2023 Fall CSED 211 Lab Report

Lab: Cache lab

Student number: 20220302

Name: 김지현

1. Introduction

- A cache simulator is a program or tool that models the behavior of a cache memory system. Its primary purpose is to analyze and predict the performance of a cache in terms of hit rate, miss rate, and other metrics.
- Cache optimization involves techniques and strategies to improve the performance of a cache memory system. The goal of cache optimization is to minimize the miss rate.

2. System Design/ Algorithm

1. Part A - cache simulator

In part A of the cache lab, we will display the number of hits, misses, and evictions. From the write-up (page 7), we'll use "get opt" function to parse the command line arguments. We will use the following header files.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <getopt.h>
#include "cachelab.h"
```

Figure 1 header files

First, I made 3 structures: lineE, setS, and Caches. First, lineE structure contains a Boolean variable called "valid" which will function as a valid bit, and two integer variables " tag" and "frequency." "tag" functions as a tag bit required for the search procedure. "frequency" of other remaining cache lines decreases every time a certain cache line x is referred.

```
typedef struct {
  bool valid;
  int tag;
  int frequency;
} lineE;
```

Figure 2 lineE structure

Next, setS structure contains a pointer lineE called “lines.” As a result, the structure has similar structure as Fig

```
typedef struct {
  lineE* lines;
} setS;
```

Figure 3 setS structure

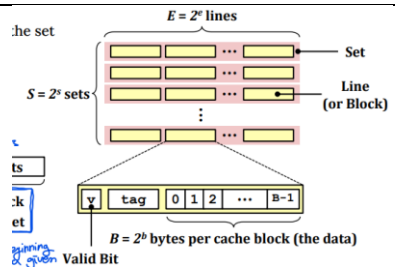

Figure 4 Diagram for set and lines

Lastly, the “Caches” structure contains the sets, the number of sets, and the number of lines. Size_t data type is used for the size of something in memory. In this case, we are measuring the size of sets and lines within the cache memory. Therefore, use size_t data type.

```
typedef struct {
  setS* sets;
  size_t setNum;
  size_t lineNum;
} Caches;
```

Figure 5 cache structure

Globally, I declared the cache variable, two integer variables – set bit and block bits, and 3 size_t variables – hits, misses, and evictions. “setBits” and “blockBits” count the set index and block offset. “hits”, “misses”, and “evictions” count for the numbers that will be printed.

```
Caches cache = {};
int setBits = 0, blockBits = 0;
size_t hits = 0, misses = 0, evictions = 0;
```

Figure 6 Global variables

There will be 3 functions: simulate, update, and main. “simulate” function is used for searching the cache lines. “update” function is used to update the frequency variable in the lineE structure. Finally, in the main function, the program parses the command line arguments.

```
void simulate(int addr);
void update(setS *set, size_t line_no);

int main(int argc, char *argv[]) {
```

Figure 7 Functions for Part A

“getopt” is a C library function that is used to analyze the command-line option. Command line options are the strings starting with ‘-‘ in Linux. Optstring is the string that represents the option character.

In the main function, two parameters are passed – argc and argv. An integer variable “argc” is the number of strings pointed to by the pointer array of “argv”. When passed onto the getopt function, the function accepts the options s, E, b, and t with additional arguments for each. For example, the command line argument will look like “… -s 1 -E 1 -b 1 -t traces/yi2.trace.”

First, declare the File pointer called “inputfile”. Then, in the for loop, run the getopt function. Optarg is set to point option argument when the getopt function analyzes the option that contains option argument. Using the switch case, change the values for “setBits”, “lineNum” for cache, “blockBits” for each input ‘s’, ‘E’, and ‘b.’ Specifically, when assigning the “setBits” variable, also assign the “setNum” of cache into $2^{setBits}$ using the left shift. Then, for the input ‘t’, open the trace file and save it into inputfile. Otherwise, return 1.

Finally, check whether any of the above values (“setBits”, “lineNum” for cache, “blockBits”, and inputfile”) are empty. If so, return 1.

Then, using the ‘malloc’ function, dynamically allocate the sets and lines. Then, read the inputfile using fscanf function and save the option into a character variable “kind” and the options argument into an integer variable “addr.”

For each data load(L) and store(S), run the simulate function once. For data modify(M), run the simulate function twice because you have to read and write. And ignore any cache access instructions (I).

Lastly, print the result using the ‘printSummary’ function (pass the number of hits, misses, evictions). Then, close the inputfile and deallocate the cache using the ‘free’ function.

```c
int main(int argc, char *argv[]) {

    FILE* inputfile = 0;

    for (int opt; (opt = getopt(argc, argv, "s:E:b:t:")) != -1;) {
        switch (opt) {
        case 's':
            setBits = atoi(optarg);
            cache.setNum = 2 << setBits;
            break;
        case 'E':
            cache.lineNum = atoi(optarg);
            break;
        case 'b':
            blockBits = atoi(optarg);
            break;
        case 't':
            if (!(inputfile = fopen(optarg, "r"))) { return 1; }
            break;
        default:
            return 1;
        }
    }

    if (!setBits || !cache.lineNum || !blockBits || !inputfile){
        return 1;
    }
```

```c
    cache.sets = malloc(sizeof(setS) * cache.setNum);
    for (int i = 0; i < cache.setNum; i++) {
        cache.sets[i].lines = calloc(sizeof(lineE), cache.lineNum);
    }

    char kind;
    int addr;

    while (fscanf(inputfile, " %c %x%*c%*d", &kind, &addr) != EOF) {
        if (kind == 'I') {
            continue;
        }

        simulate(addr);

        if (kind == 'M') {
            simulate(addr);
        }
    }

  printSummary(hits, misses, evictions);

  fclose(inputfile);

  for (size_t i = 0; i < cache.setNum; ++i){
    free(cache.sets[i].lines);
  }
  free(cache.sets);

  return 0;
}
```

Figure 8 Main function

Next, I'll explain the 'simulate' function. This is the function that distinguishes whether the hit, miss, or eviction occurs when the cache is accessed by the tag and set index values. If there is a line in the set of the set index of the address value whose tag value is the same as the tag value of the address, and the line is valid, then the "hit" increases. If no such line exists, then the "miss" increases.

If the lines are all valid but the miss occurs, then it means that the memory should be replaced. Therefore, the number of evictions will increase. When the eviction occurs, the line where the "frequency" value becomes the smallest, which will be 0, is found in the set and the value is replaced in "update" function. In all cases, when accessing the line, the "frequency" value of the line is updated through the "update" function. This is to use the LRU (least recently used) replacement policy when choosing which cache line to evict.

```
void simulate(int addr) {

    size_t setIndex = (0x7fffffff >> (31 - setBits)) & (addr >> blockBits);
    int tag = 0xffffffff & (addr >> (setBits + blockBits));

    setS* set = &cache.sets[setIndex];

    for (size_t i = 0; i < cache.lineNum; i++) {
        lineE* line = &set->lines[i];

        if (!line->valid){
            continue;
        }

        if (line->tag != tag){
            continue;
        }

        hits++;
        update(set, i);
        return;
    }

    ++misses;
```

```
    for (size_t i = 0; i < cache.lineNum; i++) {
        lineE* line = &set->lines[i];

        if (line->valid) { continue; }

        line->valid = true;
        line->tag = tag;
        update(set, i);
        return;
    }

    evictions++;

    for (size_t i = 0; i < cache.lineNum; ++i) {
        lineE* line = &set->lines[i];

        if (line->frequency) { continue; }

        line->valid = true;
        line->tag = tag;
        update(set, i);
        return;
    }
}
```

Figure 9 simulate function

Finally, the 'update' function is used to update the "frequency" value for LRU policy. LRU policy is when all LRU values are initialized to 0, and the larger the LRU value is, the more recently the memory is used. Therefore, I saved the number of lines minus 1 to the "frequency" value so that there will be only one line with "frequency" value equals to 0.

```
void update(setS *set, size_t L) {
  lineE *line = &set->lines[L];

  for (size_t i = 0; i < cache.lineNum; i++) {
    lineE *newLine = &set->lines[i];
    if (!newLine->valid){
        continue;
    }

    if (newLine->frequency <= line->frequency){
        continue;
    }

    (newLine -> frequency)--;
  }

  line->frequency = cache.lineNum - 1;
}
```

Figure 10 update function

2. Part B – Optimization

For the part B if the cache lab, I added these three functions for each cases of blocking: 32 X 32, 64 X 64, and 61 X 67.

```
void transpose_32_32(int M, int N, int A[N][M], int B[M][N]);
void transpose_64_64(int M, int N, int A[N][M], int B[M][N]);
void transpose_61_67(int M, int N, int A[N][M], int B[M][N]);
```

Figure 11 Three user defined functions

For the "transpose_submit" function, I divided up into three cases for the above three cases and directed each case into the user-defined function.

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    if (N == 32){
        transpose_32_32(M, N, A, B);
    }
    else if (N == 64){
        transpose_64_64(M, N, A, B);
    }else{
        transpose_61_67(M, N, A, B);
    }
}
```

Figure 12 transpose submit function

For 32 X 32 blocking, it divides into 8 X 8 sections, and the eviction occurs when the A[i][j] equals B[j][i]. Therefore, I separated the case when A[i][j] equals B[j][i] to minimize the miss.

```
/* 32 × 32 */
char trans_3232_desc[] = "A 32 X 32 transpose";
void transpose_32_32(int M, int N, int A[N][M], int B[M][N])
{
    int col, row, i;
    int t1, t2, t3, t4, t5, t6, t7, t8;
    for (col = 0; col < N; col += 8) {
        for (row = 0; row < M; row += 8) {
            for (i = col; i < col + 8; i++)
            {
                t1 = A[i][row];
                t2 = A[i][row + 1];
                t3 = A[i][row + 2];
                t4 = A[i][row + 3];
                t5 = A[i][row + 4];
                t6 = A[i][row + 5];
                t7 = A[i][row + 6];
                t8 = A[i][row + 7];

                B[row][i] = t1;
                B[row + 1][i] = t2;
                B[row + 2][i] = t3;
                B[row + 3][i] = t4;
                B[row + 4][i] = t5;
                B[row + 5][i] = t6;
                B[row + 6][i] = t7;
                B[row + 7][i] = t8;
            }
        }
    }
}
```

Figure 13 Case for 32 X 32

For 64 X 64 blocking, it is difficult to optimize as the way I used in 32 X 32 blocking. Therefore, first, separate the entire matrix into 8 X 8 parts, then the upper half and the lower half (each will be 8 X 4 parts) share the same set index. Therefore, these two must be handled separately.

```
/* 64 × 64 */
char trans_6464_desc[] = "A 64 X 64 transpose";
void transpose_64_64(int M, int N, int A[N][M], int B[M][N])
{
    int col, row, i, j;
    int t1, t2, t3, t4, t5, t6, t7, t8;
    for (col = 0; col < N; col += 8) {
        for (row = 0; row < M; row += 8) {
            for (i = col; i < col + 4; i++) {

                t1 = A[i][row];
                t2 = A[i][row + 1];
                t3 = A[i][row + 2];
                t4 = A[i][row + 3];
                B[row][i] = t1;
                B[row + 1][i] = t2;
                B[row + 2][i] = t3;
                B[row + 3][i] = t4;

                t5 = A[i][row + 4];
                t6 = A[i][row + 5];
                t7 = A[i][row + 6];
                t8 = A[i][row + 7];
                B[row][i + 4] = t5;
                B[row + 1][i + 4] = t6;
                B[row + 2][i + 4] = t7;
                B[row + 3][i + 4] = t8;
            }

            for (j = row; j < row + 4; j++) {
                t1 = A[col + 4][j];
                t2 = A[col + 5][j];
                t3 = A[col + 6][j];
                t4 = A[col + 7][j];

                t5 = B[j][col + 4];
                t6 = B[j][col + 5];
                t7 = B[j][col + 6];
                t8 = B[j][col + 7];

                B[j][col + 4] = t1;
                B[j][col + 5] = t2;
                B[j][col + 6] = t3;
                B[j][col + 7] = t4;

                B[j + 4][col] = t5;
                B[j + 4][col + 1] = t6;
                B[j + 4][col + 2] = t7;
                B[j + 4][col + 3] = t8;

                t1 = A[col + 4][j + 4];
                t2 = A[col + 5][j + 4];
                t3 = A[col + 6][j + 4];
                t4 = A[col + 7][j + 4];
                B[j + 4][col + 4] = t1;
                B[j + 4][col + 5] = t2;
                B[j + 4][col + 6] = t3;
                B[j + 4][col + 7] = t4;
            }
        }
    }
}
```

Figure 14 Case for 64 X 64

For 61 X 67 blocking, it is like 32 X 32 blocking, but the order is changed. First, I divide the matrix into 16 X 16 parts and for each part, check whether the element is located on the diagonal. If the element is not on the diagonal, then it is directly copies to the corresponding location in B. If it is, its value and position are stored in the diag_val and diag_pos variable. Finally, after all the elements in each block have been processed, if the block is located on the

diagonal of the matrix, the diagonal element value stored in the diag_val and diag_pos variables is copied to the corresponding location in B. This prevents the diagonal element values from being copied incorrectly.

```
/* 61 × 67 */
char trans_6167_desc[] = "A 61 X 67 transpose";
void transpose_61_67(int M, int N, int A[N][M], int B[M][N])
{
    int col, row, i, j;
    for (col = 0; col < N; col += 8) {
        for (row = 0; row < M; row += 8) {
            for (j = row; j < row + 8 && j < M; j++) {
                for (i = col; i < col + 8 && i < N; i++)
                    B[j][i] = A[i][j];
            }
        }
    }
}
```

Figure 15 Case for 61 X 67

Lastly, I added the above function into the “register” function.

```
void registerFunctions()
{
    /* Register your solution function */
    registerTransFunction(transpose_submit, transpose_submit_desc);

    /* Register any additional transpose functions */
    registerTransFunction(trans, trans_desc);

    //new
    registerTransFunction(transpose_32_32, trans_3232_desc);
    registerTransFunction(transpose_64_64, trans_6464_desc);
    registerTransFunction(transpose_61_67, trans_6167_desc);

}
```

Figure 16 Register function

The overall outcome becomes like this:

```
[jihyunk@programming2 cachelab-handout]$ 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         287
Trans perf 64x64           8.0         8        1235
Trans perf 61x67          10.0        10        1931
          Total points    53.0        53
[jihyunk@programming2 cachelab-handout]$
```

Figure 17 Overall points earned