Cache Lab: Understanding Cache Memories

20190084 권민재, CSED211

Part A: Writing a Cache Simulator

개요

- 이 프로그램은 s (Number of set index bits), E (number of lines per set), b (Number of block bits), t (파일 이름)를 입력으로 받는다. 이 중 특히 t 같은 경우는, valgrind trace가 담긴 파일의 파일 이름이다.
- 이 프로그램은 이 입력에 따라 캐시를 시뮬레이션 하고, 그 결과로 hit, miss, eviction의 횟수를 출력한다.

구조

자료 구조

Cache

```
typedef struct {
    int numSet;
    int numLine;
    Set *setList;
} Cache;
```

`Cache` 는 메인 캐시를 표현하기 위한 구조체로, `numSet`, `numLine`, `arrSet` 으로 구성된다.

- `numSet` 은 캐시의 set 개수로, 2^s 가 저장된다.
- `numLine` 은 set 마다의 line 개수로, E가 저장된다.
- `setList` 는 `Set` 자료형의 포인터로, `numSet` 개의 set들이 동적할당 된 배열을 가리킨다.

Set

```
typedef struct {
    Block *blockList;
} Set;
```

`Set` 은 캐시의 set을 표현하기 위한 구조체로, `blockList`로 구성된다.

- `blockList` 는 각 set 에 존재하는 `block` 배열이며, E 개의 block들이 동적할당 된 배열을 가리킨다.

Block

```
typedef struct {
    int valid;
    unsigned long long tag;
    int lruCount;
} Block;
```

`Block` 은 캐시의 block을 표현하기 위한 구조체로, `valid`, `tag`, `lruCount` 로 구성된다.

- `valid` 는 이 블럭이 유효한지 그 여부를 나타낸다.
- `tag` 는 블럭의 tag를 나타낸다.
- `lruCount` 는 LRU 기반의 캐시를 구현하기 위해, 각 블럭 별로 지정되는 LRU 값, 즉 일종의 타임스탬프로서 작용한다.

전역 변수

```
Cache cache = {};
int s = 0, E = 0, b = 0;
int LRU = 0;
int hits = 0, misses = 0, evictions = 0;
```

- `cache` 는 이 프로그램에서 시뮬레이션 할 캐시를 의미한다.
- `s`, `E`, `b` 에는 이 프로그램의 인자로 들어온 s, E, b 가 저장된다.
- `LRU` 는 LRU 로 캐시를 구현하기 위한 전역 LRU 카운터이다.
- `hits`, `misses`, `evictions` 는 hit, miss, eviction의 횟수로, 이 프로그램의 결과값이 담긴다.

알고리즘

1. Parse Input

우선 프로그램이 실행되면, `getopt` 를 이용하여 인자로 들어온 s, E, b, t를 저장한다. 만약, 인자가 하나라도 없다면 1을 반환하며 프로그램을 종료한다. 그리고, t를 파일 이름으로 해서 파일을 여는데 실패했다면, 1을 반환하면서 프로그램을 종료한다.

2. Init Cache

initCache()

캐시를 초기화 하는데에는 함수 `initCache`가 이용된다. 이 함수에서는 캐시의 `numSet`에 2^s, `numLine`에 E를 저장하고, `setList`에 `numSet` 개의 set으로 구성된 배열을 동적할당 한다. 그리고, `setList`의 각 set에 대해 `numLine` 개의 block으로 구성된 배열을 동적할당 한다. 각 block들의 `valid`, `tag`, `lruCount`는 0으로 초기화 된다.

3. Simulate Cache

simulate(FILE* file)

캐시를 시뮬레이션 하는데 있어서 우선 이 함수가 불리게 된다. 이 함수는 `file`을 한 줄 씩 읽으면서 경우에 따라 `cacheAccess`를 호출하여 캐시에 액세스하는 행위를 시뮬레이션 한다. 우리는 valgrind trace의 operation으로 I, L, S, M을 입력받는데, 이 중 **I 가 입력되었을 때에는** 캐시에 액세스할 필요가 없으므로, 무시하고, **나머지 경우에 대해서는** 우선 기본적으로 1번 캐시 액세스를 진행한다. 하지만, 이 중에서도 **M 의 경우에는** 데이터를 수정한 경우이므로, 캐시에 1번 더 액세스를 해야한다. 즉, 요약하면 이 함수에서는 파일을 한 줄 씩 읽으면서 인스트럭션을 해석하는데, 이때 `cacheAcess`는 **I일 때 0번, L이나 S 일때 1번, M일 때 2번** 호출된다.

cacheAccess(unsigned long long addr)

이 함수는 어떤 주소를 가지고 캐시에 액세스하는 행위를 시뮬레이션 한다. 우선, 그를 위해서 `addr` 로부터 인덱스와 태그를 구해내고, eviction을 수행할 기준이 되는 LRU `evictionLru`를 충분히 큰 값으로 설정한다. `addr`의 인덱스는 `addr`를 `b` 만큼 오른쪽으로 밀고, 그 값을 $2^s - 1$과 AND 함으로써 구할 수 있다. 또한, `addr`의 태그는 `addr`을 $s + b$ 만큼 오른쪽으로 shift해서 구할 수 있다. 이제 `addr`의 인덱스를 이용해 캐시의 `setList` 중에서 접근할 `set`을 결정하고, 해당 set의 모든 block에 대해 탐색을 수행하면서, 해당 블록이 `valid`하고 `addr`의 태그와 같다면 hit가 발생했음을 기록한다.

block들에 대해 탐색이 종료된 후, 해당 탐색에서 hit가 발생했다면, `hits` 를 1 증가시키고, hit가 발생한 블럭의 `lruCount`에 전역 LRU 카운터 `LRU` 를 할당한다.

만약, hit가 발생하지 않았다면, miss가 발생했다는 뜻이므로, 우선 `misses`를 1 증가시킨다. 이후, set의 블럭들 중에서 `lruCount`가 가장 작은 블럭을 찾는다. 만약, 이 블럭이 `valid` 하다면, eviction이 일어나야 하므로 `evictions`를 1 증가시킨다. 이후, 해당 블럭의 `valid`를 1, 태그를 현재 `addr` 의 태그, 그리고 `lruCount`를 현재 전역 LRU 카운터 `LRU`로 설정한다.

마지막으로, 전역 LRU 카운터 `LRU`를 1 증가시키고 함수를 종료한다.

4. Finalize

앞서 **Simulate Cache** 과정을 거치면서 캐시를 시뮬레이션 하는데 있어서 hit, miss, eviction의 횟수를 모두 구했으므로, `printSummary` 함수를 이용해서 `hits`, `misses`, `evictions`를 출력하고 파일을 닫는다. 또한, `freeCache` 함수 또한 호출하여 할당된 메모리를 정리한다.

freeCache()

이 함수에서는 앞서 `initCache`에서 캐시의 시뮬레이션을 위해 동적 할당한 메모리를 해제한다. 그래서 우선 각 `setList`의 `Block`들을 할당 해제 하고, 이후 `setList`를 할당 해제하여 동적 할당한 모든 메모리를 정리한다.

결과

```
                        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

TEST_CSIM_RESULTS=27
```

토론 및 개선

- 이 코드에서는 구조체를 이용하여 캐시를 구현했는데, 구조체를 이용하여 캐시를 구현하고 나니, 2차원 배열으로도 충분히 이 가상 캐시를 구현할 수 있다는 것을 알았다. 이후, 2차원 배열으로도 캐시를 구현해보고자 한다.
- 이 과제에서는 **전역변수에 대해 제한을 두지 않고 있어서**, 전역변수들을 이용하여 문제를 해결하고 있는데, 이 전역변수들의 사용을 줄일 필요성이 있다.

Part B: Optimizing Matrix Transpose

개요

- 이 코드는 32×32, 64×64, 61×67 의 행렬에 대해 transpose 연산을 수행하되, 최대한 적은 miss가 일어나도록 최적화하는 코드이다.
- s = 5, E = 1, b = 5인 캐시를 사용한다.

알고리즘

크기가 $32 \times 32, 61 \times 67$인 행렬의 최적화

크기가 $32 \times 32, 61 \times 67$ 인 행렬에 대해서는 같은 전략을 사용해서 과제에서 요구하는 만큼의 miss를 달성할 수 있다. 모든 최적화에서 당연한 것이지만, 우리는 eviction을 최소화 하는 방향을 통해 최적화를 수행할 것이다. 이를 위해 우선, 캐시를 활용하는 가장 기본적인 최적화 전략인 **blocking**을 이용한다. 이 캐시는 associativity가 1인 direct cache이며, 한 라인의 크기는 $2^5 = 32\text{byte}$ 이다. 32바이트에는 8개의 int가 적재될 수 있으므로, 우리는 이 연산을 최적화 하는데 있어서 각 행렬을 크기가 8×8 인 행렬로 쪼개서 transpose를 수행하는 방법을 통해 miss를 최대한 줄일 수 있다. 이를 구현한 것이 함수 `blocking` 이다.

하지만, 8×8 인 행렬로 쪼개서 transpose를 수행하는 것만으로는 과제에서 요구하는 miss를 달성할 수 없다. 이를 위해서는 eviction이 일어나는 지점을 더 찾아서 최적화해야 하는데, 이를 찾아보면 같은 인덱스로 값을 옮길 때, 즉 `B[x][y] = A[y][x]` 를 수행할 때 eviction이 발생하는 것을 알 수 있다. 이를 피하기 위해, 우리는 행렬의 대각선이 포함된 행렬에 대해서는, row와 column의 인덱스가 같은 값에 대해 레지스터를 활용하여 최적화해야 함을 알 수 있다. 그래서, 대각선이 포함된 부분 행렬에 대해서는 함수 `blocking_diag()` 를 통해 최적화를 수행했다.

그러므로, 8×8 인 행렬로 쪼개서 transpose를 수행하되, 기본적으로는 `blocking`을 이용하나, 대각선이 포함된 부분 행렬이라면 `blocking_diag`를 이용하여 전치를 수행한다.

함수

blocking(int M, int N, int A[N][M], int B[M][N], int ROW, int COL)

Num. of Local Variable 3 + 2 = 5 (2 from `transpose_submit`)

이 함수는 대각선이 포함되지 않은 8×8 부분 행렬에 대해 transpose를 수행한다. 이 함수는 주어진 `ROW`, `COL` 로 부터 M, N을 넘지 않는 범위에 대해 기본적인 연산을 통해 8×8 만큼의 transpose를 수행한다.

blocking_diag(int M, int N, int A[N][M], int B[M][N], int ROW, int COL)

Num. of Local Variable 3 + 3 = 6 (2 from `transpose_submit`)

이 함수는 대각선이 포함된 8×8 부분 행렬에 대해 transpose를 수행한다. `B[x][y] = A[y][x]`에서 eviction이 발생하므로, 이를 피하기 위해 row와 col에 대해 반복하되, 두 값이 같다면 transpose를 수행하지 않는다. 대신, col의 반복이 시작하기 전에 `A[row][row]`의 값을 변수에 미리 담아두고, col의 반복이 종료된 이후에 `B[row][row]`에 미리 담아둔 값을 넣는다. 이를 통해 eviction을 줄이고, miss를 최적화 할 수 있다.

결과

32×32

```
./test-trans -M 32 -N 32

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255
```

```
Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151

Summary for official submission (func 0): correctness=1 misses=287

TEST_TRANS_RESULTS=1:287
```

61×67

```
./test-trans -M 61 -N 67

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6225, misses:1960, evictions:1928

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3756, misses:4423, evictions:4391

Summary for official submission (func 0): correctness=1 misses=1960

TEST_TRANS_RESULTS=1:1960
```

크기가 64×64인 행렬의 최적화

크기가 64×64 인 행렬에서는 단순히 8×8 의 행렬로 쪼개서 최적화하는 것으로는 좋은 성과를 이뤄낼 수 없다. 왜냐하면, 64×64 행렬을 8×8 행렬로 쪼갰을 때, 해당 행렬의 위쪽 반 4×8과 아래쪽 반 4×8 들의 주소들이 거의 같은 set을 공유하기 때문이다. 그렇기 때문에, 우리는 8×8 의 행렬으로 쪼갠 부분 행렬을 다시 반씩 나눠서 transpose를 수행함으로서 최적화를 진행해야 한다. 하지만, 여기서 더 극한으로 최적화를 진행하기 위해, 우리가 사용할 수 있는 로컬 변수의 수를 모두 이용할 것이다. 4×1 크기의 한 줄을 변수에 미리 담아두고, 해당 줄을 변수들을 이용해 마지막에 옮김으로써, miss를 최소화 할 것이다. 이를 구현한 것이 바로 함수 `blocking_64`이다.

함수

blocking_64(int M, int N, int A[N][M], int B[M][N], int ROW, int COL)

Num. of Local Variable. 3 + 9 = 12 (2 from `transpose_submit`)

이 함수에서는 우선 행렬 A의 오른쪽 위 4개의 값을 변수에 미리 저장한다. 이후, 행렬 A의 왼쪽 반을 값 위에서부터 읽어나가면서 값 4개씩 transpose를 수행하고, 행렬 A의 오른쪽 반을 아래에서부터 읽어나가면서 값 4개씩 transpose를 수행한다. 이때, 오른쪽 반의 맨 윗 줄은 이미 변수에 저장해뒀으므로, 반복에서 제외된다. 이후, 모든 반복이 종료된 이후에, 미리 저장해 둔 변수를 이용하여 오른쪽 반의 맨 윗줄에 대해 transpose를 수행해서 transpose를 마무리한다.

결과

```
./test-trans -M 64 -N 64

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6842, misses:1355, evictions:1323

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3474, misses:4723, evictions:4691

Summary for official submission (func 0): correctness=1 misses=1355

TEST_TRANS_RESULTS=1:1355
```

결과

```
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        1960
          Total points    52.4        53
```

토론 및 개선

- 64 x 64 크기의 캐시를 최적화할 때, 대각선 성분에 대해 접근을 막아서 eviction이 일어나지 않도록 더 최적화할 필요성이 있다.