Project 4: Multi-level Cache Model and Pefomance Analysis

201811061 박동진

1. 소스 코드에 대한 간략한 설명

이 소스 코드는 C++로 작성되었습니다.

struct BlockLRU 와 struct ListLRU를 이용해 doubly-linked list로 LRU Replacement를 구현하였습니다.

struct BlockRan 와 struct ListRan를 이용해 Random Replacement를 구현하였습니다.

-struct BlockLRU

//멤버 변수

int64_t data, addr

bool dirty

BlockLRU* prev, next

data는 tag matching을 할 때 사용하는 변수입니다. addr은 trace파일의 addr을 블록마다 저장하기 위해서 사용하였습니다. dirty는 dirty, clean eviction을 판별하기 위해 사용하였습니다. prev, next는 doubly-linked list를 구현하기 위해 사용했습니다.

-struct ListLRU

//멤버 변수

BlockLRU* head, tail

int size, dirtyEvict, cleanEvict, writeHit, writeMiss, readHit, readMiss

head, tail은 doubly-linked list의 맨 앞과 맨 뒤의 노드를 가리킵니다. size는 way수를 의미합니다. dirtyEvict와 cleanEvict는 set에서 eviction의 수를 축적하기 위해 사용하였습니다. Hit, Miss 또한 그것들의 수를 축적하기 위해서 사용하였습니다.

//멤버 함수

BlockLRU* LRUSearch : doubly-linked list를 탐색하며 tag matching을 하는 역할을 합니다. matching되면 그 블록의 주소를 반환합니다.

void LRUHit : Cache hit이 발생했을 때, doubly-linked list를 업데이트하는 역할을 합니다. 매개변수로 받은 블록은 hit으로 접근되었으므로 doubly-linked list에서 head의 next 노드로 업데이트 해줌으로써 tail로 갈수록 LRU의 replacement 대상이 될 수 있게 해주었습니다. rmBlock과 addBlock함수를 포함합니다.

void rmBlock : 매개변수로 받은 블록을 없애주는 역할을 합니다.

void addBlock : head의 next로 블록을 추가해주는 역할을 합니다.

int64_t LRUMiss : Cache Miss가 발생했을 때, 매개변수로 받은 블록을 cache에 로드해주는 역할을 합니다. 이때, 블록이 접근된 것이므로 doubly-linked list에서 head의 next에 배정합니다. 블록을 로드한 뒤, 블록의 수가 way의 수보다 클 때, tail의 prev 블록을 evict 해줍니다.

int listLen : doubly-linked list에서 블록의 수를 알아내는 역할을 합니다.

int64_t rmTail : tail의 prev 블록을 evict 하는 역할을 합니다. dirty의 조건에 따라 eviction 합니다.

void makeDirty : 블록을 dirty로 만드는 역할을 합니다. L1에서 dirty eviction이 일어날 때, L2의 같은 블록이 write가 될 수 있도록 합니다.

bool rmL1Block : L2에서 블록이 evict될 때, 같은 블록이 L1에 존재할 경우 L1에서도 해당 블록을 evict하는 역할을 합니다. 여기서 dirty eviction이 될 경우 true를 return합니다.

void minusClean : 해당 set에서 clean Eviction의 수를 하나 줄입니다. L2에서 clean 블록이 evict 될

때, 같은 블록이 L1에서는 dirty일 경우, L2 역시 dirty eviction이 일어나기 때문에 clean eviction을 하나 줄이고 dirty eviction을 하나 증가해야 합니다. 이는 plusDirty 함수가 담당합니다.
void plusDirty : 해당 set에서 dirty Eviction의 수를 하나 증가합니다.

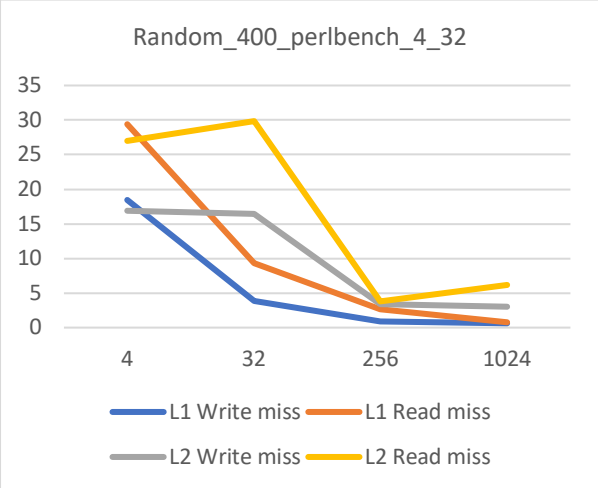
-struct BlockRan
//멤버 변수
int64_t data, addr
bool dirty, valid
data는 tag matching을 할 때 사용하는 변수입니다. addr은 trace파일의 addr을 블록마다 저장하기 위해서 사용하였습니다. dirty는 dirty, clean eviction을 판별하기 위해 사용하였습니다. valid는 set에서 유효한 인덱스를 판별하기 위해서 사용하였습니다.

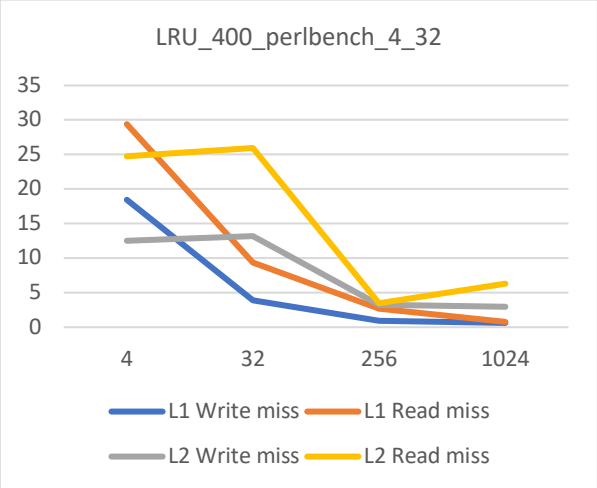
-struct ListRan
//멤버 변수
BlockRan* way
int size, dirtyEvict, cleanEvict, writeHit, writeMiss, readHit, readMiss
way로 BlockRan의 배열을 만들었습니다. size는 way수를 의미합니다. dirtyEvict와 cleanEvict는 set에서 eviction의 수를 축적하기 위해 사용하였습니다. Hit, Miss 또한 그것들의 수를 축적하기 위해서 사용하였습니다.
//멤버 함수
int RanSearch : way를 loop로 돌면서 tag matching을 합니다. matching되면 인덱스를 리턴합니다.
void RanHit : cache hit이 발생했을 때, 작동하는 함수입니다. Write access이면서 L1의 요청일경우, dirty 비트를 추가합니다.
int64_t RanMiss : cache miss가 발생했을 때, way에 블록을 로드할 공간이 없다면, 랜덤으로 블록을 evict합니다. 공간이 있다면, invalid 한 공간을 찾아 블록을 로드합니다.
int listLen : way에서 블록의 수를 알아내는 역할을 합니다.
bool rmL1Block, void minusClean, void plusDirty : ListLRU의 멤버함수와 역할이 동일합니다.

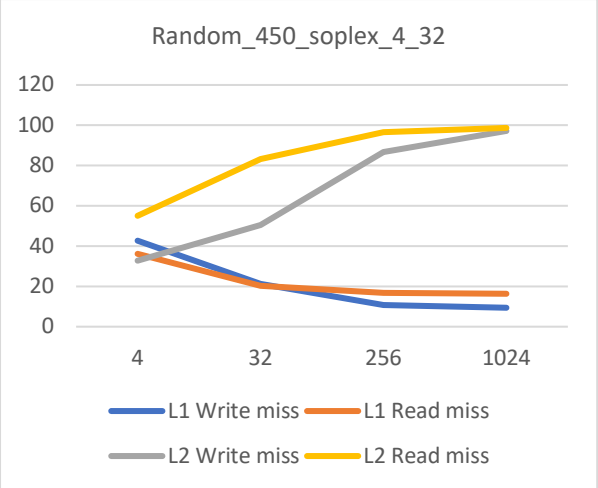
2. Two Level Cache - 트레이스 파일 및 패러미터에 따른 결과 분석

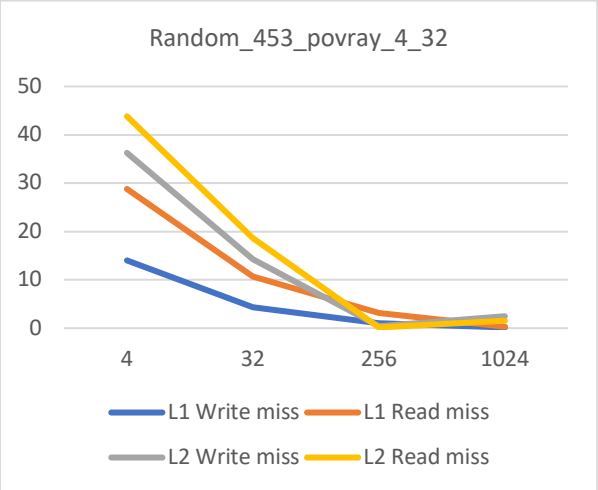
(1) 연관 정도 4, 블록 크기 32B
-그래프의 x축은 캐시의 용량이고 그래프의 y축은 miss rate입니다. 우선 6개의 트레이스 파일에 대하여 연관 정도를 4, 한 블록의 크기를 32B로 고정하고 캐시 용량을 4, 32, 256, 1024 KB로 변경해가며 측정한 miss rate를 LRU와 Random 방식으로 나누어 그래프를 기록하였습니다.
-트레이스 종류에 따른 분석 : perlbench, povray, astar 트레이스는 캐시 용량이 늘어남에 따라 L1 miss rate와 L2 miss rate가 줄어드는 경향이 있습니다. 반면, soplex, xalancbmk에서는 L1 miss rate는 줄어들지만, L2 miss rate는 오히려 증가하는 모습을 볼 수 있습니다. 이는 L1 miss, L2 miss가 일어나서 L1과 L2에 블록을 로드 했을 때, L1에서 hit만 발생하여 L2 접근이 이뤄지지 않아 L2 miss rate가 줄어들지 않고 고정 된 것으로 해석할 수 있습니다. 다시 말하자면, 일정 시점 이후에는 동일한 블록 접근만 반복 했다고 말할 수 있습니다. libquantum 트레이스 에서 L2 read miss rate와 L2 write miss rate가 100%입니다. 이는 n번은 겹치는 블록, n번은 겹치지 않는 블록을 접근하여 L1 miss, L2 miss로 L1, L2로 블록을 로드 하면서, 중간에 L1 hit를 일으키는 작업을 한 것으로 해석할 수 있습니다. 그래서 L1의 miss rate는 50%인 것입니다. 2n번의 블록 접근이 끝난 후, L1 cache에 있는 블록만 write

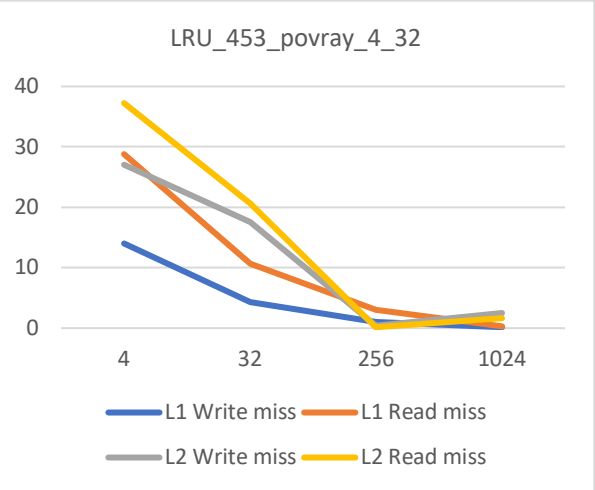
를 하여 L1 write miss rate는 0%입니다. 따라서 soplex, xalancbmk, libquantum 트레이스의 데이터 접근 방식은 너무 반복적이기 때문에 cache의 성능 향상을 위한 데이터 지표로 볼 수 없습니다. L1에만 접근하기 때문에 L2에서의 cache miss rate를 분석할 수 없습니다. multi level cache의 동작을 명확하게 분석하기 위해서는 perlbench, povray, astar 트레이스를 관찰하는 것이 더 효율적인 방법입니다.

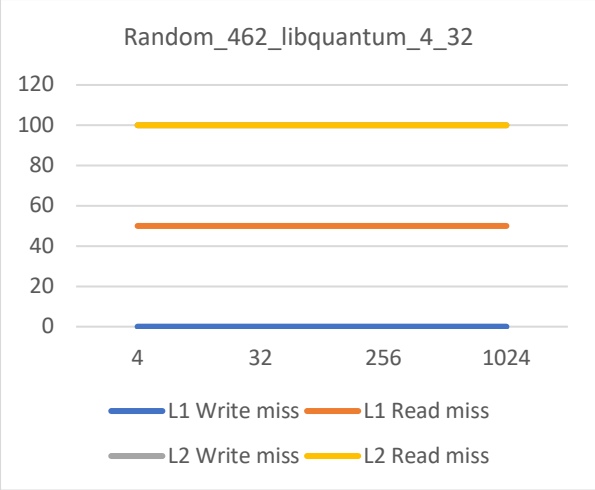


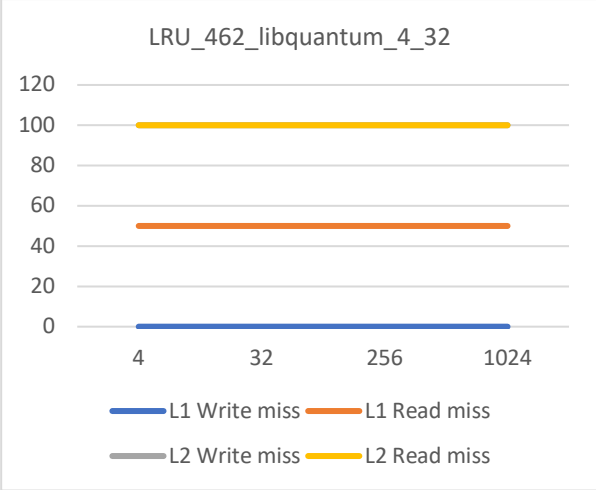




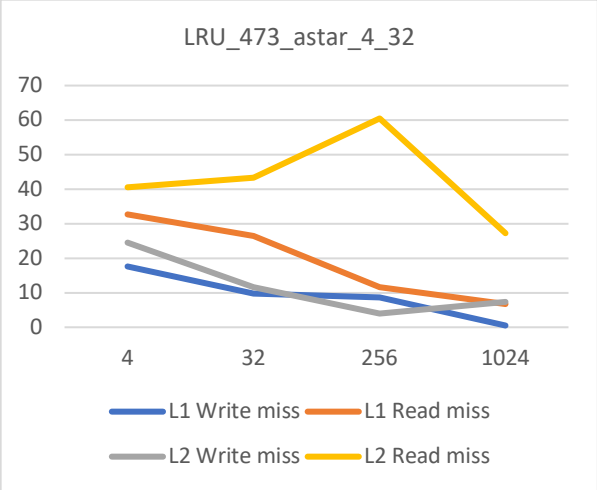


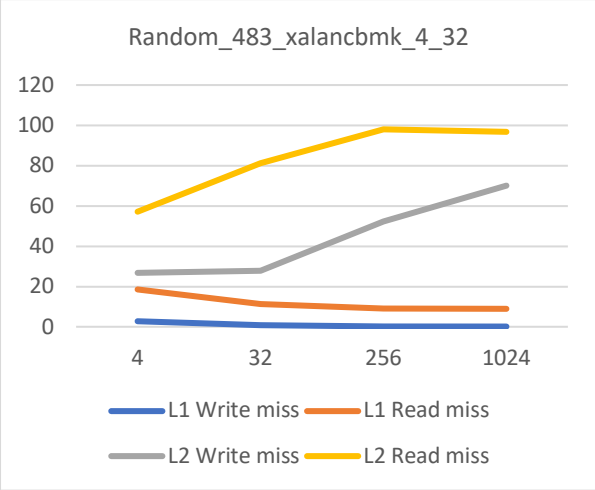


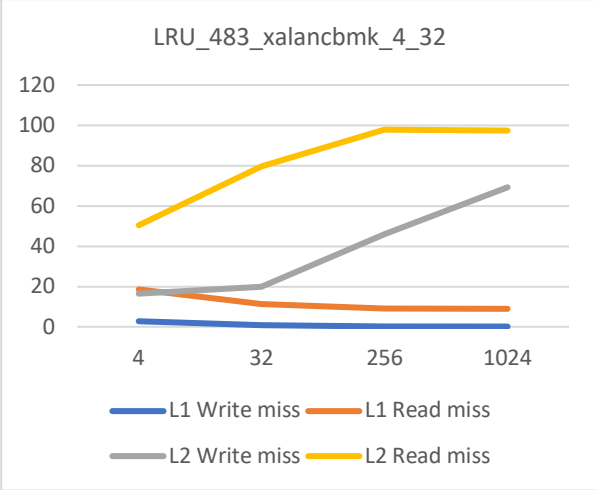




-libquantum 트레이스에서는 L2 Read Miss rate와 L2 Write Miss rate의 값이 똑같습니다.





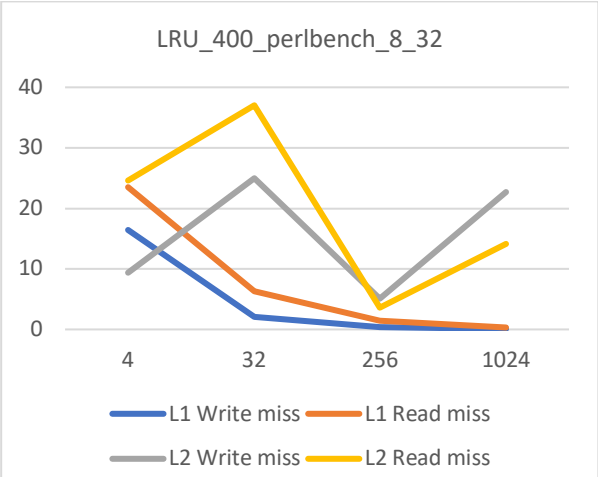


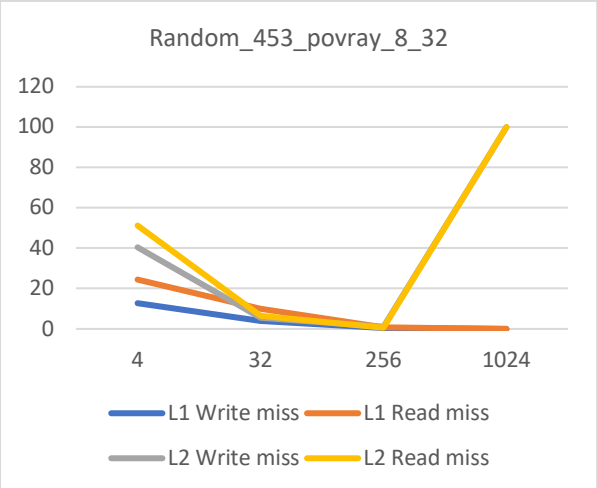
(2) 연관 정도 8, 블록 크기 32B (perlbench, povray, astar)

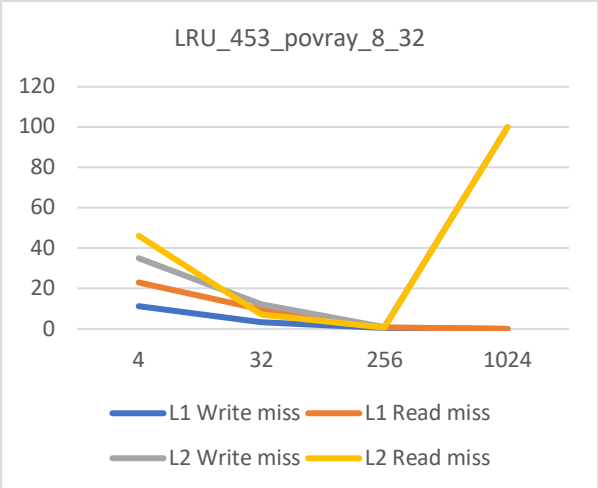
-(1)번 분석에서는 L1의 연관 정도가 1이기 때문에 L1의 miss rate는 LRU와 Random 방식이 같았습니다. 하지만 L2의 연관 정도는 4이기 때문에 L2의 miss rate는 LRU가 Random 방식보다 낮은 경향을 보였습니다. LRU와 Random의 miss rate 차이는 연관 정도가 커지면 더 두드러질 것입니다. 그래서 (2)번 그래프 분석을 진행할 것입니다. L2의 연관 정도는 8, L1의 연관 정도는 4입니다. perlbench, povary, astar 트레이스 파일에 대하여 연관 정도를 8, 한 블록의 크기를 32B로 고정하고 캐시 용량

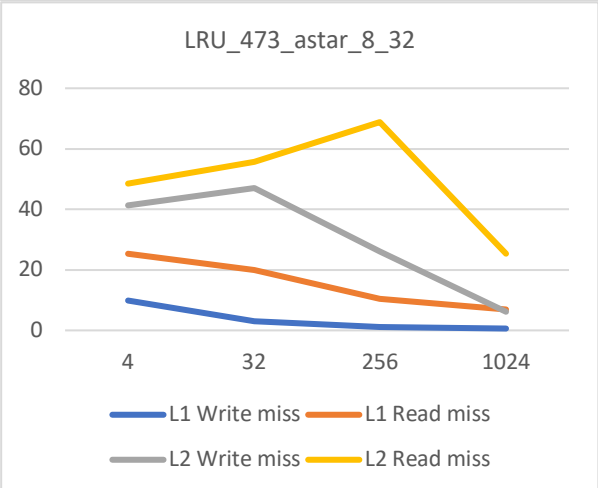
을 4, 32, 256, 1024 KB로 변경해가며 측정한 miss rate를 LRU와 Random 방식으로 나누어 그래프를 기록하였습니다.

-캐시 사이즈에 따른 분석 : 캐시 사이즈가 증가함에 따라 miss rate가 감소하는 경향성을 가집니다. 캐시 사이즈가 증가하면 set index가 증가하여 index가 겹칠 확률이 떨어져서 conflict miss가 줄어 듭니다. perlbench, povray의 1024KB 그래프에서 miss rate가 증가하는 이유는 L1 miss rate가 0으로 수렴하기 때문에 L2에 접근할 수 없으므로 L2의 miss rate가 높은 채로 유지 되는 것입니다.



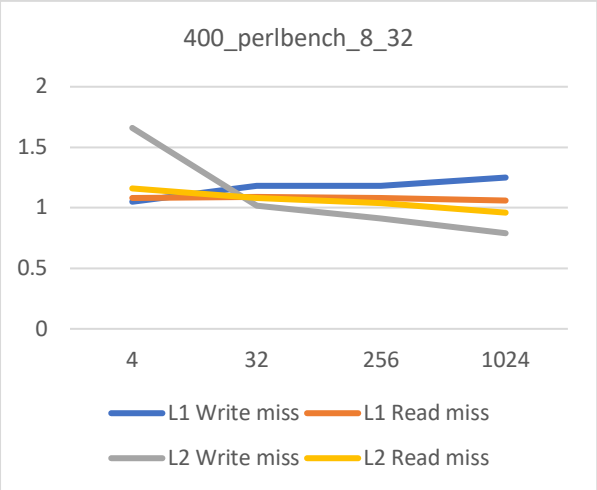


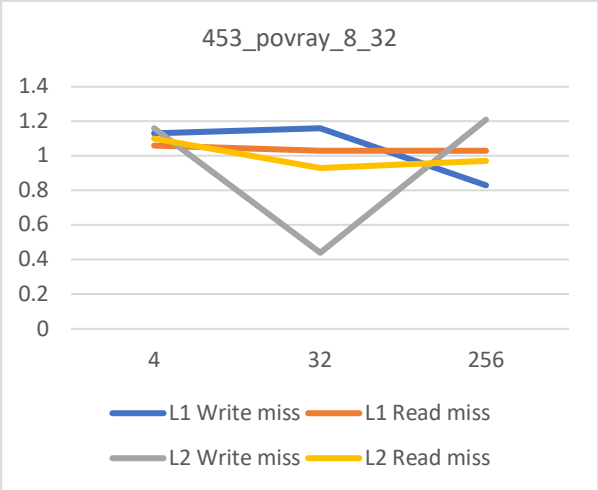


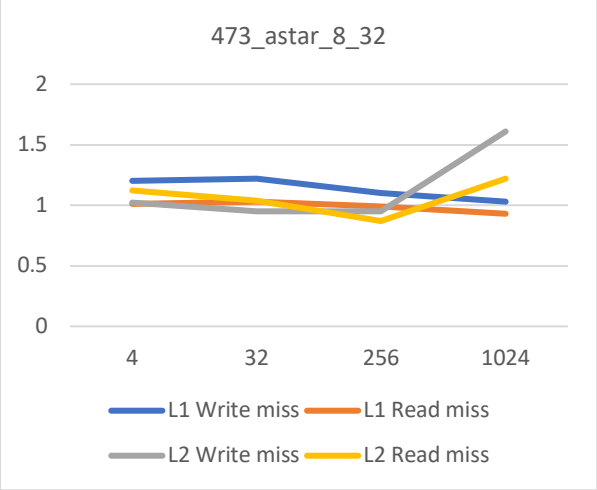


-Random Miss rate / LRU Miss rate 의 그래프

x축은 캐시 사이즈, y축은 (2)의 그래프에서 Random Miss rate / LRU Miss rate의 연산 값입니다. 해당 트레이스 파일에서 Random Miss rate가 LRU Miss rate보다 얼마나 큰지에 대한 값입니다. 즉 1보다 클수록 LRU에서의 Miss rate가 Random에서의 Miss rate보다 작다는 뜻입니다. 1보다 작을 수록 LRU에서의 Miss rate가 Random에서의 Miss rate보다 크다는 뜻입니다.







-블록 교체 정책에 따른 분석 : LRU와 Random 방식의 miss rate 차이를 명확하게 파악하기 위해 Random Miss rate / LRU Miss rate 그래프를 첨부합니다. 그래프를 보면, 대부분 1보다 큰 것을 알 수 있다. 그만큼 LRU 방식이 miss rate를 감소시켰다는 뜻입니다. 따라서 cache miss를 대략 1.3배 감소시킬 수 있습니다. Random방식이 아닌 temporal locality를 활용할 수 있는 LRU 방식을 사용해야 합니다. 캐시 용량을 증가하면서 miss rate를 감소시킬 수 있지만, hardware의 cost의 증가도 고려해야 합니다. LRU 방식을 software로 구현한다면, cost의 증가 없이 miss rate를 감소시킬 수 있습니다.

3. 컴파일/실행 환경, 방법

-컴파일/실행 환경

이 소스코드(main.cpp)는 Ubuntu 20.04.4 64bit 환경(VirtualBox)에서 작성되었습니다. 컴파일러는 g++ 9.4.0을 사용하였습니다.

-컴파일/실행

터미널에서 main.cpp와 input파일이 있는 디렉토리로 이동합니다. 터미널에 g++ -o runfile main.cpp를 입력합니다. 터미널에 ./runfile <-c capacity> <-a associativity> <-b block_size> <-lru 또는 -random> <trace file>을 입력합니다. 예를 들어 input파일이 400_perlbench.out 일경우, 터미널에 ./runfile -c 4 -a 4 -b 4 -lru 400_perlbench.out 를 입력합니다.