Cache LAB

정성태 2018-15515

Part A. Cache Simulator

typedef's

```c
typedef unsigned long long ad;

/* Opcode of request line */
typedef enum {
    ld, st, md
} opcode;

/* Result of cache access */
typedef enum {
    hit, miss, evict
} result;

/* A single cache block */
typedef struct {
    int valid;
    ad tag;
    int counter;
} block;
```

코드 작성에 사용하기 위해 위와 같이 몇몇 type을 새로 정의했다. ad는 unsigned long long에 대한 alias로, 메모리의 주소 값을 다룰 때 사용한다. opcode는 trace file에서 각 line의 가장 앞에 나오는 L, S, M 등의 opcode를 저장하기 위해 사용한다. result는 cache access의 결과가 어땠는지를 전달하기 위해 사용한다. 마지막으로 block은 cache block 하나를 나타내는 struct로, valid와 tag, LRU counter로 이루어져 있다.

global variables

```c
/* Program parameters */
static int s;           // number of set bits
static int b;           // number of block bits
static int E;           // set associativity
static char *filename;  // name of trace file
```

프로그램 실행 시에 입력되는 argument들은 static global variable로 정의했다. 이 변수들에 적절한 값을 넣는 역할은 parse_arg()가 담당한다.

main routine

main()에서는 먼저 parse_arg()를 호출해 실행 시 입력된 argument의 값을 변수에 저장한다. 그런 다음 malloc()을 사용해서 cache block들의 2차원 배열 형식으로 cache의 내용을 저장할 공간을 할당한다. 그러고 나서는 init_counter()를 호출해 각 cache block의 LRU counter 값을 적절히 초기화해준다. 이 함수는 한 set에 들어 있는 block들의 counter 값은 모두 서로 달라야 한다는 조건을 만족할 수 있게 하는 역할을 한다.

그런 다음에는 입력된 trace file을 open 한 뒤 parse_line()을 호출해 각 line의 내용을 parsing 한다. 이제 access()를 호출하면서 line의 내용을 전달하고, 그 결과에 따라 hits, misses, evictions의 값을 적절히 변경해주면 된다. trace file 내부의 모든 line에 대한 처리가 끝나면 loop가 종료되며, 처음에 malloc()으로 할당했던 cache를 다시 free 해준 뒤 결과를 출력하면 프로그램이 끝난다.

parse_arg()

main()으로 입력된 argc와 argv를 그대로 받아서 적절히 static global variable의 값을 초기화하는 역할을 한다. -s, -E, -b, -t 등의 argument를 처리할 수 있다.

parse_line()

trace file의 각 line을 처리하는 역할을 한다. line에서 opcode와 메모리 주소 정보를 추출한 뒤 (크기 정보는 생략 가능) parameter로 입력된 opcode와 ad의 포인터가 가리키는 위치에 적절한 값을 넣어준다. L, S, M이 아닌 I가 입력된 경우에는 무시하고 다음 line을 받는다.

access()

메모리 주소 정보를 받아서, 해당 위치에 접근하는 상황에서 cache 내용을 어떻게 변경해야 할지를 시뮬레이션 하는 역할을 하는 함수이다.

```
static int hash(ad addr, int nsets) {
    /* Remove lowest b bits */
    addr >>= b;

    /* Retrieve cache index */
    return addr % nsets;
}
```

먼저 주소 값으로부터 cache의 어떤 set에 속하는지(setno)와 tag의 값을 계산한다. setno의 계산은 따로 정의한 함수 hash에서 담당한다.

```
static int find_match(block *set, ad tag) {
    int i = 0;

    for (i=0; i<E; i++) {
        if (set[i].valid && set[i].tag == tag) {    // tag matches
            return i;
        }
    }
    return -1;
}
```

그러고 나면 해당 set 안에 valid이면서 tag가 일치하는 block이 존재하는지를 검사한다. 이 작업은 find_match()가 수행한다. 만약 일치하는 block이 발견되면 다른 변경 사항 없이 LRU counter의 값만 적절히 수정해주면 된다. 이때는 lru_count()를 호출한다. 그러고 나서는 result type의 hit을 반환한다.

만약 tag가 일치하는 block이 없다면 디스크에서 새로운 block을 읽어 와야 한다. 그러기 위해서 find_empty()를 호출해 invalid인 block을 탐색한다. 만약 빈 block이 발견되면 valid를 1로 바꾸고 tag를 수정한 뒤 LRU counter의 값을 업데이트하면 된다. 이때는 miss를 반환한다.

만약 빈 block이 존재하지 않는다면, LRU policy에 따라 block 하나를 선택해 cache에서 제거해서 공간을 확보해야 한다. 이때 victim을 선택하는 일은 find_victim()에서 수행한다. 이제 해당 block의 tag와 LRU counter 값을 수정하고 나서 결과로 evict를 반환하면 된다.

lru_count()

cache set 하나와 그 set 내에서 접근하고자 하는 block을 가리키는 index를 받아서, 그 block에 접근하고자 하는 상황에 맞추어 LRU counter 값을 수정해주는 함수이다. 각 set의 block들은 0부터 E-1까지의 서로 다른 counter 값을 가진다. 가장 큰 값인 E-1은 가장 최근에 접근한 block임을, 0은 가장 오래전에 접근한 block임을 나타낸다. 따라서 특정 block에 접근한 경우에는 그 block의 counter 값을 E-1로 설정하고, 나머지 block들 중 기존의 target block의 counter보다 큰 값을 가지는 것들을 1씩 감소시키면 된다. 이 방식에 맞추어 find_victim()은 counter의 값이 0인 block을 찾아내는 역할을 한다.

실행 결과

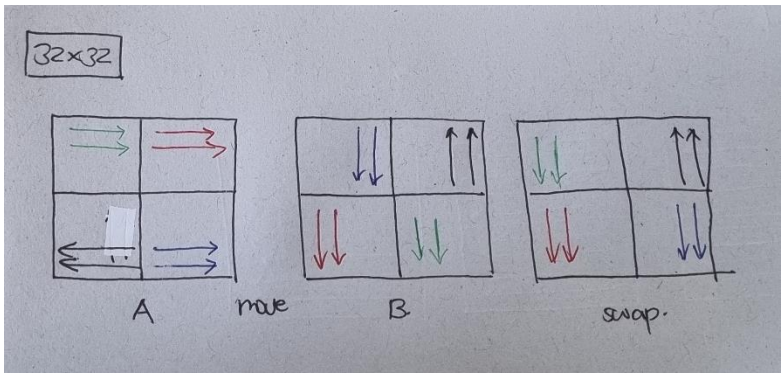
```
Part A: Testing cache simulator
Running ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       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
```

csim-ref와 동일하게 정상적인 결과를 출력함을 확인했다.

Part B. Matrix Transpose

각 testcase에 대해 최적의 결과를 내기 위해 입력된 행렬의 크기에 따라 다른 방식을 적용했다.

32 x 32



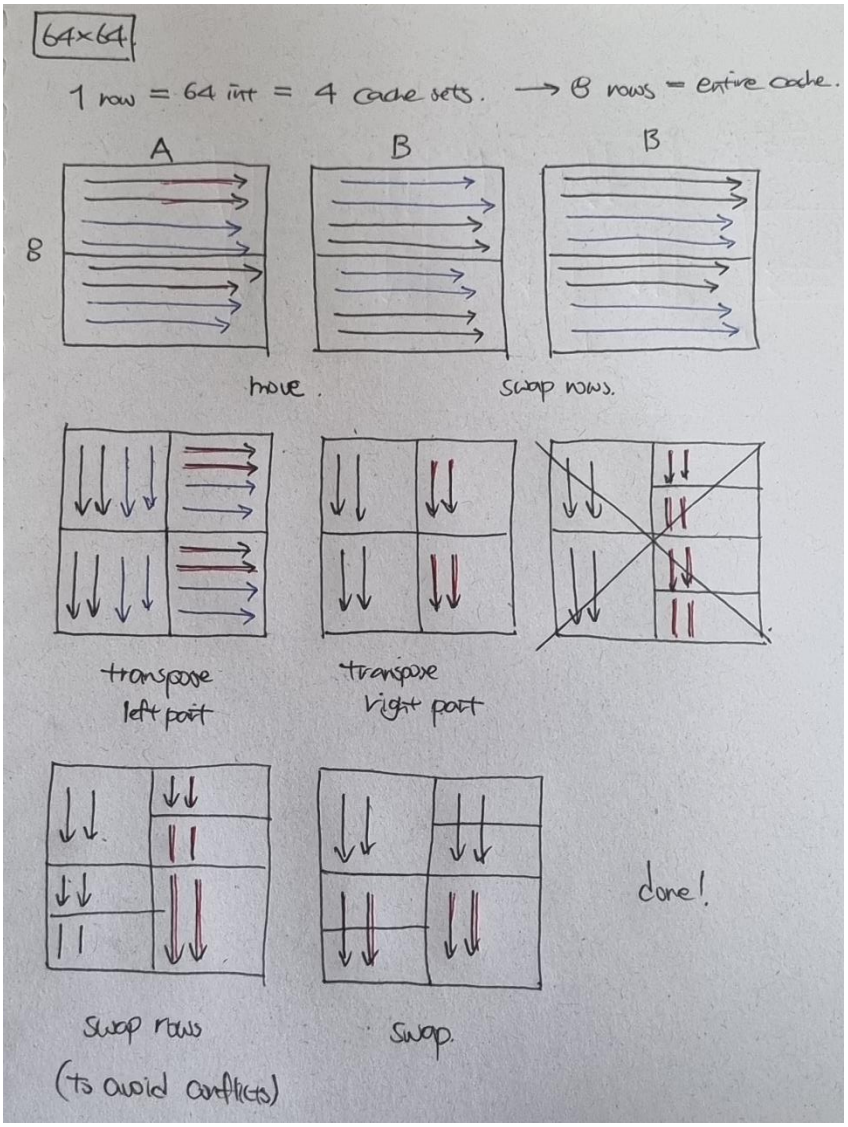
32 x 32 행렬은 8 x 8 크기의 작은 블록으로 분할할 뒤 각 block에 대해 transpose를 수행하는 방식을 사용했다. 이때 블록의 크기를 8로 한 것은 cache의 block size가 32 bytes(= 8 int's)임을 고려한 것이다.

그림의 큰 사각형은 8 x 8 블록 하나를 나타낸다. 블록 하나를 다시 4 x 4 크기의 네 부분으로 나누어 처리했는데, 이는 conflict를 줄이기 위한 조치이다.

먼저 A 내부의 각 부분을 transpose 해서 B로 옮긴다. 이때 좌상단과 우하단은 서로 교차해서 옮기게 된다. 이는 A와 B의 좌상단, 우하단이 각각 cache 내부의 같은 set으로 연결되어 발생할 수 있는 conflict를 차단하기 위함이다. 따라서 먼저 서로 교차되는 위치로 옮긴 뒤 마지막에 한 번 더 swap을 수행함으로써 transpose를 완료하게 된다.

이제 이 과정을 모든 블록에 대해 동일하게 수행하면 행렬 전체의 transpose가 완료된다.

64 x 64



64 x 64 행렬 역시 32 x 32 행렬과 유사하게 8 x 8 크기의 블록 여러 개로 나누어 처리하는 방식을 사용한다. 다만 이때는 블록 내부의 1, 2행과 5, 6행이 같은 cache set에 속하게 되므로 이를 고려해주어야 한다.

각 블록을 처리하는 세부 절차는 다음과 같다.

1. 먼저 A의 블록을 그대로 B의 transpose 된 위치에 해당하는 블록으로 옮긴다. 이때 conflict를 차단하기 위해 1, 2행과 3, 4행, 5, 6행과 7, 8행을 교차해서 옮긴 뒤 다시 swap을 수행한다.
2. 블록을 4 x 4 크기의 네 부분으로 나눈 뒤 각 부분에 대한 transpose를 수행한다.
3. 이제 왼쪽 아래 부분과 오른쪽 위 부분을 서로 바꾸면 완료된다. 이때도 conflict를 피하기 위해 각 부분을 위, 아래의 두 부분으로 쪼개어 옮긴 후 swap을 수행한다.

61 x 67

61 x 67 행렬의 경우 크기가 8 의 배수가 아니기 때문에 블록으로 나누어 처리하는 방식을 사용하기 어렵다. 그래서 단순히 여러 부분으로 나눈 뒤 각 부분에 대해 행과 열을 따라 transpose 를 수행하되, conflict 를 발생시키는 요인이 되는 diagonal entry 만 별도로 처리하는 방식을 사용했다. A 의 각 행을 B 의 각 열로 옮기면서, diagonal entry 는 바로 B 로 옮기지 않고 따로 저장해 두었다가 이동이 완료되면 마지막에 B 로 옮기는 것이다. 각 부분의 크기는 여러 값을 대입해 실험해본 결과 최적의 결과를 낸 16 x 16 을 채택했다.

실행 결과

```
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           7.5         8         318
Trans perf 64x64           6.4         8        1443
Trans perf 61x67          10.0        10        1855
          Total points    50.9        53
2018-15515@sp4:~/lab04_cache$
```

Review

새로 알게 된 점

- cache 의 eviction policy 인 LRU 가 구체적으로 어떻게 작동하는지 알게 되었다.
- matrix transpose 를 수행할 때 cache-efficiency 를 달성하기 위해 diagonal entry 의 처리가 중요하다는 점을 알게 되었다.

어려웠던 점

- cache의 spec을 고려하면서 conflict miss를 최소하기 위해 transpose의 절차를 설계하는 과정이 어려웠다. 행렬의 크기에 따라 conflict의 발생 양상이 달라진다는 점이 난이도를 높이는 요인으로 작용했다.