2021 SYSTEM PROGRAMMING

Lab4 Report – Cache Lab

자유전공학부 2012-13311 안 효 지

1. How to implement

<Part A-cache simulator>

0) Cache structure

```
typedef struct{
    bool valid;
    u_int64_t tag;
    size_t LRUcnt;
} Line;

typedef struct{
    Line* lines;
} Set;

typedef struct{
    Set* sets;
} Cache;
```

```
/*
 <Overall structure>
     |     |     | ------------
    cache --> | [set 1] | --> [line1], [line2], ... , [lineE]
     |     |     | [set 2] | --> [line1], [line2], ... , [lineE]
     |     |     |    ...  |
     |     |     | [set s] | --> [line1], [line2], ... , [lineE]
     |     |     | ------------
*/
```

캐쉬는 2차원 배열이므로 row에는 set을, column에는 line을 둔다. Cache와 set은 포인터 형태로, 각각 set의 배열과 line의 배열을 가리키도록 구성하였다.

우리의 과제는 hits/misses/evictions만 count하면 되기 때문에 실제로 데이터가 들어가는 block(offset)은 공간을 할당할 필요가 없다. 따라서, input으로 들어오는 size도 무시해도 되며, Line에는 valid, tag, LRUcnt만 존재하도록 설계했다.

1) Main function

(1) Parsing command line

Getopt()를 이용해 command line을 parsing한다. Handout에 나와있는 예제를 활용하였다. 모든 필수 option의 입력을 보장하기 위해 opt_cnt라는 변수를 만들어 입력된 옵션의 개수를 세었다. 그리고 캐시 구조상 불가능한 값(E == 0, s == 0, b == 0), file open의 실패, 필수 option이 다 들어오지 않은 상태는 오류처리를 해주었다. 그리고 verbose를 구현하기 위해 bool type global var인 verbose를 정의하고, -v 옵션이 insert 된 경우에만 verbose = true를 하여 필요한 자료들이 print되도록 만들었다.

(2) Initializing cache

Cache안에 들어있는 set의 사이즈*개수만큼 malloc으로 할당을 한다. 어짜피 곧 line의 공간이 할당되면 그 array를 가리키는 pointer가 들어올 것이므로 굳이 0으로 initialize는 하지 않아도 된다.
그리고 line size * 개수만큼 calloc으로 할당을 한다. Line 안의 내용은 0으로 초기화가 되어야 처음에 tag, valid bit을 비교할 수 있으므로 calloc을 사용하였다.

(3) Parsing file input by fscanf

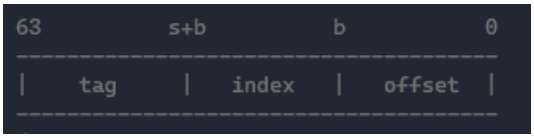
파일로 들어오는 L, S, M, I 등의 인풋을 파싱한다. I는 본 과제에서 무시한다. I를 제외한 나머지 identifier들은 맨 앞에 space가 들어가는데, `fscanf(filep, " %c %lx,%d", &identifier, &addr, &size)`를 이용해 파싱을 해보려 했으나, 앞에 space가 없는 I까지도 이 조건에 해당되는 것으로 처리되어서 그냥 switch문을 통해 I를 처리에서 제외해주었다.
Modify인 'M'은 Load->Store를 하는 것으로 보아, lookup을 두 번 호출하였고, 나머지 L, S는 lookup을 한 번만 호출하였다.

(4) Close and free

모든 작업을 마친 후에는 파일을 닫고, 할당했던 heap을 free해준다.

2) lookup function

(0) Extracting Tag and Index



64bit address에서 tag와 index를 추출한다. 왼쪽과 같은 구조이므로 tag는 addr >> (s+b) 해주면 된다. 어짜피 addr는 unsigned이므로 >>는 자동으로 logical shift right로 수행이 된다. 따라서 (addr >> (s+b)) 에다가 & 0xffffffffffffffff 를 해줄 필요는 없다.
index를 추출하기 위해, 처음에는 Tag bit만큼 <<를 하고, Tag+offset bit만큼 >>를 하려고 했다`index = (addr << (63-(s+b))) >> (s+2*b);`. 요즘에는 arithmetic operation을 bitwise operation으로 컴파일러가 알아서 바꾸어준다고는 하지만, 그래도 불필요한 arithmetic oper대신 bit oper를 사용하기 위해 `(0xffffffffffffffff >> (64-s)) & (addr >> b)`으로 코드를 짰다.

(1) index를 이용해 해당하는 set을 선택

(2) cache hit

set의 line들을 iterate하면서 valid bit == 1인지, input과 tag가 일치하는지 확인한다. 일치하면 hit이므로 LRU cnt만 update해주면 된다. Hit_cnt도 ++해준다.

(3) cache miss

Hit이 안 된 경우를 miss라고 한다. 이 때에는 비어있는 line이 있는지 확인하고, 있으면 거기에 넣는다. Valid = false이면 empty라는 의미이다. Miss_cnt++.

(4) no empty lines

Empty line이 없는 경우 replace policy에 의거하여 replace를 해야 한다. 본 과제에서는 LRU(lease recently used)를 사용한다. Evict_cnt++. 나머지는 LRUupdate()에서 설명.

3) LRUupdate()

	1st	2nd	3rd	4th	5th
L1	0	v 3	2	v 3	2
L2	0	0	v 3	2	1
L3	0	0	0	0	v 3
L4	0	0	0	0	0

일단 처음은 모두 0을 갖는다. 가장 최근에 사용된 라인은 E-1의 LRU값을 갖는다. 현재 사용되고 있는 line의 LRU보다 큰 LRU는 -1씩 해준다. 4th를 예로 들면, 업데이트 되기 전에 L1의 LRU는 2였다. 따라서, 3rd에서 2보다 큰 값을 갖고 있던 L2는 -1을 해주고 나머지는 그대로 둔다. 그리고 L1은 E-1의 값으로 리셋을 해준다. 5th도 마찬가지로, 현재 사용되는 L3의 업데이트 전 LRU는 0이므로, 4th에서 0보다 큰 값을 갖고 있던 L1과 L2는 각각 -1을 해준다. 그리고 L3는 E-1의 값으로 리셋을 해준다. 이렇게 하는 경우, LRU가 0인 라인 아무거나 evict하면 된다.

<Part B – Cache friendly>

먼저, E = 1이므로 라인은 하나인 direct mapped cache이고, Index(set)은 총 32개이다. b = 5이므로 block size는 32byte로서 총 8개의 integer를 하나의 블록에 저장한다. 따라서, 매트릭스를 8x8의 submatrix로 쪼개도록 할 것이다.

1) 32 X 32 matrix

1. 32×32 matrix
   - 아래 표는 32x32를 8x8로 쪼갠 것이다.
   - miss/eviction이 일어나는 경우는 index가 같으나 tag가 다른 경우이다.
   - cache set=32이므로 총 4개의 submatrix가 들어갈 수 있다. (하나의 submatrix에는 8개의 set과, 각 set마다 1개의 blk가 있으므로)
   - A[][], B[][]가 메모리상에 연속하여 위치하므로 주소를 계산하여 tag, index를 구하면 아래와 같다. 편의상 index는 mod 4한 값으로 표현한다.

A[][] (tag/index%4)

0/0	0/1	0/2	0/3
1/0	1/1	1/2	1/3
2/0	2/1	2/2	2/3
3/0	3/1	3/2	3/3

B[][] (tag/index%4)

4/0	4/1	4/2	4/3
5/0	5/1	5/2	5/3
6/0	6/1	6/2	6/3
7/0	7/1	7/2	7/3

(index%4=0 이면 index 0, 4, 8, 12, 16, 20, 24, 28이 들어있다는 의미)

1) 일단, 맨 위 행의 submatrix를 보면,
   (0/0 -> 0/0), (0/1 -> 5/0), (0/2 -> 6/0), (0/3 -> 7/0)으로 옮겨지면 된다.
   row=col 인 경우를 제외하고 tag는 다르지만 index는 같은 경우가 없으므로 eviction이 일어나지 않는다. 따라서 submatrix 상의 row ≠ col인 경우에는 그냥 trans를 해도 된다.
2) row=col인 (0/0 -> 4/0), (1/1->5/1), (2/2 -> 6/2), (3/3 -> 7/3)의 경우엔 일단 원소가 row≠col한 경우는 trans하고, row=col의 경우는 index가 같으므로 불필요한 eviction을 방지하기 위해 tmp에 받아놓는다.
   2중 loop 중 deeper loop가 종료되면 B에 넣는다.

```
      (tag:index)
A[0][0] (0:0) → B[0][0] (4:0) : 이 때 B[0][0]에 값을 넣으면 eviction 발생.
A[0][1] (0:0) → B[1][0] (4:4) : 그리고 다시 A[0][1]에 접근하면 또 eviction.
A[0][2] (0:0) → B[2][0] (4:8)
A[0][3] (0:0) → B[3][0] (4:12)
            ......

그러니 miss와 eviction을 줄이기 위해 B[0][0](4:0)에는 이 시점에서 값을 넣는다.

A[1][0] (0:4) → B[0][1] (4:0) : 그럼 방금 (4:0)에 접근했었으니 여기서 hit가 됨.
A[1][1] (0:4) → B[1][1] (4:4) : 이것도 맨 마지막에 넣음.
A[1][2] (0:4) → B[2][1] (4:8)
A[1][3] (0:4) → B[3][1] (4:12)
            ......
```

2) 61x67 matrix

이거는 8x8 submatrix로 나눠서 돌리기만해도 만점이 나온다. 다만, column을 shallow loop으로, row를 deep loop로 해야 만점을 받을 수 있다.

```
for (int j = col; j < col+8 && j < M; j++){
    for (int i = row; i < row+8 && i < N; i++){
        B[j][i] = A[i][j];
    }
}
```

3) 64x64 matrix

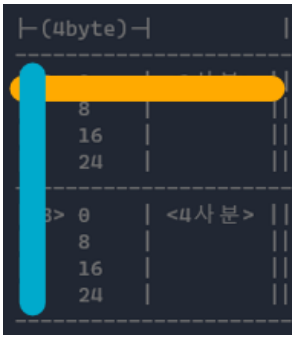
```
3. 64×64 matrix
 - 아래는 8x8 submatrix로 나눈 것을 다시 4x4로 나눈 그림이다.

      ├(4byte)┤          |------(8byte)------|
      ----------------------------------------
      |<2> 0    | <1사분> ||    1    |        |
      |    8    |         ||    9    |        |
      |    16   |         ||    17   |        |
      |    24   |         ||    25   |        |
      ----------------------------------------
      |<3> 0    | <4사분> ||    1    |        |
      |    8    |         ||    9    |        |
      |    16   |         ||    17   |        |
      |    24   |         ||    25   |        |
      ----------------------------------------
```

- cacheblock은 8byte이므로 2,1사분면/ 3,4사분면이 캐쉬 한 줄에 들어간다.

- 32x32에서는 row==col의 경우를 제외하고, input의 읽는 cache index와 output에 쓰는 cache index가 달랐기 때문에 원소 하나씩 읽고->쓰고->읽고->쓰고 순서대로 해도 됐다. 하지만 이 경우엔 1사분면의 원소 하나를 idx0에서 읽고, output 3사분면의 idx0에 쓰고, input 1사분 idx0에 가서 그 다음 원소를 읽고, 또다시 output 3사분 idx8에 쓰는 행위를 반복하면, 불필요한 eviction이 많이 일어난다.

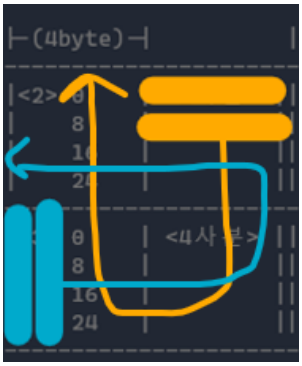
- 따라서, input 1사분면의 원소 4개를 한 번에 읽어서 local var에 받아놓고 ouuput의 3사분면에 한 번에 쓰려고 했다. 하지만 생각해보니 총 12개의 local var을 사용할 수 있으므로 8개씩 읽고 한번에 쓰기로 하였다. 그러면 8개를 어떤 구조로 읽을까.

(1) 행과 열 그대로 읽고 쓰기



이 방법은 썩 좋지 못한 방법이다. 노란색으로 읽을 때에는 1개의 miss 만 나지만, 파란 색으로 쓸 때에는 8개의 miss가 난다. Miss를 줄이기 위해서 output array에 쓰이는 순서를 (상,하)->(하,상)->(상,하)… 를 반복해도 한 열당 최소 4+1의 miss가 발생하므로 효율적이지 못하다.

(2) 4x4 행렬의 두 행을 읽고 두 열로 옮기기



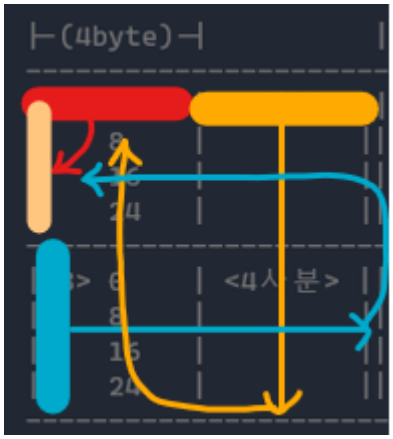
이 방법은 순서에 따라 miss cnt의 차이를 보인다. 당연히 miss가 가장 적게 하려면 input과 output 각각 배열의 상하부를 jump하는 횟수를 적게 해야 한다. 가장 적은 jump횟수를 보이는 두 가지 경우에서도 약간의 miss 차이를 보였다.

① A1->A2->A3->A4(B3->B2->B1->B4): 1427번. Input[][]에서 1번, output[][]에서 2번 jump를 한다.

② A1->A4->A3->A2(B3->B4->B1->B2): 1411번. Input[][]에서 2번, output[][]에서 1번 jump를 한다.

Jump의 총 합은 같지만 output에서 점프를 하는 게 더 효율적인 것으로 보인다.

(3) 원소 4개씩 옮기되, 미리 특정 원소 4개를 var에 담아놓고 그 것을 맨 마지막에 output[][]에 넣기



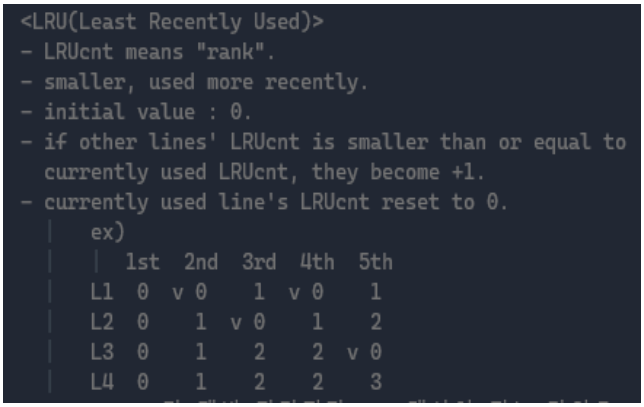
노란색은 input[][]에서 읽는 순서와 방향이다. 파란색은 output[][]에 쓰는 순서와 방향이다. 빨간색은 맨 처음에 담아놓는 원소 4개이며 파란색이 저 순서로 다 돌고 왔을 때 마지막에 output에 넣는다.

저렇게 하는 이유는, 맨 처음에 빨간색 부분에 접근을 할 때, 8byte의 저 라인이 다 캐쉬에 올라온다. 그러면 그 다음에 노란색 부분에 접근을 할 때 cache hit가 된다. 이렇게 되면 row!=col 인 경우에 miss 횟수를 (2)의 ②에 비해서 1번 줄일 수 있는데, 총 64개의 8x8 block 중, row==col은 8개이므로 나머지 블록에서 각 1개씩, 총 56개가 줄어 1411-56 = 1355가 나온다.

이와 유사한 방법으로 하면 더 줄일 수 있을 것 같기도 한데, local var에 제한이 있어 다른 방법이 떠오르지 않는다.

2. What was difficult

① LRU를 두 가지 방법으로 구현했었다. 표에 나와있는 값은 update 후의 값이다.




```
void LRUupdate(Set* set, Line* line)
{
    for (u_int64_t i = 0; i < lineN; ++i){
        Line* lineItr = &set->lines[i];
        if (lineItr->LRUcnt > line->LRUcnt){ continue; }
        (lineItr->LRUcnt)++;
        line->LRUcnt = 0;
    }
}
```

첫 번째 방법은 initial LRUcnt는 0이고, 가장 최근에 use되었을수록 0에 가까운 값을 가지도록 구현하였다. LRU의 범위는 [0, E)이다. 2nd에서 L1이 사용된 경우, 기존의 L1 LRU인 0보다 작거나 같은 LRU들은 +1씩 해주고 L1 LRU는 0으로 초기화를 해준다. 5th를 예로 들어보면, L3가 사용되어야 하기 때문에 그 전 L3의 LRU인 2보다 작거나 같은 L1, L2, L4에 1씩 더해주고, L3는 0으로 리셋을 시킨다. 이러면 evict당해야 하는 라인은 다른 라인들보다 큰 LRUcnt를 갖게 된다.

처음에는 line->LRUcnt =0을 for문 밖에 위치시켰었다. 그랬더니 LRU의 업데이트가 제대로 이루어지지 않았다. 2nd처럼 L1부터 iteration을 돌면, 나 자신과 비교를 하기 때문에 작거나 같은 것에 해당되어 +1이 되어버린다. 따라서 기준이 망가지므로 그 다음 라인과 LRU값을 비교할 때 잘못된 계산을 하게 된다. 그래서 for문 안으로 넣어서 해결을 하였고, 지금 생각해보면 그냥 pointer로 역참조를 하는 것이 아니라 그냥 value로 값을 받아놓았어도 된다는 생각이 든다. 하지만, 여전히 남아있는 이 방법의 문제는 4th에서 보이는 것처럼 E-1보다 작은 최댓값을 보이는 경우가 있다는 것이다. 이러한 경우에는 모든 라인의 LRU를 비교하여 Max 값을 찾아내야 하는데 이는 너무나 비효율적이다. 따라서, evict 대상이 0을 갖도록 다시 설계를 한 것이다.

② 32x32 matrix는 쉬웠고, 61x67 matrix는 운이 좋은 건지 32x32 구현할 때 사용했던 코드로도 처리가 가능했다. 하지만 64x64는 꽤 시간이 걸렸는데, 여러 방법을 시도해보았지만 만점을 받지는 못했다. 어떤 기발한 수를 써야 1300 아래로 miss가 날 수 있는지 너무 궁금하다.

3. What was surprising

① getopt()를 사용할 때, verbose option에 argument를 받지 않으려면 `:`을 제외하면 된다는 것을 새로 알게 되었다.

② %lx가 unsigned long int hexadecimal인 것을 새로 알게 되었다. 보통 과제를 할 때에는 4byte address를 사용해서 %x 또는 long type인 %lu 정도만 사용했었는데, 이번에 처음으로 8byte address 조건에서 과제를 해 본 덕분이다.

③ cache friendly code를 컴퓨터구조 시간에 배우기는 했었지만 직접 코드를 짜본 것은 이번이 처음이다. 직접 여러가지 경우를 구현해보며 꽤나 흥미롭게 과제를 하였다.

4. Result screen shot

```
stu3@ubuntu:~/workspace/4.cachelab/cachelab_handout$ ./driver.py
Part A: Testing cache simulator
Running ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27


Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           7.4         8        1355
Trans perf 61x67          10.0        10        1931
          Total points    52.4        53
```