МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №8

по курсу «ЭВМ и периферийные устройства»

ВЛИЯНИЕ КЭШ-ПАМЯТИ НА ВРЕМЯ ОБРАБОТКИ МАССИВОВ

Выполнил: студент 2-го курса гр. 17208

Гафиятуллин А.Р.

Новосибирск, 2018

1. ЦЕЛИ РАБОТЫ:

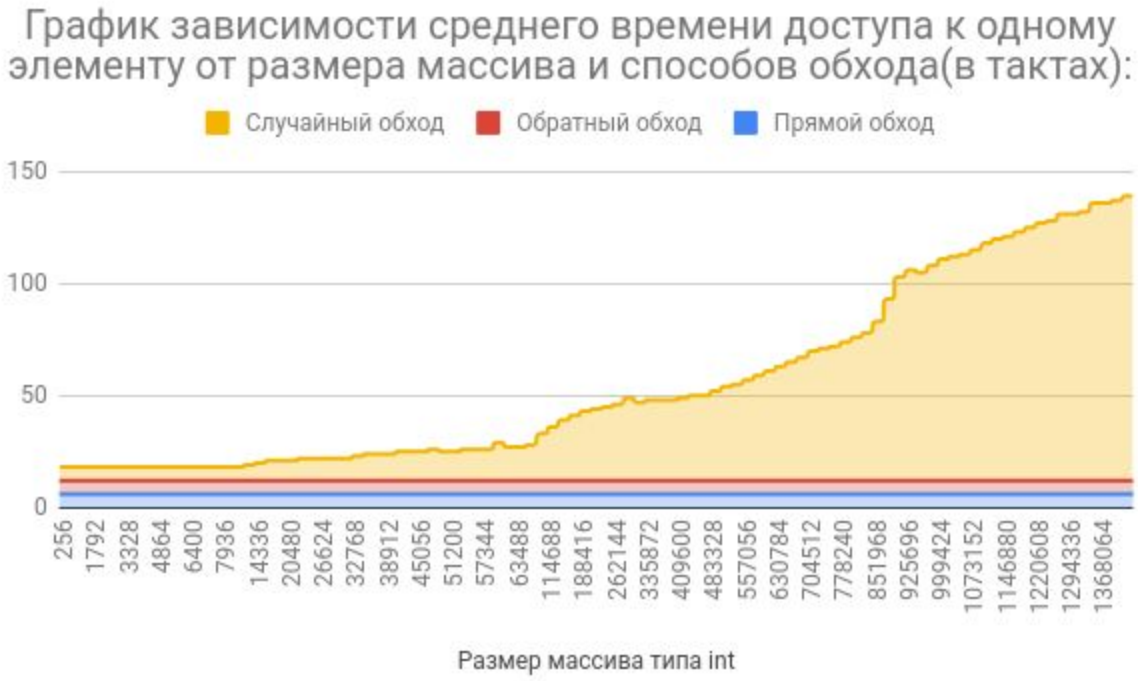
1. Исследование зависимости времени доступа к данным в памяти от их объема;
2. Исследование зависимости времени доступа к данным в памяти от порядка их обхода.

2. ХОД РАБОТЫ:

1. Для достижения поставленных целей была написана программа, многократно выполняющая обход массива заданного размера тремя различными способами:
    a. **прямой обход:** в каждой i-ой ячейке массива хранится индекс следующей (i + 1)-ой ячейки, а в последней ячейке нулевой индекс для зацикливания;
    b. **обратный обход:** в каждой i-ой ячейке массива хранится индекс предыдущей (i - 1)-ой ячейки, а в нулевой ячейке последний индекс для зацикливания;
    c. случайный обход: заполнение начинается с нулевой ячейки, в каждой i-ой ячейке массива хранится случайный индекс той ячейки, которая еще не использовалась при обходе на данном этапе заполнения массива(внутренний цикл ищет такую, что позволяет обходить все элементы массива в случайном порядке), а последняя незаполненная ячейка ссылается на нулевую ячейку для зацикливания.
2. Тестирование программы проходило на Linux-машине с Elementary OS(Ubuntu-based, Linux kernel 4.15.0-39-generic), процессор Intel Core i5-7200U CPU @3.1GHz 2 ядра(физических), загруженность - около 130-140 процессов.
3. **Информация о кэше процессора:**

a. **L1 cache:** 32 Кбайт для данных и 32 Кбайт для инструкций на одно ядро процессора;
b. **L2 cache:** 256 Кбайт на одно ядро процессора;
c. **L3 cache:** 3072 Кбайт объединенного кэша для всех ядер.

4. График зависимости среднего времени доступа к одному элементу от размера массива и способов обхода:



Впервые среднее время доступа к элементу при случайном обходе начало постоянно расти при размерах массива больших 8192 элементов, что соответствует 32 Кбайтам кэш-памяти данных 1 уровня.

Во второй раз резкий скачок среднего времени доступа к элементу массива произошел при размерах массива больших 65536 элементов, что соответствует 256 Кбайтам кэш-памяти 2 уровня.

В третий раз резкий скачок среднего времени доступа к элементу массива произошел при размерах массива больших 827392 элементов, что примерно соответствует 3072 Кбайтам кэш-памяти 3 уровня.

При размерах массива, больших кэш-памяти 1 уровня, часть данных вытесняется в кэш более низкого уровня. При случайном обходе массива часто возникают кэш-промахи контроллера при запросе процессором необходимых ему данных. Кэш-контроллер вынужден проверять наличие необходимых данных в более медленной кэш-памяти 2 уровня. Если они там есть, то запрос процессора удовлетворяется, а иначе(если размер массива больше кэш-памяти 2 уровня) кэш-контроллер ищет данные в еще более медленной кэш-памяти 3 уровня, а потом(если данных нет даже на 3 уровне) уже запрос передается в оперативную память. Этим объясняется отсутствие роста времени при случайном обходе, когда массив помещается в кэш-память 1-го уровня - контроллер не промахивается.

5. Полный компилируемый листинг реализованной программы и команда для ее компиляции:

```
#include <stdio.h>  //for printf
#include <stdlib.h> //for malloc
#include <time.h>   //for time
#include <memory.h> //for memset

#define SINGLE_CORE_LEVEL1_CACHE_SIZE 32    //KB
#define SINGLE_CORE_LEVEL2_CACHE_SIZE 256   //KB
#define LEVEL_3_CACHE_SIZE 3072 //KB
#define MAX_SIZE 10 * 1024 * 1024 / 4 //10 MB of int
#define CYCLES_AMOUNT 100

typedef unsigned long long ull;

inline ull rdtsc();

ull rdtsc(){
    unsigned int lo, hi;
```

```
    asm volatile("rdtsc\n" : "=a"(lo), "=d"(hi));
    return ((ull)hi << 32) | lo;
}

void init_direct_bypass(int *array, int size){
    for(int i = 0; i < size; i++)
        array[i] = (i + 1) % size;
}

void init_reverse_bypass(int *array, int size){
    for(int i = size - 1; i > 0; i--)
        array[i] = (i - 1) % size;
    array[0] = size - 1;
}

void init_random_bypass(int *array, int size){
    memset(array, -1, sizeof(int) * size);
    int current_elem = 0;
    for(int i = 0; i < size - 1; i++){
        int j = rand() % size;
        while(array[j] != -1 || current_elem == j)  //search for unused array
element
            j = rand() % size;
        array[current_elem] = j;
        current_elem = j;
    }
    array[current_elem] = 0;
}

ull test_access_speed(volatile int *array, int size){
    for(int j = 0, k = 0; j < size; j++)
        k = array[k];
    size *= CYCLES_AMOUNT;
    ull start = rdtsc();
```

```
    for(int j = 0, k = 0; j < size; j++)
        k = array[k];
    ull end = rdtsc();
    return  end - start;
}

int main(){
    int current_size = 256;   // 1 KB
    int *test_array;
    int step = 256;
    ull ticks_amount;
    for(; current_size < MAX_SIZE; current_size += step){
        srand(time(NULL));
        test_array = malloc(sizeof(int) * current_size);
        //test direct access
        init_direct_bypass(test_array, current_size);
        ticks_amount = test_access_speed(test_array, current_size);
        printf("Direct bypass, array size: %d, amount of ticks: %llu\n",
current_size,
               ticks_amount / (current_size * CYCLES_AMOUNT));
        //test reverse access
        init_reverse_bypass(test_array, current_size);
        ticks_amount = test_access_speed(test_array, current_size);
        printf("Reverse bypass, array size: %d, amount of ticks: %llu\n",
current_size,
               ticks_amount / (current_size * CYCLES_AMOUNT));
        //test random access
        init_random_bypass(test_array, current_size);
        ticks_amount = test_access_speed(test_array, current_size);
        printf("Random bypass, array size: %d, amount of ticks: %llu\n",
current_size,
                ticks_amount / (current_size * CYCLES_AMOUNT));

printf("-----------------------------------------------------------------
```

```
----------\n");
        if(current_size == SINGLE_CORE_LEVEL1_CACHE_SIZE * 1024 / 4){
            step *= SINGLE_CORE_LEVEL2_CACHE_SIZE /
SINGLE_CORE_LEVEL1_CACHE_SIZE;
            printf("LEVEL 2:\n");
        }
        else if(current_size == SINGLE_CORE_LEVEL2_CACHE_SIZE * 1024 / 4){
            step *= LEVEL_3_CACHE_SIZE / SINGLE_CORE_LEVEL2_CACHE_SIZE;
            printf("LEVEL 3:\n");
        }
        free(test_array);
    }
    return 0;
}
```

Команда компиляции: gcc -O1 cache.c -o cache

3. ВЫВОДЫ:

1. Исследовали зависимость времени доступа к данным в памяти от их объема;
2. Исследовали зависимость времени доступа к данным в памяти от порядка их обхода;
3. Кэш-память имеет многоуровневую архитектуру, в которой каждый следующий уровень обладает бОльшим количеством более медленной памяти;
4. Кэш-контроллер в некоторых случаях умеет предсказывать обход данных и организовывать оптимальный по времени доступ к ним;
5. Программы, обрабатывающие большие массивы данных, могут работать гораздо быстрее, если учитывать особенности кэш-памяти и стараться повысить шанс предсказания кэш-контроллером порядка их обхода.