CSED211: Cache Lab Report

20230499/KimJaeHwan

1. Overview

이번 랩에서는 코드를 실행하면서 메모리와 CPU의 상호작용 중에서 캐시가 어떤 역할을 하는지 알아보고, Valgrind 도구로 만들어진 메모리 추적 값을 이용하여 캐시를 시뮬레이션 해 본다. Part. A에서는 Valgrind 명령의 출력값을 입력으로 받아 캐시 hit, miss, eviction이 얼마나 발생했는지 시뮬레이션하는 코드를 작성한다. Part. B에서는 캐시 시뮬레이터를 이용해, 특정 크기의 행렬을 전치(transpose)하면서 발생하는 캐시 miss를 최소화하는 것을 목표로 한다.

2. Part A. Cache simulator (csim.c)

A. getopt – 명령줄 옵션 파싱(parsing). (csim:128~155)

csim은 h, v, s, E, b, t 옵션을 받을 수 있으며, s, E, b, t 옵션은 인자를 추가로 받음과 동시에 반드시 필요한 옵션이다. 위 옵션들을 파싱하기 위해 getopt.h 라이브러리를 사용하였다.

print_help() 함수는 도움말(사용법, usage)를 출력하는 함수이다. -h 옵션을 받는 경우, 필수 옵션에 인자가 없거나 옵션이 없는 경우, 정해진 6가지의 옵션 이외의 다른 옵션이 주어지는 경우 도움말을 출력한다. h 이외의 올바른 옵션이 입력되었으면 cache의 s, E, b 또는 전역변수 verbose, 파일 이름을 저장하는 traceFile 변수 등에 값을 저장한다.

B. Cache 구조. cache:(csim:11~23), setup():(csim:47~66), cleanup():(csim:69~69)

CACHE 구조체를 선언하고, CACHE 구조체 타입으로 cache 변수를 선언하여 캐시로써 사용하였다. cache 구조체는 LINE 구조체, LINE형 포인터 line, 그리고 주소를 분리할 때 사용하는 마스크와, hit, miss, eviction 카운터, 그리고 옵션으로 주어지는 s, E, b 변수를 갖는다. (set_index_mask만 실제로 사용하였다.)

char* 형으로 주어진 msg_ 변수는 각각 "hit", "miss", "miss eviction" 값이 저장되며, C++이 아닌 C 언어에서 char*, const char* 형식의 문자열을 다루는 것이 미숙하여 이런 방법을 선택하였다.

LINE* 타입의 line 포인터는 setup() 함수에서 s와 E에 따라 필요한 메모리를 할당받으며, setup() 함수에서 해제된다. 두 함수는 각각 프로그램의 시작과 끝에서 한 번씩만 호출된다.

LINE* line 포인터에 할당되는 메모리는 LINE 구조체의 크기 * 2^s * E 바이트 만큼 필요하며, 2^s * E개의 라인이 필요하다.

C. trace 파일로부터 입력받기.

주어진 writeup pdf 파일에서 자세한 설명을 읽고 그에 따라 구현하였다.

valgrind 도구를 이용해 생성되는 흔적은 다음과 같은 구조이다. (정규표현식)

```
[ ]?[ILMS][ ]+[0-9a-f]+,[0-9]+
```

공백 및 콤마(,) 로 구분된 세 부분은 각각 명령어([ILMS]) / 주소(16진수 정수, [0-9a-f]+) / 크기([0-9]+)이다.

명령어가 I인 경우는 무시하며, 명령어가 L 및 S, 데이터를 불러오거나 저장하는 경우 주소에 한 번 접근하며, 명령어가 M, 즉 데이터를 수정하는 경우 데이터에 접근하고 수정하므로 두 번 접근한다. 따라서 각 때에 따라 access() 함수를 통해 주소에 한 번 또는 두 번 접근하도록 구현하였으며, verbose 옵션이 있으면 결과를 출력하도록 구현하였다.

D. access() - 접근하는 메모리 주소를 통해 hit, miss, eviction 판단. (csim:71~124)

가장 먼저 적절한 비트연산을 통해 tag와 set_index를 분리하였다. tag는 주어진 주소를 s+b만큼 오른쪽으로 시프트 연산을 취해 얻을 수 있으며, set_index는 다음 방법으로 분리하였다.

```
set_index = (addr >> cache.b) & ((1 << cache.b) - 1)
```

cache의 각 라인은 valid, tag, lru 값을 가지며, 초기에는 모두 0으로 초기화되어 있다. 또한 cache는 lru_counter의 값을 가지며, 추후 eviction 판별에 사용될 것이다.

가장 먼저 cache hit인지 검사한다.

cache.line에서 set_index로부터 E개의 라인을 순회하면서 valid가 1이면서 입력된 주소의 태그와 동일한 태그가 있는 라인이 있는지 확인한다. 만약 존재한다면 hit이 발생한 경우로 lru를 lru_counter의 값으로 업데이트하고 lru_counter의 값은 1 증가시킨다.

그 다음으로는 cache miss (not eviction)인지 확인한다.

역시 cache.line에서 set_index로부터 E개의 라인을 순회하면서, valid하지 않은 라인이 있으면 miss가 발생한 경우이다. 이 경우 해당 줄의 valid는 1로 설정하고 tag와 lru 값을 업데이트한다.

hit과 miss 두 경우가 아니라면 남은 경우는 miss eviction이다.

이 경우 역시 set_index의 E개의 줄을 순회하면서 가장 낮은 lru를 가진 라인을 탐색한다. 해당 라인이 가장 오랫동안 사용되지 않은 라인으로, 그 줄을 입력된 주소에 해당하는 태그로 업데이트하고 lru에 새로운 순서를 할당한다.

이러한 로직을 통해 캐시를 시뮬레이션할 수 있었다.

3. Part B. Matrix transpose (trans.c)

이 문제에서는 세 가지 종류의 행렬을 전치하는 데 최적화하는 것을 목표로 하며, 세 가지 경우를 각각

최적화하는 것이 허용된다.

A. 32×32 matrix.

조건: 32 × 32: 8 points if m < 300, 0 points if m > 600

기본적으로 주어지는 row-wise 전치 함수를 실행한 후 로그를 시뮬레이션하면 L 명령, 즉 행렬 A에서 값을 로드할 때에는 대부분 hit이지만, 행렬 B에 값을 저장할 때 대부분 miss eviction이 발생한다. 반대로, column-wise 전치 함수는 행렬 A에서 값을 로드할 때에 대부분 miss eviction, 행렬 B에 값을 저장할 때 hit이 발생한다.

```
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N]) {
        int i, j, tmp;
        for(i = 0; i < N; i++) {
                for(j = 0; j < M; j++) {
                        tmp = A[i][j];
                        B[j][i] = tmp;
                }
        }
}
```

두 가지 경우 모두 cache hit은 869회, miss는 1,184회 발생한다. 여기서 캐시 miss를 줄이기 위하여 블로킹 및 더 많은 임시 변수를 사용하였다.

a. 블로킹의 경우, 실행 옵션으로 b = 5이므로 32바이트가 같은 태그 및 set에 들어가며, int 자료형은 4바이트 자료형이므로 총 8개의 연속된 값이 같은 set 내에 포함될 것으로 판단하였다. 따라서 다음과 같이 수정하였다.

```
int i, j, tmp0, tmp1;
for(i = 0; i < 32; i+=8) {
        for(j = 0; j < 32; j+=8) {
                for(tmp0 = 0; tmp0 < 8; tmp0++) {
                        for(tmp1 = 0; tmp1 < 8; tmp1++) {
                                B[j+tmp1][i+tmp0] = A[i+tmp0][j+tmp1];
                        }
                }
        }
}
```

블로킹을 통해 수정한 코드는 cache miss의 수가 344개까지 줄었지만, 아직 충분하지 않다.

b. 여기서 더욱더 줄이기 위해서, 더 많은 임시 변수를 사용하였다. 문제에서 제약 조건으로 지역 변수를 최대 12개까지 사용할 수 있다는 제한이 있다. 이는 반복문에 사용되는 변수도 포함되는데, 위 코드에서 4번의 반복문을 사용하고 있으므로 8개의 지역 변수를 사용할 수 있다. 따라서 반복문 인덱스를 바꾸고 tmp0부터 tmp7까지 선언하여 사용해 주었더니 miss 횟수를 300회 이하인 288회로 줄일 수 있었다.

```
int i, j, a, tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
for(i = 0; i < 32; i+=8) {
	for(j = 0; j < 32; j+=8) {
		for(a = 0; a < 8; a++) {
			tmp0 = A[i+a][j+0]; tmp1 = A[i+a][j+1];
			tmp2 = A[i+a][j+2]; tmp3 = A[i+a][j+3];
			tmp4 = A[i+a][j+4]; tmp5 = A[i+a][j+5];
			tmp6 = A[i+a][j+6]; tmp7 = A[i+a][j+7];
			B[j+0][i+a] = tmp0; B[j+1][i+a] = tmp1;
			B[j+2][i+a] = tmp2; B[j+3][i+a] = tmp3;
			B[j+4][i+a] = tmp4; B[j+5][i+a] = tmp5;
			B[j+6][i+a] = tmp6; B[j+7][i+a] = tmp7;
		}
	}
}
```

B. 64×64 matrix.

조건: 64 × 64: 8 points if m < 1300, 0 points if m > 2000

열과 행의 길이가 둘 다 64인 행렬은 목표 miss 횟수까지 줄이는 데에 실패하였고, 최소 miss 횟수는 1636회이다.

가장 먼저 시도한 것은 위 32 × 32 행렬 전치 코드에서 반복문 변수 i와 j의 범위를 32에서 64까지 늘리는 것인데, 이 경우 3585 hits, 4612 misses를 받았다. 원인을 찾아보기 위해 csim-ref를 이용해 로그를 시뮬레이션해보니, 한 반복 내에서 행렬 A의 값을 L, 로드할 때 8번 중 1번 miss가 발생하며, 행렬 B에 값을 저장할 때 8번 모두 miss가 발생함을 확인할 수 있었다.

이를 해결하기 위해 블로킹의 블록 크기를 조절하는 것을 시도하였다. 8개의 임시 변수를 사용하는 것은 1개의 임시 변수를 사용하는 것보다 나빠질 가능성이 없다는 생각이 들었기 때문에, 블로킹을 반복해서 수정하였다. 많은 임시 변수를 사용함으로써 한 블록 내에서 로드 및 저장의 순서를 조절하는 것인데, 1개의 임시 변수 (row-wise 또는 column-wise)는 절반 정도가 miss가 발생한다. 그러나 32 × 32 행렬을 전치할 때 8개의 임시 변수를 좋은 방향으로 활용하여 코드를 개선했기 때문에, 변수 수의 문제는 아니라고 생각했다.

블록 크기를 적절히 수정하면서 최적의 값을 찾으려 시도하였다. 이때 블록 크기는 「Using Blocking to Increase Temporal Locality」에 따라 64의 약수가 되도록, 2, 4, 8, 16 등의 값에 대해 miss 발생 횟수를 비교하였다. 그 결과 4 × 4를 한 블록으로 했을 때 miss의 수가 1652가 나옴을 확인하였다. 또한, 행렬 A 값의 로드 순서와 행렬 B에 저장하는 순서를 최대한 캐시 친화적으로 수정한 결과 miss 발생 횟수를 1636회까지 줄일 수 있었다.

```
int i, j;
int tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
for(i = 0; i < M; i += 4) {
        for(j = 0; j < N; j += 4) {

tmp0 = A[j][i];       tmp1 = A[j][i+1];    tmp2 = A[j][i+2];    tmp3 = A[j][i+3];
tmp4 = A[j+2][i];     tmp5 = A[j+2][i+1];  tmp6 = A[j+2][i+2];  tmp7 = A[j+2][i+3];
B[i][j] = tmp0;       B[i+1][j] = tmp1;    B[i+2][j] = tmp2;    B[i+3][j] = tmp3;
B[i][j+2] = tmp4;     B[i+1][j+2] = tmp5;  B[i+2][j+2] = tmp6;  B[i+3][j+2] = tmp7;
tmp0 = A[j+1][i];     tmp1 = A[j+1][i+1];  tmp2 = A[j+1][i+2];  tmp3 = A[j+1][i+3];
tmp4 = A[j+3][i];     tmp5 = A[j+3][i+1];  tmp6 = A[j+3][i+2];  tmp7 = A[j+3][i+3];
B[i][j+1] = tmp0;     B[i+1][j+1] = tmp1;  B[i+2][j+1] = tmp2;  B[i+3][j+1] = tmp3;
B[i][j+3] = tmp4;     B[i+1][j+3] = tmp5;  B[i+2][j+3] = tmp6;  B[i+3][j+3] = tmp7;

        }
}
```

목표치인 1,300회 이하로 줄이는 방법을 생각해 봤으나, 도저히 떠오르지 않아 여기서 그만두게 되었다.

다만 재미있는 것은, 여러 시도 중 캐시-적대적 프로그래밍에 성공하였다. 해당 코드를 여기 첨부한다. 이 코드가 유의미하게 사용되길 희망해 본다.

```
// hit: 1, miss: 8196 Wow! maybe this is the world's worst code! omg :(
int i, j;
for(i = 0; i < 8; i ++) {
        for(j = i; j < M*N; j += 8) {
                B[j%64][j/64] = A[j/64][j%64];
        }
}
```

C. 61×67 matrix.

조건: 61 × 67: 10 points if m < 2000, 0 points if m > 3000

위 문제는 이전 64×64 행렬 전치 문제를 풀면서 다음 문제는 얼마나 어려울까 두려워했지만, 생각보다 쉽게 풀 수 있었다. 역시 이전과 마찬가지로 8×8 크기의 블록으로 나누어 행렬을 뒤집도록 코드를 작성하였다. 대신 이전과 다르게 행렬의 열과 행이 깔끔하게 어떤 수로 나누어 떨어지지 않기 때문에, 그에 대한 처리를 추가로 작성하였다.

```
int i, j;
int ii, jj;
int tmp0, tmp1, tmp2, tmp3;
for(i = 0; i < M; i += 8) {
        for(j = 0; j < N; j += 8) {
                for(ii = i; ii < i + 8 && ii < M; ii += 4) {
                        for(jj = j; jj < j + 8 && jj < N; jj++) {
                                if(ii <= M) tmp0 = A[jj][ii];
                                if(ii+1<= M) tmp1 = A[jj][ii+1];
                                if(ii+2<= M) tmp2 = A[jj][ii+2];
                                if(ii+3<= M) tmp3 = A[jj][ii+3];
                                if(ii <= M) B[ii][jj] = tmp0;
                                if(ii+1<= M) B[ii+1][jj] = tmp1;
                                if(ii+2<= M) B[ii+2][jj] = tmp2;
                                if(ii+3<= M) B[ii+3][jj] = tmp3;
                        }
                }
        }
}
```

위 코드를 사용하여 행렬을 전치한 결과 2124번 miss가 발생하였다. 이를 해결하기 위해 이전 문제와 마찬가지로 블록 크기를 적절히 조절한 결과, 반복문 내 변수 i와 j가 8이 아니라 16씩 더하도록 하면 miss 수가 목표치인 2000 이하의 1914회로 낮아지는 것을 확인하였고, 문제를 풀 수 있었다.

```
int i, j;
int ii, jj;
int tmp0, tmp1, tmp2, tmp3;
for(i = 0; i < M; i += 16) {
        for(j = 0; j < N; j += 16) {
                for(ii = i; ii < i + 16 && ii < M; ii += 4) {
                        for(jj = j; jj < j + 16 && jj < N; jj++) {
                                if(ii <= M) tmp0 = A[jj][ii];
                                if(ii+1<= M) tmp1 = A[jj][ii+1];
                                if(ii+2<= M) tmp2 = A[jj][ii+2];
                                if(ii+3<= M) tmp3 = A[jj][ii+3];
                                if(ii <= M) B[ii][jj] = tmp0;
                                if(ii+1<= M) B[ii+1][jj] = tmp1;
                                if(ii+2<= M) B[ii+2][jj] = tmp2;
                                if(ii+3<= M) B[ii+3][jj] = tmp3;
                        }
                }
        }
}
```

4. Result

다음은 채점 프로그램 driver.py의 실행 결과이다. Part. A에서는 캐시 시뮬레이터를 잘 작성하여 모든 테스트를 통과하였지만, Part. B에서는 64×64 행렬을 전치하는데 miss를 완전히 줄이지 못해 53점 만점에 총 49.2점을 획득하였다.

```
root@LAPTOP1807:~/cachelab# python2 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         288
Trans perf 64x64           4.2         8        1636
Trans perf 61x67          10.0        10        1914
          Total points    49.2        53
```

추가로, 아래는 작성한 csim과 주어진 csim-ref 사이에서, ls 명령어를 시뮬레이션한 결과로, 둘이 일치함을 알 수 있다.

```
root@subvnic:/root/cachelab# ./csim-ref -s 5 -E 1 -b 5 -t ./traces/custom_ls.trace
hits:153312 misses:59333 evictions:59301
root@subvnic:/root/cachelab# ./csim -s 5 -E 1 -b 5 -t ./traces/custom_ls.trace
hits:153312 misses:59333 evictions:59301
```

5. Reference

[1] Computer Systems: A Programmer's Perspective, 3/E (CS:APP3e), Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University.

[2] [Writeup] CSED211, Fall 2024 Cache Lab: Understanding Cache Memories, Instructor. Prof. Jisung Park.

[3] CS:APP2e Web Aside MEM:BLOCKING: Using Blocking to Increase Temporal Locality, Randal E. Bryant and David R. O'Hallaron