Computer architecture
Cache simulator Assignment Report

1. How to implement the cache simulator
1) 주요 data structure
먼저 instruction cache, data cache, L2 cache(inclusive), L2 cache(exclusive)에 대하여 총 4개의 3차원 배열을 동적할당 하였습니다. 첫 index로 들어가는건 cache주소의 index,즉 set의 순서의 번호이며, 두번째 factor는 한개의 set내에서의 block의 번호입니다. 한개의 set에는 최대 way의 수만큼의 blcok이 들어갑니다. 세번째 factor로는 한 블럭내의 각각의 자료를 가르키는 번호가 들어갑니다.(ex. valid bit=block[0],tag = block[1], dirty bit = block[2]).
또한 이러한 자료구조들은 함수 show_mr을 실행시킬때마다 구현됩니다.
이에 해당하는 부분의 코드는 다음과 같습니다.

```
122 void show_mr(int bsize, int way, int csize,int what_cache,char *link,int option) {
123     FILE *trace;
124     char str[20];           //파일위치를 다룰때 사용
125     long addr,blockaddr;    //addr에 string내의 16진수 숫자가 담기며, 이 byteadress를 blocksize로 나눈것이 blockaddr이다.
126     float L2_rate_in2, L2_rate_ex2;
127     int tmp,index,block,tag,i_cnt=0,i_num_hit=0,d_cnt=0,d_num_hit=0,num_wbin=0,num_wbex=0,victimtag,vblockaddr;
128     int num_set= get_set(bsize,way,csize),num_setL2 = get_set(bsize,8,16384);   //각각의 상황에 따른 L1 cache와 L2cache의 세트의 수를 계산
129     int ***icache = (int***) malloc(sizeof(int**)*num_set);
130     int ***dcache = (int***) malloc(sizeof(int**)*num_set);
131     int ***L2cache_in= (int***) malloc(sizeof(int**)*num_setL2);
132     int ***L2cache_ex= (int***) malloc(sizeof(int**)*num_setL2);
133     for ( index = 0 ; index < num_set ; index++) {
134         icache[index] = (int**)malloc(sizeof(int*)*way);           //index는 (set의 순서)를 의미한다.
135         dcache[index] = (int**)malloc(sizeof(int*)*way);
136         for( block = 0; block <way ; block++) {
137             icache[index][block] = (int*)calloc(sizeof(int),3);        //block은 block의 순서를 따지며
138             dcache[index][block] = (int*)calloc(sizeof(int),3);        //icache[i][j][0]부터 ...[2]까지 각각 valid bit,tag 정보,dirty bit를 의미
139         }                                                          //valid bit와 dirty bit의 0으로 초기화가 필요하므로 malloc함수 대신 calloc함수 사용
140     }
141     for ( index = 0 ; index < num_setL2 ; index++ ) {
142         L2cache_in[index] = (int**)malloc(sizeof(int*)*8);
143         L2cache_ex[index] = (int**)malloc(sizeof(int*)*8);
144             for(block=0; block<8; block++){
145                 L2cache_in[index][block] = (int*)calloc(sizeof(int),3);
146                 L2cache_ex[index][block] = (int*)calloc(sizeof(int),3);
147             }
148     }
149     int *L2_rate_in = (int*)calloc(sizeof(int), 3);
150     int *L2_rate_ex = (int*)calloc(sizeof(int),3);
151     float i_rate, d_rate;
152     char link2[30] = "trace/";
153     strcat(link2,link);
154     trace = fopen(link2,"r");
```

즉,다시 말해 예를 들어 dcache[3][2][0]은 data cache의 4번째 set의 3번째 block의 validbit의 정보를 담고있는 변수입니다.

또한 i_cnt, i_num_hit는 각각 instruction cache의 접근 횟수와 캐시의 히트 횟수를 카운팅해줌으로써, miss rate을 계산할 때 1-hit rate= 1-(히트 횟수/접근횟수)와 같은 방식으로 miss rate을 산정하여 printf에 정보를 줍니다. 이는 data cache, L2 cache에도 비슷한 방법으로 구현이되었습니다.

```
234     i_rate = (float)i_num_hit/i_cnt;
235     d_rate = (float)d_num_hit/d_cnt;
236     L2_rate_in2=(float)L2_rate_in[0]/L2_rate_in[1];
237     L2_rate_ex2=(float)L2_rate_ex[0]/L2_rate_ex[1];
238     //print시간
239     if(what_cache==0) printf("%5.2f%% |",(1-i_rate)*100);
240     else if(what_cache==1) {
241         if(option==1) printf("%7d|",num_wbin);
242         else if(option==2) printf("%7d|",num_wbex);
243         else printf("%5.2f%% |",(1-d_rate)*100);
244     }
245     else if(what_cache==2) printf("%5.2f%% |",(1-L2_rate_in2)*100);
246     else printf("%5.2f%% |",(1-L2_rate_ex2)*100);
247
```

num_wbin, num_wbex는 data cache의 memory write수를 카운팅해주는 변수입니다. 구동과정중에 이와 같이 카운팅이 됩니다.

```
else if(str[0]=='0'|| str[0]=='1') {                        //data read
    for(block=0; block<way ;block++) {              //index가 맞는 줄의 모든 block을 뒤진다.
        if(dcache[index][block][0]==1 && dcache[index][block][1]== tag) {
            d_num_hit++;                            //valid bit가 1이고 tag도 매칭되어 hit한 상황
            if(str[0]=='1')dcache[index][block][2]=1;// dirtybit만 1로만들고 지연쓰기 예약하고 캐시에쓰기
            LRUupdate(dcache[index],block,way);
            break;                                  //hit 했으므로 더이상 볼것없다 이 명령에 대한건 이제종료
        }
        else if (dcache[index][block][0]==0) {                  //tag matching도 아니고(miss) 빈 공간이있다
            dcache[index][block][0] = 1;
            dcache[index][block][1] = tag;
            dcache[index][block][2] = 0;
            //L1에 requested block 추가후 L2에 대한 접
            if(str[0]=='1' )num_wbin += addtoL2in(L2cache_in,blockaddr,L2_rate_in,num_setL2,icache,dcache,num_set,way,1,0,1);
            else addtoL2in(L2cache_in,blockaddr,L2_rate_in,num_setL2,icache,dcache,num_set,way,1,0,0);
            num_wbex += accesstoL2ex(L2cache_ex,blockaddr,L2_rate_ex,num_setL2,icache,dcache,num_set,way,1,0);
            break;
        }
        else if (block==way-1 && dcache[index][block][0] == 1) { //miss이고 전부 꽉 차있어서 evict해야하는 상황
            victimtag=dcache[index][0][1];
            vblockaddr=victimtag*num_set + index;
            num_wbin += addtoL2in(L2cache_in,vblockaddr,L2_rate_in,num_setL2,icache,dcache,num_set,way,1,1,0);
            num_wbex += accesstoL2ex(L2cache_ex,vblockaddr,L2_rate_ex,num_setL2,icache,dcache,num_set,way,1,1);

            if(str[0]=='1') {
                tmp = LRUevict(dcache[index],way,tag,0);
                num_wbin += tmp;
                num_wbex += tmp;
            }
            else LRUevict(dcache[index],way,tag,0);

            if(str[0]=='1') num_wbin += addtoL2in(L2cache_in,blockaddr,L2_rate_in,num_setL2,icache,dcache,num_set,way,1,0,1);
            else addtoL2in(L2cache_in,blockaddr,L2_rate_in,num_setL2,icache,dcache,num_set,way,1,0,0);
            num_wbex += accesstoL2ex(L2cache_ex,blockaddr,L2_rate_ex,num_setL2,icache,dcache,num_set,way,1,0);

        }
    }
    d_cnt++;
```

str[0]이 1일 때 즉 data cache에 write 명령이 들어갈때만 카운트가 되게하였습니다.

victimtag와 vblcokaddr은 evict되는 block에 대하여 작업을 시행하기 위하여 따로 만든 변수입니다. 아래는 victimtag와 vblockaddr이 사용되는 코드의 부분입니다.

```
else if (block==way-1 && icache[index][block][0] == 1) { //miss이고 전부 꽉 차있어서 evict해야하는 상황
    //L1의 block eviction에 앞서 victimblock에 대한 처리를 해준다. option ==1
    //blcok eviction에 따른 L2캐시 접근이므로 missrate엔 반영이 되지 않게 option을 1로 한다.
    victimtag=icache[index][0][1];
    vblockaddr=victimtag*num_set + index;
    addtoL2in(L2cache_in,vblockaddr,L2_rate_in,num_setL2,icache,dcache,num_set,way,0,1,0);
    accesstoL2ex(L2cache_ex,vblockaddr,L2_rate_ex,num_setL2,icache,dcache,num_set,way,0,1);
    // inclusive: Block eviction from L1 cache: move the victim block to L2 cache (do not update the LRU order of L2 cache.)
    // exclusive: Block eviction from L1 cache: move the victim block to L2 cache, and update the LRU order of L2 cache

    LRUevict(icache[index],way,tag,0);  //제거하고 requested block추가한다.
    addtoL2in(L2cache_in,blockaddr,L2_rate_in,num_setL2,icache,dcache,num_set,way,0,0,0);
    accesstoL2ex(L2cache_ex,blockaddr,L2_rate_ex,num_setL2,icache,dcache,num_set,way,0,0);
}
```

victimtag를 이용하여 vblockaddr이라는 evict되는 block의 blcokaddress를 구해줍니다. blocksize는 L1 cache와 L2 cache가 같으므로 이 vblockaddr이라는 변수를 addtoL2in, accesstoL2ex라는 L2 cache를 다루는 두 함수에 인자로 넘겨줍니다.

2) main algorithm

먼저 제가 구현한 LRU(Least Recent Used)전략을 설명하면 다음과 같습니다.

초기 상태의 set에 들어오는 blcok은 set내의 0번째에 위치시킵니다. 그다음에 오는 블록들은 차례차례 set내의 1번째,2번째..에 위치하고, 이렇게 위치한 블록들에 대하여 접근을 하는 일이 있으면 가장 뒤쪽인 max번째 블록으로 보냅니다.

따라서 LRU정책에 의해 block을 evict하는 경우 항상 set내의 0번째 block에 대하여 evict를 진행하게됩니다.

즉, 한 개의 set내의 blcok의 번호가 작을수록 즉 0번째가 가장 오래된 block이고, blcok의 번호가 클 수록 가장 최신 blcok이라고 할 수 있습니다.

그다음 명령을 인식하는 과정은 다음과 같습니다.

show_mr함수에서 파일 IO로 명령을 하나하나 읽어들이고 그에 따라 작업을 수행합니다.

먼저 구해놓은 num_set이라는 변수를 통해 cache주소의 index와 tag를 산출하고, 어떤 캐시로 접근을 할 것인지는 str[0]을 통해 유추하고, 그에 따라 index에 해당하는 set의 모든 block에 대하여 for문을 돌립니다.

- validbit가 1이고, tag가 매칭되는 경우 cache hit가 발생한 것으로 취급을 하고, cache hit count를 해주며, LRU순서를 update해주고 해당 for문을 break를 통해 빠져나옵니다.
- validbit가 1인 곳에서 tag가 매칭되지 않고, valid bit가 0인 빈 공간이 나타나는 경우, cache miss가 발생한 것으로 취급을 하고 cache hit의 count를 해주지 않으며 빈공간에 해당 주소의 block을 담습니다. 또한 L1 miss임으로 L2에 접근을 하는 함수 addtoL2in(inclusvie), accesstoL2ex(exclusive)들을 호출합니다.
- 그리고 마지막 경우로 tag matching이 안되어 cache miss긴 한데, 모든 set내의 공간이 꽉 차있는 경우입니다. 이 경우는 for문이 way-1번째까지 갔을 때도 valid bit가 1일 때 인식해내며, 이때는 다음과 같은 동작을 수행합니다.

먼저 LRU정책에 따라 삭제될 예정인 victimblock의 address를 다시 계산해줍니다. 그리고 victimblock에 대한 L2캐시의 process를 수행해줍니다.

그 과정이 끝나면, L1캐시내에서 LRU정책에 따른 가장 오래된 블록을 evict하고 requested block을 추가해줍니다. 그리고 L1 miss임으로 그에 따른 L2 cache에 대한 함수를 호출합니다.

```
while (fgets(str,20,trace)) {
    addr = get_num(str);                                    //예전에 잘못된 것 : word_addr(=n/4)/way **way는 영향을 끼치지 않는다 num_set을
    blockaddr = addr/bsize;
    index = blockaddr%num_set;                          // n/bsize는 tag부분 + index부분
    tag = blockaddr/num_set;                           // 그부분을 나눠줄수있는것이 num_set으로 나누기
    if(str[0]=='2') {                                       //instruction read
        for(block=0; block<way; block++) {                 //index가 맞는 줄의 모든 block을 뒤진다.
            if(icache[index][block][0]==1 && icache[index][block][1]== tag) {
                i_num_hit++;                                //valid bit가 1이고 tag도 매칭되어 hit한 상황
                LRUupdate(icache[index],block,way);        //접근하였으므로 LRU상의 가장 최신으로 업데이트해줌
                break;                                      //hit 했으므로 더이상 볼것없다 그냥 읽고 빠져나오자.
            }
            else if (icache[index][block][0]==0) {          //앞의 valid bit가 1인공간을 뒤졌으나 tag에 맞는 block이 존재하지 않았다
                                                            //->cache miss이고, 또한 빈 공간이 존재한다.
                icache[index][block][0] = 1;                //가장 최신블록에 블록을 추가
                icache[index][block][1] = tag;
                icache[index][block][2] = 0;
                //L1 miss이므로 이에 따라 L2 cache에 접근한다.
                addtoL2in(L2cache_in,blockaddr,L2_rate_in,num_setL2,icache,dcache,num_set,way,0,0,0);
                accesstoL2ex(L2cache_ex,blockaddr,L2_rate_ex,num_setL2,icache,dcache,num_set,way,0,0);
                break;
            }
            else if (block==way-1 && icache[index][block][0] == 1) { //miss이고 전부 꽉 차있어서 evict해야하는 상황
                //L1의 block eviction에 앞서 victimblock에 대한 처리를 해준다. option ==1
                //block eviction에 따른 L2캐시 접근이므로 missrate엔 반영이 되지 않게 option을 1로 한다.
                victimtag=icache[index][0][1];
                vblockaddr=victimtag*num_set + index;
                addtoL2in(L2cache_in,vblockaddr,L2_rate_in,num_setL2,icache,dcache,num_set,way,0,1,0);
                accesstoL2ex(L2cache_ex,vblockaddr,L2_rate_ex,num_setL2,icache,dcache,num_set,way,0,1);
                // inclusive: Block eviction from L1 cache: move the victim block to L2 cache (do not update the LRU order of L2 cache.)
                // exclusive: Block eviction from L1 cache: move the victim block to L2 cache, and update the LRU order of L2 cache

                LRUevict(icache[index],way,tag,0);  //제거하고 requested block추가한다.
                addtoL2in(L2cache_in,blockaddr,L2_rate_in,num_setL2,icache,dcache,num_set,way,0,0,0);
                accesstoL2ex(L2cache_ex,blockaddr,L2_rate_ex,num_setL2,icache,dcache,num_set,way,0,0);
            }
        }
        i_cnt++;
    }
```

위의 셋중의 분기가 끝나면 i_cnt를 카운팅해주어 instruction cache에 접근했다는 것을 구현해줍니다.
위와 같은 방식으로 data cache에 대해서도 코드가 작동하게 됩니다.
그리고 show_mr함수 마지막에는 함수의 인자 what_cache(0:instruction cache, 1:data cache: 2: L2cache inclusive 3:L2cache exclusive)와 ,
option(1:data cache's the number of memory write at inclusvie L2 cache, 2: data cache의 memory write number at exclusive L2 cache, 3: data cache(L1 cache)'s miss rate)에 따라 표안의 내용을 출력해줍니다.

```
234     i_rate = (float)i_num_hit/i_cnt;
235     d_rate = (float)d_num_hit/d_cnt;
236     L2_rate_in2=(float)L2_rate_in[0]/L2_rate_in[1];
237     L2_rate_ex2=(float)L2_rate_ex[0]/L2_rate_ex[1];
238     //print시간
239     if(what_cache==0) printf("%5.2f%% |",(1-i_rate)*100);
240     else if(what_cache==1) {
241         if(option==1) printf("%7d|",num_wbin);
242         else if(option==2) printf("%7d|",num_wbex);
243         else printf("%5.2f%% |",(1-d_rate)*100);
244     }
245     else if(what_cache==2) printf("%5.2f%% |",(1-L2_rate_in2)*100);
246     else printf("%5.2f%% |",(1-L2_rate_ex2)*100);
247
```

함수 LRUevict의 작동원리는 다음과 같습니다. 인자 option이 0인 경우 LRU정책에 따른 가장 오래된 blcok을 evict하고 최신블록을 집어넣고, option이 1인 경우 해당되는 blcok에 대하여 삭제만을 진행합니다.
인자로 int이중포인터 set를 받아 캐시내의 특정 set에 대한 정보를 확보하여 tag를 할당하거나 비교에 이용하여 작업을 진행합니다.
기본적으로 for문을 돌며 임의의 한개의 set내에서 block을 전부 수색하면서 block의 index가 작은쪽으로 복사를 계속 하며 땡겨서 지우는 형식이 option0일때와 1일때에 공통적으로 적용이 됩니다. 함수 LRUevict의 코드는 다음과 같습니다. return되는 인자 result에는 victim block의 dirty bit가 1인 경우 counting을 해주게 됩니다.

```
int LRUevict(int** set,int way,int tag,int option) {

    int result = 0,block;
    if(option== 0) {                //option 0
        if(set[0][2] ==1) {   //dirty bit on, 지연쓰기
            set[0][2]=0;      //동시에 dirtybit초기화
            result++;           //memory block wirte를 co
        }
        //0에서 way-2까지 설정
        for(block=0; block<way-1; block++) {
            set[block][0]=set[block+1][0];
            set[block][1]=set[block+1][1];
            set[block][2]=set[block+1][2];
            if(set[block][0] == 0) break;       //block0
        }                                       // way-1
                                                // 전부다
                                                // 따라서
                                                // 그외 w
        set[block][0] = 1;      //마지막 블록은 최신 블록
        set[block][1] = tag;    //최신블록갱신
        set[block][2] = 0;
    }
```

LRUupdate도 LRUevict와 비슷한 방식으로 앞으로 땡겨서 LRU순서를 조정하고 해당되는 블록을 최신블록의 위치에 위치시킵니다.

addtoL2in함수와 accesstoL2ex함수는 L2cache의 inclusive와 exclusive에 대하여 다룬 함수로써 option이 0일때와 1일때를 적용하는 경우를 제외하면 L1 cache에서 작동한 방식과 비슷하되 PDF의 L2 정책에 따라 작동하게 구현하였습니다.

On inclusive policy, the L2 cache must include all the data in the L1 cache. (L1 ⊂ L2)

- L1 hit: update the LRU order of L1 cache blocks.
- L1 miss & L2 hit: add the requested block in L1 cache, update the LRU order of L2 cache blocks.
- L1 miss & L2 miss: add the requested block in both L1 cache and L2 cache.
- Block eviction from L1 cache: move the victim block to L2 cache (do not update the LRU order of L2 cache).
- When a block is evicted from L2 cache, the same block in L1 cache must also be evicted.

On exclusive policy, a cached block should be in either L1 cache or L2 cache, never in both. (L1 ∩ L2 = ∅)

- L1 hit: update the LRU order of L1 cache blocks.
- L1 miss & L2 hit: add the requested block in L1 cache, remove the block in L2 cache.
- L1 miss & L2 miss: add the requested block only in L1 cache.
- Block eviction from L1 cache: move the victim block to L2 cache, and update the LRU order of L2 cache

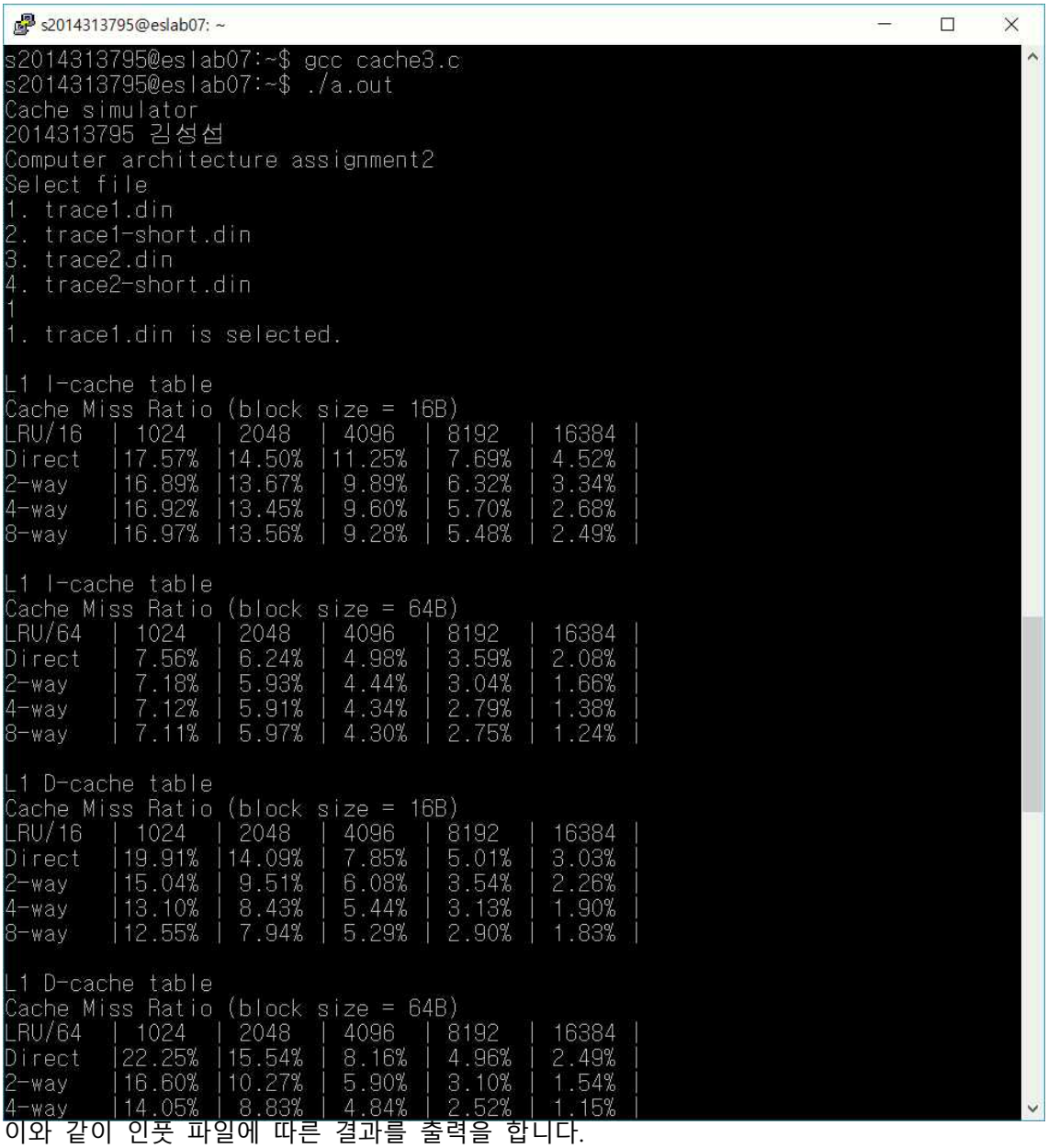
```
int addtoL2in(int*** L2cache, int blockaddr,int* L2rate,int num_setL2,int*** icache, int*** dcache, int num_setL1,int wayL1,int isDcache,int option,int
        //option:0 -> L1 miss& L2 hit or miss, missrate 계산이 반영
        //option:1 -> L1 eviction에 수반되어 발생한 함수,따라서 missrate 계산x
        //option:1 -> Block eviction from L1 cache: move the vitim block to L2 cache(do not update the LRU order of L2 cache)
```

option인자가 0이면 L1cache miss가 발생했을때 호출된 경우로 L2 cache hit miss를 따져 missrate계산에 반영이 되게 하였습니다.
반면 1인 경우는 eviction에 따른 호출로 인식하여 L2 cache의 miss rate계산에 산정이 되지 않고 block을 할당하는 부분만 작동하게 구현을 하였습니다.

cache simulator 작동법
./a.out으로 프로그램을 구동하면

```
s20143137795@eslab07:~$ ./a.out
Cache simulator
2014313795 김성섭
Computer architecture assignment2
Select file
1. trace1.din
2. trace1-short.din
3. trace2.din
4. trace2-short.din
```

이와 같이 printf가 작동을 하고 1,2,3,4 네개의 숫자중에 입력을 하면



이와 같이 인풋 파일에 따른 결과를 출력을 합니다.

trace파일은 상대경로 기준 trace/에 trace1.din, trace1-short.din, trace2.din trace2-short.din 으로 저장된 상태여야합니다.

3) 결과 table

1.trace1.din

L1 I-cache table

Cache Miss Ratio (block size = 16B)

LRU/16	1024	2048	4096	8192	16384
Direct	17.57%	14.50%	11.25%	7.69%	4.52%
2-way	16.89%	13.67%	9.89%	6.32%	3.34%
4-way	16.92%	13.45%	9.60%	5.70%	2.68%
8-way	16.97%	13.56%	9.28%	5.48%	2.49%

L1 I-cache table

Cache Miss Ratio (block size = 64B)

LRU/64	1024	2048	4096	8192	16384
Direct	7.56%	6.24%	4.98%	3.59%	2.08%
2-way	7.18%	5.93%	4.44%	3.04%	1.66%
4-way	7.12%	5.91%	4.34%	2.79%	1.38%
8-way	7.11%	5.97%	4.30%	2.75%	1.24%

L1 D-cache table

Cache Miss Ratio (block size = 16B)

LRU/16	1024	2048	4096	8192	16384
Direct	19.91%	14.09%	7.85%	5.01%	3.03%
2-way	15.04%	9.51%	6.08%	3.54%	2.26%
4-way	13.10%	8.43%	5.44%	3.13%	1.90%
8-way	12.55%	7.94%	5.29%	2.90%	1.83%

L1 D-cache table

Cache Miss Ratio (block size = 64B)

LRU/64	1024	2048	4096	8192	16384
Direct	22.25%	15.54%	8.16%	4.96%	2.49%
2-way	16.60%	10.27%	5.90%	3.10%	1.54%
4-way	14.05%	8.83%	4.84%	2.52%	1.15%
8-way	13.60%	8.09%	4.51%	2.37%	1.08%

L2 Miss Ratio (block size = 16 B) (Inclusive)

LRU/16	1024	2048	4096	8192	16384
Direct	20.17%	25.13%	32.54%	40.08%	51.71%
2-way	22.30%	28.45%	38.59%	50.13%	66.44%
4-way	22.91%	29.47%	39.30%	55.78%	77.93%
8-way	23.03%	29.40%	40.52%	55.79%	82.66%

L2 Miss Ratio (block size = 64 B) (Inclusive)

LRU/64	1024	2048	4096	8192	16384
Direct	20.18%	26.27%	35.80%	41.11%	47.98%
2-way	23.75%	31.92%	44.24%	53.37%	60.74%
4-way	25.58%	33.39%	45.57%	59.94%	70.65%
8-way	25.84%	34.14%	46.34%	59.06%	77.22%

L2 Miss Ratio (block size = 16 B) (Exclusive)

LRU/16	1024	2048	4096	8192	16384
Direct	27.80%	32.50%	35.30%	37.54%	47.16%
2-way	30.37%	35.06%	39.92%	44.20%	58.95%
4-way	30.82%	35.10%	39.85%	48.33%	69.94%
8-way	30.74%	34.49%	39.72%	48.09%	74.28%

L2 Miss Ratio (block size = 64 B) (Exclusive)

LRU/64	1024	2048	4096	8192	16384
Direct	27.69%	33.20%	38.38%	38.51%	43.82%
2-way	31.96%	38.13%	44.55%	46.74%	52.71%
4-way	33.91%	39.00%	44.79%	51.02%	61.34%
8-way	34.22%	39.03%	44.53%	49.58%	67.01%

L1 D-cache table

Number of Memory Block Writes (Inclusive)

LRU/16	1024	2048	4096	8192	16384
Direct	4680	3781	2312	1751	1107
2-way	4198	2622	2040	1557	969
4-way	4251	2622	1879	1573	1015
8-way	4196	2477	1826	1683	1062

L1 D-cache table

Number of Memory Block Writes (Inclusive)

LRU/64	1024	2048	4096	8192	16384
Direct	3443	2785	1187	849	483
2-way	2942	1644	942	630	385
4-way	2538	1447	799	534	351
8-way	2581	1373	797	522	372

L1 D-cache table

Number of Memory Block Writes (Exclusive)

LRU/16	1024	2048	4096	8192	16384
Direct	25562	19699	8450	5111	2692
2-way	16863	9340	5060	2844	1416
4-way	14765	8089	4645	2398	1141
8-way	14020	7540	4442	2564	1102

L1 D-cache table

Number of Memory Block Writes (Exclusive)

LRU/64	1024	2048	4096	8192	16384
Direct	30910	25095	10072	5827	2626
2-way	19218	10756	4945	2077	1034
4-way	16040	8161	3307	1117	520
8-way	15433	6961	2923	1097	406

2.trace2.din

L1 I-cache table

Cache Miss Ratio (block size = 16B)

LRU/16	1024	2048	4096	8192	16384
Direct	11.63%	6.91%	4.67%	3.15%	1.47%
2-way	10.80%	6.17%	3.36%	1.74%	0.80%
4-way	9.98%	5.26%	3.11%	1.43%	0.45%
8-way	9.59%	4.67%	2.95%	1.31%	0.40%

L1 I-cache table

Cache Miss Ratio (block size = 64B)

LRU/64	1024	2048	4096	8192	16384
Direct	6.01%	3.86%	2.29%	1.48%	0.67%
2-way	5.74%	3.51%	1.61%	0.84%	0.38%
4-way	5.74%	3.15%	1.30%	0.77%	0.25%
8-way	5.86%	3.03%	1.23%	0.82%	0.21%

L1 D-cache table

Cache Miss Ratio (block size = 16B)

LRU/16	1024	2048	4096	8192	16384
Direct	15.45%	11.10%	8.70%	3.81%	2.69%
2-way	10.87%	4.73%	2.93%	1.14%	0.81%
4-way	9.05%	3.33%	1.32%	0.93%	0.68%
8-way	8.33%	3.33%	1.24%	0.91%	0.64%

L1 D-cache table

Cache Miss Ratio (block size = 64B)

LRU/64	1024	2048	4096	8192	16384
Direct	15.95%	11.35%	8.24%	3.48%	2.24%
2-way	11.40%	5.54%	3.91%	1.45%	0.34%
4-way	9.93%	3.20%	1.34%	0.40%	0.26%
8-way	10.50%	2.89%	1.35%	0.37%	0.25%

L2 Miss Ratio (block size = 16 B) (Inclusive)

LRU/16	1024	2048	4096	8192	16384
Direct	5.71%	8.86%	11.51%	16.43%	26.60%
2-way	6.71%	11.75%	18.91%	33.43%	54.78%
4-way	7.46%	14.40%	21.81%	39.86%	85.33%
8-way	7.82%	15.94%	22.78%	42.00%	92.52%

L2 Miss Ratio (block size = 64 B) (Inclusive)

LRU/64	1024	2048	4096	8192	16384
Direct	6.24%	9.36%	12.15%	14.34%	17.26%
2-way	7.39%	13.14%	20.49%	28.34%	43.14%
4-way	7.69%	16.15%	31.71%	38.28%	61.06%
8-way	7.44%	17.06%	33.08%	33.72%	76.48%

L2 Miss Ratio (block size = 16 B) (Exclusive)

LRU/16	1024	2048	4096	8192	16384
Direct	12.05%	14.95%	13.69%	16.12%	24.52%
2-way	13.33%	18.77%	21.69%	30.84%	50.95%
4-way	14.15%	20.22%	24.74%	36.93%	80.85%
8-way	14.66%	21.91%	24.49%	39.02%	87.99%

L2 Miss Ratio (block size = 64 B) (Exclusive)

LRU/64	1024	2048	4096	8192	16384
Direct	13.06%	15.96%	14.28%	13.92%	16.00%
2-way	14.88%	20.16%	22.86%	23.06%	39.75%
4-way	15.31%	24.26%	33.75%	31.93%	57.43%
8-way	14.69%	24.69%	32.75%	28.65%	64.95%

L1 D-cache table

Number of Memory Block Writes (Inclusive)

LRU/16	1024	2048	4096	8192	16384
Direct	1646	1462	1116	723	165
2-way	1490	1120	823	403	204
4-way	1280	942	598	388	134
8-way	1165	961	528	390	120

L1 D-cache table

Number of Memory Block Writes (Inclusive)

LRU/64	1024	2048	4096	8192	16384
Direct	1265	918	600	345	68
2-way	1385	513	355	167	74
4-way	1157	384	242	121	64
8-way	1148	368	249	111	67

L1 D-cache table
Number of Memory Block Writes (Exclusive)

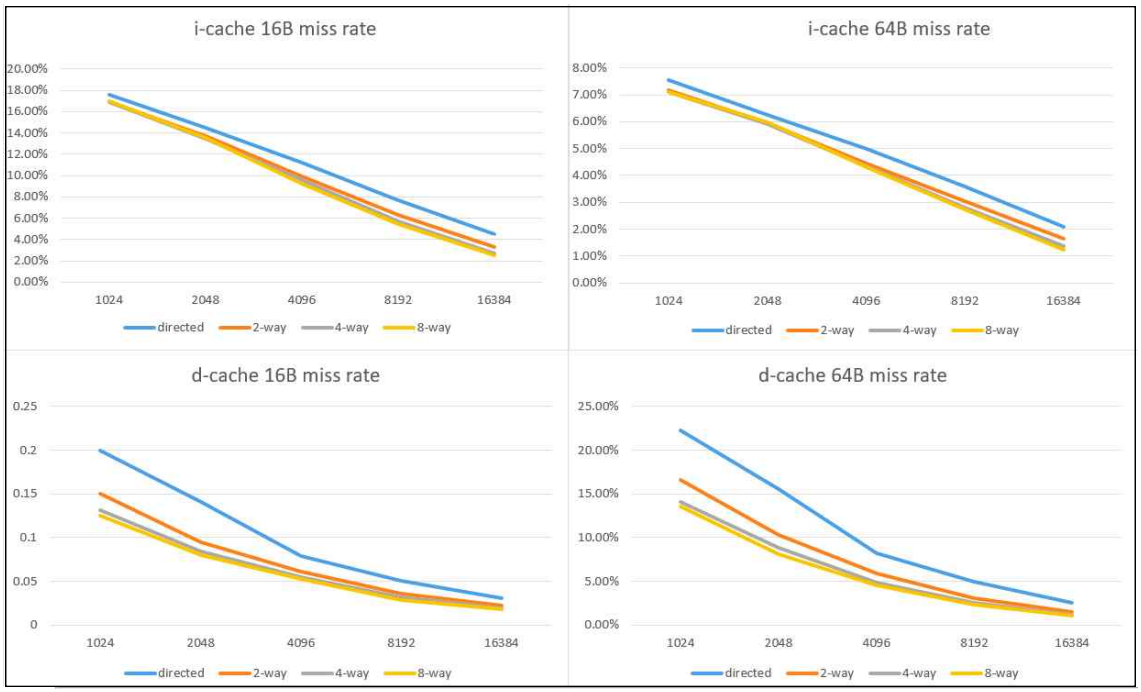
LRU/16	1024	2048	4096	8192	16384
Direct	21761	19311	15867	5936	3905
2-way	14747	5848	4018	755	257
4-way	11859	3046	1075	528	130
8-way	10762	3029	979	553	119

L1 D-cache table
Number of Memory Block Writes (Exclusive)

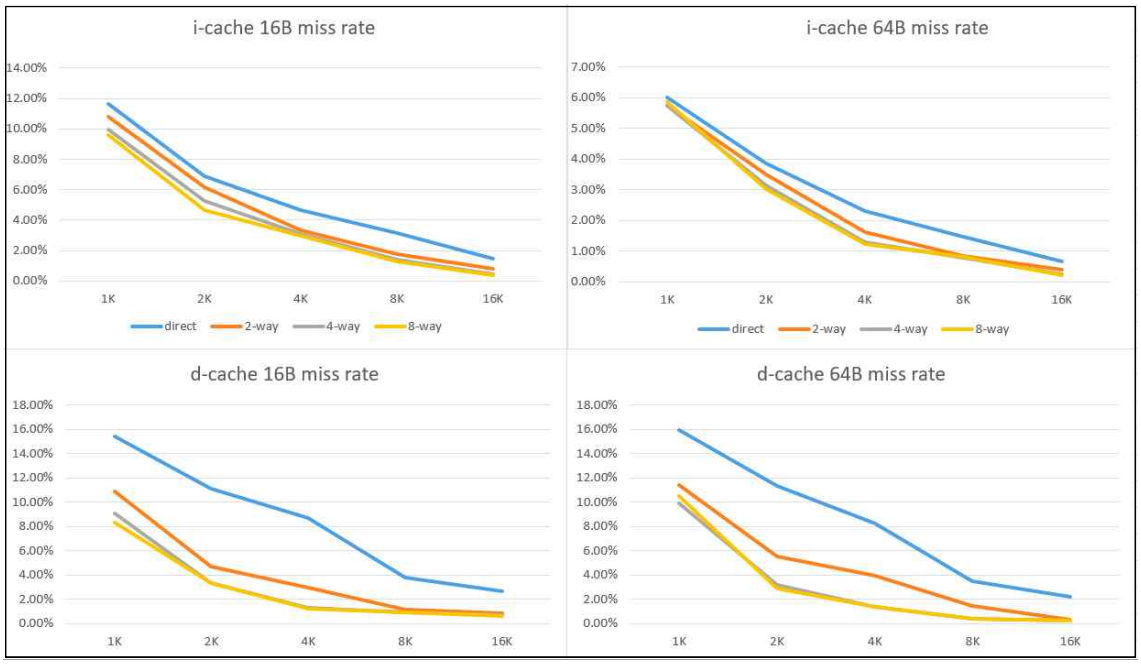
LRU/64	1024	2048	4096	8192	16384
Direct	23445	20532	15526	6136	4071
2-way	17237	7472	6291	2311	131
4-way	14414	3151	805	193	70
8-way	15719	2651	776	214	69

4. 결과 분석 그래프

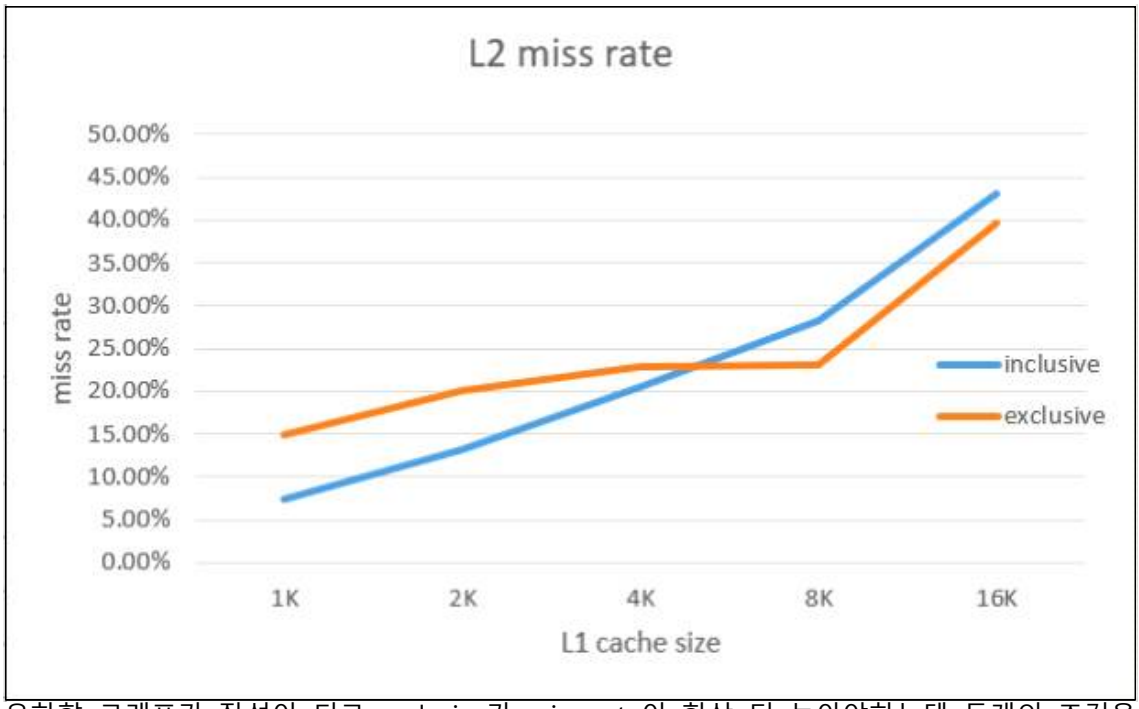
Trace1.din



Trace2.din

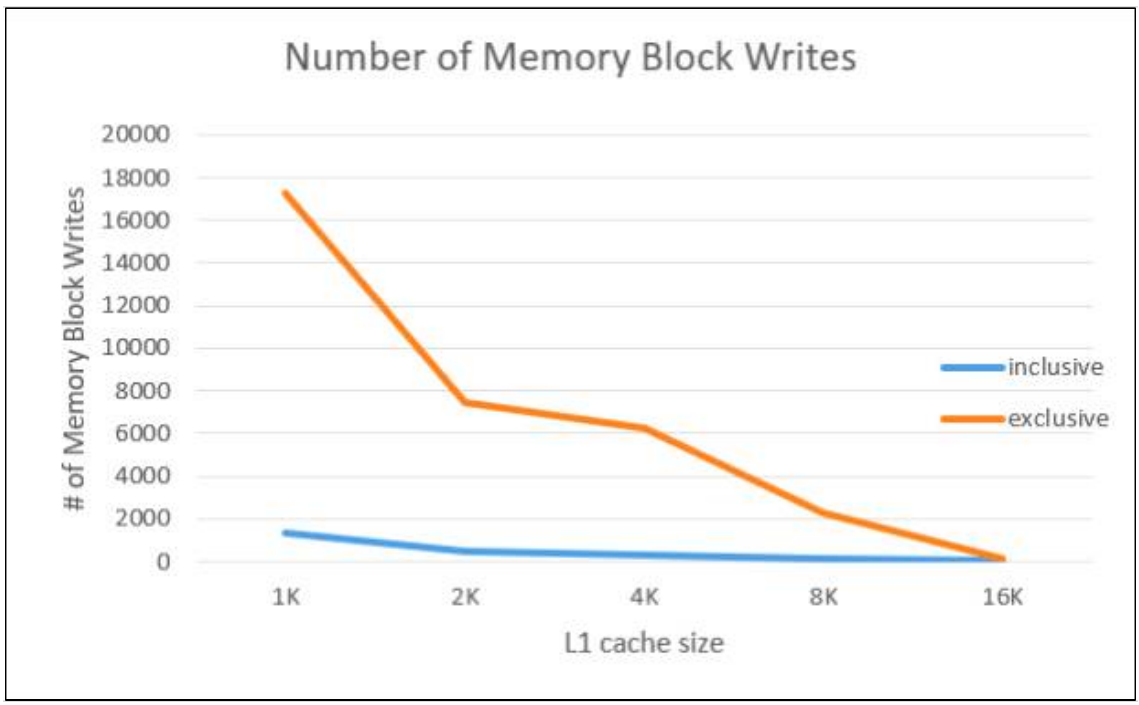


L2 miss rates graph (block size is 64 bytes and the associativity is 2-way.)



우하향 그래프가 작성이 되고 exclusive가 miss rate이 항상 더 높아야하는데 두개의 조건을 전부 만족하지 않는 그래프가 작성이 되었다. 구현상에 버그가 있는 것으로 추정이 된다.

# of memory writes (block size is 64 bytes and the associativity is 2-way.)



이론대로 inclusive가 더욱 적은 메모리 접근 횟수를 나타내고 L1 cache size가 증가함에 따라 write횟수도 감소하는 경향성을 나타내었다.