

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ**  
**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**  
**НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**  
**Факультет информационных технологий**  
**Кафедра параллельных вычислений**

**ОТЧЕТ**

**О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ**

«Влияние кэш-памяти на время обработки массивов»

студента 2 курса, группы 21206

**Балашова Вячеслава Вадимовича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:  
Кандидат технических наук  
А. Ю. Власенко

Новосибирск 2022

## Содержание

Цель.....	3
Задание.....	3
Описание работы .....	4
Заключение.....	7
Приложение 1. Исходный код функций из файла Function__GenerateArray.h.....	8
Приложение 2. Исходный код программы Direct.cpp, выполняющей прямой обход массива .....	10
Приложение 3. Исходный код программы Reverse.cpp, выполняющей обратный обход массива .....	12
Приложение 4. Исходный код программы Random.cpp, выполняющей случайный обход массива .....	14
Приложение 5. Вывод команды lscpu   grep “cache” в системе Manjaro Linux.....	16
Приложение 6. Вывод программы CPU-Z в системе Windows.....	17

## Цель

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.

## Задание

1. Написать программу, многократно выполняющую обход массива заданного размера тремя способами (прямой, обратный и случайный).
2. Для каждого размера массива и способа обхода измерить среднее время доступа к одному элементу (в тактах процессора). Построить графики зависимости среднего времени доступа от размера массива. Каждый последующий размер массива отличается от предыдущего **не более, чем в 1,2 раза**.
3. Определить размеры кэш-памяти точным образом (на основе документации по процессору используемой машины; утилите, отражающей характеристики процессора; системному файлу...)
4. На основе анализа полученных графиков:
  - оценить размеры кэш-памяти различных уровней, обосновать ответ, сопоставить результат с известными реальными значениями;
  - определить размеры массива, при которых время доступа к элементу массива при случайном обходе больше, чем при прямом или обратном; объяснить причины этой разницы во временах.
5. Составить отчет по лабораторной работе.

## Описание работы

Ход выполнения работы:

1. Для удобства сначала был создан файл `Function__GenerateArray.h` (См. Приложение 1.), уместивший в себя 4 полезные функции на языке C++: генераторы прямого, обратного и случайных обходов массива (Генераторы случайных обходов отличаются асимптотикой). Для случайных обходов также была написана функция проверки правильности обхода (обход должен состоять из одного цикла, содержащего все индексы массива).
  - а) Для прямого обхода в элемент массива с индексом  $i$  кладется номер следующего элемента  $i+1$ , а в последний элемент массива кладется номер первого элемента  $-0$ .
  - б) Для обратного обхода в элемент массива с индексом  $i$  кладется номер предыдущего элемента  $i-1$ , а в первый элемент массива кладется номер последнего элемента  $-size-1$ .
  - с) Для случайного обхода были созданы два алгоритма:
    - i. Создается ассоциативный контейнер `std::set`. В этот контейнер помещается индекс первого элемента  $-0$ , затем случайным образом выбирается число, меньшее размера исходного массива, не содержащееся в контейнере. Это число кладется в элемент массива с индексом  $0$  и затем помещается в контейнер. На следующей итерации выполняются действия, описанные выше, но для  $i$ -го элемента массива. Цикл работает до тех пор, пока в контейнере не будут храниться все элементы массива.
    - ii. Индукционный алгоритм:
      - 1) База индукции – массив из одного элемента, в котором нулевой элемент содержит число  $0$  – индекс самого себя.
      - 2) Шаг индукции – случайным образом выбирается элемент текущего массива, запоминается его индекс. Размер массива увеличивается на единицу. В последний элемент увеличенного массива записывается значение элемента по ранее полученному индексу, а в этот случайно выбранный элемент записывается индекс последнего элемента.

В итоговой версии был использован индукционный алгоритм.

2. Были созданы 3 файла с программами на языке C++, высчитывающие среднее количество тактов, требующееся для доступа к элементу массива, для каждого размера массива и каждого из трех обходов. После каждой итерации размер массива увеличивается в 1.1 раза (см. Приложения 2, 3 и 4).
3. На основе данных команды `lscpu` была получена информация о размере кэш-памяти процессора Ryzen 7 5700U (архитектура Zen 2). На основе этого были выбраны минимальный размер массива (100 элементов) и максимальный ( $5 \cdot 10^6$  элементов) (Вывод команды `lscpu` в приложении 6.).
4. Перед запуском каждой из 3-х программ выполнялся «прогрев» процессора с помощью алгоритма перемножения матриц. Таким образом устанавливалась стабильная тактовая частота процессора.
5. На основе выведенных программой данных был построен график зависимости среднего времени доступа к элементу массива в тактах в зависимости от обхода и размера массива (См. График 1.).



6. По результатам замеров были оценены размеры разных уровней кэш-памяти: первый прирост тактов при случайном обходе начинается при приблизительно 36 KiB. Следовательно, размер кэша уровня L1 равен 32 KiB. Следующий прирост наблюдается при приблизительно 528 KiB, из чего можно сделать вывод, что кэш уровня L2 равен 512 KiB. Следующий прирост наблюдается при 4305 KiB, что означает, что кэш уровня L3 равен 4 MiB.

7. В качестве дополнительного источника информации была использована утилита CPU-Z (См. Приложение 7).

## Заключение

В ходе выполнения практической работы было выполнено исследование зависимости времени доступа к данным в памяти в зависимости от обхода массива и его размера.

Были определены размеры кэш-памяти разного уровня с помощью анализа графика, построенного по результатам измерений, и утилит CPU-Z и lscpu для Windows и Manjaro Linux соответственно. Результаты первых двух способов совпали, а третий показал гораздо большие значения размера кэш-памяти.

Это получилось потому, что L1 и L2 кэш считаются на ядро, поэтому если мы умножим объем кэша на количество ядер процессора, то получим значения из утилиты lscpu. L3 кэш выделяется сразу на все ядра, но в архитектуре Zen 2 (на которой и построен тестируемый процессор) делится на количество потоков на ядре. Поэтому для кэша уровней L1 и L2 первые два способа показывают результат в 8 раз меньший, чем третий способ, а для L3 – в 2 раза меньший.

Также, исходя из измерений, можно сделать вывод, что прямой или обратный обход массива наиболее эффективны из-за аппаратной предвыборки кэша, что значительно ускоряет обход массива, размещая его кусками в кэше первого уровня, тем самым минимизируя количество кэш-промахов. Среднее время доступа к элементу массива при прямом и обратном обходах – 2.5 тактов.

При случайном обходе массива случается большое количество кэш-промахов, так как не удается полностью размещать массив в кэш памяти сначала первого, а затем второго и третьего уровней. Из-за этого наблюдается рост времени доступа к следующему элементу массива. Время доступа к памяти:

- L1 кэш – 2.5 Тактов.
- L2 кэш – 6-9 Тактов.
- L3 кэш – 10-30 Тактов.
- Оперативная память – более 150 тактов.

## Приложение 1. (Исходный код функций из файла Function\_\_GenerateArray.h)

```
#pragma once.
```

```
#include <iomanip>
#include <iostream>
#include <random>
#include <set>
#include <vector>
```

```
inline void GenerateDirectBypass(int * array, size_t size)
{
    for (int i = 0; i < size - 1; i++) array[i] = i + 1;
    array[size - 1] = 0;
}
```

```
inline void GenerateReverseBypass(int * array, size_t size)
{
    for (int i = 1; i < size; i++) array[i] = i - 1;
    array[0] = int(size - 1);
}
```

```
inline bool CheckPermutation(const int * array, size_t size, bool print = true)
{
    if (print)
    {
        std::cout << std::left << std::setw(7) << 0 << ' ';
    }
    int k = 0;
    k = array[k];
    size_t cnt = 0;
    while (k)
    {
        if (print)
        {
            std::cout << std::left << std::setw(7) << k << ' ';
        }
        k = array[k];
        cnt++;
    }
    if (print)
    {
        std::cout << "\nPermutation status (true = correct): ";
        std::cout << std::boolalpha << (cnt == size - 1) << std::endl;
    }
    return cnt == size - 1;
}
```



```

inline void GenerateRandomBypass(int * array, size_t size)
{
    std::set<int> set;
    int j;
    set.insert(j = 0);
    for (int i = 0; i < size - 1; i++)
    {
        int k;
        while (set.contains(k = static_cast<int> (random() % size)));
        set.insert(k);
        array[j] = k;
        j = k;
    }
    array[j] = 0;
}

inline void GenerateRandomBypassLineal(int * array, size_t previousSize, size_t newSize)
{
    while (previousSize < newSize)
    {
        size_t index = random() % previousSize;
        array[previousSize] = array[index];
        array[index] = static_cast<int> (previousSize);
        ++previousSize;
    }
}

```

## Приложение 2. (Исходный код программы Direct.cpp, выполняющей прямой обход массива)

```
#include <iomanip>
#include <iostream>
#include <x86intrin.h>

#include "Function__GenerateArray.h"

using namespace std;

constexpr size_t START_SIZE = 100;
constexpr size_t FINAL_SIZE = 5e6;
constexpr size_t NUMBER_OF_ROUNDS = 5;
constexpr double ratio = 1.1;

static int buf = 0;

long double func_AverageBypassTick(const int * array, size_t size)
{
    // int k = reversed ? int(size - 1) : 0;
    int k = 0;
    for (int i = 0; i < size * NUMBER_OF_ROUNDS; i++)
    {
        k = array[k];
        if (k == 4) buf = 7;
    }

    long double minTime = UINT32_MAX;

    for (int j = 0; j < 6; j++)
    {
        k = 0;
        size_t start = __rdtsc();
        for (int i = 0; i < size * NUMBER_OF_ROUNDS; i++) k = array[k];
        size_t end = __rdtsc();
        if (k == 4) buf = k;
        minTime = min(minTime, static_cast <long double> (end - start) / (size *
NUMBER_OF_ROUNDS));
    }
    return minTime;
}

int main(int argc, char * argv[])
{
    auto * array = new int[FINAL_SIZE];
    size_t current = START_SIZE;

    while (current <= FINAL_SIZE)
```

```

{
    GenerateDirectBypass(array, current);
    cout << setw(7) << left << func_AverageBypassTick(array, current) << " ";
    << (long double) (current) * sizeof(int) / 1024 << endl;
    current = static_cast <size_t> (static_cast <double> (current) * ratio);
}

delete[] array;
cerr << buf;
    return 0;
}

```

### Приложение 3. (Исходный код программы Reverse.cpp, выполняющей обратный обход массива)

```
#include <iomanip>
#include <iostream>
#include <x86intrin.h>

#include "Function__GenerateArray.h"

using namespace std;

constexpr size_t START_SIZE = 100;
constexpr size_t FINAL_SIZE = 5e6;
constexpr size_t NUMBER_OF_ROUNDS = 5;
constexpr double ratio = 1.1;

static int buf = 0;

long double func_AverageBypassTick(const int * array, size_t size)
{
    int k = 0;
    for (int i = 0; i < size * NUMBER_OF_ROUNDS; i++)
    {
        k = array[k];
        if (k == 4) buf = 7;
    }

    long double minTime = UINT32_MAX;

    for (int j = 0; j < 6; j++)
    {
        k = 0;
        size_t start = __rdtsc();
        for (int i = 0; i < size * NUMBER_OF_ROUNDS; i++) k = array[k];
        size_t end = __rdtsc();
        if (k == 4) buf = k;
        minTime = min(minTime, static_cast <long double> (end - start) / (size *
NUMBER_OF_ROUNDS));
    }
    return minTime;
}

int main(int argc, char * argv[])
{
    auto * array = new int[FINAL_SIZE];
    size_t current = START_SIZE;

    while (current <= FINAL_SIZE)
    {
        GenerateReverseBypass(array, current);
```

```

    cout << setw(7) << left << func_AverageBypassTick(array, current) << " ";
    << (long double) (current) * sizeof(int) / 1024 << endl;
    current = static_cast <size_t> (static_cast <double> (current) * ratio);
}

delete[] array;
cerr << buf;
return 0;
}

```

## Приложение 4. (Исходный код программы Random.cpp, выполняющей случайный обход массива)

```
#include <iomanip>
#include <iostream>
#include <x86intrin.h>

#include "Function__GenerateArray.h"

using namespace std;

constexpr size_t START_SIZE = 100;
constexpr size_t FINAL_SIZE = 5e6;
constexpr size_t NUMBER_OF_ROUNDS = 5;
constexpr double ratio = 1.1;

static int buf = 0;

long double func_AverageBypassTick(const int * array, size_t size, bool reversed = false)
{
    // int k = reversed ? int(size - 1) : 0;
    int k = 0;
    for (int i = 0; i < size * NUMBER_OF_ROUNDS; i++)
    {
        k = array[k];
        if (k == 4) buf = 7;
    }

    long double minTime = UINT32_MAX;

    for (int j = 0; j < 6; j++)
    {
        k = 0;
        size_t start = __rdtsc();
        for (int i = 0; i < size * NUMBER_OF_ROUNDS; i++) k = array[k];
        size_t end = __rdtsc();
        if (k == 4) buf = k;
        minTime = min(minTime, static_cast <long double> (end - start) / (size *
NUMBER_OF_ROUNDS));
    }
    return minTime;
}

int main(int argc, char * argv[])
{
    auto * array = new int[FINAL_SIZE];
    size_t current = START_SIZE;
    size_t prevSize = 1;
    array[0] = 0;
```

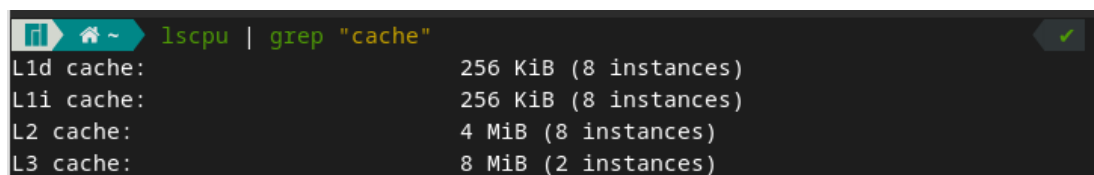
```

while (current <= FINAL_SIZE)
{
    GenerateRandomBypassLineal(array, prevSize, current);
    cout << setw(7) << left << func_AverageBypassTick(array, current) << " ";
        << (long double) (current) * sizeof(int) / 1024 << endl;
    prevSize = current;
    current = static_cast <size_t> (static_cast <double> (current) * ratio);
}

delete[] array;
cerr << buf;
return 0;
}

```

## Приложение 5. (Вывод команды `lscpu | grep "cache"` в системе Manjaro Linux)



```
lscpu | grep "cache"
L1d cache:                256 KiB (8 instances)
L1i cache:                256 KiB (8 instances)
L2 cache:                 4 MiB (8 instances)
L3 cache:                 8 MiB (2 instances)
```



## Приложение 6. (Вывод программы CPU-Z в системе Windows)

