2020년 2학기

Assignment #4

과목 : 컴퓨터SW시스템개론

담당 교수 : 김종

학과 : 컴퓨터공학과

학번 : 20190439

이름 : 오승훈

povis id : sho0927

|  |
| --- |
| < 명예서약(Honor code) >  “나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.” |

**Problem : Cache Lab\_Part A**

1. 문제의 개요

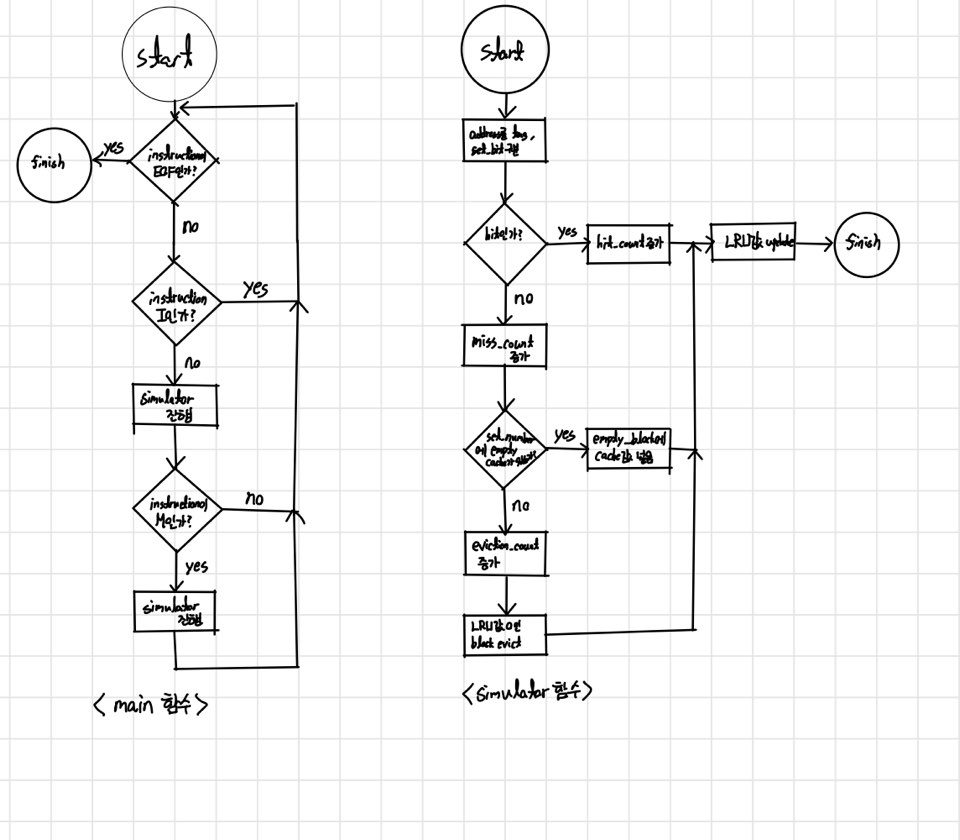
본 프로그램은 다음의 내용을 포함한다.

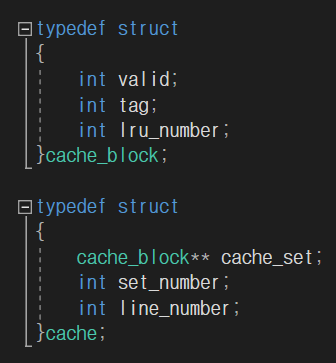
* Cache simulator를 제작하는 것을 목표로 한다.
* 각 line에는 각각의 해당되는 instruction들이 존재하는데 이 instruction들은 총 4가지의 옵션이 존재한다.

1. I : instruction load
2. M : data modify(data loading and data storing을 동시에 진행)
3. S : data storing
4. L : data loading

* 우리의 프로그램은 hit, miss, eviction의 횟수를 세는 것을 목표로 하기 때문에 I instruction은 무시해도 상관이 없고, M instruction과 S, L instruction만을 신경 써 코딩을 진행하면 된다.
* 그리고 또한 이 프로그램에서는 LRU 방식을 취하기 때문에 가장 최근에 쓴 data가 무엇인지 cache block 내에서 조절을 진행해주면 된다.
* Cache에 대한 대략적인 설명을 붙이자면 접근하려는 address가 32bit짜리라고 하고, set\_bit가 5개, block\_bit가 5개, E 값이 1이라고 한다면, 현재 MSB ~ LSB 31 ~ 0 bit 라고 가정을 한다면 LSB에서 5개가 block\_bit가 되고, 그 뒤 5개가 set\_bit 그 뒤 22개가 tag\_bit이 되는 것이다. 현재 이 상황에서 set은 총 32개가 존재하고, block\_size는 32, 그리고 E가 1이기 때문에 set당 block은 1개로 이루어진 상태의 cache가 되는 것이다.

2. 문제 구현

* 이 프로그램은 다음과 같은 구조를 가질 것이다.
* 
* 다음 flow\_chart를 보면 현재 simulator 진행 이라는 함수는 simulator 함수를 진행한다는 의미로 받아들이면 된다.
* 현재 instruction이 I인 경우에는 다음 instruction으로 넘기고, S, L인 경우 한 번의 simulation을 돌리고, M인 경우 두 번의 simulation을 돌리는데 이는 M인 경우, S, L이 동시에 일어나는 것과 같은 상황이라고 해뒀기 때문에 두 번을 돌려주면 되는 것이다.
* 그리고 또한 현재 block 함수는 다음과 같이 구현을 해 뒀다.



현재 cache는 offset이 필요없고, 이 block이 사용 중인지를 나타내는 valid 값(사용중이면 1, 아니면 0을 입력해 넣음) 과 tag 값이 무엇인지, 그리고 lru\_number가 몇 번째 인지만을 확인해주면 되는데, LRU\_NUMBER는 처음 값이 넣어질 떄 line\_number – 1 값을 넣어주고, set 값이 동일한 block이 사용이 될 때에 1씩 감소하여 같은 set 값일 때 모든 block이 찼다면 가장 마지막으로 사용된 block의 lru\_number는 0이 되도록 구현을 해뒀다.

그리고 또한 cache의 구조는 set 값과, 몇 개의 block으로 구성이 되어있는지를 확인을 해야 하기 때문에 위의 cache structure 내부의 cache\_set이 나타내고 있는 이차원 배열로 구현을 했다.

그리고 line\_number는 하나의 set 당 몇 개의 block으로 이루어졌는지를 나타내는 수이고, set\_number는 몇 개의 set이 존재하는지를 나타낸 변수이다. 이는 cache simulator를 사용할 때 위의 두 개의 변수 값들이 많이 사용이 되는 특성이 있었기 때문에 다음과 같이 구현을 해뒀다.

이제 main 함수를 보면 main 함수에서 s, E, b, t 값들을 받아 들이고, 그 후 malloc을 이용해 필요한 만큼의 cache\_block을 동적 할당을 이용하여 구현을 한다. 그러면 필요한 모든 infra가 구현이 다 되었으므로 이제 instruction과 access 하려는 address들을 받아들여 caching을 진행해주면 된다.

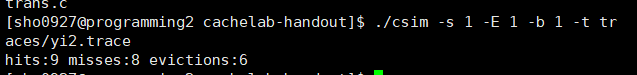
Caching의 과정은 먼저 access하려는 값을 access하려는 set number와 tag 값을 알아낸 후, access 하려는 set의 block에 valid한 값이 1인 경우 tag 값이 현재 접근하려는 값과 같은지를 체크해주고, 같으면 cache hit이므로 hit\_count값을 1 증가시키고, LRU 값을 update 해주면 된다.

Hit이 일어나지 않은 경우 miss\_count 값을 올려주고, 현재 접근하려는 set\_number에 해당하는 block이 empty한 곳이 있는지를 확인 해준 뒤, 있으면 그 block에 현재 cache 값을 넣어주면 된다.

Empty한 곳이 없다면 cache\_eviction이 일어나야하는 상황이기 때문에 eviction\_count 값을 증가시켜주고, LRU 값이 0인 값이 가장 사용되지 않은 cache 값이므로 이 값을 eviction 해주고, 현재 cache 값을 그 block에 넣어준 뒤, LRU\_UPDATE를 진행해주면 된다.

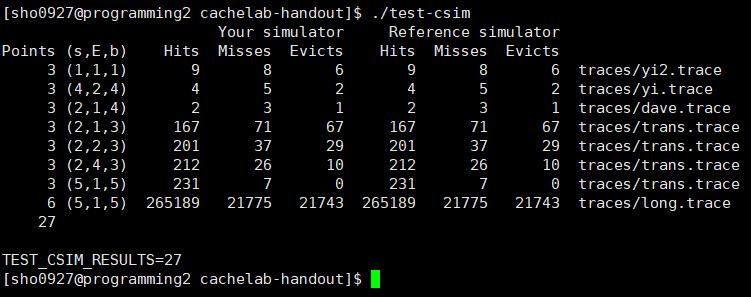
3. 구현 결과

구현을 모두 다 하면 다음과 같은 상황을 볼 수 있다.



이는 set이 총 2개, set당 block은 1개의 상황일 때 simulation을 돌려보라는 것과 같은 상황이다.

모든 상황에 대한 test 결과는 다음과 같다.



위와 같은 결과를 보인다는 것을 알 수 있고, 모든 test에 대해서 값이 성립함 역시 볼 수 있다.

**Problem : Cache Lab\_Part B**

1. 문제의 개요

본 문제는 다음과 같은 내용을 포함한다.

- hit ratio가 최대가 되도록 matrix transpose 연산을 수행하는 코드를 작성한다.

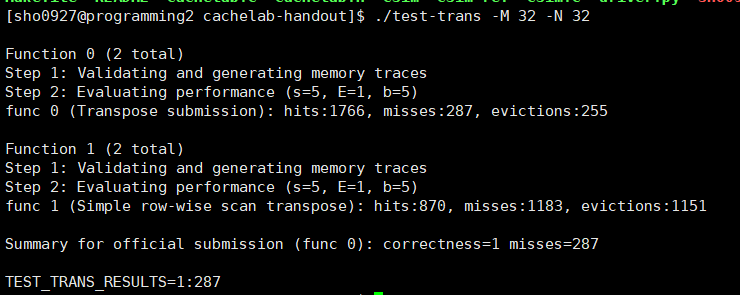
- hit ratio가 최대가 되기 위해서는 blocking을 이용하여 cache의 miss와 eviction이 적어 지게 하는 것이 효율을 높일 수 있는 방법이다.

2. 문제 해결 방법

Hit ratio를 높일 수 있는 방법은 위에서 언급했듯이, cache의 miss와 eviction을 줄여주는 것이 가장 좋은 방법이다. 그러므로 cache의 eviction을 줄여주는 방법을 구현을 해봐야할 것이다. 이때 가장 좋은 방법은 cache값에 맞는 blocking을 통하여 cache의 miss와 eviction을 줄여주는 방법인데 이 방법은 현재 우리가 사용하려는 cache인 s = 5, E = 1, b = 5인 상황에서 block이 32byte이고, 현재 우리가 접근하려는 matrix는 int 값이기 때문에 한 block당 8개의 integer 값을 담을 수 있다는 것을 알 수 있다. 그러므로, 8 \* 8의 blocking이 가장 좋은 방법이 될 것이라는 것을 알 수 있다. 그리고 또한 현재 문제에서 요구하는 32 by 32, 64 by 64, 61 by 67 모두 각각의 경우마다 eviction과 miss가 일어나는 경우가 다르기 때문에 3개의 경우 모두 나눠서 문제를 구현을 해야할 것이다. 또한 이 때 A와 B matrix는 함수의 인자로 받기 때문에 주소 값이, A의 모든 index가 끝나고 난 후, B[0][0] 값이 이어지는 상황이라고 생각하면 될 것이다.

1. 32 by 32

현재 상황에서 32 by 32에서는 모든 index 값을 계산을 해보면 1024개이고, address 값을 계산을 해보면 B matrix의 address는 A[0][0] 보다 4096이 더 큰 것을 확인 할 수 있다. 4096을 이진수 값으로 표현하면 1 0000 0000 0000 인 것을 확인 할 수 있다. A matrix는 row 방향으로 접근을 하고, B matrix는 col 방향으로 접근을 하기 때문에 차이가 생기는데 먼저 A[0][0]인 상황부터 보면 A[0][0]부터 A[0][7]까지는 하나의 캐시에 포함되어있다는 것을 볼 수 있고, 그리고 B[0][0]부터 B[7][0]은 각각 다 다른 cache값을 가진다는 것을 알 수 있는데 이 때 A[0][0] 이후 B[0][0]을 접근하게 되면 set\_number가 같기 때문에 cache miss와 eviction이 일어나게 만들고, 이는 다시 A[0][1] 을 접근할 때 또 다른 cache miss를 유도하기 때문에 효율적이지 않다는 것을 확인할 수 있다. 또한 A[1][0] 이후 A[1][1]을 접근하면 A[1][0]을 이미 cache로 담아두고, A[1][1]을 접근하는 것이기 때문에 hit가 발생하고, B[1][1]을 접근하게 되면 cache eviction이 발생하고, 다시 A[1][2]를 접근할 때 cache miss가 또 나기 때문에 이는 비효율적이라는 것을 알 수 있다. 즉 A[i][i]를 접근할 때 cache miss와 eviction이 여러 번 일어나게 만들기 때문에 A[i][i]인 값들을 따로 접근을 해주는 것이 좋다는 것을 알 수 있다. 그러므로 8\*8 blocking을 하고, 16번의 block matrix를 진행하지만 diagonal 값을 접근하는 것은 마지막에 하면 cache miss를 줄이는 방법이 될 것이다.



위와 같은 방식으로 코딩을 진행하면 위와 같은 결과값을 보인다는 것을 알 수 있다.

1. 64 by 64

64 by 64의 경우 64\*4만 해도 256이 되고, 이는 이미 주소값으로 하면 integer는 4byte이기 때문에 1024값을 지녀서 A[0][0] 과 A[4][0]이 같은 cache set\_number를 사용한다는 것을 알 수 있다. 그러므로 단순히 대각행렬만 신경 쓸 것이 아니라 여기서는 8\*8 blocking된 상태에서만 살펴본다면 다음과 같은 패턴의 상황에서 비효율적인 cache 접근이 발생한다.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| O |  |  |  | O |  |  |  |
|  | O |  |  |  | O |  |  |
|  |  | O |  |  |  | O |  |
|  |  |  | O |  |  |  | O |
| O |  |  |  | O |  |  |  |
|  | O |  |  |  | O |  |  |
|  |  | O |  |  |  | O |  |
|  |  |  | O |  |  |  | O |

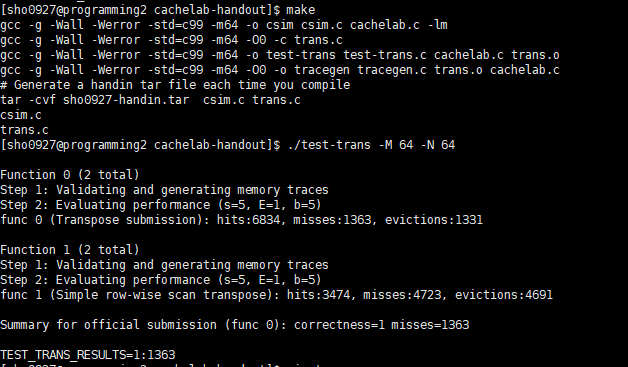
그러므로 A 행렬을 기준으로 빨간색 -> 보라색으로 가는 방향의 for문과 빨간색 -> 초록색으로 가는 for문을 차례로 적어주게 되면 최소한의 cache\_miss가 나는 것을 확인할 수 있다.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| O |  |  |  | O |  |  |  |
|  | O |  |  |  | O |  |  |
|  |  | O |  |  |  | O |  |
|  |  |  | O |  |  |  | O |
| O |  |  |  | O |  |  |  |
|  | O |  |  |  | O |  |  |
|  |  | O |  |  |  | O |  |
|  |  |  | O |  |  |  | O |

밑의 그림은 B 행렬의 값을 보여줌

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

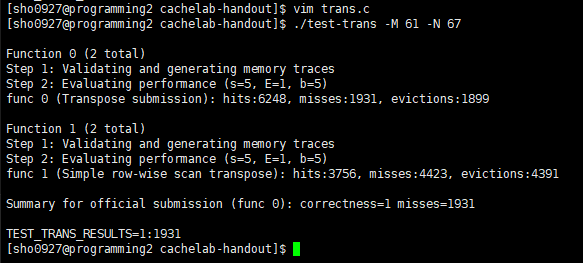
즉 4개의 column wise로 진행할 때 B에서는 cache miss가 대각행렬이 아닌 경우 발생하지 않기 때문에 더 효율적인 코드 사용이 가능하고, 그 후 빨간색에서 초록색으로 가는 이유는 현재 보라색을 사용하고 나면, cache에 A의 밑의 보라색 부분의 cache 값이 남아있고, 그렇다면 A의 가장 밑 빨간색 부분의 cache 값을 가지고 있기 때문에 cache miss를 줄일 수 있는 방법론이 되는 것이다. 그러므로 A 행렬은 U 모양을 그리도록 B 행렬은 ‘ㄷ’자형을 그리도록 만들어서 transpose를 진행하면 된다.



그러면 위와 같은 결과가 나옴을 확인할 수 있다.

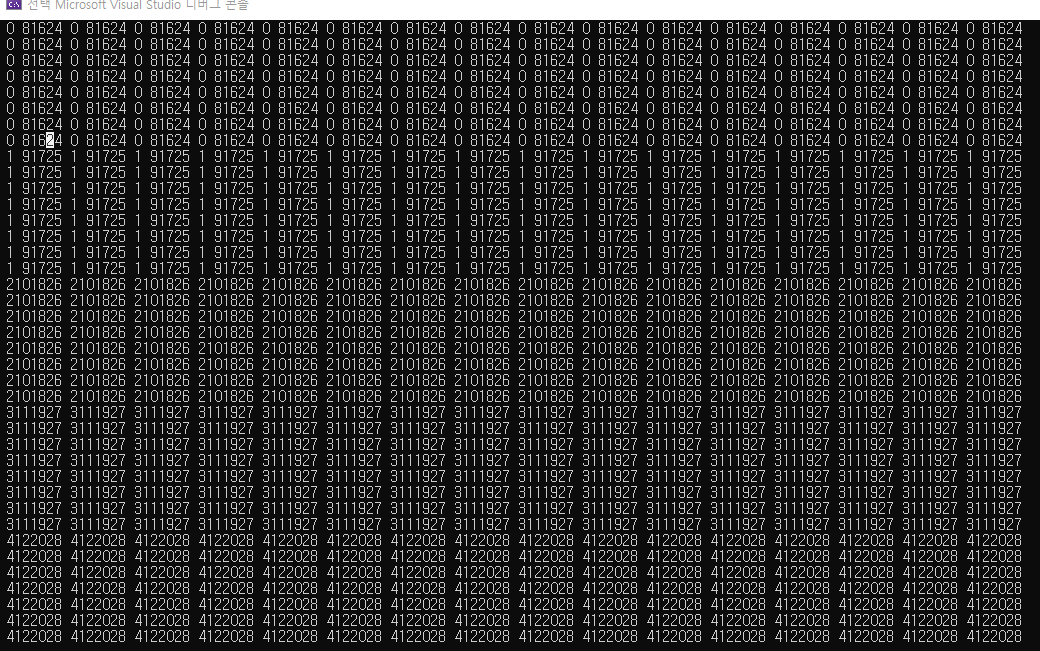
1. 61 by 67

이 경우 8 by 8 matrix로 분할하여 transpose를 수행하면 miss 횟수가 2000 아래임을 코드 진행을 통해 확인할 수 있다. 여기서 cold miss 32회, eviction 1899회가 일어남을 확인 가능하다.



3. 토론

이번 과제를 하면서 가장 아쉬운 점은 분명 64\*64 matrix의 cache를 해줄 때 miss를 줄일 수 있는 방법이 존재할 것인데 이 점을 찾아내지 못 한 것이 너무 아쉽게 작용하는 것 같다.



특히 위와 같이 set\_number에 대한 값들도 다 조사를 해본 결과 대각 행렬이 존재하는 곳에서 cache miss가 더 자주 나오기 때문에 이 값들만 줄여줘도 cache miss가 1300 밑으로 떨어질 수도 있을 것 같은데 도저히 이 밑으로 안 내려가는 것이 아쉽게 작용하는 것 같다.

또한 이번 cache 랩을 진행하면서 처음에는 cache의 모양과 똑같게 만들어서 진행을 하다 그렇게 하면 offset 값이 아무 의미 없는 값인데 structure에 존재하게 되고, 그 값이 낭비된다는 것을 깨닫고 나서 코딩이 제대로 된 것 같다.

마지막으로 코딩을 하면서 Xshell 내부 vim을 사용하여 코딩을 진행하다 보니 생각보다 너무 까다로운 코딩이었고, ‘Xshell이랑 vscode 연결만 됐어도…’ 라는 생각을 하게 만드는 ASSN이었다.