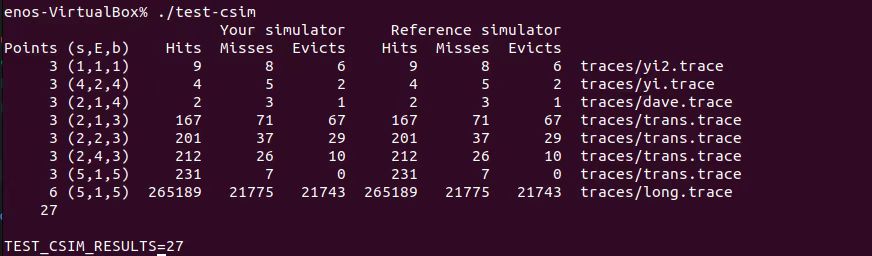
**Cache Lab**

1. **Part A Writing a Cache Simulator**

**Result:**

****

**Core Process:**

The core process of the cache simulator:

 1. Get the tag bits and the set index bits from the address. (The block offset bits can be ignored here in that it makes no difference in the simulating process)

 2. Get the reference of the indicated LRU set via the set index bits and find the specific line via the tag bits.

 3. If the indicated line can be found in the specific LRU set which means a **hit** from the cache, then increments the hit count and insert this line to the headmost position of its LRU set according the LRU strategy.

 4. If the indicated line cannot be found in the specific LRU set which means a **miss** from the cache, then increments the miss count.

Next, create a new line with the address(this simulates the process of loading a block from the memory) and insert it to the headmost position of its LRU set if the set is not full.

If the set is full which means we need to **evict** a block(increment the eviction count), we remove the backmost line of the set which indicates the least frequently used line according to the LRU strategy and insert the new line to the headmost position.

**Source Code:**

#define \_GNU\_SOURCE

#include "cachelab.h"

#include <getopt.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_FILENAME\_LENGTH 100

static int s; // Number of set index bits(S = 2 ^ s is the number of sets)

static int E; // Associativity(number of lines per set)

static int b; // Number of block bits(B = 2 ^ b is the block size)

static int S; // Number of sets

static int hit\_count = 0;

static int miss\_count = 0;

static int eviction\_count = 0;

static char\* trace\_file\_directory; // valgrind trace file directory

// A double linkedList node to simulate a line in a set

typedef struct Line {

    unsigned tag;       // Tag bits

    unsigned set\_index; // Set index bits(actually it makes no difference)

    struct Line\* prev;  // Pointer to the previous line

    struct Line\* next;  // Pointer to the next line

} Line;

// A double linkedList to simulate and arrange a set in the cache

typedef struct LRU {

    Line\* head;   // Dummy Head pointer of a LRU set

    Line\* tail;   // Dummy Tail pointer of a LRU set

    int size;     // The size of a LRU set(associativity / number of lines per set)

} LRU;

// The whole cache

static LRU\* cache;

void parseCommandLineArguments(int argc, char\*\* argv);

void intializeCache();

void insertAtHead(LRU\* current\_LRU, Line\* new\_line);

void delete(LRU\* current\_LRU, Line\* line\_to\_be\_deleted);

Line\* findByTag(LRU\* current\_LRU, unsigned target\_tag);

void update(unsigned address);

void simulate();

int main(int argc, char\*\* argv)

{

    parseCommandLineArguments(argc, argv);

    intializeCache();

    simulate();

    printSummary(hit\_count, miss\_count, eviction\_count);

    return 0;

}

/\*\*

 \* @brief Parsing command line arguments to get s(number of set index bits), E(number of lines per set),

 \* b(number of blocks bits) and trace file directory via getopt function.

 \* @param argc argument count

 \* @param argv argument vector(parameter list)

 \*/

void parseCommandLineArguments(int argc, char\*\* argv) {

    int option;

    trace\_file\_directory = (char\*)malloc(MAX\_FILENAME\_LENGTH \* sizeof(char));

    while ((option = getopt(argc, argv, "s:E:b:t:")) != -1) {

        switch(option) {

        case 's': s = atoi(optarg);

        case 'E': E = atoi(optarg);

        case 'b': b = atoi(optarg);

        case 't': strcpy(trace\_file\_directory, optarg);

        }

    }

    S = 1 << s;

}

/\*\*

 \* @brief Set each set in the cache an empty double linked list.

 \*/

void intializeCache() {

    cache = (LRU\*)malloc(S \* sizeof(LRU));

    for (int set\_index = 0; set\_index < S; set\_index++) {

        LRU\* current\_LRU = &cache[set\_index];

        // Assign memory

        current\_LRU->head = (Line\*)malloc(sizeof(Line));

        current\_LRU->tail = (Line\*)malloc(sizeof(Line));

        current\_LRU->head->next = current\_LRU->tail;

        current\_LRU->tail->prev = current\_LRU->head;

        current\_LRU->head->prev = NULL;

        current\_LRU->tail->next = NULL;

        cache[set\_index].size = 0;

    }

}

/\*\*

 \* @brief Insert a specific line to a LRU set.

 \* @param current\_LRU The LRU set to insert a line

 \* @param new\_line  The line to be inserted

 \*/

void insertAtHead(LRU\* current\_LRU, Line\* new\_line) {

    new\_line->next = current\_LRU->head->next;

    new\_line->prev = current\_LRU->head;

    current\_LRU->head->next->prev = new\_line;

    current\_LRU->head->next       = new\_line;

    current\_LRU->size++;

}

/\*\*

 \* @brief Delete(evict) a specific line in a LRU set.

 \* @param current\_LRU  The LRU set to be manipulated

 \* @param line\_to\_be\_deleted  The line to to deleted

 \*/

void delete(LRU\* current\_LRU, Line\* line\_to\_be\_deleted) {

    line\_to\_be\_deleted->prev->next = line\_to\_be\_deleted->next;

    line\_to\_be\_deleted->next->prev = line\_to\_be\_deleted->prev;

    current\_LRU->size--;

}

/\*\*

 \* @brief Find the indicated line in a LRU set via the tag bit.

 \* @param current\_LRU The LRU line to perform the search

 \* @param target\_tag  The tag bits of the line to be found

 \* @return Line\* NULL if the line cannot be found

 \*/

Line\* findByTag(LRU\* current\_LRU, unsigned target\_tag) {

    Line\* current\_line = current\_LRU->head->next;

    while (current\_line != current\_LRU->tail) {

        if (current\_line->tag == target\_tag) {

            return current\_line;

        }

        current\_line = current\_line->next;

    }

    return NULL;

}

/\*\*

 \* @brief The core process of the cache simulator.

 \* 1. Get the tag bits and the set index bits from the address.

 \* (The block bias bits can be ignored here in that it makes no difference in the \* simulating process)

 \* 2. Get the reference of the indicated LRU set via the set index bits and find \* \* the specific line via

 \* the tag bits.

 \* 3. If the indicated line can be found in the specific LRU set which means a hit \* from the cache, then

 \* increments the hit\_count and insert this line to the headmost position of its \* \* LRU set according the

 \* LRU strategy.

 \* 4. If the indicated line cannot be found in the specific LRU set which means a \* \* miss from the cache,

 \* then increments the miss\_count.

 \*    Next, create a new line with the address(this simulates the process of loading \* a block from the

 \* memory) and insert it to the headmost position of its LRU set if the set is not full.

 \*    If the set is full which means we need to evict a block(increment the eviction\_count), we remove

 \* the backmost line of the set which indicates the least frequently used line \* \* according to the LRU

 \* strategy and insert the new line to the headmost position.

 \* @param address The address of a trace of its memory access

 \*/

void update(unsigned address) {

    unsigned tag = address >> (s + b);

    unsigned set\_index = (address >> b) & (0xFFFFFFFF >> (32 - s));

    LRU\* current\_LRU = &cache[set\_index];

    Line\* target\_line = findByTag(current\_LRU, tag);

    if (target\_line != NULL) {

        // hit

        hit\_count++;

        // If the target line is already the headmost line in the set, it do not need to be moved.

        if (current\_LRU->head->next != target\_line) {

            delete(current\_LRU, target\_line);

            insertAtHead(current\_LRU, target\_line);

        }

    }

    else {

        // miss

        miss\_count++;

        Line\* new\_line = (Line\*)malloc(sizeof(Line));

        new\_line->next = NULL;

        new\_line->prev = NULL;

        new\_line->tag = tag;

        new\_line->set\_index = set\_index;

        if (current\_LRU->size == E) {

            // evict

            delete(current\_LRU, current\_LRU->tail->prev);

            eviction\_count++;

        }

        insertAtHead(current\_LRU, new\_line);

    }

}

/\*\*

 \* @brief Simulation process.

 \*/

void simulate() {

    FILE\* fp = fopen(trace\_file\_directory, "r");

    char operation;

    unsigned address;

    int size;

    while(fscanf(fp, "%c %x, %d", &operation, &address, &size) > 0) {

        switch (operation)

        {

        case 'L': update(address); break;

        case 'M': update(address);

        case 'S': update(address); break;

        }

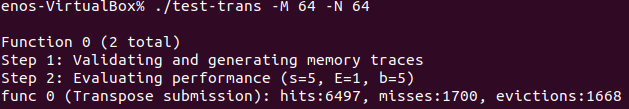
    }

}

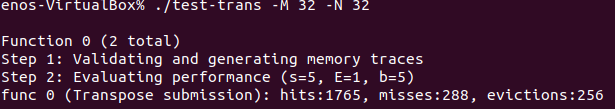
**Part B: Optimizing Matrix Transpose**

**Result:**

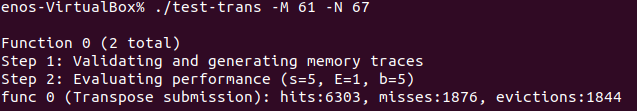
32 \* 32

****

64 \* 64

****

61 \* 67



**Blocking:**

1. Since the cache can hold 8 ints in a row, it is best to use a multiple of 8 for blocking as well. In a 32 x 32 matrix, there are 32 ints in one row, that is, 4 cache rows, so the cache can hold a total of 8 rows of the matrix. You can just use the 8's and the 8's and there's no conflict.
2. The result of 8 blocks blocking is not ideals. There is no alternative but to try to use 4 blocks, although it may not be efficient to use the cache.
3. 8 blocks blocking but need to handle the remaining part separately.

**Source Code:**

char transpose\_submit\_desc[] = "Transpose submission";

void transpose\_submit(int M, int N, int A[N][M], int B[M][N])

{

    int i, j, bi, t0, t1, t2, t3, t4, t5, t6, t7;

    if (M == 32 && N == 32) {

        for (i = 0; i < N; i += 8) {

            for (j = 0; j < M; j += 8) {

                for (bi = i; bi < i + 8; bi++) {

                    t0 = A[bi][0 + j];

                    t1 = A[bi][1 + j];

                    t2 = A[bi][2 + j];

                    t3 = A[bi][3 + j];

                    t4 = A[bi][4 + j];

                    t5 = A[bi][5 + j];

                    t6 = A[bi][6 + j];

                    t7 = A[bi][7 + j];

                    B[0 + j][bi] = t0;

                    B[1 + j][bi] = t1;

                    B[2 + j][bi] = t2;

                    B[3 + j][bi] = t3;

                    B[4 + j][bi] = t4;

                    B[5 + j][bi] = t5;

                    B[6 + j][bi] = t6;

                    B[7 + j][bi] = t7;

                }

            }

        }

    } else if (M == 64 && N == 64) {

        int i, j, bi, t0, t1, t2, t3;

        for (i = 0; i < N; i += 4) {

            for (j = 0; j < M; j+= 4) {

                for (bi = i; bi < i + 4; bi++) {

                    t0 = A[bi][0 + j];

                    t1 = A[bi][1 + j];

                    t2 = A[bi][2 + j];

                    t3 = A[bi][3 + j];

                    B[0 + j][bi] = t0;

                    B[1 + j][bi] = t1;

                    B[2 + j][bi] = t2;

                    B[3 + j][bi] = t3;

                }

            }

        }

    } else if (M == 61 && N == 67) {

        int n = N / 8 \* 8;

        int m = M / 8 \* 8;

        int i, j, t0, t1, t2, t3, t4, t5, t6, t7;

        for (j = 0; j < m; j += 8) {

            for (i = 0; i < n; i++) {

                t0 = A[i][0 + j];

                t1 = A[i][1 + j];

                t2 = A[i][2 + j];

                t3 = A[i][3 + j];

                t4 = A[i][4 + j];

                t5 = A[i][5 + j];

                t6 = A[i][6 + j];

                t7 = A[i][7 + j];

                B[0 + j][i] = t0;

                B[1 + j][i] = t1;

                B[2 + j][i] = t2;

                B[3 + j][i] = t3;

                B[4 + j][i] = t4;

                B[5 + j][i] = t5;

                B[6 + j][i] = t6;

                B[7 + j][i] = t7;

            }

        }

        for (i = n; i < N; i++) {

            for (j = m; j < M; j++) {

                t0 = A[i][j];

                B[j][i] = t0;

            }

        }

        for (i = 0; i < n; i++) {

            for (j = m; j < M; j++) {

                t0 = A[i][j];

                B[j][i] = t0;

            }

        }

        for (i = n; i < N; i++) {

            for (j = 0; j < m; j++) {

                t0 = A[i][j];

                B[j][i] = t0;

            }

        }

    }

}