МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Ульяновский государственный технический университет**»**

Кафедра «Вычислительная техника**»**

**Лабораторная работа №1**

по дисциплине «Высокопроизводительные вычисления»

«Исследование средств измерения затрат времени исполнения кода»

Выполнил: студент

гр. ИВТАПбд-41

Галацков И.А.

Проверил:

Преподаватель кафедры ВТ

Негода В.В.

Ульяновск, 2024

**Оглавление**

[1. Параметры платформы 3](#_Toc129808121)

[2. Оценка влияния режима сборки debug на время исполнения кода 3](#_Toc129808122)

[3. Исследование разрешающей способности средств измерения затрат времени 5](#_Toc129808123)

[4. Исследование повторяемости результатов измерения 6](#_Toc129808124)

[5. Исследование ошибок оценки затрат времени по эмпирической формуле с линейной зависимостью времени от размера рабочей нагрузки 12](#_Toc129808125)

[Приложение. Измерение времени выполнения функции Фибоначчи 18](#_Toc129808126)

# 1. Параметры платформы

Замеры значений времени выполнения для лабораторной работы проводились на персональном компьютере со следующими характеристиками:

1. Тип ЦП - AMD Ryzen 5 5500, 3500 MHz (35 x 100) (12 ядра / 12 потоков)
2. Оперативная память - 16 ГБ DDR4-2666 DDR4 SDRAM (24-18-18-43 @ 1333 МГц)
3. Видеоадаптер – AMD Vega 7
4. Накопитель –SSD M.2 512GB
5. Операционная система - Microsoft Windows 11 Home

# 2. Оценка влияния режима сборки debug на время исполнения кода

Для замера времени выполнения была написана функция вычисления числа Фибоначчи. Затем было проведено измерение времени выполнения данной функции с различными аргументами (n = 10 и n = 40) тремя различными способами. Результаты измерения представлены ниже. Исходный код программы находится в приложении.

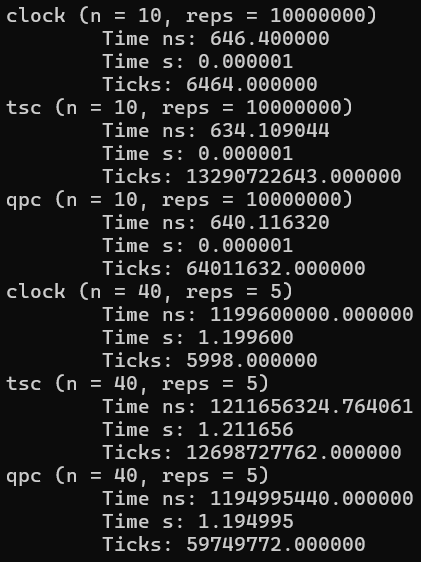


Рис.1 Результаты измерений функции Фибоначчи (debug)

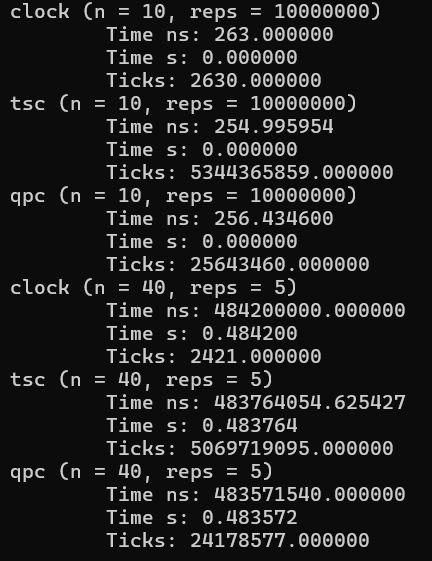


Рис.2 Результаты измерений функции Фибоначчи (сборка release)

Таблица 1 Сравнение результатов измерений функции Фибоначчи

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| t\_code ns | Debug | | Release | | T\_Debug / T\_Release | |
| n=10 | n=40 | n=10 | n=40 | n=10 | n=40 |
| clock: t\_code | 646.4 | 1199600000 | 263 | 484200000 | 2.458 | 2.477 |
| clock: t\_end – t\_start | 6464 | 5998 | 2630 | 484200 | 2.458 | 1.239 |
| QPC: t\_code | 640.116320 | 1194995440 | 256.434600 | 483571540 | 2.496 | 2.471 |
| QPC: t\_end – t\_start | 64011632 | 59749772 | 25643460 | 24178577 | 249.6 | 2.471 |
| TSC: t\_code | 634.109044 | 1211656324.8 | 254.995954 | 483764054.6 | 2.488 | 2.505 |

# 3. Исследование разрешающей способности средств измерения затрат времени

Разрешающей способностью средств измерений называют наименьшее изменение измеряемой величины, различимое с помощью данного средства измерения. Были измерены значения следующих счетчиков: clock, tsc, qpc.

Получить значение разрешающей способности можно, последовательно увеличивая объем обработки данных между двумя засечками времени до устойчивого получения значения, отличного от 0. Достаточно получения двух подряд значений, отличных от 0. Исключением является счетчик tsc, который успеет изменить свое значение при непосредственно следующем вызове, так что его разрешающей способностью будет время между двумя вызовами подряд.

Для получения значений счетчиков была написана следующая функция. Объекты измерения рассмотрены в следующей главе.

void calc\_resolutions() {

int maxLenArray = 10000;

fill(maxLenArray);

string counterNamesByType[] = { "clock", "tsc", "qpc" };

int counterStepByType[] = { 1, 1, 1 };

for (int type = 0; type <= 2; ++type) {

string currCounterName = counterNamesByType[type];

int step = counterStepByType[type];

cout << "\tРасчет разрешающей способности для счетчика " << currCounterName << "\n";

double prevNs = -1, currNs = -1, minDiff = LLONG\_MAX;

if (type == 1) {

double tclock, tsc, frequency;

tclock = clock();

while (clock() < tclock + 1) {}

tsc = \_\_rdtsc();

while (clock() < tclock + 2) {}

tsc = \_\_rdtsc() - tsc;

frequency = tsc \* CLOCKS\_PER\_SEC;

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

tsc = \_\_rdtsc();

tsc = \_\_rdtsc() - tsc;

double currDiff = (double(tsc) / frequency) \* 1e9;

if (currDiff < minDiff) {

minDiff = currDiff;

}

}

}

else {

for (int currentSize = step; currentSize < maxLenArray; currentSize += step) {

FuncMeter\* meter = getMeterByType(type, currentSize, true);

currNs = meter->meter() \* 1e9;

if (prevNs < 0) {

prevNs = currNs;

continue;

}

long long diff = abs(currNs - prevNs);

if (diff < minDiff && diff != 0) {

minDiff = diff;

}

prevNs = currNs;

delete meter;

}

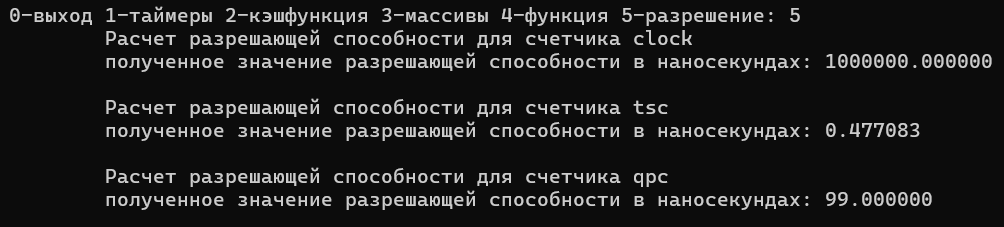
}

cout << fixed <<"\tполученное значение разрешающей способности в наносекундах: " << minDiff << "\n\n";

}

}

Лог результатов измерения разрешающей способности:



# 4. Исследование повторяемости результатов измерения

В ходе исследования повторяемости результатов, в первом варианте исследования были получены слишком малые значения, связанные с ошибкой дискретизации, что привело к погрешности, это произошло из-за малого количества тактов. Во втором варианте было принято решение увеличить объем рабочей нагрузки, добившись не менее 100 тактов, таким образом мы минимизировали погрешность организации эксперементов. Результаты представлены на скриншоте.

Для любого средства измерения повторяемость результатов в одних и тех же условиях является важнейшим свойством. При нескольких операциях измерения времени выполнения одного и того же блока CODE с одними и те ми же обрабатываемыми данными различающиеся результаты измерений получаются по нескольким основным причинам:

1) Ошибки дискретизации интервалов времени, связанные с тем, что все счетчики времени являются целочисленными.

2) Влияние фоновой нагрузки – диспетчер процессов может переключить CPU на реализацию другого процесса в интервале между стартовой и финишной засечек времени исполнения блока CODE. Если используемый счетчик тактов timeFunc считает не время процесса, а время, протекающее либо в ядре CPU, либо во всем компьютере, то в разницу между двумя засечками попадет время исполнения другого процесса.

3) Влияние кэш-промахов – первые обращения к обрабатываемым данным и обращения к тем данным, которые долго не использовались, связаны с кэш-промахами, обработка которых занимает много времени (происходит подкачка данных в кэш-память). Чем интенсивнее повторное использование одних и тех же участков памяти, тем меньше времени тратится на обращение к ним.

4) Влияние ветвлений на время конвейерной обработки потока команд в CPU. Если алгоритм предсказания переходов ошибся в своем предсказании, то значительная часть промежуточных результатов, полученных в ходе конвейерной обработки, будет аннулирована и повышение производительности за счет конвейерного распараллеливания окажется низким. Если алгоритм предсказания перехода не ошибся, то производительность конвейерного распараллеливания будет высокой; алгоритмы базируются на статистике, формируемой внутри CPU и в различные акты измерения эти алгоритмы могут действовать по-разному.

5) Влияние механизмов изменения частоты процессора как в сторону увеличения (Turbo Boost), либо снижение из-за перегрева.

Для анализа повторяемости нужно организовать серии измерении, в каждой из которых выполняется абсолютно одинаковая обработка данных. Повторяемость характеризуют среднеквадратичное отклонение и разница между наибольшими, наименьшими и средним значениями.

Для исследования повторяемости результатов измерения используется следующая формула подсчета:

*ai / ln*(*1-xi*) + *bi* \* *ln*(*1+xi*)

В программном представлении функция примет следующий вид

double calc(double x, double a, double b) {

return a / log(1-x) + b \* log(1+x);

}

Для измерения интервалов времени выполнения функции были созданы массивы X, A, B, C и следующий базовый объект, у которого будет переопределяться метод meter:

vector <double> X, A, B, Y;

void fill(int size) {

X.resize(size);

A.resize(size);

B.resize(size);

Y.resize(size);

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

int r = rand();

X[i] = 1.0 \* r / RAND\_MAX;

r = rand();

A[i] = 1.0 \* r / RAND\_MAX;

r = rand();

B[i] = 1.0 \* r / RAND\_MAX;

r = rand();

Y[i] = 1.0 \* r / RAND\_MAX;

}

}

class FuncMeter : public Meter {

public:

bool div;

FuncMeter(int size, bool divByFreq) : Meter(size) {

// init(size);

this->div = divByFreq;

}

double calc(double x, double a, double b) {

return a / log(1-x) + b \* log(1+x);

}

~FuncMeter() {

}

};

Объекты измерения, переопределяющие метод meter

class FuncQPCMeter : public FuncMeter

{

public:

FuncQPCMeter(int size, bool div) : FuncMeter(size, div) {}

double meter() {

LARGE\_INTEGER t\_start, t\_finish, freq;

\_int64 t\_code;

QueryPerformanceFrequency(&freq);

QueryPerformanceCounter(&t\_start);

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

Y[i] = calc(X[i], A[i], B[i]);

}

QueryPerformanceCounter(&t\_finish);

t\_code = t\_finish.QuadPart - t\_start.QuadPart;

return double(t\_code) / (div ? freq.QuadPart : 1);

}

};

class FuncClockMeter : public FuncMeter

{

public:

FuncClockMeter(int size, bool div) : FuncMeter(size, div) {}

double meter() {

int count = 1;

unsigned long long t\_start, t\_end;

double t\_ns, t\_diff;

\_int64 t\_code;

t\_start = clock();

for (int cnt = 0; cnt < count; ++cnt) {

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

Y[i] = calc(X[i], A[i], B[i]);

}

}

t\_end = clock();

t\_code = double(t\_end - t\_start) / count;

return double(t\_code) / (div ? CLOCKS\_PER\_SEC : 1);

}

};

class FuncTSCMeter : public FuncMeter

{

public:

FuncTSCMeter(int size, bool div) : FuncMeter(size, div) {}

double meter() {

unsigned long long tclock, tsc, frequency, t\_start, t\_end;

double t\_diff, t\_ns;

\_int64 t\_code;

tclock = clock();

while (clock() < tclock + 1) {} // ожидание завершения первого clock-интервала

tsc = \_\_rdtsc(); // значение TSC в начале второго clock-интервала

while (clock() < tclock + 2) {} // ожидание завершения второго clock-интервала

tsc = \_\_rdtsc() - tsc; // сколько тактов TSC прошло за один такт clock

frequency = tsc \* CLOCKS\_PER\_SEC;

t\_start = \_\_rdtsc();

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

Y[i] = calc(X[i], A[i], B[i]);

}

t\_end = \_\_rdtsc();

t\_code = double(t\_end - t\_start);

return double(t\_code) / (div ? frequency : 1);

}

};

Далее были написаны функции замеров с помощью Log

int lenArray = 1000000;

vector <string> counterType = { "clock", "TSC", "QPC" };

FuncMeter\* getMeterByType(int type, int size, bool divByFreq = true) {

if (type == 0) return new FuncClockMeter(size, divByFreq);

if (type == 1) return new FuncTSCMeter(size, divByFreq);

if (type == 2) return new FuncQPCMeter(size, divByFreq);

}

void func(int type) {

int lenArray = 1000000;

fill(lenArray);

cout << "\n\tПроводим измерение времени с помощью метода " << counterType[type];

FuncMeter\* metr = getMeterByType(type, lenArray, true);

Log log;

log.config({ {FILTR\_MAX, 0} });

double scale = MCS\_IN\_SEC; // Значения будем выводить в микросекундах

int nPasses = 3; // число проходов при выполнении серий измерений

cout << "\n\tПодсчет функции" << endl;

for (int n = 1; n <= nPasses; n++) {

cout << "\n\tСерия " << n << ". ";

log.series(true, 100, metr)

.calc().stat(scale, "микросекунды")

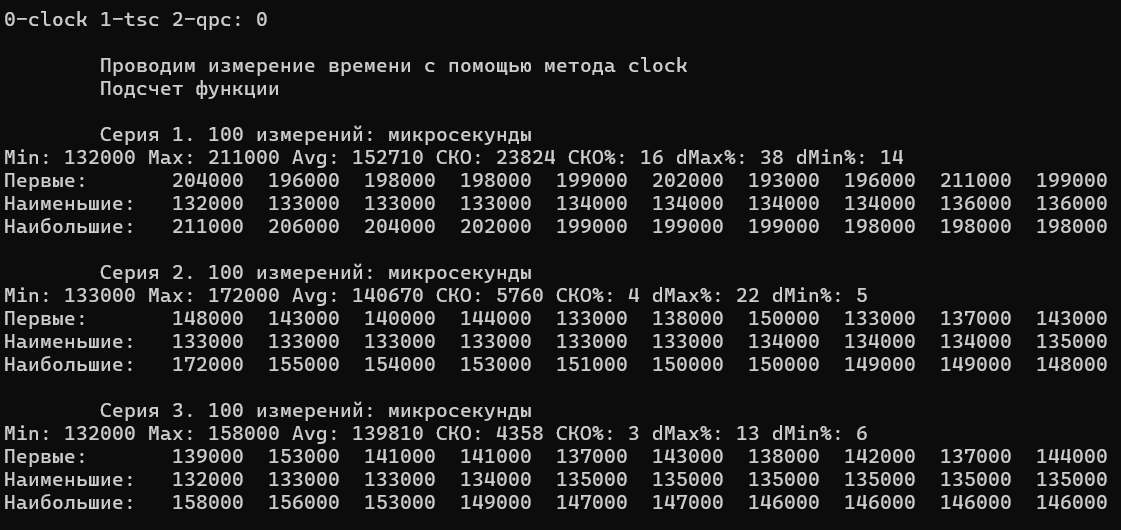
.print(scale, 10);

}

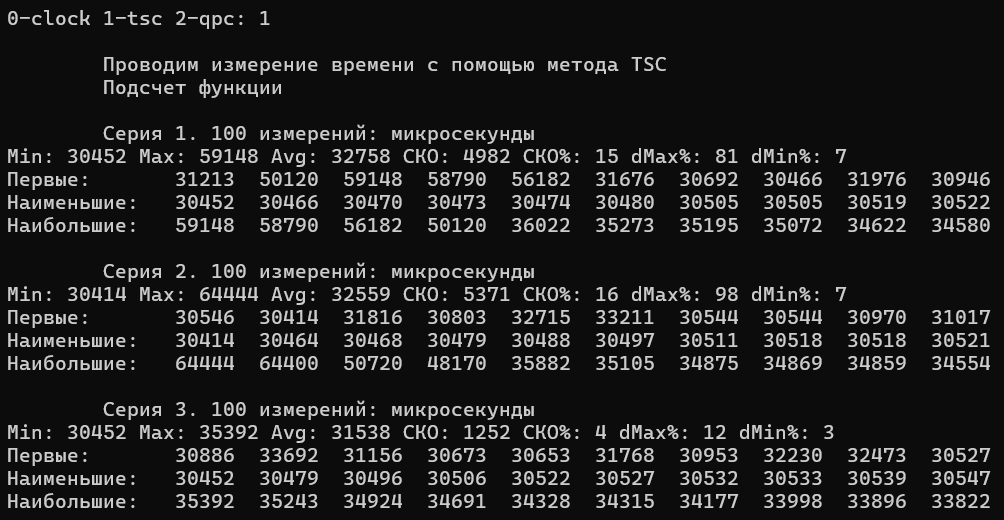
}

Данная функция проводит 3 серии измерений по 100 повторений. В результате работы программы. Были получены следующие логи:

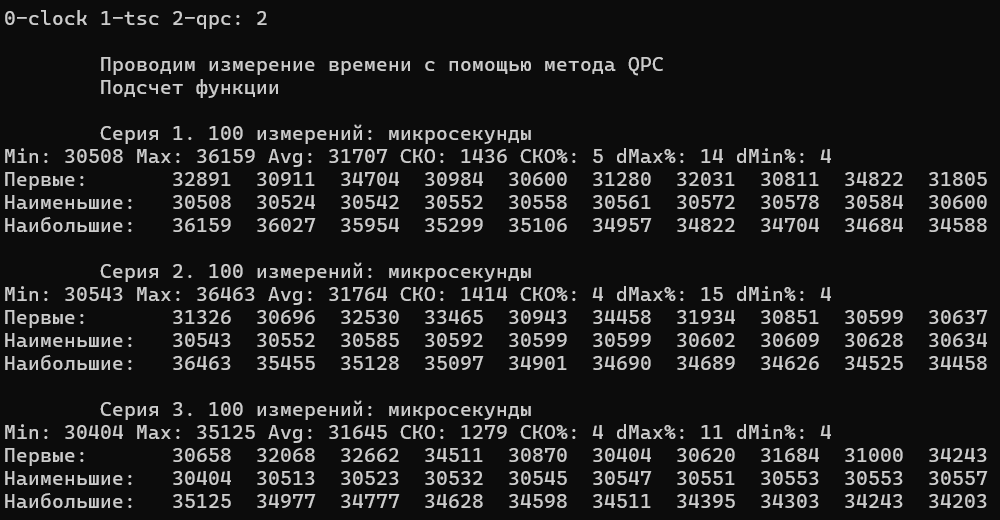
Лог замеров для Clock



Лог замеров для TSC



Лог замеров с помощью QPC



По итогу видно значительное превышение максимального значения времени над средним для всех трех счетчиков. Это объясняется тем, что первое обращение к рабочей нагрузке сопровождается большим количеством кэш-промахов, ведь в кэше эти данные будут отсутствовать, и потому данные будут браться из памяти. Однако уже после первого прохода рабочая нагрузка загрузится в кэш, откуда данные доставать намного быстрее, нежели из памяти.

# 5. Исследование ошибок оценки затрат времени по эмпирической формуле с линейной зависимостью времени от размера рабочей нагрузки

Линейная зависимость времени от объема рабочей нагрузки выражается формулой T = k \* w, где w – объем рабочей нагрузки (например: количество элементов обрабатываемого массива, количество обращений к функции, количество итераций при реализации рекуррентных формул), k – коэффициент пропорциональности. Вместо многократных экспериментальных измерений T при разной рабочей нагрузке можно использовать формулу, зная величину k. Ответ на вопрос «Насколько это возможно?» зависит от того, насколько приемлемы ошибки такой замены.

Для оценки погрешностей аналитического процесса относительно реального необходимо организовать серию измерений при разных значениях рабочей нагрузки с получением ряда значений (T(0), T(Δw), T(2\*Δw), T(3\*Δw),…), где Δw – шаг изменения рабочей нагрузки. Затем нужно оценить значение коэффициента пропорциональности либо путем линейной аппроксимации на основе полученного ряда значений, либо путем деления k = T(N\* Δw ) / N при достаточно большом N. Затем нужно построить ряд отклонений вида (T(0) - 0, T(Δw) – k\*Δw, T(2\*Δw) – k\*2\* Δw, T(3\*Δw) – k\*3\* Δw, …), где уменьшаемые берутся из исходного ряда результатов измерения, а вычитаемые получаются по эмпирической формуле. Полученный ряд отклонений нужно подвергнуть статистической обработке для получения минимума, максимума, среднего и среднеквадратичного отклонения.

Для анализа линейности была использована следующая функция

void lineFuncDiff(int maxArray, int filtrMin, int filtrMax, int type) {

Log log;

FuncMeter\* metr;

int dW = maxArray / 1000;

cout << "\n\tАнализ линейности T(n) при линейно-нарастающей рабочей нагрузке"

<< " от " << dW << " до " << maxArray << endl;

// Сначала вычисляем коэффициент для T(n) = coeff \* dW

metr = getMeterByType(type, maxArray, false);

// Серия измерений с фильтрацией и получением среднего значения

cout << "\n Оценка значения коэффициента пропорциональности при maxArray = " << maxArray << endl;

double scale = 1.0; // измеряем в тактах TSC, поэтому 1.0

log.series(true, 100, metr).config({ {FILTR\_MIN, filtrMin}, { FILTR\_MAX, filtrMax } })

.calc().stat(scale, "такты");

double coeff = log.avg / maxArray; // а вот и сам коэффициент

cout << "coeff = " << \_\_int64(coeff)

<< " тактов/один\_элемент\_массива" << endl;

// Для каждого измеренного вычисляем dT = Tизм(n) - coeff \* (n \* dW)

Log logDiff; // Протокол для Tизм - T(n)

FuncMeter\* pfm = NULL;

cout << "\n\tОтклонения измеренного времени от T(n) = coeff \* n \* dW при dW =" << dW << endl;

for (int n = 1; n <= maxArray / dW; n++) {

int size = n \* dW;

try {

pfm = getMeterByType(type, size, false);

}

catch (const exception& e)

{

std::cout << e.what() << "\n";

}

log.series(true, 100, pfm).calc();

logDiff.set(log.avg - coeff \* n \* dW);

cout << "\rn = " << n << " size = " << size << " Tизм = " << log.avg << " тактов";

pfm->~FuncMeter();

}

cout << endl;

logDiff.calc().stat(scale, "такты")

.print(O\_NATURE, scale, 10)

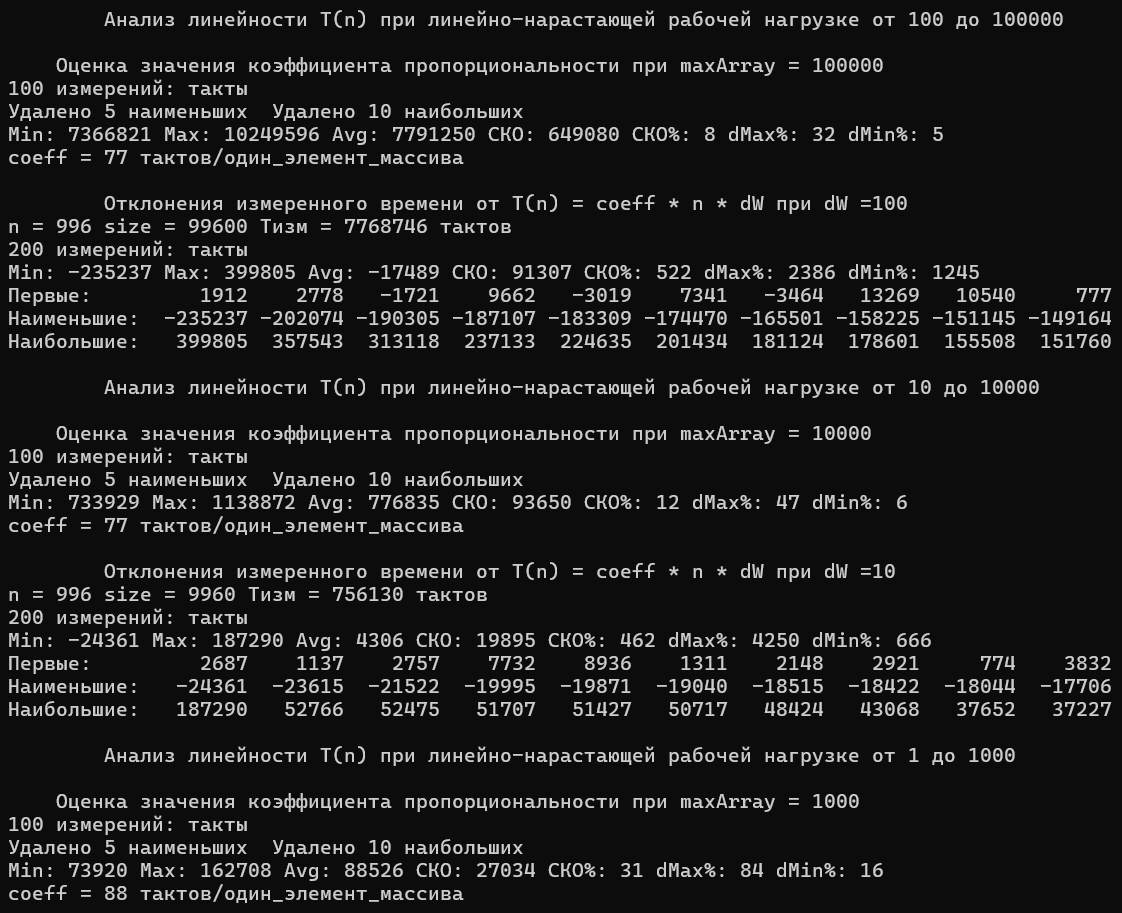
.print(O\_MIN, scale, 10)

.print(O\_MAX, scale, 10);

}

При исследовании отбрасывались 10 наибольших и 5 наименьших по времени замеров в серии из 100 замеров.

Лог для счетчика TSC:



По полученным результатам также построен график. Оранжевым выделены ожидаемые результаты, синим – фактические.

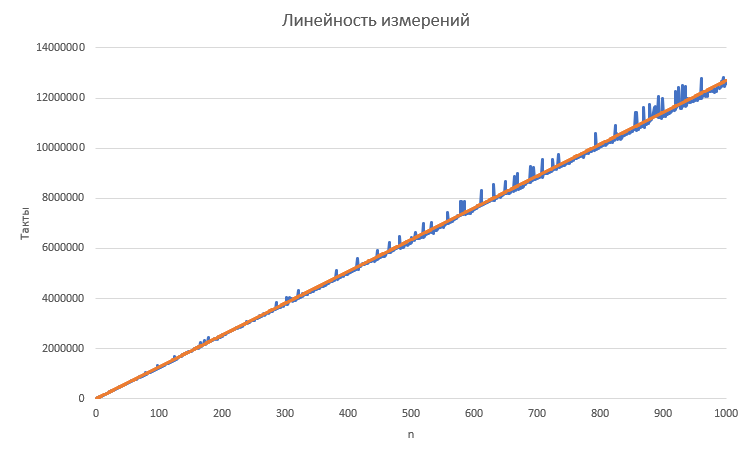


Рис.3 График тактов

По результатам анализа видно, что результаты измерений имеют пикообразные отклонения от линейного закона. Это происходит из-за кэш-промахов и ошибок алгоритма предсказания.

Далее был проведен тот же самый тест, но с фоновой нагрузкой (был открыт браузер Chrome с видеорядом, график результатов следующий)



Рис.3 График тактов с фоновой нагрузкой

По графику видно, что при фоновой нагрузке количество и амплитуда скачков возрастает в разы по сравнению с обычным выполнением. Связано это с постоянным переключением потоков процессора на фоновые задачи.

Теперь проведем исследование без фоновой нагрузки, но с фильтрацией 80 максимальных элементов. В итоге график примет следующий вид.



С изменением фильтрации фактический и прогнозируемый результат практически совпали. Это связано с тем, что вызовы функций большим количеством кэш-промахов и ошибок предсказания были отфильтрованы и не учитывались при построении графика.

# Вывод

В процессе выполнения лабораторной работы были изучены средства измерения затрат времени исполнения кода. Были проанализированы разрешающие способности средств измерения и повторяемость результатов измерений, проведено исследование ошибок оценки затрат времени по эмпирической формуле с линейной зависимостью времени от размера рабочей нагрузки.

# Приложение. Измерение времени выполнения функции Фибоначчи

#include <iostream>

#include <stdio.h>

#include <Windows.h>

using namespace std;

unsigned long long fib(unsigned \_\_int64 n) {

if (n < 2) return n;

return fib(n - 1) + fib(n - 2);

}

unsigned long long check\_clock(int n, int reps) {

unsigned long long t\_start, t\_end;

double t\_ns, t\_diff;

printf\_s("clock (n = %d, reps = %d)\n", n, reps);

int a = 0;

a = fib(n);

t\_start = clock();

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

a = fib(n);

}

t\_end = clock();

t\_diff = double(t\_end - t\_start) / reps;

t\_ns = t\_diff / CLOCKS\_PER\_SEC \* 1e9;

printf\_s("\tTime ns: %f\n", t\_ns);

printf\_s("\tTime s: %f\n", t\_ns / 1e9);

printf\_s("\tTicks: %lf\n", t\_diff \* reps);

return a;

}

unsigned long long check\_tsc(int n, int reps) {

unsigned long long tclock, tsc, frequency, t\_st, t\_end;

double t\_diff, t\_ns;

int a = 0;

a = fib(n);

printf\_s("tsc (n = %d, reps = %d)\n", n, reps);

tclock = clock();

while (clock() < tclock + 1) {} // ожидание завершения первого clock-интервала

tsc = \_\_rdtsc(); // значение TSC в начале второго clock-интервала

while (clock() < tclock + 2) {} // ожидание завершения второго clock-интервала

tsc = \_\_rdtsc() - tsc; // сколько тактов TSC прошло за один такт clock

frequency = tsc \* CLOCKS\_PER\_SEC; // частота процессора

t\_st = \_\_rdtsc();

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

a = fib(n);

}

t\_end = \_\_rdtsc();

t\_diff = double(t\_end - t\_st) / reps;

t\_ns = (t\_diff / frequency) \* 1e9;

printf\_s("\tTime ns: %f\n", t\_ns);

printf\_s("\tTime s: %f\n", t\_ns / 1e9);

printf\_s("\tTicks: %lf\n", t\_diff \* reps);

return a;

}

unsigned long long check\_qpc(int n, int reps) {

LARGE\_INTEGER t\_st, t\_end, frequency;

double t\_diff, t\_ns;

printf\_s("qpc (n = %d, reps = %d)\n", n, reps);

int a = 0;

a = fib(n);

QueryPerformanceFrequency(&frequency);

QueryPerformanceCounter(&t\_st);

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

a = fib(n);

}

QueryPerformanceCounter(&t\_end);

t\_diff = double(t\_end.QuadPart - t\_st.QuadPart) / reps;

t\_ns = (t\_diff / frequency.QuadPart) \* 1e9;

printf\_s("\tTime ns: %f\n", t\_ns);

printf\_s("\tTime s: %f\n", t\_ns / 1e9);

printf\_s("\tTicks: %lf\n", t\_diff \* reps);

return a;

}

#define N1 10

#define N2 40

#define REP\_1 10000000

#define REP\_2 5

int main() {

check\_clock(N1, REP\_1);

check\_tsc(N1, REP\_1);

check\_qpc(N1, REP\_1);

check\_clock(N2, REP\_2);

check\_tsc(N2, REP\_2);

check\_qpc(N2, REP\_2);

}