과제 #2 – Cache Simulator 구현

박준호 (201510756)
전북대학교 컴퓨터공학부
Krs0123@naver.com

요약

Cache simulator 코드는 visual의 c++로 구현하고, ubuntu의 g++로 구동하였다.
출력은 Tag, Index, Offset 각각의 bit수를 출력하고, 총 access와 hit rate를 출력한다.
각각의 입력오류에 대해 오류 메시지를 출력한다.

1. 실습 프로그램의 구성 및 동작 원리

```
//hit
Cache tmp;
for (int i = 0; i < associativity; i++)
{
    if (cache[index][i].tag == tag)
    {
        hit++;
        for (int j = i + 1; j < associativity; j++)
        {
            if (!cache[index][j].empty)
            {
                tmp = cache[index][j - 1];
                cache[index][j - 1] = cache[index][j];
                cache[index][j] = tmp;
            }
            else
                break;
        }
        return;
    }
}
```

Hit를 판별하는 부분이다. 메인에서 넘겨준 index안에 tag를 비교하여 같으면 hit수를 늘려준다.

Hit를 하면 lru를 최신화 시켜줘야 한다. 일단 사용한지 가장 오래된 tag는 cache[index]의 0번째 배열에 위치시켰고, 만약 어떠한 tag가 hit한다면 그 배열의 구조체를 차례대로 맨 끝까지 교체하며 바꿔준다.

만약 0 1 2 3 이 차례대로 저장되어 있었고 1이 hit 했다면 0 2 3 1 이 되는 것이다.

인덱스 안에 associativity 만큼 채워져 있지 않을 수도 있기 때문에 다음 구조체 배열이 비어 있는지 판별하며 교체한다.

```
//miss and write
for (int i = 0; i < associativity; i++)
{
    if (cache[index][i].empty)
    {
        cache[index][i].tag = tag;
        cache[index][i].empty = 0;
        return;
    }
}
//replacement
cache[index][0].tag = tag;
for (int i = 1; i < associativity; i++)
{
    tmp = cache[index][i - 1];
    cache[index][i - 1] = cache[index][i];
    cache[index][i] = tmp;
}
```

위에서 hit가 발생하지 않으면 무조건 miss이다. Miss는 비어 있는 자리에 tag를 추가하거나 replace 한다.

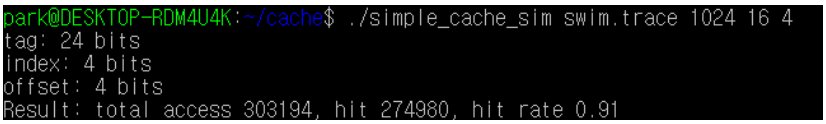
이 부분은 자리가 비어 있는 곳을 찾아 tag를 추가하는 부분이다.

Replace는 사용한지 가장 오래된 tag를 교체하는데, 0번째 배열이 가장 오래된 tag 이므로 새로운 tag로 교체시킨 후, 가장 최신이 되었으므로 index 안의 0번째 배열을 배열의 끝까지 밀어 올려 준다.

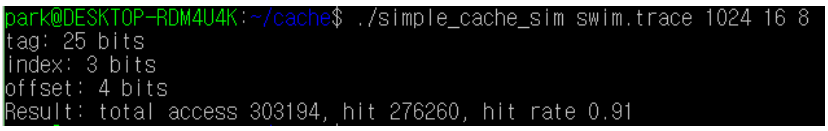
```
address = strtoul(address_str.c_str(), NULL, 16);
tag = address >> (index_bit + offset_bit);
index = (address - (tag << (index_bit + offset_bit))) >> offset_bit;
```

비트 떼는 부분이다. 이 부분을 구현하는게 lru 다음으로 힘들었다. 어떻게 떼야 될지 감도 잡히지 않았고, 이 부분을 구현하기 위해 인터넷도 엄청 찾아봤고, 혼자 고민도 엄청 많이 하며 고생 좀 하다가 이 방법이 생각이 났다.
일단 Trace 파일의 2번째 요소를 읽어 들인 후, address에 문자열을 16진수로 변환하여 저장한다. 그 다음 그 address 값을 index, offset bit만큼 오른쪽으로 shift 시켜 tag를 떼어 tag에 저장한다. 그 다음 tag값을 다시 index, offset bit 만큼 왼쪽으로 shift 시키고, 그 값을 address와 뺀다. 그러면 index, offset 비트만 남게 되고, offset 비트만큼 오른쪽으로 shift 시키면 index값이 나온다.

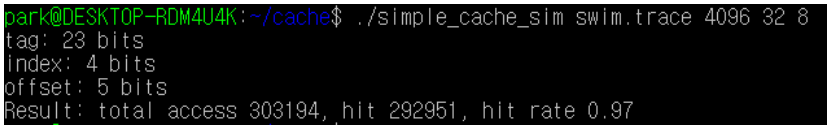
2. 결과



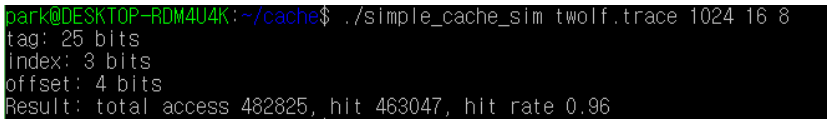
과제의 예시로 주었던 값을 입력했을 때의 결과 값.



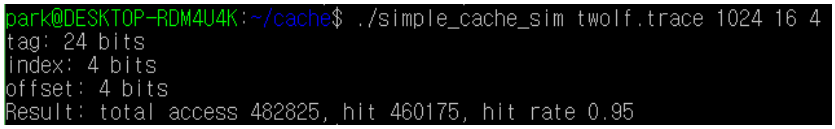
associativity 값을 올렸을 때. Hit 수가 약 1300개 정도 더 늘어났다.



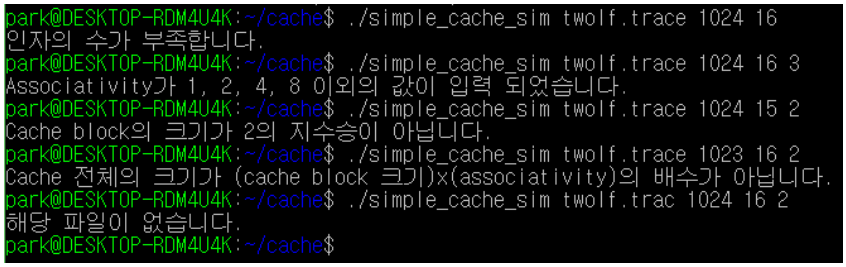
좀 많이 다른 값을 넣었을 때, 히트율이 상당히 올랐다.



다른 trace 파일을 사용했을 때의 결과 값.



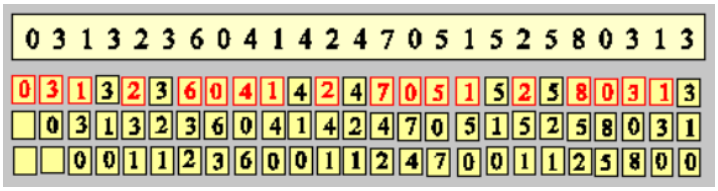
Associativity 값을 낮췄을 때. 히트 수가 줄어들었다.



예외처리 test case
5가지의 예외처리에 대해 오류 없이 잘 되는 모습.

3. 결론

처음에 lru를 구현할 때, 구조체에 lru 변수를 하나 만들고, access가 계속해서 늘어나므로 이걸 이를 이용하여 lru를 갱신 시켜봤는데, 만약에 메모리 접근이 방대하게 늘어나면 access를 lru로 쓸 때 메모리가 부족해질 것 같아 새로운 방법을 찾았다. 인터넷에 lru에 대해 찾아보니



이런 느낌으로 설명이 되어있었다. 이걸 보고 아 배열을 이용해서 lru를 구현하면 lru 변수를 사용하지 않아도 되고 깔끔하고 좋겠다 하고 영감을 얻어 구현하게 되었다.

그런데 아무리 생각 해도 lru를 깔끔하게 구현할 방법이 이 방법 말고는 없는 것 같았고,

만약에 배열이 아닌 다른 방법으로 lru를 깔끔하게 짠 방법이 있으면 나도 알고 싶다.