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

«Измерение степени ассоциативности кэш-памяти»

студента 2 курса, группы 21204

Осипова Александра Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Кандидат технических наук
А. Ю. Власенко

Новосибирск 2022

СОДЕРЖАНИЕ

ЦЕЛЬ

Экспериментальное определение степени ассоциативности кэш-памяти.

ЗАДАНИЕ

1. Написать программу, выполняющую обход памяти в соответствии с заданием.
2. Измерить среднее время доступа к одному элементу массива (в тактах процессора) для разного числа фрагментов: от 1 до 32. Построить график зависимости времени от числа фрагментов.
3. По полученному графику определить степень ассоциативности кэш-памяти, сравнить с реальными характеристиками исследуемого процессора.
4. Составить отчет по практической работе.

ОПИСАНИЕ РАБОТЫ

Программа (см. Приложение 1), выполняющая обход памяти в соответствии с заданием, была реализована на сервере кафедры.

Результаты измерений количества тактов в зависимости от числа фрагментов были записаны в csv-таблицу (см. Приложение 2).

Использовался процессор Intel Xeon X5660, имеющий следующие размеры кэша (см. Приложение 3):

L1 Data = 32 Кбайт (на одно ядро)
L1 Inst. = 32 Кбайт (на одно ядро)
L2 = 256 Кбайт (на одно ядро)
L3 = 12 Мбайт (распределено между всеми шести ядрами)

Инициализация массива и порядок его обхода

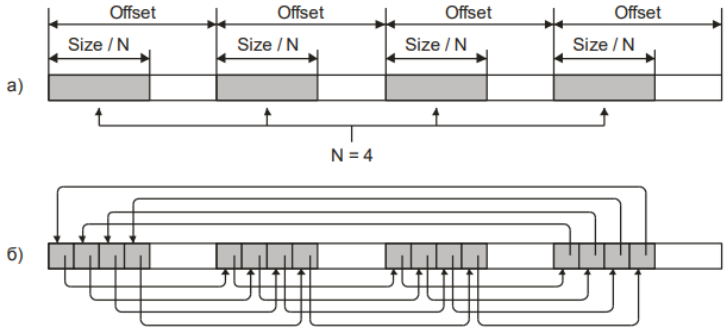


Рис. 2. Схема расположения в памяти фрагментов данных для обхода (а) и порядок обхода элементов (б)

Для каждого уровня кэш-памяти за переменную offset, отвечающую за шаг при обходе массива, было взято значение, равное размеру данного уровня кэш-памяти (в количесвте элементов типа int). Так, для L1 это 8192, для L2 это 65536, а для L3 – 3145728.

Такое значение переменной offset гарантирует, что при обходе массива мы будем считывать элементы, претендующие на одно и тоже множество в кэш-памяти, так как расстояние между ними кратно размеру банка кэша (размер кэш-памяти = k * sizeof(BANK), где k – степень ассоциотивности кэш-памяти данного уровня).

Размер фрагмента определяется как Size/N, где Size – это размер кэш-памяти данного уровня, N – число фрагментов.

Функция, отвечающая за инициализацию массива в соотвествии с заданием:

```
int* InitArray(int size, int offset, int fragmentSize, int numFragments) {
      int* arr = new int[size] {0};
```

```
	for (int i = 0; i < fragmentSize; ++i) {
		int position = i;
		for (int j = 1; j < numFragments; ++j) {
			arr[position] = position + offset;
			position = position + offset;
		}
		arr[position] = (i + 1) % fragmentSize;
	}

	return arr;
}
```

Для измерения количества тактов процессора была написана функция, использующая ассемблерную инструкцию rdtsc:

```
uint64_t GetTSC() {
	uint64_t highPart, lowPart;
	asm volatile("rdtsc\n":"=a"(lowPart), "=d"(highPart));
	return (highPart << 32) | (lowPart);
}
```

Функция, возвращающая минимальное среднее время доступа к элементу массива. Программа выполняет обход цикла, выполняющего чтение элемента массива, 50 раз (ATTEMPTS = 50). Среди всех 50 попыток выбирается минимальное значение переменной accessTime:

```
uint64_t GetTestResult(int* arr, int fragmentSize, int numFragments) {
	uint64_t start, end;
	uint64_t accessTime = 0, minAccessTime = INT_MAX;

	int k = 0;
	for (int count = 0; count < ATTEMPTS; ++count) {

		start = GetTSC();
		for (int i = 0; i < fragmentSize * numFragments; ++i) k = arr[k];// (1)
		end = GetTSC();

		if (k == 12345) std::cout << "Wow!"; // (2)

		accessTime = (end - start) / (fragmentSize * numFragments);
		if (accessTime < minAccessTime) minAccessTime = accessTime;
	}
	delete[] arr;
	return minAccessTime;
}
```

Так как программа компилируется на уровне -O1, цикл (1) может быть удален компилятором, так как он ни на что не влияет в программе. Для этого было добавлено условие (2), чтобы компилятор точно оставил необходимый цикл.

Функция, записывающая результаты измерений для данного уровня кэш-памяти. Размер обходимого массива равен cacheSize * numFragments, так как расстояние между двумя элементами разных фрагментов, претендующий на одно множество кэш-памяти, равно размеру кэша. Соответсвенно, чтобы

обойти numFragments фрагментов, необходим размер массива в cacheSize * numFragments элементов.

```
void TestCacheLevel(int cacheSize, std::ofstream& output) {
	output << GetCacheLevel(cacheSize); //записывает в csv таблицу, с каким
					    //уровнем кэша работает функция
	int* arr = nullptr;

	int offset = cacheSize;

	for (int numFragments = 1; numFragments <= 32; ++numFragments) {
		output << numFragments << ";";
		int fragmentSize = cacheSize / numFragments;
		int arraySize = cacheSize * numFragments;

		arr = InitArray(arraySize, offset, fragmentSize, numFragments);
		output << GetTestResult(arr, fragmentSize, numFragments) << ";\n";
	}
}
```

Графики, показывающие зависимость количество тактов для среднего доступа к элементу массива и числом фрагментов

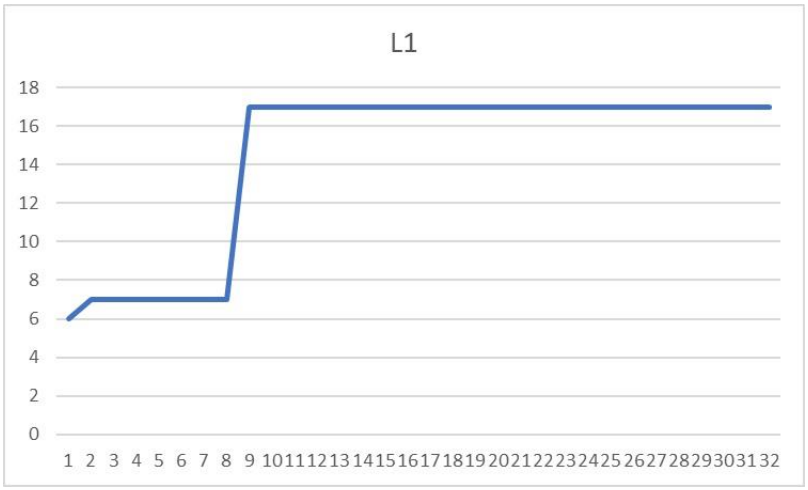


Рис. 1

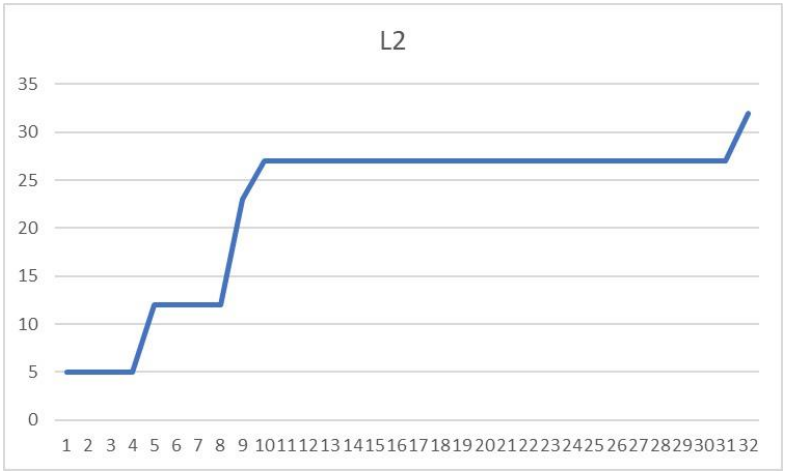


Рис. 2

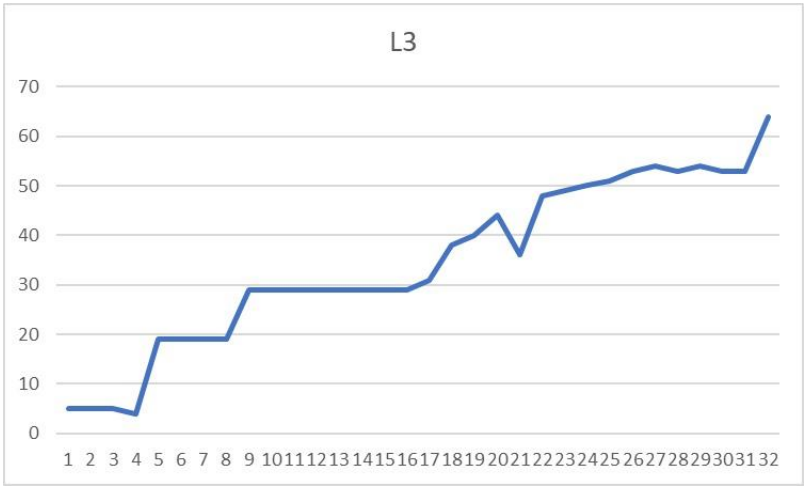


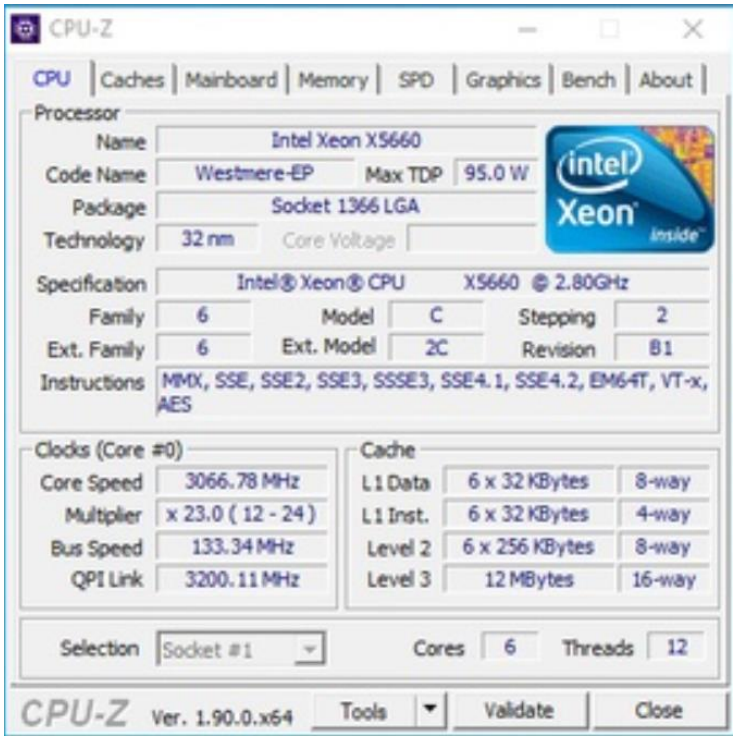
Рис.3

На графиках видно, что на 8-ми фрагментах происходит замедление времени, что соответствует степени ассоциативности L1 и L2. Также на Рис.3 можно заметить замедление после 16 фрагментов – это соответствует степени ассоциативности L3.

Увеличение после 4-х фрагментов на Рис.2 и Рис.3 соответствует степени ассоциативности буфера трансляции адресов.

Реальные значения степени ассоциативности кэш-памяти процессора

L1 Data: 8-way, L1 Inst.: 4-way
L2: 8-way
L3: 16-way



ЗАКЛЮЧЕНИЕ

В рамках данной лабораторной работы была экспериментально определенна степень ассоциативности кэша. Для этого была написана программа со специальным обходом массива, который вызывает буксование кэша. Элементы разных фрагментов, расположенные на расстояние offset = siozeof(CacheLevel) претендуют на одно множество кэш памяти. Когда фрагментов оказывается больше, чем степень ассоциативности, то кэш-контроллеру приходится вытеснять «самые ненужные данные» (алгоритм определение «ненужности» может быть разный), на месте которых он будет размещать новые. Так, для сервера кафедры были получены следующие результаты:

L1	L2	L3
8-way	8-way	16-way

Эти результаты соответствуют реальным характеристикам кэш-памяти.

Приложение 1. Листинг программы

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <fstream>
#include <climits>

#define ATTEMPTS 50

enum CacheLevel {
	L1 = 8192,   //32 [Кбайт] = 32 * 1024 [байт] = 32 * 1024 / 4 [int]
	L2 = 65536,  //256 [Кбайт] = 256 * 1024 [байт] = 256 * 1024 / 4 [int]
	L3 = 3145728 //12 [Мбайт]=12 * 1024 * 1024 [байт] = 12 * 1024 * 1024 / 4[int]
};

int* InitArray(int size, int offset, int fragmentSize, int numFragments) {
	int* arr = new int[size] {0};

	for (int i = 0; i < fragmentSize; ++i) {
		int position = i;
		for (int j = 1; j < numFragments; ++j) {
			arr[position] = position + offset;
			position = position + offset;
		}
		arr[position] = (i + 1) % fragmentSize;
	}

	return arr;
}

uint64_t GetTSC() {
	uint64_t highPart, lowPart;
	asm volatile("rdtsc\n":"=a"(lowPart), "=d"(highPart));
	return (highPart << 32) | (lowPart);
}


uint64_t GetTestResult(int* arr, int fragmentSize, int numFragments) {
	uint64_t start, end;
	uint64_t accessTime = 0, minAccessTime = INT_MAX;

	int k = 0;
	for (int count = 0; count < ATTEMPTS; ++count) {

		start = GetTSC();
		for (int i = 0; i < fragmentSize * numFragments; ++i) k = arr[k];
		end = GetTSC();

		if (k == 12345) std::cout << "Wow!";

		accessTime = (end - start) / (fragmentSize * numFragments);
		if (accessTime < minAccessTime) minAccessTime = accessTime;
	}
	delete[] arr;
	return minAccessTime;
}

void PrintArray(int* arr, int size) {
	for (int i = 0; i < size; ++i) {
		std::cout << arr[i] << " ";
	}
	std::cout << std::endl;
}
```

```
std::string GetCacheLevel(int size) {
	if (size == L1) return "L1;;\n";
	if (size == L2) return "L2;;\n";
	if (size == L3) return "L3;;\n";
}

void TestCacheLevel(int cacheSize, std::ofstream& output) {
	output << GetCacheLevel(cacheSize);

	int* arr = nullptr;

	int offset = cacheSize;

	for (int numFragments = 1; numFragments <= 32; ++numFragments) {
		output << numFragments << ";";
		int fragmentSize = cacheSize / numFragments;
		int arraySize = cacheSize * numFragments;

		arr = InitArray(arraySize, offset, fragmentSize, numFragments);
		output << GetTestResult(arr, fragmentSize, numFragments) << ";\n";
	}
}

int main() {
	std::ofstream output;
	output.open("results.csv");
	if (!output.is_open()) return 1;

	output << "NumFragmenst;Ticks;\n";

	TestCacheLevel(L1, output);
	TestCacheLevel(L2, output);
	TestCacheLevel(L3, output);

	output.close();
	return 0;
}
```

Приложение 2. Таблица с результатами измерений

NumFragmenst	Ticks
L1	
1	6
2	7
3	7
4	7
5	7
6	7
7	7
8	7
9	17
10	17
11	17
12	17
13	17
14	17

15	17
16	17
17	17
18	17
19	17
20	17
21	17
22	17
23	17
24	17
25	17
26	17
27	17
28	17
29	17
30	17
31	17
32	17
L2	
1	5
2	5
3	5
4	5
5	12
6	12
7	12
8	12
9	23
10	27
11	27
12	27
13	27
14	27
15	27
16	27
17	27
18	27
19	27
20	27
21	27
22	27
23	27
24	27
25	27
26	27

27	27
28	27
29	27
30	27
31	27
32	32
L3	
1	5
2	5
3	5
4	4
5	19
6	19
7	19
8	19
9	29
10	29
11	29
12	29
13	29
14	29
15	29
16	29
17	31
18	38
19	40
20	44
21	36
22	48
23	49
24	50
25	51
26	53
27	54
28	53
29	54
30	53
31	53
32	64

Приложение 3. Данные, полученные через CPU-Z