**ЛАБОРАТОРНАЯ РАБОТА №8**

**ВЛИЯНИЕ КЭШ-ПАМЯТИ НА ВРЕМЯ ОБРАБОТКИ МАССИВОВ**

**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.

**Задачи:** на основе разных способов обхода циклического массива: прямого, обратного и случайного, сделать среднюю оценку количества тактов для обращения к одному элементу массива. На основе полученных данных выяснить приблизительный размер кэша процессора.

Описание методов обхода массива:

* Прямой обход

Подразумевает, что в i-ом элементе массива будет лежать значение, равное i+1. В случае последнего элемента следующим элементом будет самый первый.

* Обратный обход

Прямой обход, реализованный с конца массива, т.е. подразумевает, что в i-ом элементе массива будет лежать значение, равное i-1.

* Случайный обход

Подразумевает, что значением ячейки массива с индексом i будет некоторый индекс, который невозможно предугадать, но гарантируется, что индекс не выйдет за пределы массива. Тем самым, ходя по разным индексам, мы полностью обходим массив.

В рамках требований лабораторной работы замеры среднего времени обращения к элементу массива проводились на массивах различных размеров от 1 КБ до 64 МБ, выполняясь многократно.

Шагом изменения размера массива был выбран коэффициент 1.3, чтобы все существенные изгибы на графике были отчетливо видны.

Для получения наиболее объективного времени для замеров использовались следующие приемы:

* Программа компилировалась с ключом оптимизации -O1 для исключения лишних обращений в память за счет размещения переменных в регистрах, которые могли бы исказить средний показатель времени.
* Перед измерением времени для каждого конкретного размера массива в кэш-память предварительно загружались необходимые данные посредством однократного обхода массива. Что впоследствии сказывается на среднем времени обращения к элементам, т.к. время на загрузку кэша не тратится.
* Замеры для каждого размера массива проводились несколько раз, чтобы снизить влияние посторонних процессов, запущенных на компьютере.

На основе полученных данных был составлен график среднего времени обращения к элементу массива в зависимости от размера массива, а также способа обхода массива:

**Анализ полученных данных:**

По построенному графику видно, что среднее время обращения к элементам массива при прямом и обратном обходе постоянно при любом размере массива. Это связано с тем, что в большинстве современных микропроцессоров реализована аппаратная предвыборка данных. Она устроена таким образом, что при последовательном обходе очередные данные считываются из оперативной памяти еще до того, как к ним произошло обращение. Кэш-контроллеры с высокой вероятностью распознают последовательный обход памяти и обеспечивают эффективную предварительную загрузку данных в кэш-память. Как следствие, вероятность кэш-промахов значительно снижается.

В случае же случайного обхода массива среднее время обращения растет вместе с увеличением размера массива. Это связано с тем, что из-за сложного порядка обхода массива аппаратная предвыборка данных работает некорректно, т.к. кэш-контроллер не распознает порядок правильно, либо не работает совсем.

В случае малых размеров массива время при случайном обходе совпадает со временем прямого и обратного обхода, т.к. массив полностью помещается в кэш-память, и, следовательно, время обращения низкое. С ростом размеров же учащаются кэш-промахи, и требуется более длительное обращение к оперативной памяти.

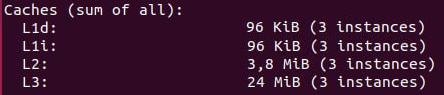
**Анализ размера кэша процессора, исходя из данных графика:**

На графике видно, что первый скачок времени обращения к элементам происходит между массивами размерами от 85 до 145 КБ. На основе этой информации можно сделать вывод, что в этом месте уже происходит обращение ко второму уровню кэш-памяти.

Следующий скачок происходит между размерами в 3382 и 5716 КБ. Соответственно, можно сделать вывод, что между ними лежит граница следующего уровня кэш-памяти.

И последний скачок можно наблюдать после 21226 КБ. Следовательно, там находится граница следующего уровня.

Для проверки предположений, установленных выше, сравним характеристики процессора, полученные с помощью команды **lscpu:**

****

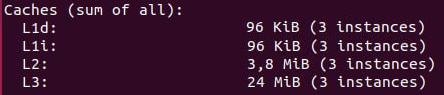
Эти характеристики подтверждают предположения.

**Приложение 1. Листинг компилируемой программы**

1. #include <iostream>
2. #include <set>
3. #include <cstdlib>
4. #include <ctime>
5. **#include <fstream>**
6. #include <limits.h>
7. #include <cmath>
8. #include <random>
9. #include <x86intrin.h>
11. using namespace std;
13. void goForward(unsigned int \*array, unsigned int n){
14. for(unsigned int i = 0; i < n - 1; i++){
15. **array[i] = i + 1;**
16. }
17. array[n - 1] = 0;
18. }
20. **void goBackwards(unsigned int \*array, unsigned int n){**
21. for(unsigned int i = n - 1; i > 0; --i)
22. array[i] = i - 1;
23. array[0] = n - 1;
24. }
26. void goRandom(unsigned int \*array, const unsigned int arraySize) {
27. auto \*indexes = new unsigned int[arraySize];
28. for (unsigned int i = 0; i < arraySize; ++i) {
29. indexes[i] = i;
30. **}**
31. srand(time(NULL));
32. for (long long i = arraySize - 1; i >= 0; --i) {
33. unsigned int j = 0;
34. if (i > 0) {
35. **j = rand() % i;**
36. }
37. swap(indexes[i], indexes[j]);
38. }
40. **for (unsigned int i = 0; i < arraySize - 1; ++i) {**
41. array[indexes[i]] = indexes[i + 1];
42. }
43. array[indexes[arraySize - 1]] = indexes[0];
45. **delete[] indexes;**
46. }

49. void run(const unsigned int \*array, unsigned int n, unsigned int k) **{**
50. int m = 0;
51. for(unsigned int j = 0; j < n \* k; ++j)
52. m = array[m];
53. }
55. int main(){
56. ofstream ticks;
57. ofstream mem;
58. ofstream log;
59. **log.open("/home/nextples/CLionProjects/EVM-lab8/log.txt");**
60. mem.open("/home/nextples/CLionProjects/EVM-lab8/mem.txt");
61. ticks.open("/home/nextples/CLionProjects/EVM-lab8/ticks.txt");
62. unsigned int minSize = 256; //1 Kb
63. **unsigned int k = 10;**
64. int maxSize = 8 \* 1024 \* 1024 \* 2; //64 Mb
65. unsigned int currentSize = minSize;
66. **unsigned int \*array;**
67. **currentSize = minSize;**
68. for (; currentSize < maxSize; currentSize = (unsigned int)(currentSize \* 1.3)){
69. array = (unsigned int \*)malloc(currentSize \* sizeof(unsigned int));
70. **goForward(array, currentSize);**
72. unsigned long long minTime = ULLONG\_MAX;
73. run(array, currentSize, 1);
74. for (int j = 0; j < 3; j++) {
75. **unsigned long long start = \_\_rdtsc();**
76. run(array, currentSize, k);
77. unsigned long long end = \_\_rdtsc();
78. size\_t Time = end - start;
79. **if (Time < minTime) {**
80. minTime = Time;
81. }
82. }
83. **mem << currentSize \* 32 / 8 / 1024 << endl;**
84. ticks << minTime / (k \* currentSize) << endl;
86. log << "Forward lookup for " << currentSize \* 32 / 8 / 1024
87. << " Kb, taken: " << minTime / (k \* currentSize) << endl;
89. cout << "Forward lookup for " << currentSize \* 32 / 8 / 1024
90. << " Kb, taken: " << minTime / (k \* currentSize) << endl;
91. free(array);
92. }
93. **mem << "\n\n\n\n" << endl;**
94. ticks << "**\n\n\n\n**" << endl;
95. currentSize = minSize;
96. **for (; currentSize < maxSize; currentSize = (unsigned int)(currentSize \* 1.3)){**
97. array = (unsigned int \*)malloc(currentSize \* sizeof(unsigned int));
98. goBackwards(array, currentSize);
99. unsigned long long minTime = LLONG\_MAX;
100. **run(array, currentSize, 1);**
101. for (int j = 0; j < 3; j++) {
102. unsigned long long start = \_\_rdtsc();
103. run(array, currentSize, k);
104. **unsigned long long end = \_\_rdtsc();**
105. unsigned long long Time = end - start;
106. if (Time < minTime) {
107. minTime = Time;
108. }
109. **}**
110. mem << currentSize \* 32 / 8 / 1024 << endl;
111. ticks << minTime / (k \* currentSize) << endl;
112. **log << "Backwards lookup for " << currentSize \* 32 / 8 / 1024**
113. << " Kb, taken: " << minTime / (k \* currentSize) << endl;
114. cout << "Backwards lookup for " << currentSize \* 32 / 8 / 1024
115. << " Kb, taken: " << minTime / (k \* currentSize) << endl;
116. **free(array);**
117. }
118. mem << "**\n\n\n\n**" << endl;
119. ticks << "**\n\n\n\n**" << endl;
120. **currentSize = minSize;**
121. for (; currentSize < maxSize; currentSize = (unsigned int)(currentSize \* 1.3)){
122. array = (unsigned int \*)malloc(currentSize \* sizeof(unsigned int));
123. **goRandom(array, currentSize);**
124. unsigned long long minTime = LLONG\_MAX;
125. run(array, currentSize, 1);
126. for (int j = 0; j < 3; j++) {
127. unsigned long long start = \_\_rdtsc();
128. run(array, currentSize, k);
129. unsigned long long end = \_\_rdtsc();
130. unsigned long long Time = end - start;
131. **if (Time < minTime) {**
132. minTime = Time;
133. }
134. }
135. mem << currentSize \* 32 / 8 / 1024 << endl;
136. **ticks << minTime / (k \* currentSize) << endl;**
137. log << "Random lookup for " << currentSize \* 32 / 8 / 1024
138. << " Kb, taken: " << minTime / (k \* currentSize) << endl;
139. cout << "Random lookup for " << currentSize \* 32 / 8 / 1024
140. **<< " Kb, taken: " << minTime / (k \* currentSize) << endl;**
141. free(array);
142. }
143. mem << "**\n\n\n\n**" << endl;
144. **ticks << "\n\n\n\n" << endl;**
145. log.close();
146. return 0;
147. }

**Приложение 2. Фактические размеры кэш-памяти**

****