## Литература

1. Руководство по разработке для WindowsPhone 8.1 [Электронный ресурс] / METANIT.COM: – Режим доступа: http://metanit.com/sharp/windowsphone/ – Дата доступа: 05.05.2016.
2. Разработка приложений для WindowsPhone 7.5 / С.В. Пугачев, С.И. Павлов, Д.В. Сошников. – СПб.:БХВ-Петербург, 2012. – 384с.: ил.

УДК 004.07

## АНАЛИЗ ВЛИЯНИЯ СТРУКТУРЫ ДАННЫХ НА СКОРОСТЬ ВЫПОЛНЕНИЯ ГРАФИЧЕСКИХ АЛГОРИТМОВ

Романенко Р.А.

Научный руководитель – Борисова И.М., ст. преподаватель

Кэш – неотъемлемая часть современного процессора. Зачастую проблемы с производительностью приложений связаны с неправильным использованием кэш-памяти. Современные процессоры умеют предсказывать последовательность обработки данных и автоматически оптимизировать процесс загрузки. Разумеется, даже в таком случае использование правильного представления данных будет давать прирост в скорости работы алгоритмов.

Требования к эффективной структуре данных: линейное расположение данных, данные отсортированы по частоте доступа, небольшой размер, простой и предсказуемый порядок доступа к данным

Автор рассмотрел создание эффективной структуры данных на примере словаря.

```
struct Node {
    key_type        key;
    value_type      value;
};
```

Классический алгоритм поиска значения по ключу (псевдокод):

```
for node in node_list do
        if node.key == key          // Вопрос на засыпку:
            return node.value // Как часто выполняется эта строка?
        else
            continue
```

Следует отметить, что данный алгоритм не выполняет сложных вычислений, а значит, по большей части зависит от скорости загрузки данных. Чем быстрее будет загружена очередная ячейка, тем быстрее оператор if сможет приступить к сравнению ключей. При $N$ итераций цикла нужно обработать $N$ ключей и только для одного из них вернуть значение. Допустим, что размеры ключа и значения одинаковы, а словарь полностью помещается в кэш-память, следовательно, половина кэш-памяти будет использована впустую.

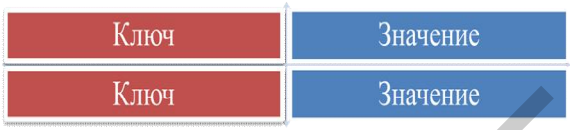


Рисунок 1. Структура кэша в момент поиска по словарю

Красный – данные, используемые при поиске значения (высокая частота доступа). Синий – данные, возвращаемые для данного значения (низкая частота доступа). Это приводит к замедлению скорости работы алгоритма и приложения в целом, так как используются только операции загрузки и сравнения. Процессор загружает данные т.н. «словами».

До этого мы допускали, что пара ключ-значение помещается в одно машинное «слово», рассмотрим обратную ситуацию. Допустим, что ячейка словаря не помещается в одно «слово», тогда возникает следующая проблема: накладные расходы на загрузку ненужных данных.

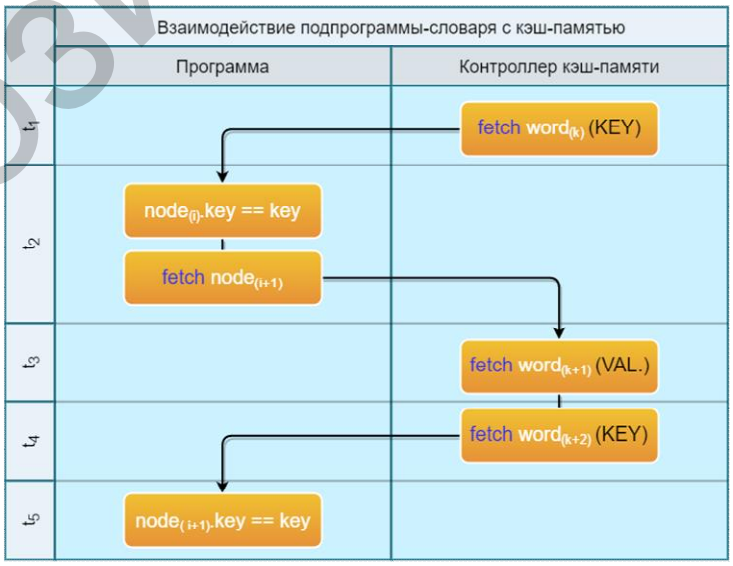


Рисунок 2. Таблица зависимости различных операций от времени

$(t_1 = t_2 = t_3 = t_4 = t_5)$

Ключу ячейки с индексом *i* соответствует машинное слово word с индексом *k*. Инструкция fetch занимается загрузкой данных: программа запрашивает загрузку данных ячейки (node), а контроллер кэш-памяти выполняет запрос, считывая данные «словами» из кэш-памяти. В данном случае контроллеру кэш-памяти приходится загружать по 2 машинных слова для обработки каждого последующего элемента, так как каждому последующему ключу предшествует значение для предыдущего ключа. Получается, что для *N* элементов словаря, контроллеру кэш-памяти придётся загрузить *(N-1)*2 + 1* машинных слов. Это ведёт к серьёзному замедлению скорости работы алгоритма и приложения, так как в момент загрузки данных контроллером, программа не выполняется. Простым решением является замена «ячеек» на две линейные структуры данных, одна из которых содержит ключи, а другая – значения.

В языке C++ в качестве линейных структур могут использоваться массивы и вектора. Список не является линейной структурой данных. Он состоит из ячеек, каждая из которых содержит указатели на предыдущую и последующую ячейку списка. Такая реализация приводит к проблемам, описанным выше.

Возможная реализация алгоритма поиска на C++:

```
#include <vector>
#include <algorithm>
std::vector<key_type>   keys;
std::vector<value_type>values;
…
auto keys_end = keys.cend();
for (auto itr = keys.cbegin(); itr != keys_end; ++itr)
{
        if (key == *itr)
                return values[std::distance(keys.begin(), itr)];
}
```

В данном случае решаются описанные выше проблемы: цикл запрашивает данные из линейной последовательности, каждый следующий ключ гарантированно находится в кэше, нет необходимости загружать лишние данные из кэша, так как он содержит только ключи. Соответственно необходимые данные всегда находятся в кэше, за исключением двух случаев: первый доступ к вектору keys – до первого

доступа данные не загружаются в кэш, первый и единственный доступ к вектору values – до доступа данные не загружаются в кэш.

Был проведён тест для стандартной (std::map) и собственной реализаций словаря в идеальных условиях:

1. Тестовое приложение не выделяло память динамически во время и после теста различных реализаций словарей
2. Для обеих реализаций сохранялась последовательность добавления, каждый элемент добавлялся в конец списка (значения и ключи в диапазоне [0; 4095])

В словарь заносились ключи типа double с соответствующими значениями типа unsigned long. Словарь содержал 4096 элементов, количество замеров – 2048.

| Вставка 4096 элементов в словарь | |
|---|---|
| std::map | 642 мкс. |
| Своя реализация | 243 мкс. |
| Поиск по ключу в словаре из 4096 элементов | |
| std::map | 308 нс. |
| Своя реализация | 222 нс. |

Рисунок 3.Результаты тестирования.

Итог: Скорость вставки элементов увеличилась на 164%.Скорость поиска элемента по ключу увеличилась на 39%

**Литература**

*1. 1.The Gap between Processor and Memory speeds http://gec.di.uminho.pt/discip/minf/ac0102/1000gap_proc-mem_speed.pdf*

*2. Latency Numbers Every Programmer Should Know [Электронный ресурс] https://gist.github.com/jboner/2841832*