CSED211 Intro. to Computer SW Systems : Cache Lab Report

November 24, 2025

Student 20240505 Hyunseong Kong

1. Overview

이번 과제는 캐시 시뮬레이션, 효율적인 행렬 전치 알고리즘을 구현하는 것을 목적으로 한다. 캐시 시뮬레이션에는 valgrind로 만들어진 trace 파일이 사용되며, 행렬 전치는 block size가 32Byte이고 전체 크기가 1KB 크기의 캐시가 주어짐을 가정한다.

2. Cache Simulation

문제에서 요구하는 것은 `csim-ref`와 동등한 작업을 수행하는 프로그램을 구현하는 것이다. 이를 위해, 하나의 Set과 Line을 각각 나타내는 아래의 두 구조체를 정의하였다.

```
typedef struct {
    int valid;
    unsigned long tag;
    unsigned long lru; // LRU timestamp
} Line;
```

```
typedef struct {
    Line *lines;
} Set;
```

또한 입력받은 파라미터를 저장하고, 캐시를 시뮬레이션하며 그 결과를 저장하기 위한 전역 변수를 다음과 같이 정의하였다.
파라미터를 입력받기 위한 전역 변수:

```
char verbose = 0;
unsigned s = 0;
unsigned E = 0;
unsigned b = 0;
char* filePath = "";
```

캐시 시뮬레이션을 위한 전역 변수:

```
Set *cache = NULL; // cache
unsigned S = 0; // set count(=2^s)
unsigned long globalTime = 0; // global time for LRU
```

최종 출력을 위한 전역 변수:

```
int hitCount = 0;
int missCount = 0;
int evicCount = 0;
```

2-1. printUsage

Function Prototype:

```
void printUsage(char *argv0);
```

Explanation: 이 함수는 잘못된 파라미터를 입력(파라미터가 완전하지 않은 경우)하거나, `-h`를 입력하였을 때 사용 방법을 출력하기 위해 정의하였다. 함수 내부는 고정된 출력값을 가지며, 0번째 입력값인 argv0(프로그램을 실행하기 위해 입력한 이름)를 받아서 사용 방법 내부에 적절히 출력하였다. 출력값은 `csim-ref`의 것을 참고하여 만들어졌다.

2-2. argParser

Function Prototype:

```
char argParser(int argc, char *argv[]);
```

Explanation: 이 함수는 파라미터의 개수 argc와 각 파라미터를 저장한 argv를 받아서, getopt를 이용해 파라미터 입력을 처리했으며, char flag를 이용해 각 파라미터가 입력될 때마다 비트에 기록하는 방식으로 총 4개의 파라미터(s, E, b, t)가 전부 올바르게 입력되었는지 확인한다.
함수는 main에서 1번 실행되며, 그 반환값에 의해 코드를 진행할지 아니면 printUsage를 호출하고 종료할지를 결정한다. 함수가 1을 반환하면 올바르지 않은 파라미터가 입력되었다는 의미이다.

2-3. initCache

Function Prototype:

```
void initCache();
```

Explanation: 이 함수는 올바른 파라미터가 설정된 이후, 캐시 시뮬레이션을 시작하기 위해 main에서 1번 실행되는 함수이다. 주어진 파라미터 s, E, b를 이용해 우선, 전역변수 cache를 $S = 2^s$개의 set을 할당하며, 그 후 cache의 각 i번째 lines에 다시 E개의 Line 변수를 malloc으로 할당한다.
하나의 Line은 valid, tag, lru로 구성되어 있는데, 여기서 valid는 현재 Line이 접근가능한지를 체크하며 초기에 전부 0으로 설정되어 우연히 tag가 겹치는 경우를 방지한다. lru는 eviction 횟수를 판별하는데 이용하는 것으로, 정수 형태로 시각이 기록되며, 가장 빠른 시간이 선택되게 된다. 이는 accessData에서 후술한다.

2-4. freeCache

Function Prototype:

```
void freeCache();
```

Explanation: 이 함수는 프로그램의 종료 시에, malloc으로 할당했던 모든 메모리를 해제하여 메모리 누수를 방지하기 위한 함수로 main의 마지막 부분에서 1번 실행된다.

2-5. accessData

Function Prototype:

```
void accessData(unsigned long address);
```

Explanation: 이 함수는 1번의 캐시 작업을 처리하는 함수이다. 우리가 받는 모든 입력 데이터는 '연산종류 주소,데이터크기'로 구성되어 으며, 이 함수는 주소를 입력받는다. 연산의 종류는 캐시 시뮬레이션에 필요하지 않으며, 이는 우리가 실제 데이터 입출력은 수행하는 것 대신에, 캐시에 접근해서 얼마나 많은 충돌이 발생하는지에만 관심이 있기 때문이다.
각 accessData의 실행은 다음의 작업을 수행한다:

1. setIndex, tag 계산. 여기서 tag는 address에 들어가있음을 명심하자.
2. setIndex에 대응되는 set을 찾기.
3. hitLine, emptyLine, lruLine 찾기. 각각의 작업은 현재 set에서 모든 line을 순차적으로 돌며 valid와 tag를 통해 판별한다. hitLine은 `valid == 1`이면서 tag가 같은 것이고, emptyLine은 `valid == 0`인 첫번째 line이다. 마지막으로, lruLine은 순차적으로 돌면서 `valid == 1`이고 lru 값이 최소인 것을 찾는다.
4. hit이 존재하면, 끝. 아니면, miss이며 이때 emptyLine이 없다면 lruLine을 대체한다. 이때 eviction이 발생한다.

2-6. fileAnalyze

Function Prototype:

```
void fileAnalyze(FILE *fp);
```

Explanation: 이 함수는 파일 포인터를 받아서 파일을 한 줄 씩 실행하며, 위의 accessData를 실행하는 함수이다. 이 함수가 최종적으로 캐시 시뮬레이션을 진행하며 각 줄을 sscanf로 받는다. 이때 2번째 인자로 `" %c %lx,%u"`를 입력하며, 이는 앞의 empty string을 strip하는 역할을 수행한다. 이제 operation이 I이면 그대로 다음 줄로 넘기고, 그렇지 않으면 L 또는 S일 때 accessData를 1번, M일 때 accessData를 2번 실행(M은 쓰고 읽는 연산을 수행하기 때문에 2번 실행해야 함)한다.

3. Matrix Transpose

miss를 가능한 한 줄이기 위해서 모든 파트에서 행렬 곱셈 최적화에서 배운 block matrix를 활용한다. 과제 안내사항에 의하면 주어진 cache는 direct mapped cache이기 때문에, 하나의 set에 하나의 line만이 들어간다. line 내 block 크기는 32Byte이고, 전체 크기가 1KB 크기의 캐시가 주어졌기에 set은 총 32개이다.

3-1. M=32, N=32

우선, 초기 시도로 8x8 격자로 쪼개어 block 단위로 전치를 수행하였다. 각 블록 내에서 다시 for문을 활용하여 값을 하나씩 옮겼으며, 결과는 다음과 같다:

```
func 0 (Transpose submission): hits:1709, misses:344, evictions:312
func 1 (Simple row-wise scan transpose): hits:869, misses:1184, evictions:1152
```

온전한 점수를 받기 위해서는 miss가 300 미만으로 떨어져야 하기 때문에, 최소 45회 이상의 miss를 더 줄여야 함을 알 수 있다(달리 말해, 그런 방법이 존재한다). 지금 코드는 캐시 상에서 불필요한 충돌이 발생하는 것이라 판단하였고, 이를 더 분석하기 위해 A와 B가 저장되는 메모리 주소를 확인하고자 하였다. 이에, ./tracegen을 gdb를 이용해 열고, M과 N에 각각 32을 입력하여 실행하였다. 그렇게 확인한 초기 주소는 다음과 같다:

```
Breakpoint 1, transpose_submit (M=32, N=32, A=0x555555559100 <A>, B=0x555555599100 <B>) at
trans.c:24
```

두 주소의 차이는 0x40000이다. 캐시는 주소의 6번째부터 10번째까지(s=5) 값을 읽어들여 set의 index로 삼기 때문에, A와 B 각각의 시작 주소를 근거로 두 행렬의 시작점이 캐시 상에서 불러와지는 주소는 같다고 할 수 있다.

$0, \cdots, 32$	$1, \cdots, 33$	$2, \cdots, 34$	$3, \cdots, 35$
$0, \cdots, 32$	$1, \cdots, 33$	$2, \cdots, 34$	$3, \cdots, 35$
$0, \cdots, 32$	$1, \cdots, 33$	$2, \cdots, 34$	$3, \cdots, 35$
$0, \cdots, 32$	$1, \cdots, 33$	$2, \cdots, 34$	$3, \cdots, 35$

위는 각 (i,j)-블록 내에서, row의 set index의 범위를 나타내는 표이다. 각 간격은 4칸이며, 이에 (0,0)-블록의 경우, $0, 4, 8, 12, \cdots, 28, 32$번 set index를 사용하게 된다. 32개 캐시 블록이 256개 int 타입을 저장할 수 있으며, 블록 1개당 정수 64개가 있기 때문이다.
이때, A와 B를 블록으로 나눴을 때 그 대각성분의 주소는 전부 겹치게 된다. 반면, 대각 성분이 아니라면 겹치지 않는다. 이를 해결하기 위해서는 겹치지 않는 각 블록은 for문을 활용해서 연산하되, 대각성분은 for문을 다르게 설계하여야 한다. 더 정확하게, 대각 블록은 값을 하나 꺼내어 하나 넣을 때마다 충돌이 발생하며, 이를 해결하기 위해 레지스터를 활용해 한 줄 단위로 값을 꺼내고 넣어 충돌을 최소화한다.
컴파일러는 함수의 지역 변수를 레지스터를 이용한 코드로 변환해주기 때문에, 지역 변수를 이용한다. 이때, 과제의 제한조건에 의해 최대 12개의 지역 변수만을 선언할 수 있다. 블록 단위 쪼개기, 블록 내부 index를 지정하는데 4개의 지역 변수가 이미 사용되므로, 최대 8개의 지역 변수를 추가로 정의하여 활용할 수 있다.
최종적으로 완성한 코드는 지역변수 tmp0 - tmp7를 정의하고, 대각 블록을 전치할 때 A의 row 한 줄을 지역 변수 8개에 통째로 받아온다. 그리고 이렇게 저장한 값을 다시 B의 column에 통째로 쓴다. A의 row가 1줄씩 불러와질 때, 해당하는 B의 row가 다시 덮어쓰이지만, 이는 이전의 발생한 모든 줄에서의 충돌에 비하면 훨씬 개선된 결과이다.
최종적으로 달성한 점수는 아래와 같다:

```
func 0 (Transpose submission): hits:1765, misses:288, evictions:256
```

3-2. M=64, N=64

우선, 테스트를 위해 3-1과 동일하게 8x8 격자로 쪼갠 뒤, block 단위로 전치를 수행하였다. 결과는 다음과 같다:

```
func 0 (Transpose submission): hits:3585, misses:4612, evictions:4580
func 1 (Simple row-wise scan transpose): hits:3473, misses:4724, evictions:4692
```

기본적인 전치와 성능 차이가 거의 없는 것을 확인해볼 수 있다. 블록의 set index 범위를 나타내는 아래의 표를 통해 그 이유를 알 수 있다.

$0, 8, 16, 24$	$1, 9, 17, 25$	$2, 10, 18, 26$	$3, 11, 19, 27$	$\cdots$
$0, 8, 16, 24$	$1, 9, 17, 25$	$2, 10, 18, 26$	$3, 11, 19, 27$	$\cdots$
$0, 8, 16, 24$	$1, 9, 17, 25$	$2, 10, 18, 26$	$3, 11, 19, 27$	$\cdots$
$0, 8, 16, 24$	$1, 9, 17, 25$	$2, 10, 18, 26$	$3, 11, 19, 27$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

(0,1)-블록을 보자. 숫자 64가 공교롭게도 캐시 크기와 맞아떨어지기 때문에, 처음 4개 row와 마지막 4개 row의 set가 완전히 겹치게 된다. 이는 B에서 값을 적을 때, column-wise한 현재의 구현을 그대로 따라갈 경우, B의 짝지어진 row들이 서로 겹치게 되면서 A에서 B로 값을 옮길 때 큰 conflict가 발생하게 된다.
가장 우선적으로 고려한 것은 블록 크기를 4x4로 바꾸는 것이다. 이렇게 될 경우, B에 값이 들어갈 때 충돌이 덜 발생하게 된다. 결과는 다음과 같다:

```
func 0 (Transpose submission): hits:6401, misses:1796, evictions:1764
```

이렇게만 수정해도 miss는 크게 감소한다. 그러나, 과제에서 요구하는 기준은 1300 미만이기 때문에 추가적인 최적화가 필요하다는 것을 알 수 있다. 지금 상태에서 가장 큰 문제점은 캐시 블록 크기가 int를 8개 저장할 수 있는 반면 블록 크기는 4로 설정되어 있다는 점이다. 행렬 A의 경우, 다음 블록을 가져와도 이전 블록이 캐시에 들어가있는 경우 그대로 쓰인다. 달리 말해, A는 저장 방식이 row-major이고 읽는 방식 또한 row-major이기 때문에 좋은 locality를 갖게 된다. 반면, 행렬 B는 전치된 형태로 저장해야 하기 때문에, column 1줄을 쓰려면 각 row를 계속 새로 가져와야 하고, 이 파트는 추가적으로 최적화가 가능하기 때문에 이 부분을 수정해야 함을 알 수 있다.
핵심은 블록을 다시 가로세로 2등분씩 총 4등분하여 submatrix를 생각하는 것이다. 우선, 블록 내 왼쪽 위 부분은 기존의 방식으로 전치 후 B로 옮긴다. 이는 왼쪽 위 부분 행렬을 마치 처음부터 블록 크기가 4x4인 것처럼 생각하면 기존과 동일한 방법으로 해결이 가능함을 알 수 있다. A의 오른쪽 위 부분은 전치를 하되, 다시 B의 오른쪽 위에 저장한다. 이렇게 하면 A와 B의 위쪽 4개 row들만 캐시에서 가져왔기 때문에 어떠한 충돌도 발생하지 않는다. 또한, B에서 오른쪽 위의 값들은 row-wise하게 저장되었기 때문에 다시 값을 옮길 때 충돌이 적게 발생하게 된다.
그 다음에는, B의 오른쪽 위 부분을 왼쪽 아래 부분으로 옮기면서 동시에 A의 왼쪽 아래 부분을 B의 오른쪽 위 부분으로 옮긴다. A는 아래쪽 4개의 row가 한번만 불러와진 후에 변경되는 것이 없으며, B의 경우 오른쪽 위와 왼쪽 아래를 가져올 때 충돌이 발생한다. 이때, 값을 중간에 지역 변수에 저장함으로써 한 줄 당 1번씩 총 4번의 충돌만 발생하게 한다.
마지막으로는, A의 오른쪽 아래를 전치해서 B로 넣는다. A는 위 단계에서 이미 불러와졌기 때문에 충돌이 발생하지 않고, B 또한 위의 각 줄 당 충돌에서 충돌이 끝나면 아래쪽 줄만 남기 때문에 충돌이 발생하지 않는다.
위의 방법을 순서로 수행하면 매우 최적화된 행렬 전치 코드를 얻을 수 있다. 3-1처럼 대각 성분을 별도로 최적화하지는 않았지만, 여전히 기준을 만족하는 결과를 얻을 수 있기 때문에 최적화를 더 수행할 필요는 없다. 최종적으로 달성한 점수는 아래와 같다:

```
func 0 (Transpose submission): hits:8953, misses:1292, evictions:1260
```

3-3. M=61, N=67

M과 N이 2의 거듭제곱 꼴이 아니기 때문에 캐시에서 데이터가 겹치는 부분이 그리 크지 않다. 이에, 결론적으로만 말하면 3-1과 정확히 같은 방식으로 풀 수 있다. 여기서는 심지어 대각선 부분도 크게 겹치지 않는다. 따라서 위의 문제들처럼 복잡한 알고리즘을 고려할 필요가 없다. 블록 크기를 8로 설정하면 다음과 같은 결과를 얻는다.

```
func 0 (Transpose submission): hits:6060, misses:2119, evictions:2087
```

과제가 요구하는 기준은 2000 미만이기 때문에, 약간의 최적화를 더 할 필요는 있다. 블록 크기를 9로 설정하면 miss가 2093로 줄어든다. 여기서 힌트를 얻어 블록 크기를 하나씩 늘려가며 기준인 2000 이내로 들어오는 값이 존재하는지 찾았다. 블록 크기를 14로 설정하면 기준에 부합하는 값을 찾을 수 있음을 알게 되었다.
블록의 크기가 너무 작으면 locality를 덜 활용하며, 반면에 블록 크기가 너무 크면 블록 내에서 사용하는 캐시 line이 많아져서 블록 내 충돌이 발생할 수 있음은 쉽게 추론해볼 수 있다. 따라서 문제를 풀기 위해서는 적절한 균형점이 필요한데, 블록 크기 14이 이러한 균형점에 해당하는 것으로 생각된다. 최종적으로 달성한 점수는 아래와 같다:

```
func 0 (Transpose submission): hits:6182, misses:1997, evictions:1965
```

4. Result

```
[hyunseong@programming2 cache]$ python 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         287
Trans perf 64x64           8.0         8        1291
Trans perf 61x67          10.0        10        1996
          Total points    53.0        53
[hyunseong@programming2 cache]$
```

위의 스크린샷은 포스텍 programming2 서버에서 driver.py를 실행한 결과를 나타낸다. 이 보고서를 빌어 어떠한 규칙도 어기지 않고 모든 문제를 적법한 방법으로 해결하였음을 알린다.

5. References

[1] getopt: `https://man7.org/linux/man-pages/man3/getopt.3.html`