**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра вычислительной техники**

**ОТЧЕТ**

**по лабораторной работе № 1**

**по дисциплине «Параллельные алгоритмы и системы»**

**Тема: Хэш-таблица с закрытой адресацией**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9305 |  | Есин А.Ю. |
| Преподаватель |  | Пазников А.А. |

Оглавление

[Цель работы 3](#_Toc131734312)

[Основные положения 3](#_Toc131734313)

[Ход работы 3](#_Toc131734314)

[2. Замена логического «И» 8](#_Toc131734315)

[3. Метод разрешения коллизия с помощью линейного пробинга 8](#_Toc131734316)

[Вывод 9](#_Toc131734317)

[Приложение 9](#_Toc131734318)

# Цель работы

Оптимизировать методы вставки и получения элемента из хэш-таблицы с закрытой адресацией.

# Основные положения

Хэш-таблица была реализована для базовых примитивов типа std::string. При этом для хранения данных в таблице используется двойной односвязный список std::list (т.е. список, внутри которого еще один список). Сами элементы представляют собой пару ключ-значение std::pair.

При инициализации таблицы задается поле capacity, которое хранит в себе размерность таблицы. Хэш-функция в данной реализации представляет собой сумму ASCII кодов в ключе.

Также был написан testbench, который сначала «разогревает» процессор, а затем начинает основную работу по добавлению элементов в таблицу и получению элементов из нее N раз, где N = 1000000. При этом value внутри ячейки лежит в диапазоне от 0 до 10000.

# Ход работы

В первую очередь необходимо было найти hotspot. Это было сделано с помощью утилиты perf командами record и report. Результат поиска hotspot приведен на рисунке 1.

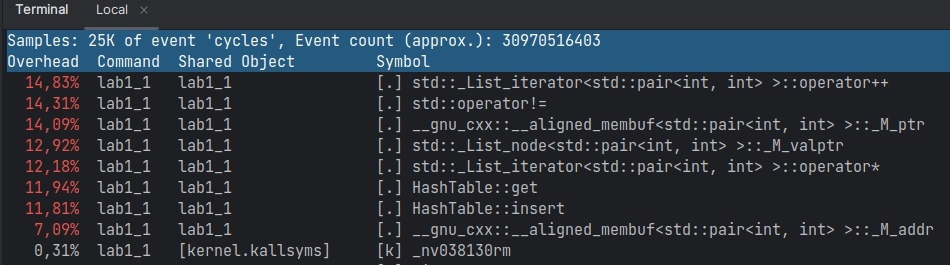


Рисунок 1. Отчетность утилиты perf

Измерения будут происходить по заранее написанному тесту, где 10000 циклов уходит на прогрев процессора, 10000 циклов уходит на вставку и получение элементов из таблицы. При этом внутри программы замеряется время выполнения, а с помощью утилиты perf в системе Linux будет произведен замер других характеристик, таких как кэш-миссы

1. **Сравнительный анализ**

Результаты сравнительного анализа приведены в таблице 1, а также представлены на рисунках 2-7

Таблица 1. Сравнительный анализ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Time elapsed | Cpu-migration | Cache-misses | Cache-references | Компилятор |
| Без оптимизации | 6,964 | 11 | 1,652% | 306613169 | gcc |
| Замена остатка на «&» | 6,142 | 29 | 0,772% | 723552910 | gcc |
| Разрешение коллизий с помощью пробинга | 5,143 | 5 | 0,933% | 514771127 | gcc |
| Без оптимизации | 7,199 | 12 | 3,2% | 187927566 | clang |
| Замена остатка на «&» | 6,285 | 5 | 0,843% | 625366131 | clang |
| Разрешение коллизий с помощью пробинга | 5,315 | 24 | 1,027% | 523280405 | clang |

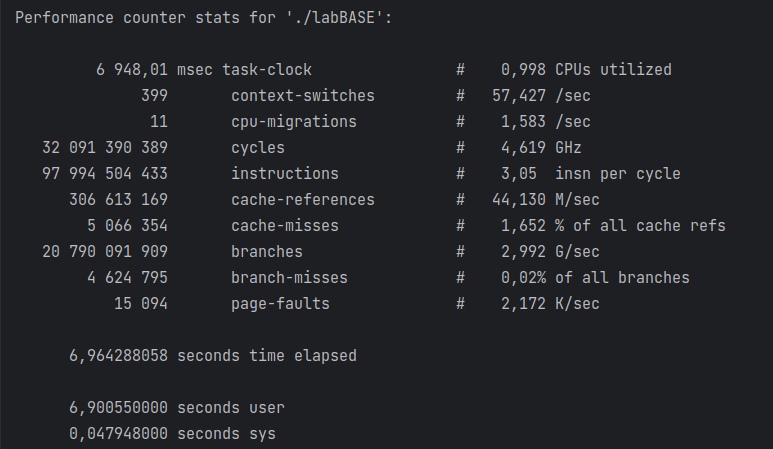


Рисунок 2. Без оптимизации g++

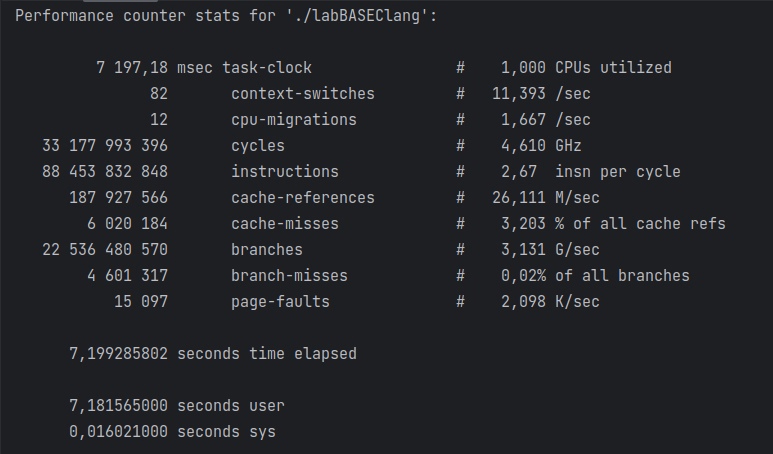


Рисунок 3. Без оптимизации clang

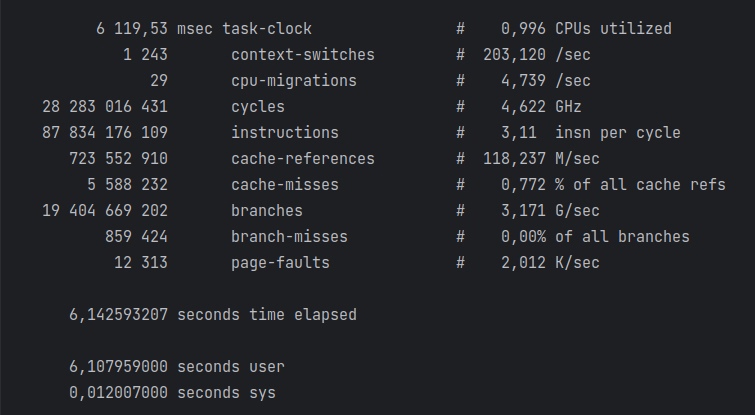


Рисунок 4. Логическое «И» g++

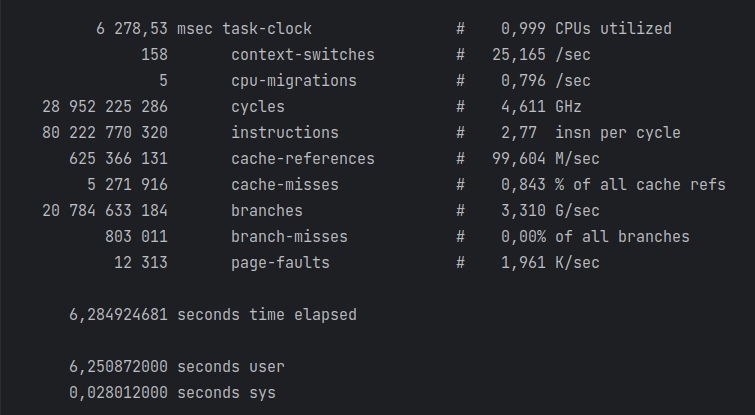


Рисунок 5. Логическое «И» clang

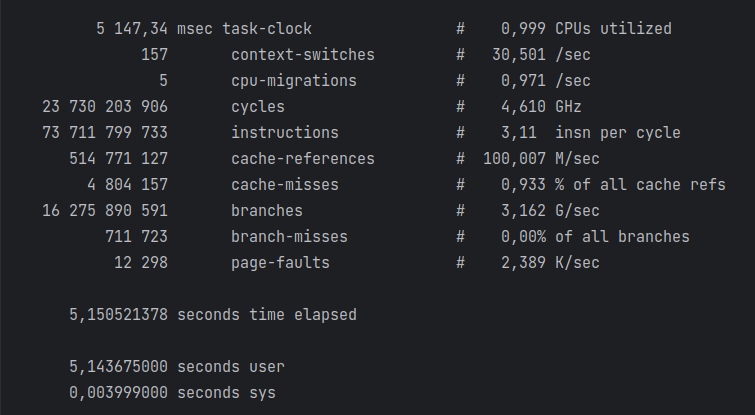


Рисунок 6. Пробинг g++

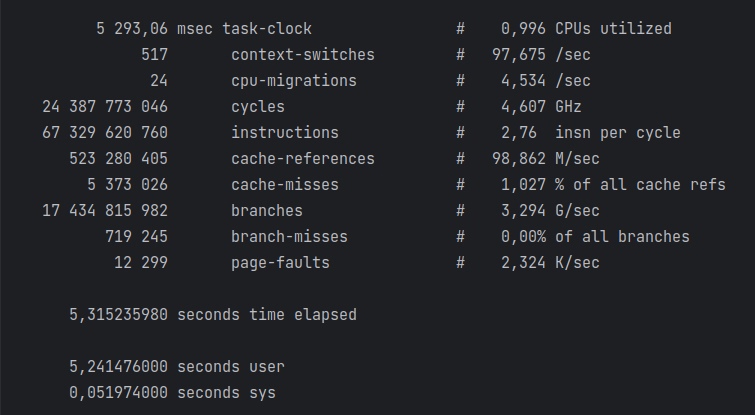


Рисунок 7. Пробинг clang

# Замена логического «И»

Замена остатка от деления на логическое "И" (англ. bitwise AND) в хеш-функции - это одна из техник, используемых для вычисления хеш-кода (хеш-значения) ключа в хеш-таблице. Эта техника может использоваться для получения более равномерного распределения хеш-кодов и более эффективного разрешения коллизий (столкновений).

Одним из распространенных способов вычисления хеш-кода ключа является взятие остатка от деления результата некоторой арифметической операции (например, суммы кодов символов ключа) на размер хеш-таблицы. Однако, в некоторых случаях, такая операция может приводить к неравномерному распределению хеш-кодов, особенно если размер хеш-таблицы не является степенью двойки.

Замена остатка от деления на логическое "И" может быть использована для более равномерного распределения хеш-кодов, особенно при использовании размеров хеш-таблицы, являющихся степенями двойки.

# Метод разрешения коллизия с помощью линейного пробинга

Линейный пробинг (англ. linear probing) - это одна из техник решения коллизий (столкновений) в хеш-таблицах, которая используется для разрешения конфликтов при вставке нового элемента в ячейку хеш-таблицы, которая уже содержит другой элемент.

При использовании линейного пробинга, если происходит коллизия (т.е. вычисленный хеш-код указывает на уже занятую ячейку), новый элемент помещается в следующую свободную ячейку в таблице. Таким образом, элементы, имеющие одинаковый хеш-код, размещаются в последовательных ячейках хеш-таблицы.

Для таблицы, приведенной выше, были использованы следующие метрики:

* Time elapsed – общее время выполнения записи с помощью perf
* Cpu-migration – количество переключений ядер процессора
* Cache-misses – количество обращений к кэшу, когда необходимых данных в нем нет
* Cache-references – общее количество обращений к кэшу

# Вывод

В ходе выполнения лабораторной работы был оптимизирован алгоритм операции вставки и поиска в хэш-таблице с закрытой адресацией.

Оригинальный алгоритм показал, что метрики Cache-misses и Task-clock можно минимизировать.

Лучший результат дало применение метода линейного пробинга, так как с его помощью решается основная проблема хэш-таблиц – коллизии при добавлении элемента в таблицу.

## **Приложение**

#include <iostream>

#include <list>

#include <vector>

#include <chrono>

#include <algorithm>

class HashTable {

private:

std::list<std::list<std::pair<std::string, std::string>>> table; // vector of linked lists to store key-value pairs

int capacity; // capacity of the hash table

int hash(std::string key) const; // hash function to calculate the index for a given key

public:

HashTable(int capacity) : capacity(capacity) {

table.resize(capacity); // initialize the table with empty linked lists

}

void insert(std::string key, std::string value) {

int index = hash(key);

auto iter = table.begin();

std::advance(iter, index);

for (auto& kvp : \*iter) { // check if the key already exists

if (kvp.first == key) {

kvp.second = value; // update the value

return;

}

}

// key doesn't exist, so add a new key-value pair to the linked list

iter->push\_back(std::make\_pair(key, value));

}

std::string get(std::string key) {

int index = hash(key);

auto iter = table.begin();

std::advance(iter, index);

for (auto& kvp : \*iter) { // search for the key in the linked list

if (kvp.first == key) {

return kvp.second; // return the value if the key is found

}

}

return ""; // key not found

}

void remove(std::string key) {

int index = hash(key);

auto iter = table.begin();

std::advance(iter, index);

iter->remove\_if([&key](auto& kvp) { return kvp.first == key; }); // remove the key-value pair if the key is found

}

};

// Simple hash function that just returns the key modulo the capacity of the table

int HashTable::hash(std::string key) const {

int sum = 0;

for (char c : key) {

sum += c;

}

return sum % capacity;

}

/\*!

\* Generates some random data and puts it in hash table

\* @param size : number of created entries

\* @param table : table to add data to

\*/

void generateTestData(int size, HashTable& table) {

for (int i = 0; i < size; ++i) {

table.insert(std::to\_string(i + 10), std::to\_string(i));

}

}

/\*!

\* Gets all the data from hash table and prints it

\* @param size : number of entries

\* @param table : table-source of data

\*/

void getTestData(int size, HashTable& table) {

for (int i = 0; i < size; ++i) {

printf("number: %s",table.get(std::to\_string(i + 10)).c\_str());

}

}

/\*!

\* Removes all data from hash table

\* @param size : number of deleted entries

\* @param table : source table

\*/

void deleteTestData(int size, HashTable& table) {

for (int i = 0; i < size; ++i) {

table.remove(std::to\_string(i + 10));

}

}

/\*!

\* Function that launches git test

\* @param warmUpIterations : Number of iterations that don't count (used to warm up the cpu)

\* @param mainIterations : Number of counted iterations

\* @param maxInt : Max number that can be in map

\* @param map : Map to torture

\*/

void gitTest(int warmUpIterations, int mainIterations, int maxInt, HashTable& map) {

//warmup

std::string result = "";

for (long i = 0; i < warmUpIterations; i++) {

int num1 = (int)i % maxInt;

int num2 = num1 \* 2;

int num3 = num1 + 1;

map.insert(std::to\_string(num1), std::to\_string(num2));

std::string value = map.get(std::to\_string(num3));

if (!value.empty()) {

result += value;

}

}

//main test

result = "";

auto start = std::chrono::high\_resolution\_clock::now();

for (long i = 0; i < mainIterations; i++) {

int num1 = (int)i % maxInt;

int num2 = num1 \* 2;

int num3 = num1 + 1;

map.insert(std::to\_string(num1), std::to\_string(num2));

std::string value = map.get(std::to\_string(num3));

if (!value.empty()) {

result += value;

}

}

auto stop = std::chrono::high\_resolution\_clock::now();

auto testTime = std::chrono::duration\_cast<std::chrono::duration<double>>(stop - start);

//std::cout << result << std::endl;

std::cout << "Test time: " << testTime.count() << "seconds" << std::endl;

}

/\*!

\* Function that launches handmade tests

\* @param size : number of entries in table

\* @param table : tested hash map

\*/

void mapTest(int size, HashTable& table) {

generateTestData(size, table);

getTestData(size, table);

deleteTestData(size, table);

}

int main() {

HashTable ht(1000002);

//option 2: git tests

gitTest(100000, 1000000, 10000, ht);

return 0;

}