는 member function이다.
evict_block	add_block에서 빈 way가 없다면 lru와 random을 사용해 block을 evict하는 member function이다. 이때, 이 Block이 dirty Block일 경우에는 L2 Cache의 access_addr를 이용해 기록한다.

- class L2Cache

  L2 Cache의 역할을 담당하는 Cache의 child class이다.

L1	L1 Cache의 위치를 담고 있는 pointer이다.
L2Cache : constructor	capacity, associativity, block size, lru 여부, emptyBlock의 주소를 받아서 L2Cache의 member variable과 cache vector를 초기화하는 역할을 한다.
~L2Cache : deconstructor	L2Cache를 만들면서 동적할당 된 요소들을 동적할당 해제한다.
init_cache_pointer	L1 Cache의 위치를 저장하는 member function이다.
miss_handler	miss가 발생했을 시 memory에서 해당하는 Block을 받아오는 member function이다. 이번 경우에는 memory를 구현하기 보다는 해당하는 주소가 속한 Block을 새로 만들어서 반환한다.
add_block	새로운 Block을 Cache에 저장하는 member function이다.
evict_block	add_block에서 빈 way가 없다면 lru와 random을 사용해 block을 evict하는 member function이다. 이때, 해당하는 Block이 L1에도 있는 Block이라면, L1Cache의 evict_block을 불러와 이를 처리한다.

C. 03datamanage.cpp

00main.cpp에서 Project 2의 코드를 사용한 부분이 있는데, 이 부분을 지원하기 위해서 Project 2에서 가져온 코드이다.

함수명	기능
DecimalToHex	unsigned int를 16진수 형태의 문자열로 바꾸는 함수다. Project 1과는 달리, isSigned 인자가 추가되어 기본값인 false일때는 unsigned int로 처리하고, true일때는 int형으로 명시적 casting 후 처리한다.
SplitLine	Project 1에서와 동일하게 string str가 들어오면 char seperator를 기준으로 분리한 후 이를 담은 vector를 반환한다.
StrHaveChar	string str에 char c가 있는지 여부를 확인하고 있으면 true, 없으면 false를 반환한다.
NumToBit	String 형식의 수를 받으면 이 중 least significant bit를 size의 크기에 맞춰서 string 형식으로 반환한다.
StrIsNum	string str이 숫자인지 판별한다. isHex의 기본값 false일때는 숫자를 10진수를 기준으로 판별하고, isHex가 true일때는 16진수를 기준으로 판별한다.
BitExtended	string 형식의 bit를 받으면 이를 isSignEx가 true일때는 sign extend, false일때는 zero extend한 후 그 결과를 string 형식의 bit로 반환한다.

D. Code flow

이번 project에서 cache는 다음과 같이 동작한다.

i. 우선 L1 Cache에 tracefile 속 memory address와 읽기, 쓰기 여부를 meminfo에 넣어 전달한다.

ii. hit일 경우에는 L1 Cache에서 해당하는 Block의 pointer를 반환하고, miss일 경우에는 miss_handler를 불러 L2에서 해당하는 Block을 불러온다.

iii. L1 Cache의 miss_handler에서 L2에 접근해 L2에서 hit가 난다면, 해당하는 Block을 add_block을 이용해 저장한다. 이때, Block을 evict해야 한다면 evict_block을 이용해 Block을 evict한 이후 새 Block을 add_block을 이용해 저장한다. 이때, evict되는 Block이 dirty block일 경우에는 Block의 정보를 L2 access_addr를 이용해 L2 Cache로 넘긴다.

iv. 만약 L2에서도 miss가 난다면, L2 miss_handler에서 새로운 Block을 받아온 이후, L2에 해당 Block을 저장한 이후 L1으로 Block을 넘긴다. 이때, 새로운 Block을 저장할 때 Block을 evict해야 한다면 evict policy에 따라 evict한다. 이때, 해당 Block의 정보를 L1 evict_block으로 넘겨서, L1에도 해당 Block이 있다면 같이 evict한다.

v. i.에서 iv.까지 모든 과정을 tracefile 속 모든 줄에 대해 처리한다면, 파일로 출력할 변수들을 파일로 출력하고 프로그램을 종료한다.

E. Output

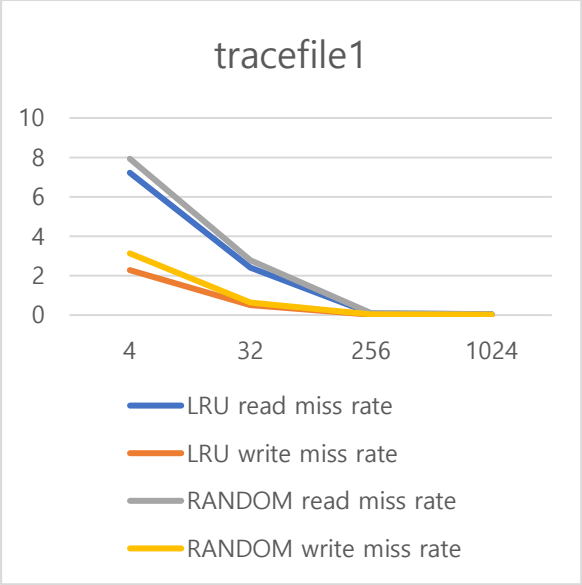
tracefile들이 Cache에 접근할 때 Cache의 Capacity, associativity, block size의 변화에 따른 전체 Cache의 miss rate(L2 Cache (read, write) miss 횟수를 (read, write) access 횟수로 나눈 값)를 구했다.

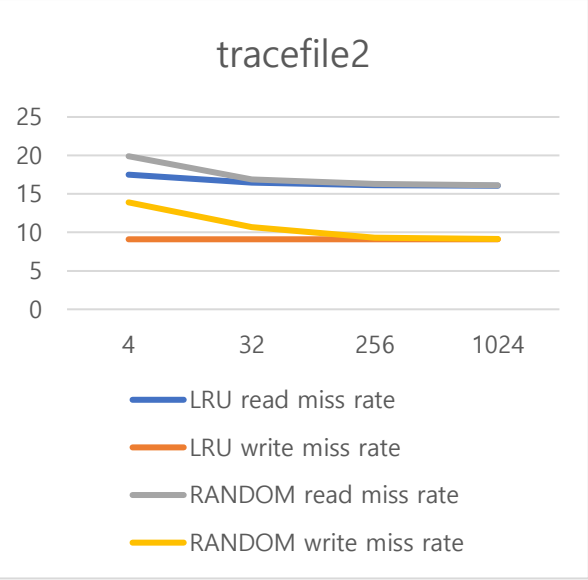
그래프의 제목은 각각 다음 tracefile을 나타낸다.

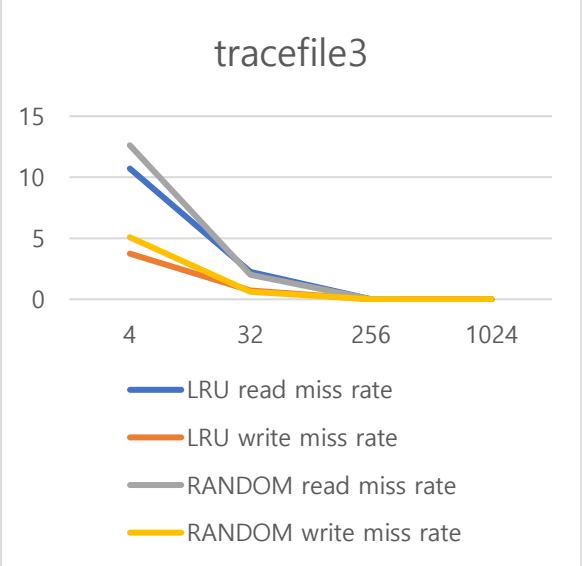
- tracefile1 : 400_perlbench.out
- tracefile2 : 450_soplex.out
- tracefile3 : 453_povray.out
- tracefile4 : 462_libquantum.out
- tracefile5 : 473_astar.out
- tracefile6 : 483_xalancbmk.out

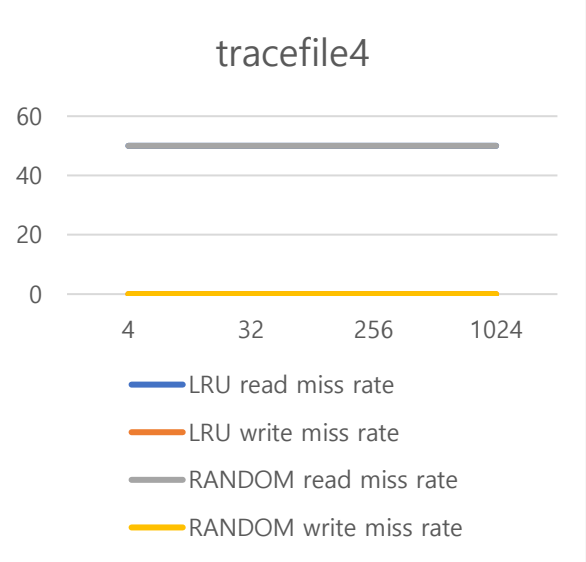
우선 Cache의 Capacity에 따른 tracefile별 read, write miss rate를 그래프로 구해보았다. 이때, Cache의 Capacity는 L2 Cache를 기준으로 4, 32, 256, 1024KB이고, associativity는 4로, block size는 32byte로 고정했다.

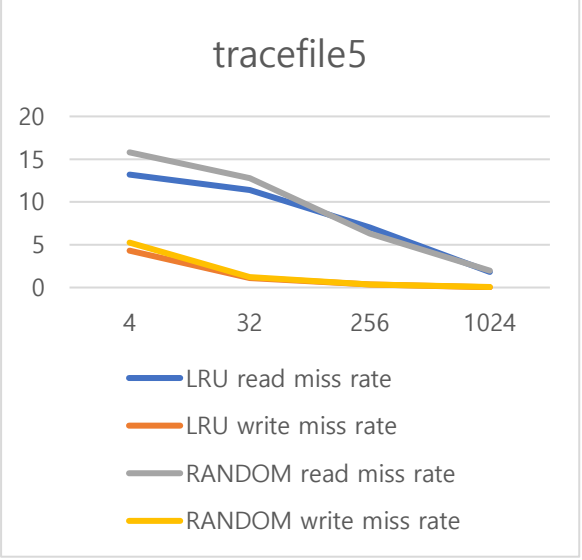
evict policy로 LRU를 적용했을 때, Random을 적용했을 때 각 tracefile 별 read, write miss rate는 다음과 같다. 그래프의 가로축은 Cache Capacity(KB), 세로축은 miss rate(%)이다.

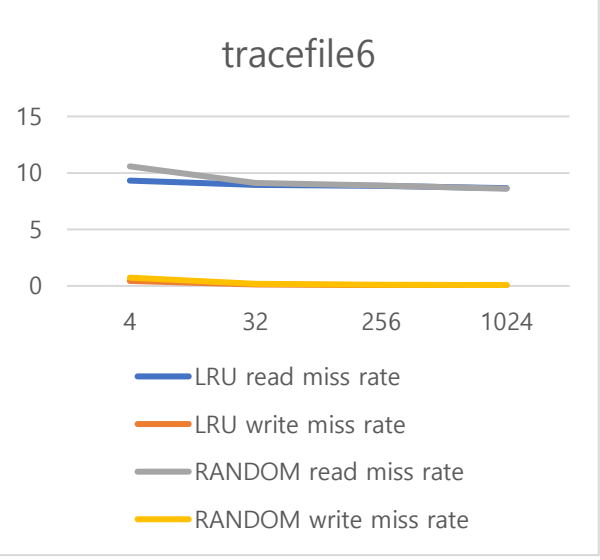










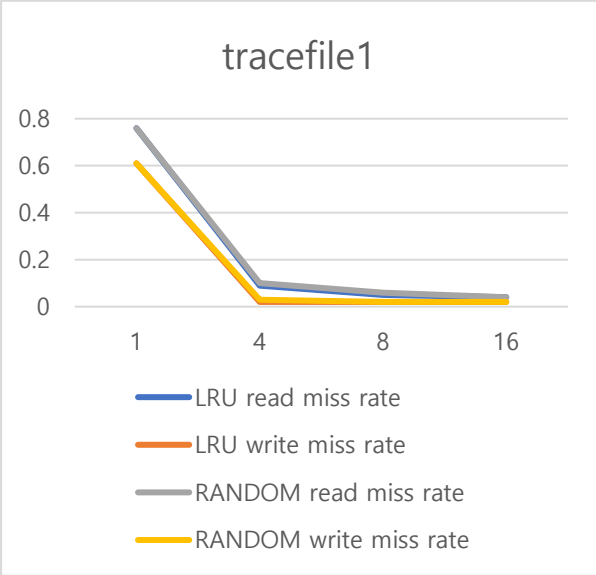


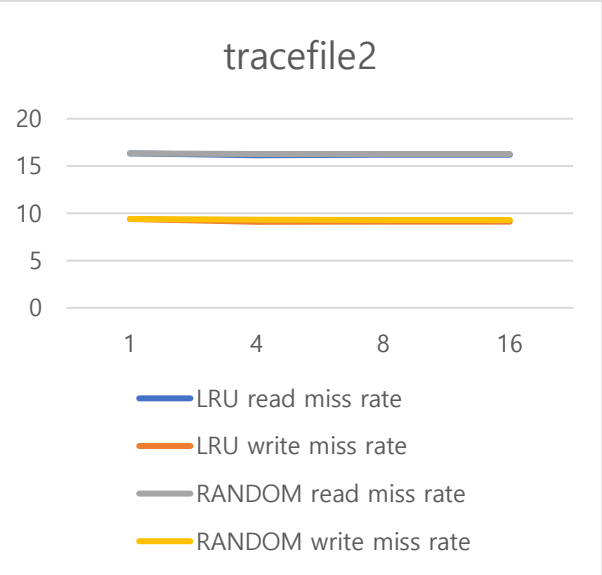
위 그래프로 보아, tracefile4를 제외한 대부분의 tracefile에서 LRU policy의 cache read, write miss rate가 Random policy보다 작은 것을 알 수 있다. 또한, tracefile2, tracefile4, tracefile6을 제외한 나머지 tracefile에서 Cache의 Capacity를 늘리면 cache read, write miss rate가 줄어드는 것을 볼 수 있다. 이는 Cache의 size가 늘어나면 evict를 하지 않아도 새로운 Block을 저장할 수 있기에 miss rate가 줄어든다고 해석할 수 있다. 다만 Cache의 capacity를 늘려도 miss rate가 안 주는 tracefile2,

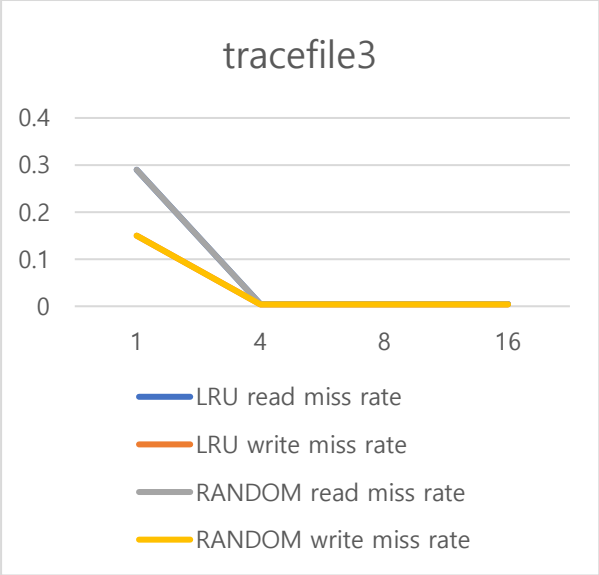
tracefile4, tracefile6의 경우는 Cache의 capacity보다는 spacial locality가 높은 데이터에 접근하려고 해서 miss rate의 변화가 없다고 볼 수 있다.

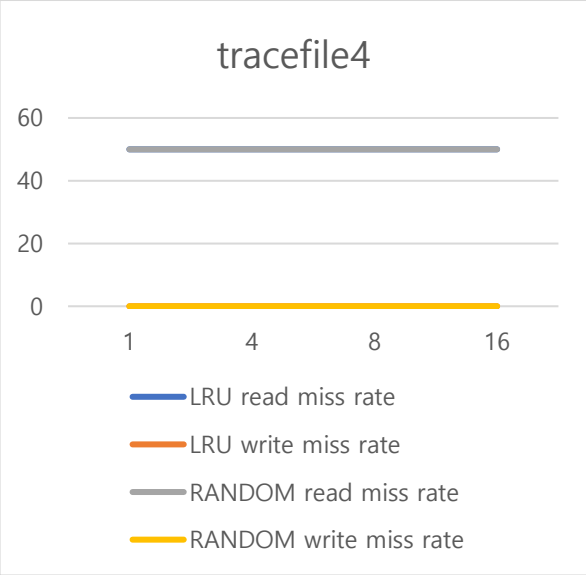
다음으로는 Cache의 associativity에 변화를 주었을 때 tracefile 별 read, write miss rate을 그래프로 구했다. 이때, Cache capacity는 256KB로, block size는 32Byte로 고정했다.

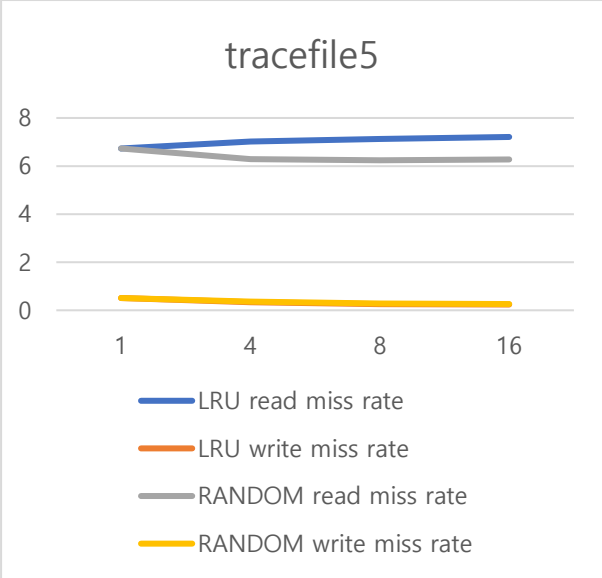
evict policy로 LRU를 적용했을 때, Random을 적용했을 때 각 tracefile 별 read, write miss rate는 다음과 같다. 그래프의 가로축은 Cache Associativity, 세로축은 miss rate(%)이다.

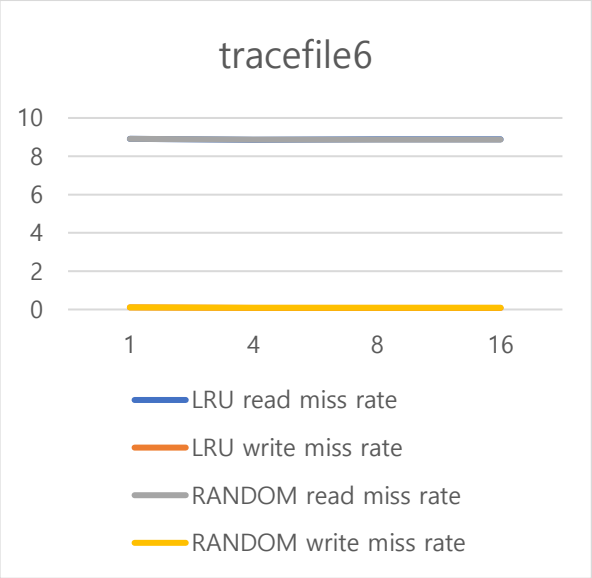








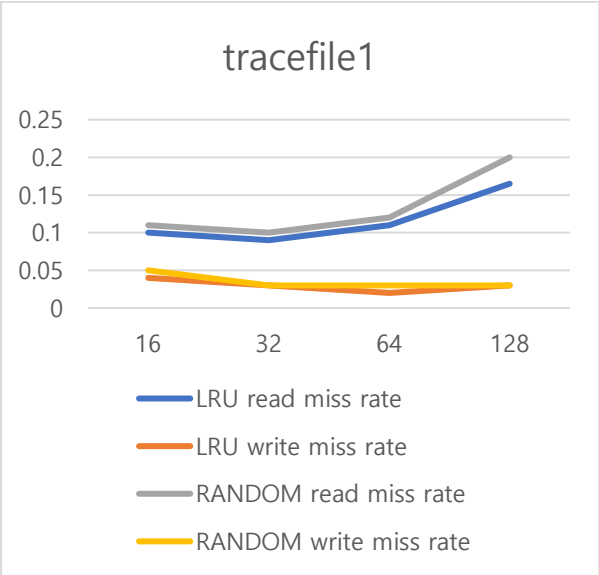


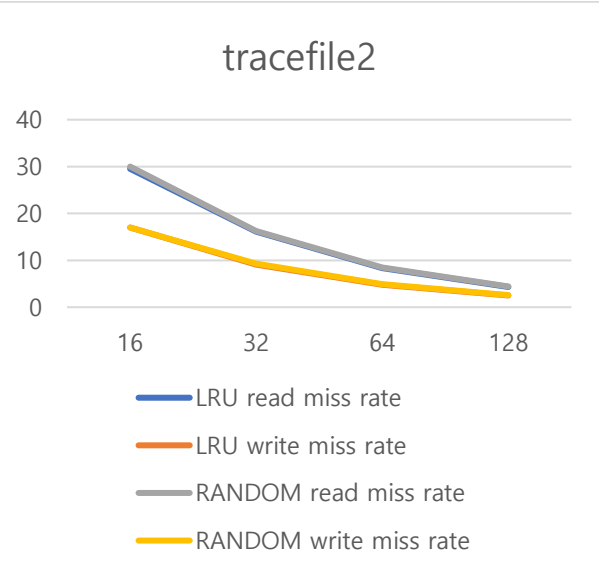


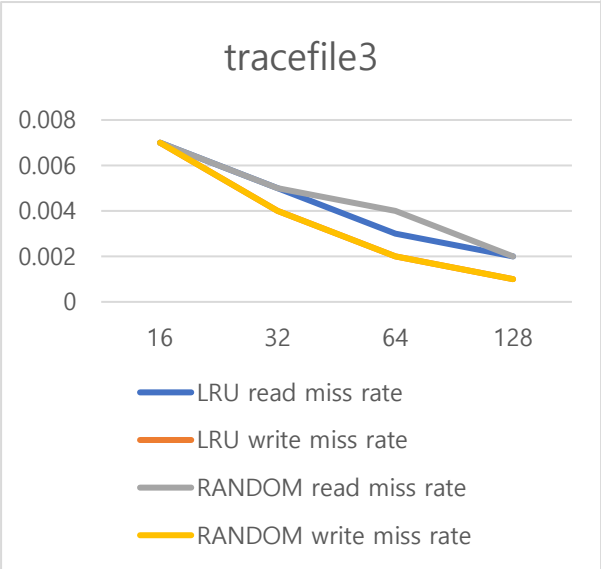
위 그래프를 보면 이전 capacity와는 달리 tracefile1과 tracefile3에서만 associativity의 변화에 따라 miss rate가 줄어든다. 또한 LRU와 Random policy간의 차이가 적거나 거의 없고, tracefile5의 경우에는 LRU read miss rate가 random read miss rate보다 크다. 이는 tracefile1, 3을 제외한 나머지 tracefile의 경우 서로 다른 set에 해당하는 data들의 접근이 더 자주 일어나 associativity가 늘어난 이점을 살리지 못한다고 해석할 수 있다. 또는 Block들간의 spacial locality가 높은 데이터들에 접근하려고 할 때 또한 높은 associativity를 살릴 수 없다. 또한 LRU와 Random의 miss rate 차이가 적은 이유는 Cache Capacity를 256KB로 고정해서 그렇다. 이는 앞선 Capacity 변화에서도 확인할 수 있는데, Capacity가 늘어날수록 LRU의 miss rate와 Random의 miss rate간의 차이가 거의 없어졌다.

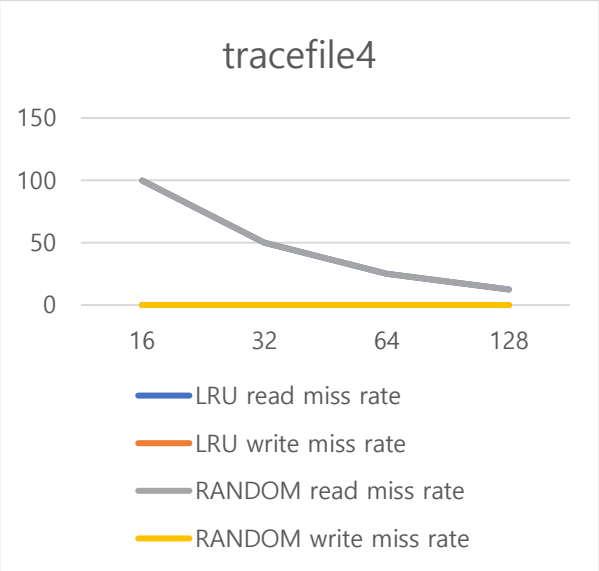
마지막으로 Cache의 block size에 변화를 주었을 때 tracefile 별 read, write miss rate을 그래프로 구했다. 이때, Cache capacity는 256KB로, associativity는 4로 고정했다.

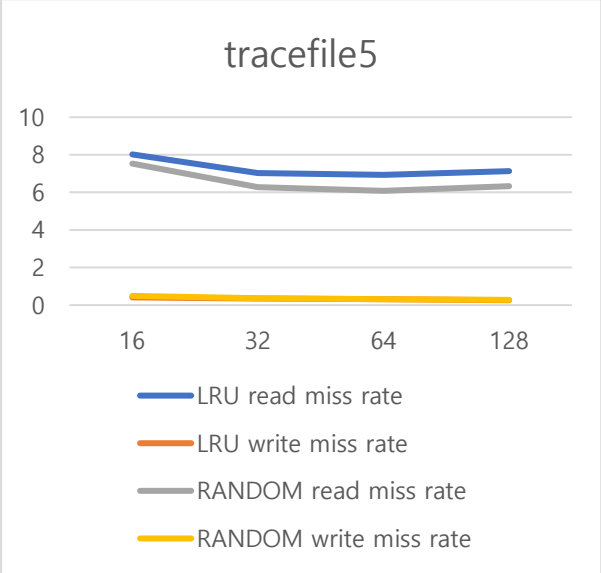
evict policy로 LRU를 적용했을 때, Random을 적용했을 때 각 tracefile 별 read, write miss rate는 다음과 같다. 그래프의 가로축은 Cache Block size(Byte), 세로축은 miss rate(%)이다.

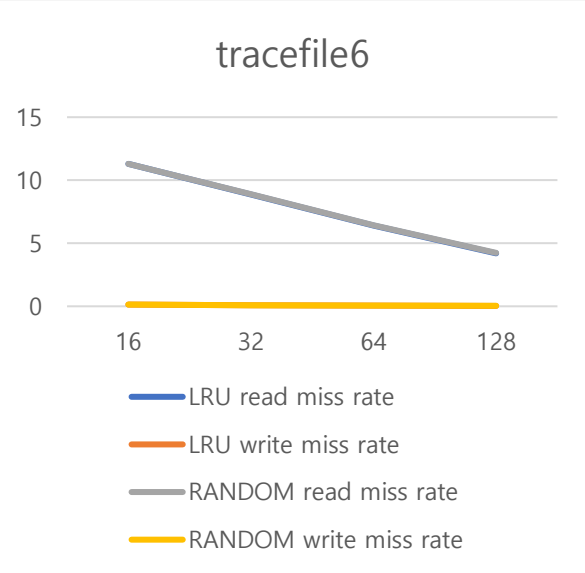












block size에 변화를 주었을 때는 앞서 miss rate가 변화하지 않았던 tracefile2, 4, 6에서 miss rate가 감소했다. 이는 tracefile2, 4, 6에서 접근하는 데이터들의 spatial locality가 높았기 때문으로 볼 수 있다. 하지만 tracefile1과 tracefile5에서는 오히려 block size를 키우면 miss rate가 증가하는 경향을 보였는데, 이는 이 데이터들의 spatial locality가 낮은 상황에서 큰 block 단위가 오히려 miss rate를 증가시켰다고 해석할 수 있다.