**ЛАБОРАТОРНАЯ РАБОТА №9**

**ИЗМЕРЕНИЕ СТЕПЕНИ АССОЦИАТИВНОСТИ КЭШ-ПАМЯТИ**

**Цель работы:** экспериментальное определение степени ассоциативности кэш-памяти.

**Задание:**

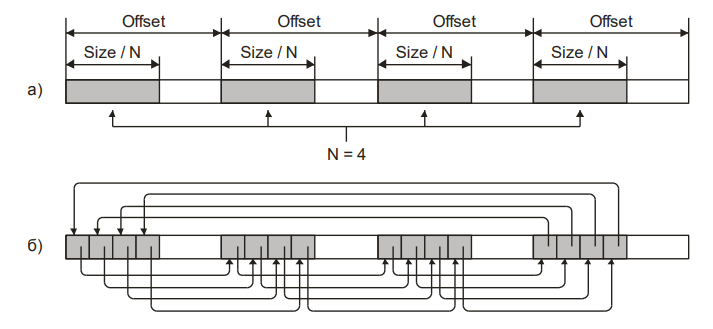
1. Написать программу, выполняющую обход памяти в соответствии с заданием.
2. Измерить среднее время доступа к одному элементу массива (в тактах процессора) для разного числа фрагментов: от 1 до 32. Построить график зависимости времени от числа фрагментов.
3. По полученному графику определить степень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

**Описание работы**

Кэш-память устроена следующим образом: при обращении процессора к оперативной памяти она передает некоторый блок данных (блочная передача данных). Такой блок данных называется строкой. Строка размещается в ячейке кэш-памяти соответствующего размера. Однако память устроена таким образом, что каждая строка из памяти может быть записана только в определенные ячейки кэша, объединение которых называется множеством. Ассоциативностью кэша называется число разных строк кэша, в которые можно отобразить одну строку памяти.

Если одному множеству сопоставляется некоторое число строк памяти, превосходящее объем множества, то ячейки должны быть перезаписаны, на что требуется время, и, соответственно, быстродействие программы падает.

Для определения степени ассоциативности кэша экспериментальным путем напишем программу, обходящую массив данных определенным образом, вызывающим «пробуксовку кэша». Массив будет разбит на несколько блоков определенного размера, равного объему кэш-памяти. Структура массива (а) и метод его обхода (б) представлены на схеме:



Здесь: Size – объем кэш-памяти, N – количество блоков в массиве, Offset – смещение между начали соседних блоков (равен размеру кэш-памяти).

Обход фрагментов данных организуем таким образом, чтобы подряд происходили обращения к элементам разных фрагментов, отстоящим на заданное смещение. Сначала последовательно производится чтение всех первых элементов во всех фрагментах, затем всех вторых, затем – всех третьих и т.д. Т.е. массив должен образовывать односвязный список.

Вычислив среднее время обращения к одному элементу и построив график можно найти моменты, когда после определенного числа считанных блоков время начинает резко возрастать. Количество блоков и будет соответствовать степени ассоциативности кэша. Это объясняется тем, что данные считываются из разных участков памяти и претендуют на одно и то же множество в кэше. После определенного момента объем памяти одной группы становится полностью заполненным, и кэш начинает перезаписываться. Именно количество этих блоков и является степенью ассоциативности.

**Измерения:**

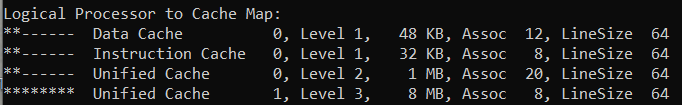
Полный листинг программы, компилируемой с ключом -O1 для размещения переменных в регистрах, приведен в приложении 1.

На основе полученных данных был составлен следующий график зависимости:

Линии LVL1, LVL2 и LVL3 отличаются между собой, по сути, размером отступа, соответствующим размеру кэша этого уровня.

Как можно видеть на графике, для всех трех уровней кэша скачок в среднем времени обращения к элементу массива происходит между 12 и 13 блоками, что говорит о том, что степень ассоциативности всех трех уровней кэш-памяти равна 12.

Сравним полученные данные с настоящими показателями процессора (работа выполнялась на процессоре Intel Core i5-11300H)



Как можно видеть из системных данных, 12 равна лишь степень ассоциативности кэша первого уровня (нас интересует Data Cache). Степень кэшей 2 и 3 уровня соответственно равны 20 и 8.

Такое расхождение с графиком, скорее всего, объясняется алгоритмами работы процессора и операционной системы. Можно предположить, что процессор преимущественно использовал кэш именно первого уровня, т.к. быстродействие с ним наиболее быстрое.

Перепады между 4 и 5 блоками соответствуют степени ассоциативности TLB-кэша (буфер ассоциативной трансляции).

**Вывод:** в ходе выполнения лабораторной работы были изучены некоторые основные понятия, связанные с кэш-памятью, а также экспериментально измерена степень ассоциативности кэш-памяти.

**Приложение 1. Полный листинг компилируемой программы**

1. #include <iostream>
2. #include <fstream>
3. #include <chrono>
5. **using namespace std;**
7. #define CACHE\_SIZE 8 \* 1024 \* 1024 // in bytes
8. #define NUMB\_OF\_RUNS 8
10. **volatile int\* GenerateArray(int numbOfChunks) {**
11. int offset = CACHE\_SIZE;
12. int chunkSize = CACHE\_SIZE / numbOfChunks;
13. volatile int\* array = new int[offset \* numbOfChunks / sizeof(int)];
14. for (int i = 0; i < numbOfChunks; i++) {
15. **for (int j = 0; j < chunkSize / sizeof(int); j++) {**
16. int currentInd = offset / sizeof(int) \* i + j;
17. if (i == numbOfChunks - 1) {
18. array[currentInd] = (j + 1) % (chunkSize / sizeof(int));
19. }
20. **else {**
21. array[currentInd] = currentInd + offset / sizeof(int);
22. }
23. }
24. }
25. **return array;**
26. }
28. long GetAvgTime(volatile int\* array) {
29. volatile int numOfElements = CACHE\_SIZE / sizeof(int); // Number of elements to be accessed
30. **volatile int indOfNext = 0;**
31. for (int i = 0; i < numOfElements; i++) { // warm up
32. indOfNext = array[indOfNext];
33. }
34. indOfNext = 0;
35. **chrono::system\_clock clock;**
36. auto start = clock.now();
37. for (int i = 0; i < numOfElements \* 8; i++) {
38. indOfNext = array[indOfNext];
39. }
40. **auto end = clock.now();**
41. long averageTime = (end - start).count() / (numOfElements \* 8); // in ns
42. return averageTime;
43. }
45. **int main() {**
46. ofstream numbOfChunkLog;
47. ofstream avgTimeLog;
48. numbOfChunkLog.open("/home/nextples/CLionProjects/EVM-lab9/numb\_of\_chunk\_log.txt");
49. avgTimeLog.open("/home/nextples/CLionProjects/EVM-lab9/avg\_time\_log.txt");
50. **for (int numOfChunks = 1; numOfChunks <= 32; numOfChunks++) {**
51. volatile int\* array = GenerateArray(numOfChunks);
52. long averageTime = 0;
53. for (int i = 0; i < NUMB\_OF\_RUNS; i++) {
54. averageTime += GetAvgTime(array);
55. **}**
56. cout << "Number of chunks: " << numOfChunks << "; average time: " << averageTime / NUMB\_OF\_RUNS << " ns**\n**";
57. numbOfChunkLog << numOfChunks << endl;
58. avgTimeLog << averageTime / NUMB\_OF\_RUNS << endl;
59. delete[] array;
60. **}**
61. }