UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Wall-time based performance assessment and comparison framework

Author:
Yurii LABA

Supervisor:
Oleg FARENYUK

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

APPLIED SCIENCES FACULTY

Lviv 2019

Declaration of Authorship

I, Yurii LABA, declare that this thesis titled, "Wall-time based performance assessment and comparison framework" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Wall-time based performance assessment and comparison framework

by Yurii LABA

Abstract

In this study, we present the framework to analyze and compare the performance of different parts of code. The work of the framework is based on the wall-time measurements and statistical analysis of the obtained results. Wall-time is most relevant metrics of code efficiency but difficult to measure reliably so statistical analysis was used to refine and check results. This framework was used to analyze performance of various implementations of threads and mutexes on several different platforms with x86 and ARM CPUs. In the future, the framework will be used to study other low and high-level concurrent APIs. We are currently working on performance of the thread-safe queues.

Acknowledgements

I would like to thank Oleg Farenyuk who helped me to solve trivial and non-trivial tasks. Also, I want to thank Rostyslav Hryniv, who helped with statistical researches and Oles Dobosevych, who helped to build the work process.

Contents

List of Figures

List of Tables

List of Abbreviations

CPU	**C**entral **P**rocessing **U**nit
RAM	**R**andom **A**ccess **M**emory
ISA	**I**nstruction **S**et **A**rchitecture
CISC	**C**omplex **I**nstruction **S**et Computer
RISC	**R**educed **I**nstruction **S**et **C**omputer
OS	**O**peration **S**ystems
RAII	**R**esource acquisition is initialization

Dedicated to my family

Chapter 1

Introduction

The data volumes to process and the complexity of the corresponding algorithms are constantly increasing. At the same time, the performance of the single processor (CPU) in the last decade almost does not increase [*1*, Hennessy, 2013]. Today, instead of increasing productivity of the CPU, the number of the CPUs increases – dual-core, quad-core, or larger. Therefore, one of the main ways of effective solution of large-scale tasks is the use of parallel computing technologies[Boyd-Wickizer et al., 2010]. When the task is performed sequentially, only one core is used, while others are idle. Parallel computing allows to speed up the solution of computational tasks, maximizing the use of resources of computing systems, in particular - processors.

Today concurrency is used in various spheres. For example, in the cloud compute instances – containers of the resources – are small. So, a little web app is basically a concurrent app. If it designed well, you can increase the number of servers to work with more customers or to change your server to more efficient which can lead to loss of information, databases, etc. To create an efficient desktop app, you also need to deal with concurrency because every CPU is dual, quad or more cores now. The same story with mobile development, the latest flagships have dual or more cores too. Xbox 360, Sony's PS4 are also multicore systems,

To talk about the effectiveness of the application of certain methods of increasing productivity and optimization, we need somehow measure the increase in efficiency. One of the main performance metrics is latency – time to perform a specific task [Hennessy, 2013]. The best criterion is the so-called wall-time – the real execution time measured independently of the clock processor[Hennessy, 2013]. More accessible but less relevant is processor time – the number of processor counts – the number of executed commands, the number of accesses to memory, cache misses, etc.

Due to the importance of the measurement of productivity, there are many relevant tools - profilers, APIs to access processor performance counters and operating systems counters. Profiles such as perf (Linux) [*3*], MS Windows Performance Toolkit (Windows) [*4*], VTune (Windows, Linux) [*5*], allow getting runtime and other metrics for individual functions or even smaller parts of the program, giving recommendations for improving programs. APIs to access counters, such as the Performance Application Programming Interface (Linux) [Dongarra et al., 2001, Terpstra et al., 2010], MS Windows Performance Counters [*6*], Processor Counter Monitor (Linux, Windows) [*7*], allows measurements to be made directly in the code.

However, because of the great complexity of modern computer systems, measurement analysis itself is a complex task – computational times are not directly reproducible. This is due to many factors: multitasking at the operating system level [A. Tanenbaum, 2014, Durbhakula, 2018], technologies for increasing the efficiency of the CPU, such as memory cache, reordering commands, etc., energy saving technologies such as dynamic clock rate changes, such as Intel Turbo Boost[*9*], etc.

Although the problem is recognized among researchers [Hennessy, 2013, *10*, P. Fleming, 1986], there is no satisfactory and universal approach.

We proposed a methodology for comparing the efficiency of various programming languages constructs or different APIs based on statistical analysis. This methodology has been applied to compare the efficiency of different APIs to create threads, synchronize between threads and share data between threads.

1.1 Main definitions

1.1.1 What is the CPU?

One of the main components of the computer is the CPU. The essence of the CPU's work is that it receives the instructions and needed data from the memory and/or input-output devices. Then CPU based on received input executes the instruction and stores the output somewhere. Figure 1.1 shows the main parts of the CPU: CU – control unit, ALU – arithmetic logic unit, and registers.

Now let's take a look at what kind of work each component performs — the task of CU it to do the fetching and decoding of the instructions. The ALU performs all arithmetic and logical operations - executes the instructions. After it, the result is written to a specific location. Registers are another essential part of the CPU. ALU and CU stores needed to execute the instruction data in registers. Register - is a very fast memory within the processor. Various kinds of registers are in the CPU, one of the most important – program counter, which stores the following instruction to execute on the processor.

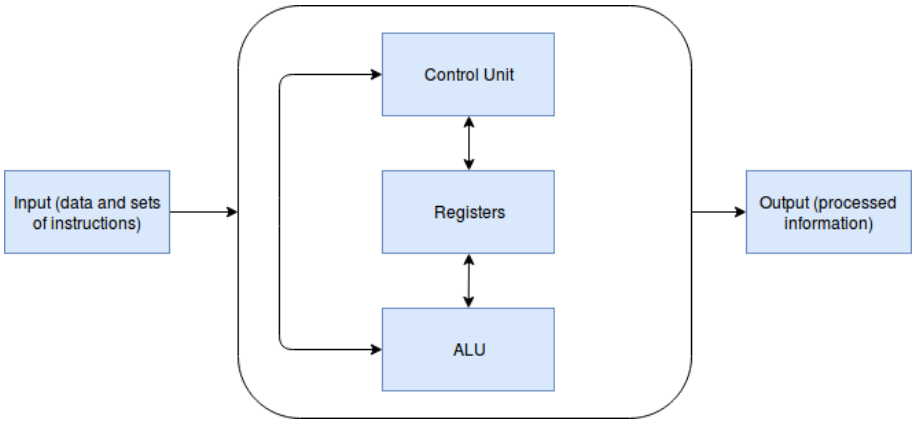


FIGURE 1.1: The main parts of the CPU

1.1.2 What affects the performance of the CPU?

CPU performance is determined by several factors. One of the most important is the number of the cores in CPU. Each core has CU, ALU, and registers so CPUs with multiple cores can execute multiple programs simultaneously. But cores need to communicate with each other; it takes some time, so an increasing number of cores does not mean increasing in productivity in the same ratio.

Another factor which affects the CPU performance is the speed of the core – CPU frequency. This value indicates how many instruction cycles the CPU can handle within a second. For example, if core speed is equal to 3 MHz, this means that

CPU performs three million cycles per second. It is important to understand that the number of cycles executed per second is not equal to the number of instructions executed per second. Per one cycle CPU can fetch the instruction and decode another instruction but don't execute any instruction. And we should keep in mind, that there are limits to how fast a CPU can be. Often frequency changes dynamically depending on what tasks are running. For example, Intel realized this feature (called Intel Turbo Boost) in the Core i5, Core i7, Core i9, and Xeon series. Also, CPU frequency can be increased via BIOS, which theoretically means that the computer will work faster. But if the CPU executes the instructions too quickly, this means that data can be corrupted. Also, the CPU can overheat if it is working faster than it was designed to work.

Memory is also significant for CPU performance. CPU directed by instructions can command I/O subsystem to load data from HDD to RAM but this requires a relatively long time. RAM helps to accelerate the process of obtaining data, it stores the data that are currently being used.

But RAM speed is not enough for modern processors. There is another type of memory, faster than RAM – cache memory, extremely fast memory. its speed is much close to the speed of the CPU than RAM. Cache memory can temporary store instructions and data that is usually used by CPU.

There a few levels of cache memory. First level, L1, is the smallest and fastest to access; it's usually the part of the CPU chip. Second level, L2, and third level, L3, caches are bigger than L1, CPU need more time to access them. These caches are located between the CPU and RAM.

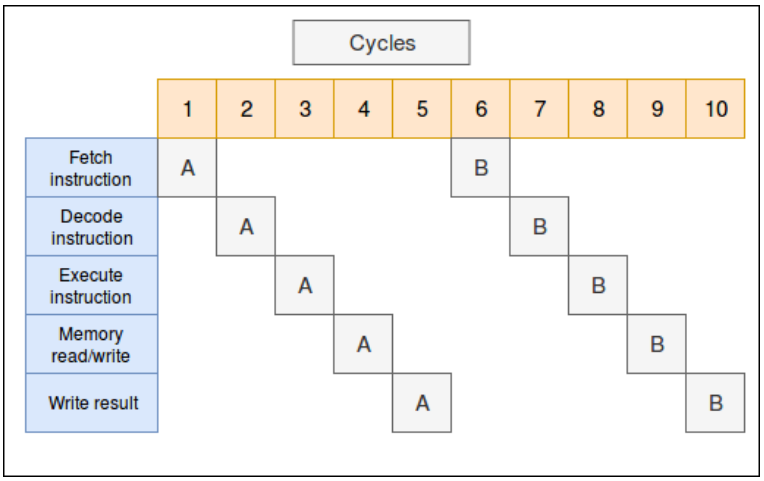
Each CPU core has its own L1 cache but may share L2 and L3 caches. As L1 (and sometimes L2 or even L3) is always personal for each CPU core, the concept of cache coherence can also affect the work of the CPU. Cache coherence is the situation when a few cores work together, and the same memory is in their caches, and both want to write/read to it. The algorithms for solving such situations are called cache negotiation protocols (MESI[Papamarcos and Patel, 1984] and MOESI[Papamarcos and Patel, 1984]).

Also, CPU performance depends on the type of CPU instruction set architecture (ISA). There are two main approaches to create an ISA – complex instruction set computer (CISC) and reduced instruction set computer (RISC). CISC often has much more instructions which are less orthogonal than RISC instructions, so to perform a complex task RISC needs more instructions but it is easier for them to reach large clock speed. Currently, the only common CISC architecture is x86, and a classic c example of RISC architecture is ARM.

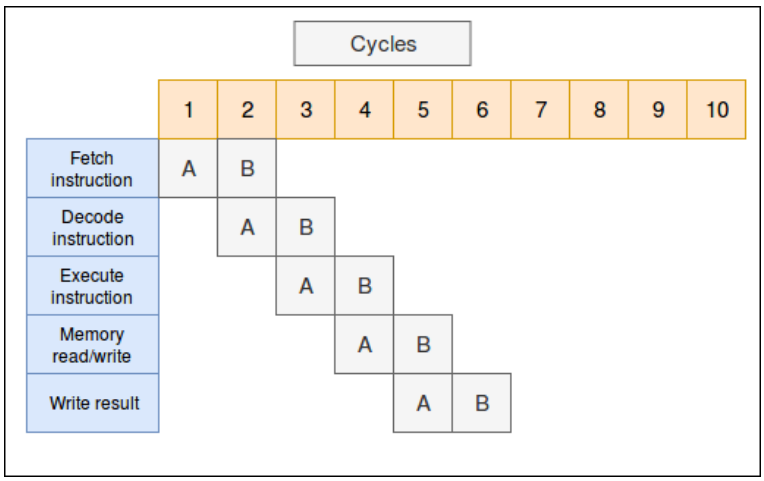
Instruction pipelining, superscalar architecture, branch prediction, out-of-order execution are used to achieve an efficient implementation of both RISC-type or and CISC-type CPUs.

Instruction pipeline is crucial to achieving high efficiency in modern CPU. The CPU process the instructions in a several steps. For example, it can be: fetch an instruction, decode it, execute it, access memory, write back the results. In the figure 1.2a CPU process each instruction from start to end without a pipeline. It will take ten cycles to process two instructions. Modern processors achieve higher performance using instruction pipelines. Because the different section of hardware can handle different steps of processing the instruction, it allows processing a separate instruction simultaneously. Figure 1.2b shows the execution of instruction with the pipeline. In one instruction cycle, a processor do fetching of instruction B and in

the same time decoding instruction A. With a pipeline it will take six cycles to process two instructions. To conclude, the strength of the pipeline is that it increases throughput without affecting the latency.



(A) Execution without pipeline



(B) Execution with pipeline

FIGURE 1.2: Comparison of execution instructions with and without a pipeline

In a superscalar architecture, several instructions executes simultaneously during a single clock cycle by multiple instruction pipelines.

Out-of-order execution is another essential thing for CPU performance. All modern processors have this feature. The essence of this technique is that the processor allows commands to be executed not in order until it affects the final result.

Branch prediction is used to help efficiency feed the pipeline with instructions and out-of-order execution allows unrelated instructions execution while waiting for data from memory or cache for other.

Branch prediction is one of the techniques CPU's implements which tries to guess result of the jump instruction and load corresponding instructions to provide work for pipeline. When branch is mispredicted, pipeline should be flushed. Branch predictors are very important in modern processor to achieve higher performance. There are a few types of prediction: static and dynamic. The static prediction does not depend on the previous commands story. It can be predetermined in processors,

or the compiler gives the hints to the processor. The static prediction is simple in realization, however not very effective — maximum efficiency - 75-80 percentages. The dynamic prediction depends on the previous history of execution. The dynamic predictors can use counters, two-way prediction, etc. There are also hybrid variants of predictors: cyclic predictors, return from function predictors, etc. There are also architectures that use the neural network to predict branches, for example, AMD Rysen.

Processor with the help of OS can even execute several programs on one core. This is achieved by frequent switching between tasks - preemption. Preemptive scheduler is a part of the system which can interrupt tasks and after a while continue them, keeping the execution state.

1.1.3 Process

Modern operation systems (OS) represent programs to be executed using the abstraction of **process**. The process is a computer program example, which has its own set of resources – code, data, registers, address space, etc. The process consists of the image of the program – a machine code that will be executed, process memory (often virtual), which includes code and data of the process, its stack, and a heap. The stack is a part of the process, where all variables, objects, etc that are defined in the process are set. The heap is dynamic memory; you can allocate a block and free it at any time. Also, the process consists of operating system descriptors associated with the process: file descriptors, windows, etc., permissions of the process, including its owner and the state (context) of the processor – in the first place, the contents of its registers. One of the most important registers is the program counter, that shows which instruction CPU has to execute next.

The system with multiple cores or CPUs can run multiple processes simultaneously. Also, several processes can work on single CPU. Pseudo-parallel execution will be achieved by frequent switching of the CPU between processes creating the illusion that threads are running in parallel. There is only one physical counter of commands, so when starting each process, its logical counter of commands is loaded into a real counter.

Processes are isolated from each other, because each of them has its own memory address space. It cannot directly access shared data in another process. Independence between processes means that switching between them takes a relatively long time: to save and load registers, memory maps, etc., but it's worth it. One process cannot corrupt another, which means that when you have a problem with the process (working application), you can close it without affecting other processes.

1.1.4 Thread

The process is executed by the thread. The thread has a counter of commands, which monitors what kind of instruction is to be executed. It has registers that contain current variables. Although the thread can be performed within the process, they are different concepts. The processes are used to group resources into a single entity, threads – for execution on a central processor.

One of the important features of the threads is that the time required to create them is much less time consuming to create processes (10-100 times faster). When the threads work within the same CPU, for CPU-bound tasks – when significant computations are performed, they do not bring any performance gains, but when a

significant amount of time is spent on I/O waiting, the threads allow to overlap in time, accelerating the work of the application.

By implementation threads can be divided into kernel threads and user threads. The difference between them is that that the first are implemented within the kernel of the OS, and others are implemented in the user-space.

Several threads can be executed within single process. Because of all threads within the process have the same address space, communication between them can be very fast, which is a significant advantage. But there is also a significant disadvantage – threads can easily affect the work of each other. For example, a situation may easily occur when one executing thread writes a value into a variable and another attempts to read it at the same time. Such a situation is called a data race and may cause a lot of bugs. To avoid this and similar situations, threads need to be synchronized using various methods. OS and libraries of concurrent-ready programming languages provides different synchronization tools.

1.1.5 How to synchronize threads?

As threads within the same process have the same address space, they have to be synchronized when accessing shared resources. In concurrency, there is a problem called race condition. In terms of C++11, when two or more threads are running and simultaneously refer to the same variable, and one of this thread write something in this variable – congratulations, you have a race condition, and in that case, you'll have undefined behaviour. Read/write operations aren't atomic. Operation is called an atomic if it can't be interrupted and the other threads can't see its intermediate result, operation "var += 1" isn't atomic by default. But atomicity can be achieved using different synchronization primitives. Atomicity of very complex operation can be achieved, but what's the price for it?

The standard C++ library provides explicit synchronization tools. Among them, the most important is the mutexes. Mutexes allow organizing exclusive (serialized) and synchronous access to shared data. The simplest class of mutexes in C++ is std::mutex. Class std::mutex has a few methods to work with mutex: lock(), try_lock() and unlock(). lock() acquires the mutex. If the mutex is blocked, it blocks execution until it succeeds in acquiring it. Does not return anything. try_lock() – tries to acquire the mutex. In any case, the return takes place immediately – the thread is not blocked. If it managed to acquire the mutex – returns true, otherwise – false. unlock() – method releases (unlocks) the mutex.

Also, there is one crucial thing to know – if in the process of running the code between lock() and unlock() will throw an exception, the mutex will remain blocked – there will be a so-called deadlock. Of course, we could use try-catch(...) constructions and there release the mutex. But such decisions are bulky and vulnerable to mistakes. C++ provides corresponding RAII wrappers – std::lock_guard, std::unique_lock and so on.

Another way of synchronization is conditional variables. A conditional variable allows the thread to notify another thread, that there is a work for it and this thread will wake up. When you implement the conditional variable, you will meet the following situations: the thread will not be awakened in response to the signal-notification and the thread may be awakened even if there wasn't any notification. The second situation is more acceptable so no notification is missed but spurious wake-ups can occur. C++ provides corresponding std::conditional_variable class.

A barrier is another synchronization methods. It has two variable: a variable that tracks whether threads can move further or not, and counter of threads number,

reached the barrier. The barrier waits until all threads reach it and only after it allows them to continue the execution.

Semaphore is another synchronization method. It's an atomic counter which can be within 0 and any positive custom number. Over the semaphore you can perform the following operations: initialize it (set the number), decrease or increase semaphore's number.

As of C++17 standard C++ does not provide barriers or semaphores but third-party libraries do.

Chapter 2

Related works

The theme of code productivity analysis is quite popular among researchers, but there are quite a few official articles. Researches on this theme is usually published on relevant sites but not in official resources.

The aim of the paper called "An Analysis of Linux Scalability to Many Cores" (Boyd-Wickizer et al., 2010) is to discover how well traditional kernel designs allow the different applications to scale. They've tested two types of applications: bad scaled (memcached, Apache serving static files, and multicore MapReduce library Metis) and well-scaled applications that were designed for parallel execution (gmake, PostgreSQL, Exim mail server and Psearchy file indexer). The chosen set of applications stresses essential parts of many kernel components: page cache, process manager, scheduler, memory manager, etc. To avoid execution bottlenecks caused by hard disk writes, authors used tmpfs (Snyder, 1990) filesystem.

Researchers tried to identify and fix kernel implementations bottlenecks which arose during the application testing. For example, they've detected the problem in the Metis application that there were super per-page soft page faults contend on a per-process mutex. This problem was fixed by protecting each super-page memory mapping with own mutex.

Based on the experiments and researches with previously mentioned applications, authors have created a benchmark, called MOSBENCH, to measure the scalability of the operating system. They used it to compare scalability of the latest version of the kernel at the time of writing the article and their improved version of the kernel called PK. In figure 2.1[*12*], presented results of their comparison. Most applications scale better with the improvements, scalability bottlenecks are caused by imperfect load balance, hardware bottlenecks, etc.

In the paper called "Capriccio: Scalable Threads for Internet Services" created by "Von Behren et al., 2003 the researchers realized scalable user-level thread package for use in the servers with high concurrency called capriccio. The package implemented for Linux, it uses POSIX threads API. They included various interesting techniques: linked stack management - minimization of the wasted stack space and resource-aware scheduling - adaptation to the system's current resource usage. Researches also use various ready-made solutions, for example, to efficiently context switches they've used Edgar Toernig's coroutine library, which provides extremely fast context switches.

Researchers compared the results of their implementation with different thread packages. Capriccio implementation shows better results compered to Linux Threads. To study the scalability of the new package, they run single producer-consumer microbenchmark and used condition variables to synchronize threads. Capriccio showed good results, especially on the large numbers of producers/consumers. Input/Output performance is good too, researches conducted pipetest: concurrently pass a number of tokens to fixed number of pipes and measured throughput.

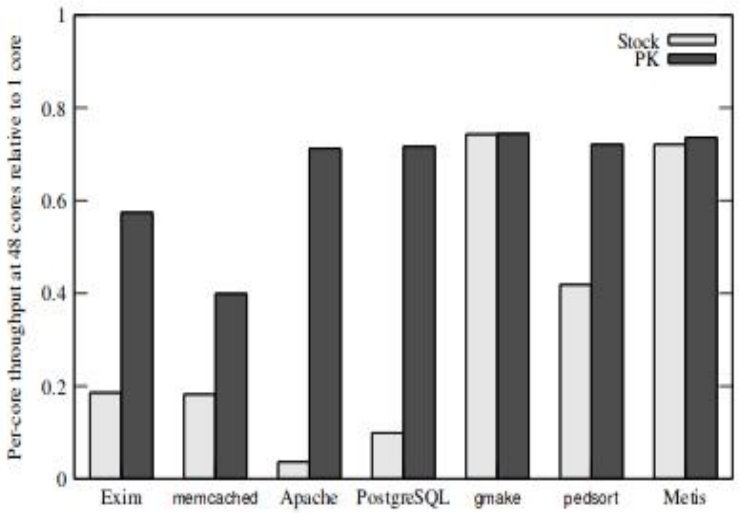
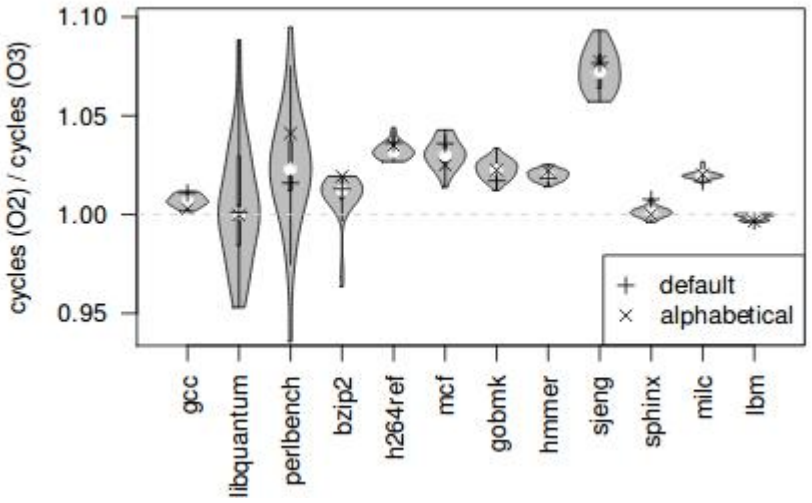


FIGURE 2.1: MOSBENCH results summary (taken from Boyd-Wickizer et al., 2010)

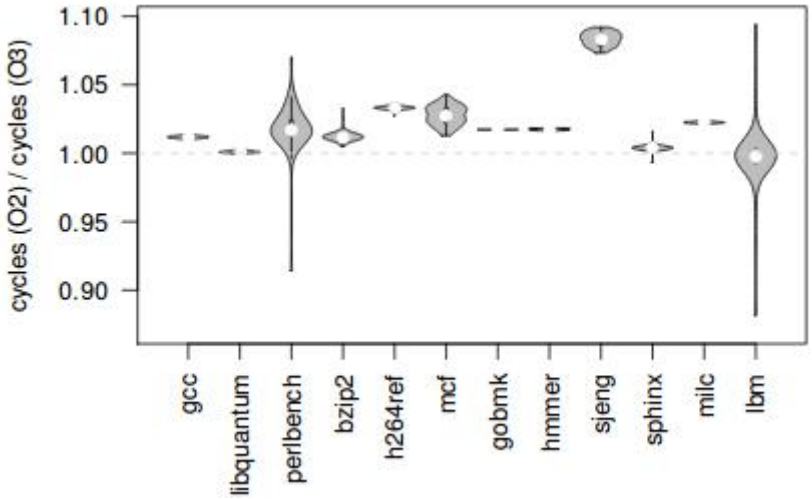
Capriccio package was also applied to Apache 2.0.44 web server and and improved its performance by 15 percent.

Next paper I want to mention is called "Producing Wrong Data Without Doing Anything Obviously Wrong!" created by Diwan, Hauswirth, and Sweeney, 2009. The paper aim is to discover the effect called measurement bias and proof that the performance is often vulnerable to the experimental setup in which the performance is measured. The measurement bias is something external to the program which can affect its performance. For example, UNIX environment. Researchers empirically found that its changes affect the performance. There are many others reasons which may cause measurement bias: room temperature can affect CPU clock speed, previously mentioned UNIX environment size, link order, etc. Authors chose last two to test. In order to detect the measurement bias they used benchmarks from SPEC CPU2006 [13] (gcc, libquantum, perlbench, bzip, gobmk, milc, lbm, etc.). All experiments were conducted using different best practices: ran experiments in a minimal environment, deleted all unused environmental variables, repeat all experiments multiple times to ensure that produced data is valid, etc. All benchmarks are one-thread applications. For the experiments authors used Pentium 4, a Core 2 workstation and m5 simulator. Each benchmark was run 5,940 times.

Authors discovered speed up of O3 optimization over O2 with different link orders. Tests were conducted on the previously chosen benchmarks. In figure 2.2a presented received results. Each violin represents data for all the link orders for one benchmark. The width of y-value is proportional to the number of times y was observed; x-value represents the range of obtained speedup (or speed down). From this figure, we can conclude that the violins result for five benchmarks (libquantum, perlbench, bzip2, sphinx, lbm) is below 1, which means that conclusions about the benefit of the O3 optimization may be false. Data was obtained on Core 2. On the Pentium 4 results are even more interesting: all results are below 1. The same result was obtained for m5 simulator using the O3CPU. There are many reasons which may cause this phenomenon: link order can affect the alignment of code, causing problems within different hardware buffers, cause conflict misses in the instruction cache, etc.



(A) Link order



(B) UNIX environmental size

FIGURE 2.2: Received bias in Intel's C compiler on Core 2

Figure 2.2b represented results of discovering the effect of changing the UNIX environment size on the speedup of O3 over O2 across all previously mentioned benchmarks. From this figure, we can conclude that the violins result for four benchmarks (libquantum,perlbench,sphinx, lbm) are less than 1, which means that conclusions about the benefit of the O3 optimization may be false in this case too. This is the data obtained on Core 2. On the Pentium 4 results are also interesting: outcomes of six benchmarks are below 1. In this case, measurement bias can be caused because of affecting the starting address of the C stack by changing UNIX environmental variable. The second reason, which applies only to perlbench benchmark, is that it copies contents of the UNIX environment to the heap, thus changing the UNIX environmental variable can change the alignment of heap-allocated structures in various hardware buffer.

To conclude, in the comparison between two systems, measurement bias arises when experimental setup on one of the system is better than another. So measurement bias can be the reason to choose one of the systems as better while it's not

true. Also, as the authors demonstrated, this effect is commonplace because they've received it on different benchmarks and different architectures.

In another paper called "Just how accurate are performance counters?" created by W. Korn, P.J. Teller, and G. Castillo in 2001, authors present methodology for determining the accuracy of different system counters. They discovered results obtained from MIPS R12000, SimpleScalar's sim-outorder simulator based on their benchmarks. Performance data were collected via perfex and libperfex. The results of their benchmarks indicate that obtained data with performance counters may lead to wrong conclusions. Accuracy of the counters depends upon the interface used, the application being measured, and the events being measured. For example, wrong results can be obtained because of measuring code hasn't enough granularity to ensure that overhead given by counters doesn't dominate. In another paper called Accuracy of Performance Monitoring Hardware created by Maxwell et al., 2002 this work was extended to the three new platforms: POWER3, IA-64, and Pentium. For exploring usability and accuracy issues, PAPI interface for hardware counting was used.

PAPI (Performance Application Programming Interface) is a standard specification used for hardware performance counters. In the paper called "Using PAPI for hardware performance monitoring on Linux systems" (Dongarra et al., 2001) capabilities of PAPI were discussed in the context of Linux. There are two interfaces in PAPI. The high-level interface was designed for relatively simple measurements: stop, start and read counters of different events, that are in the predefined list. The low-level interface allows managing hardware events in the user-defined groups. The whole list of hardware counters available in PAPE is in papiStdEventDefs.h file. Modern processors have a short number of hardware events which can be accessed simultaneously. This limitation dramatically reduces the amount of information that a user can receive after a single program run, but multiplexing hardware counters can help with such a restriction. PAPI uses a high-resolution interval timer to multiplex. PAPI uses perfctr to access the counters on Linux/x86 platforms. Supported processors are all 5 and 6 family processors by Intel, AMD K7 Athlon, Cyrix 6x86MX, MII, and III, WinChip C6, 2, 2A, 2B, and 3. With PAPI you can access different timers: for example timers which return time, measured from an arbitrary point in virtual units. Also, you can get information about the hardware on which the program is running: the number of CPUs, the cycle time of the CPU, etc PAPI can also provide the information about he start and end addresses of the text, data, etc.

The PAPI project also developed a graphical display of PAPI performance data. But this tool is more for demonstrating the capabilities of PAPI. Great tool to collect statistical program counter data is Visual Profiler. It can also graphically display the results on Linux Intel machines. vprof also provides access to a list of PAPI hardware counters.

Another way to monitor the performance is to use different libraries for benchmarking — for example, Google Benchmark, Hayai library, Celero library, Nonius library, etc. One I want to discuss in more details is Google Benchmark [*10*]. The user can test his or her code snippet and get the following output: wall-clock time in nanoseconds (it can be changed), CPU time also in nanoseconds and number of invoked iterations. The benefits of this library are the following: very easy to use and to customize, has benchmarks for the multithreaded program, has manual timing, which is very useful when a code is executing on GPU or the other devices where standard CPU timing is not relevant. But this Google Benchmark provides only information about the performance of user's code, don't analyse it to weaknesses or different faults, don't give any statistical information about the received results.

Chapter 3

Framework overview

3.0.1 How to measure time?

To speak about assessing the efficiency of some methods that improve performance, we need somehow to measure this performance. One of the main performance metrics is latency, the time to perform a specific task. Time measurements is quite complicated due to the complexity of modern computer systems: preemption, technologies for increasing the efficiency of the CPU, like caches, out-of-order execution, pipeline, branch predictor, etc., energy saving technologies, for example, CPU frequency changes depending on what tasks are running. This section describes some possible approaches to measure the execution time of some code fragment of the multithreaded program.

Each process and thread can be executed in the user space[A. Tanenbaum, 2014], where thread executes the code, and in the kernel space[A. Tanenbaum, 2014], where thread carry out system the calls. Both Windows and POSIX provide access to the time spent in each mode. But in this research wall-clock time is investigated. Wall-clock time gives an estimate of how much time a code snippet was performed taking into account all the features of the processor that were mentioned in the section "What affects the performance of the CPU?". For better reproducibility it is important to reboot the computer before each measurement and to ensure that there is a minimum number of other executed processes.

POSIX-compatible OSes and MS Windows provide idea of the process time. This time includes the execution time of all process threads – so the measured time in this way may be greater than the physical time that the program was executed. The thread time includes only the time of its execution. The process time measures the time during CPU works on the certain task.

Now let's take a closer look at what functions can be used to get process and thread times in POSIX. First of them to mention is **times()**.

The function stores the current process execution time using the pointer being passed to it. Time is returned in certain abstract units, the quantity of which can be obtained per second using sysconf(`_SC_CLK_TCK`), where `_SC_CLK_TCK` is the number of this abstract units.

Usage of the function:

```
#include <sys/times.h>

struct tms {
 clock_t tms_utime;
 clock_t tms_stime;
 clock_t tms_cutime;
 clock_t tms_cstime;
};
```

```
clock_t times(struct tms *buf);

tms proc_t;
times(&proc_t);

proc_t_sys  = proc_t.tms_stime ;
proc_t_user  = proc_t.tms_utime;
proc_t_total = proc_t_sys + proc_t_user;
```

The next function you can use in POSIX is **getrusage()**. The structure that this function returns looks like this:

```
#include <sys/time.h>
#include <sys/resource.h>

struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};

int getrusage(int who, struct rusage *usage);
```

The argument **who** may acquire the values **RUSAGE_SELF**, then the function will return the information about the process, **RUSAGE_CHILDREN** – function will return information about child processes, and in linux one more option is possible - **RUSAGE_THREAD** – return the thread information. Times are stored in the following structure:

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

The rusage structure contains a lot of other useful information: hard page faults, soft page faults etc. but it is not used in this research. This function can be used for the current process and thread in the following way:

```
rusage p_times;
times(&p_times);
```

```
getrusage(RUSAGE_SELF, &p_times);
#ifdef  RUSAGE_THREAD
rusage t_time;
getrusage(RUSAGE_THREAD, &t_time);
#endif

user_process_time = p_times.ru_utime.tv_sec + p_times.ru_utime.tv_usec;
sys_process_time = p_times.ru_stime.tv_sec + p_times.ru_stime.tv_usec ;
total_process_time = sys_process_time + user_process_time;

#ifdef  RUSAGE_THREAD
user_thread_time = t_time.ru_utime.tv_sec + t_time.ru_utime.tv_usec;;
sys_thread_time  = t_times.ru_stime.tv_sec + t_times.ru_stime.tv_usec ;
total_thread_time= thread_time_user + thread_time_sys;
#endif
```

For MS Windows similar functions are**GetProcessTimes()** and **GetThreadTimes()**. The output of the GetProcessTimes() is almost the same as the output of the previously mentioned function times(). The output of the GetProcessTimes() is similar to getrusage()/clock_gettime().

There is another way, much easier to measure the time - function **clock()** from standard C and C ++ libraries. It returns the value in abstract units, to convert this units in the seconds, you need to divide it into a CLOCKS_PER_SEC constant. In the POSIX standard it is 1 000 000, for Windows 1000. The start point is somewhere in the past, so you should only look at the difference between two clock () calls:

```
std::clock_t start = std::clock();

//beginning of measurement
//end of measurement

double duration = (std::clock() - start);
double c_duration = duration\static_cast<double>(CLOCKS_PER_SEC);
```

The solution is the cross-platform and simple. However, it has many disadvantages, for example, for POSIX it returns processor time, for windows – wall time. Wall time is different from processor time, it measures the real time execution of the program for the user. This is like a wall clock that goes until the program is running, and does not stop even when the CPU switches to another task.

One of the ways to measure wall time is to use chrono library contains various timers, including **chrono::high_resolution_clock**. It is a timer with best available resolution. Its static method now() returns the current time in the form std::chrono::time_point. With two moments of time, you can get their difference – an object of type std::chrono::duration, and than it can be converted to seconds, miliseconds, etc. To check whether the timer is steady – whether it counts accounts for system time changes (due to user intervention, daylight saving time, etc.) or not, you can by using the static field is_steady. chrono::high_resolution_clock can be used as follows:

```
auto start = std::chrono::high_resolution_clock::now();
//beginning of measurement
//end of measurement
auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration_cast<std::chrono::microseconds>
```

```
                                    (end - start).count();
```

The problem is that both the compiler and processor have the right to rearrange commands to optimize a code. And there may be a situation where the measurement point is shifted excessively backward or forward relative to the beginning of the code being measured. You can forbid the compiler and processor to do this by inserting the appropriate barriers. Of course, at the same time, we make task of optimization of our code harder for the compiler and CPU and the barriers themselves take some time to execute. However, for our task, this should not be a problem. C++11 includes the atomic_thread_fence() memory barrier. Measurements using the barrier will look like this:

```
#define OUR_FENCE()

do{std::atomic_thread_fence(std::memory_order_seq_cst);}while(false)

inline std::chrono::high_resolution_clock::time_point
    get_current_wall_time_fenced() {
    if(!std::chrono::high_resolution_clock::is_steady
                                    && PRINT_NON_STEADY)
    std::cerr << "Warning:
    std::chrono::high_resolution_clock is not steady." << std::endl;
    OUR_FENCE();
    auto res_time = std::chrono::high_resolution_clock::now();
    OUR_FENCE();
    return res_time;
}
```

Another option to measure performance is hardware counters of CPU cycles number. High-resolution timers often rely on them. One of the counters for x86 architecture is **Time Stamp Counter**, a 64-bit register that contains the number of cycles calculated from computer restart. The content of this counter can be obtained using the instruction RDTSC. But in modern processors, there are many problems with similar counters. The processor can reorder commands so the RDTSC cannot be executed if we would like it. Modern processors change their clock frequency, and the counter runs at a fixed frequency. So the counters are difficult to use when it comes to measuring the time of code execution.

Another way to analyse the performance of Linux is **perf**. This is a profiler that is built into the kernel but has a user-space utility. It can count various events - perf stat command, print the most used functions - perf top, etc. One of the most important results that perf allows you to get is task-clock – execution time, taking into account the time of all threads. Other important results are context-switches, cpu-migrations, page-faults, number of executed cycles and instructions.

Another, the significant and powerful profiler is VTune[5]. It is a commercial application for analyzing the performance of programs for computers based on x86 processors. It has both a graphical user interface and command line support. Vtune helps to understand where the program spends most of the runtime, detect hardware bottlenecks. You can also get a call tree, view at the assembler code, see what functions are most time-consuming. Vtune also shows information about threads, what exactly they perform, how they synchronize and interact with each other.

In our research we are investigating wall time. chrono::high_resolution_clock timer is used with appropriate memory barriers to measure wall time. Also, framework for measuring time returns the value of thread user/kernel time, process user/kernel times using previously discussed **getrusage()**. These values are not used in this study, but they may be useful in future studies.

The code is based on what was learned on the course of architecture of computer systems.

During our research results of perf were also collected for every single program run. This data can also be useful in future studies.

Having time measurements results, we need to use it depending on what is being discovered. For example in one of our experiments – investigating thread creation time – we are interested in just thread creation time, so here we used the minimum time, the time that was spent only on the thread creation, and not on any other things. In another experiment - capture and release of the mutex by many threads in the one program, we are not interested in the least time we have received. The least time corresponded to the situation when the thread was fortunate no one threads occupied the mutex, and there was no competition for it. In this experiment, we want to investigate the threads competition for the mutex, so here the most optimal is the average time

3.0.2 Cullen and Frey Graph

To draw some conclusions from the received data we used hypothesis testing technique. For example, this technique can be used to understand to what theoretical distribution obtained data belongs.

Cullen and Frey graph helps to understand what theoretical distributions the data obtained precisely do not belong and to which distribution obtained data are more likely to belong.

Cullen and Frey graph use skewness and kurtosis to produce the result. Kurtosis is a coefficient that characterizes how much the distribution has a pronounced peak relative to the normal distribution. Skewness is a coefficient that characterizes the asymmetry of the obtained empirical distribution.

As already mentioned, skewness and kurtosis are not reliable characteristics of the distribution. You can apply the bootstrap procedure to take into account the inaccuracy of the estimated skewness and kurtosis. Values of skewness and kurtosis are computed on the bootstrap samples. The procedure consists of iteratively replacing a certain amount of data from the distribution. But the problem is, that some skewness and kurtosis have very big variance cannot be solved by bootstrapping. Cullen and Frey Graph is used to pre-evaluate the distribution. Figure 3.1 depicts the example of a graph.

3.0.3 Kolmogorov-Smirnov test

The goodness test of fit is a statistical hypothesis test is used to determine if some obtained data represents the data of the actual population. There are a few goodness tests of fit which are commonly used in statistics: Kolmogorov-Smirnov Test, Anderson-Darling Test, Chi-squared Test, etc.

In this research we used Kolmogorov-Smirnov Test to determine if the distribution of the obtained data belongs to some theoretical distributions (normal distribution, lognormal, exponential etc.).

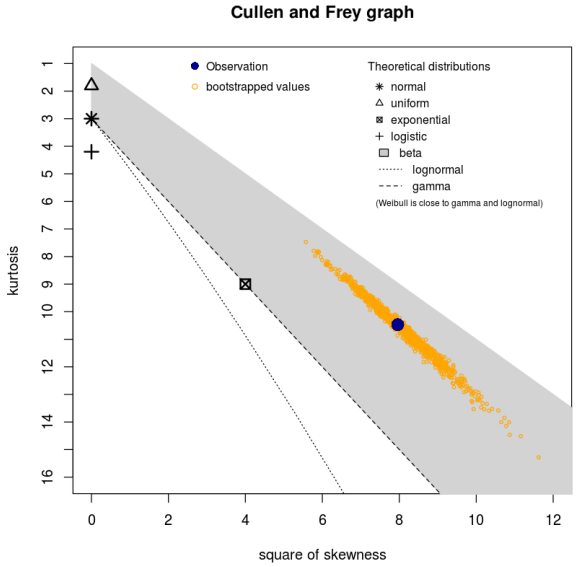


FIGURE 3.1: Time example of Cullen and Frey Graph build on obtained data.

Basically, K-S tests null hypothesis (obtained sample belongs to some specified distribution) against alternate hypothesis (obtained sample doesn't belong to some specified distribution).

To be more precise, it compares the empirical distribution function of some obtained data and the cumulative distribution function which is associated with null hypothesis. K-S test measures maximum vertical distance between functions.

The cumulative distribution function F(x) of a random variable X is $P(X \leq x)$, where x is an arbitrary real number. The empirical distribution function $G\{(x)\}$ gives the $P(X \leq x)$ based on the results of observations. It's a step function, that increases by $1/N$ at each data point. Suppose we have N observations $x_1, \ldots, x\_n$, the empirical distribution function is defined as:

$$G_{obs}(x) = \frac{\text{number of elements in the sample} \leq x}{N} \tag{3.1}$$

The statistic of K-S is defined as:

$$D = \sup_x |F_{exp}(x) - G_{obs}(x)| \tag{3.2}$$

The statistic, D, compares data to what is expected under the null hypothesis. When D is calculated, the next step is to compare the critical value to it (critical values for K-S test can be found here[14]). If D is less than critical value, than null hypothesis has to be rejected, and alternate hypothesis has to be accepted.

Another measure of the K-S test results is p-value. If the test statistic is in rejection region, where rejection region is a set of points where null hypothesis should be rejected, the p-value is smaller than the significance level. A small p-value (typically ≤ 0.05) is an evidence against the null hypothesis, so it can be rejected.

3.0.4 Wilcoxon Paired Signed Rank Test

Another important test we used is the Wilcoxon Paired Signed Rank Test. All experiments were carried out with the same input parameters on different machines, so we have to use the paired version of the test. We used this test to determine if two obtained samples from two different experiments belong to the same statistical population. Unlike paired Student's t-test, Wilcoxon test can be applied if data isn't normally distributed.

Wilcoxon Paired Signed Rank Test tests the null hypothesis (the obtained samples belongs to the same statistical population) against alternate hypothesis the obtained samples doesn't belong to the same statistical population).

The test procedure looks like this: we have two paired sets of data A and B. Than we compute the difference between every element of two sets ($B_i - A_i$) and take absolute of this value. Each absolute value is assigned a rank relative to the increasing order. Then we need to sum up all ranks of positive and negative values (T_{pos}, T_{neg}). The statistic of Wilcoxon Paired Signed Rank Test is defined as:

$$D = \min(T_{pos}, T_{neg}) \tag{3.3}$$

If D is less than critical value (Critical values for Wilcoxon Paired Signed Rank Test can be found here[15]), than null hypothesis has to be rejected, and alternate hypothesis has to be accepted.

Chapter 4

Experiments

4.1 Threads creation

Threads are one of the main elements of low-level programming. An important part of their strength is that, along with the preemption, it might be more efficient to create more threads than there are CPUs in a processor. We used our framework to compare the efficiency of creating threads by different APIs: POSIX threads Linux, std::thread implementation by GCC, std::thread implementation by clang, boost::thread and tbb::thread.

On different machines (Table 4.1) we run the program that creates different number of threads and measures the time it takes to run them. Threads immediately go into standby. For each number of threads created, the program was re-executed five hundred times.

Device	**OS distribution/kernel**	**CPU**	**freq(MHz)/Cores/ RAM(Gb)**
Dell Inspirion 5521	Ubuntu 16.04/4.15.0	Intel Core i7-3517U	800-3000/2/8
HP Probook 440 G5	Ubuntu 18.04/4.15.0	Intel Core i5-8250U	400-3400/8/16
Raspberry Pi Model 3B+	Raspbian 2018/4.14.79	Broadcom BCM2837B0, Cortex-A53 (ARMv8)	600-1400/4/1

TABLE 4.1: Computers on which researches were conducted.

In the figures A.10 the minimum run times for each API are shown. In this experiment only time for thread creation is important. From this results it can be concluded that the creation of the first threads requires a relatively long time, obviously, there is a certain "warm up" of the system - the allocation of memory structures, loading pages with the corresponding code, etc. It is also obvious that POSIX threads Linux, gcc std::thread, clang std::thread and boost::thread showed a very similar results, but tbb::thread need more time to create first threads than the others. So, it's less effective to use tbb when a small number of threads executes the program.

The figure 4.1 depicts the examples of the histograms of the received time results of one thread creation when creating 100 threads in one program. All results for all computers and thread APIs that were tested are shown here A.1, A.2, A.3 It is obvious that there is a certain peak time on each histogram. In order to find out which theoretical distributions the obtained data more likely belongs to, Cullen

and Frey Graph was constructed. The Kolmogorov-Smirnov test was used to determine whether the distribution is normal, lognormal, exponential, gamma or neither. The K-S test showed that none of the obtained data belongs to the above-mentioned distributions – p-value in all tests turned out to be less than 0.05.

FIGURE 4.1: Examples of the histograms of the received time results of one thread creation when creating 100 threads in one program

Therefore, to determine the differences between the thread time creation for different APIs Wilcoxon signed rang test was applied. In all tests, the Wilcoxon test showed a negative result (p-value in all tests turned out to be less than 0.05), which means that the difference between thread time creation for different APIs is statistically significantly different from zero. Though, the overhead costs of std::thread, boost::thread, gcc std::thread, clang std::thread or tbb::thread are small compared to posix threads.

Since the distribution of the obtained data is not normal, it makes no sense to talk about the variance.

Some of the results of this experiment were published earlier[Y. Laba, 2019].

4.2 Mutex try_lock()

In the next experiment, we investigated the time to request the same mutex from many threads simultaneously. Mutexes are one of the main primitives to synchronize threads.

The experiment was conducted on various implementations: POSIX, std implementation by GCC, std implementation by clang, boost and tbb on the computers described in the table 4.1. Each program was re-executed a thousand times. Conditions of the experiments were as follows: before the mutex was requested by many threads, it was captured by the main thread. Every thread tried to capture the mutex for 10,000 iterations. In this experiment, we used the function try_lock() – signature depends on what API was used. This function immediately returns an answer whether the mutex is captured or not and if not, thread capture it. In our experiment, the mutex is already captured. We are investigating time to request the same mutex by many threads. It makes sense to use the average time here. The minimum time would correspond to the situation when the thread was fortunate and nobody tried to capture the mutex. It is interesting for us to investigate how much time it takes for a thread of one or another implementation to request the mutex when there is a competition between threads. In the figures A.11 the average run times to request

mutex by a thread when a different number of threads tries to request it are shown for each API.

From these graphs, it can be concluded that the POSIX implementation is not the fastest.

The figure 4.2 depicts examples of the histograms of the received time results of the mutex request by thread, when 100 threads tries to request it. All results for all computers and thread APIs that were tested are depicted here A.4, A.5, A.6.

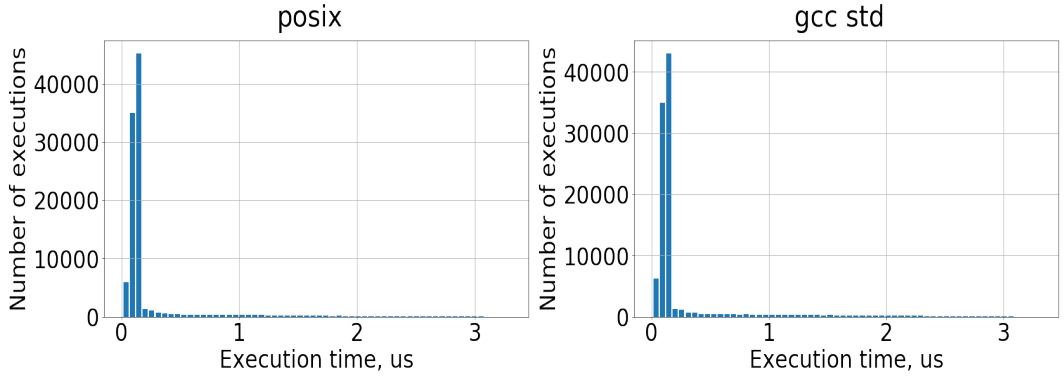


FIGURE 4.2: Examples of the histograms of the received time results of the mutex request by thread, when 100 threads try to request it in one program

The Cullen and Frey Graph and the Kolmogorov-Smirnov test were applied to determine which of the theoretical distributions the data obtained in this experiment belongs to. The Kolmogorov-Smirnov test showed very small p-value (less than 0.05) when checking if obtained data belongs to normal, lognormal, exponential and gamma distribution. Again, our data isn't normally distributed therefore it makes no sense to talk about variance. The Wilcoxon signed rang test was also applied. In all tests, it showed a negative result, which means that there are time overheads depending on the implementation.

4.3 Mutex lock()/unlock()

The next experiment is also related to the mutexes, but we investigated the time to capture and release the same mutex from many threads simultaneously. The experiment was also conducted on various implementations: POSIX, std implementation by GCC, std implementation by clang, boost and tbb on the computers (Table 4.1). Each program was re-executed a thousand times. Every thread captured and immediately released the mutex for 10,000 iterations. In this experiment, we used the functions lock() and unlock() – signature depends on what API was used. lock() captures mutex and unlock releases it. lock() doesn't return anything if mutex is occupied by another thread, it will just wait. We are investigating time to capture and release the same mutex by many threads. It also makes sense to use the average time here. The minimum time would correspond to the situation when the thread was fortunate and mutex was free. It is interesting for us to investigate how much time it takes for a thread of one or another implementation to capture and release the mutex when there is a competition between threads. In the figures A.12 the average run times to capture and release mutex by a thread when a different number of threads try to request it are shown for each API. The received results are quite

obvious, the larger the number of threads that capture and release the mutex, the longer is the execution time.

The figure 4.3 depicts examples of the histograms of the received time results of the occupating and releasing a mutex by thread, when 100 threads try to capture and release it. All results for all computers and thread APIs that were tested are depicted here A.7, A.8, A.9.

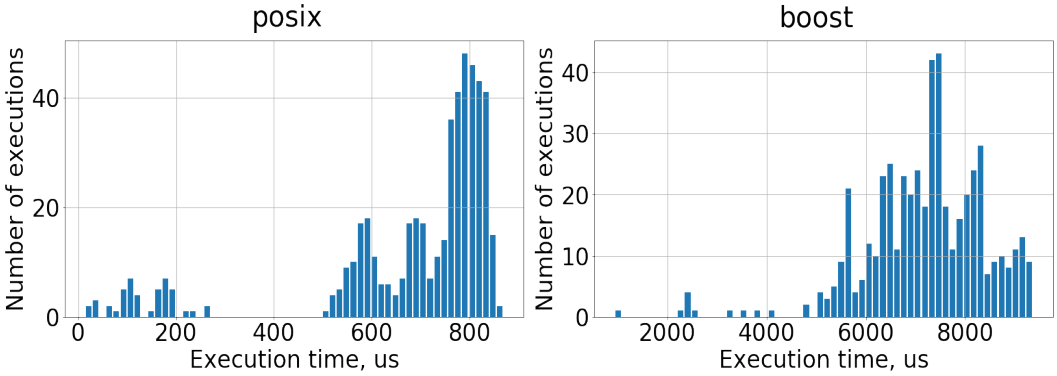


FIGURE 4.3: Examples of the histograms of the received time results of the capture and release mutex by thread, when 100 capture and release mutex in one program

The Kolmogorov-Smirnov doesn't show that the obtained distributions belong to normal, lognormal or gamma distribution. We can't speak about variance in this case too, In all tests, the Wilcoxon test showed a negative result, which means that there are time overheads depending on the implementation.

In all the experiments, the code was compiled with -O3 optimization.

Chapter 5

Conclusions

In this research, we developed and tested a framework for testing various code snippets. We have tested the framework on several tasks.

None of the obtained distribution results belongs to the normal distribution so we can not talk about variance. We also found that in all the experiments the Wilcoxon test showed negative results. It indicates that there is a time overhead between different threads and mutexes implementations. Respectively, there is a difference in performance. We empirically showed that the POSIX implementation is not always the fastest. For example, in the mutex try_lock() experiment for Dell the fastest is std implementation by clang, for HP the fastest is also std implementation by clang, and for Raspberry Pi the fastest is tbb.(Figures A.11).

This framework is planned to be used in the near future for other tasks, for example, future-promise. We are currently running an experiment and apply this framework to various implementations of thread-safe queues (boost lockfree queue, tbb concurrent queue). Unfortunately, due to the time constraints, the results of these experiments are not part of this work.

Appendix A

Plots

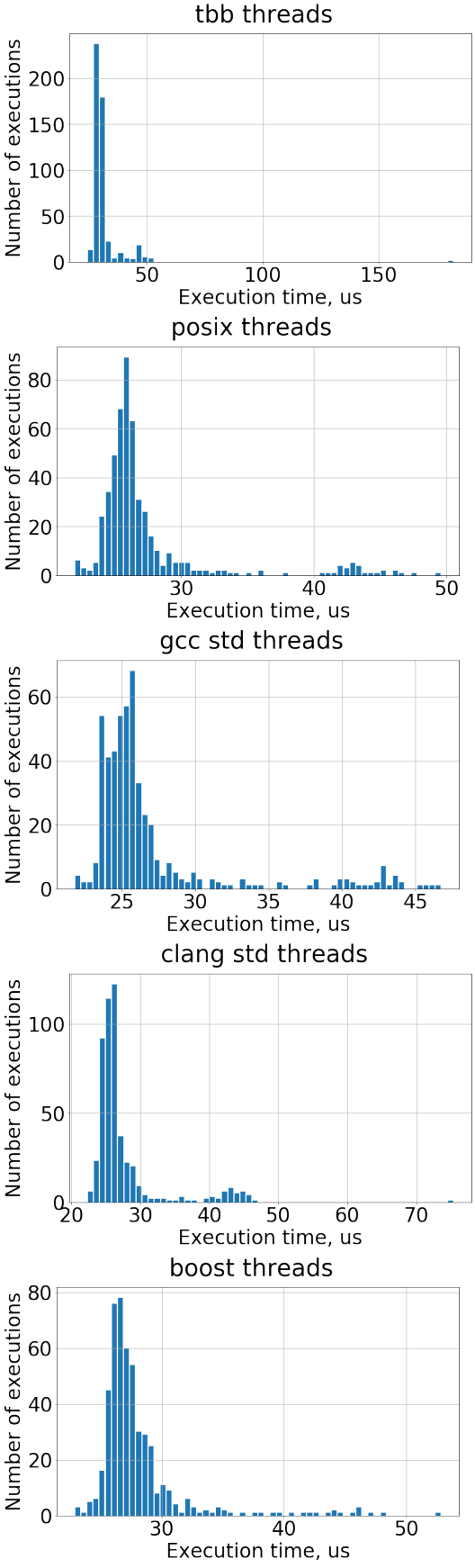


FIGURE A.1: Histograms of the received time results of one thread creation when creating 100 threads in one program on Dell Inspirion 5521.

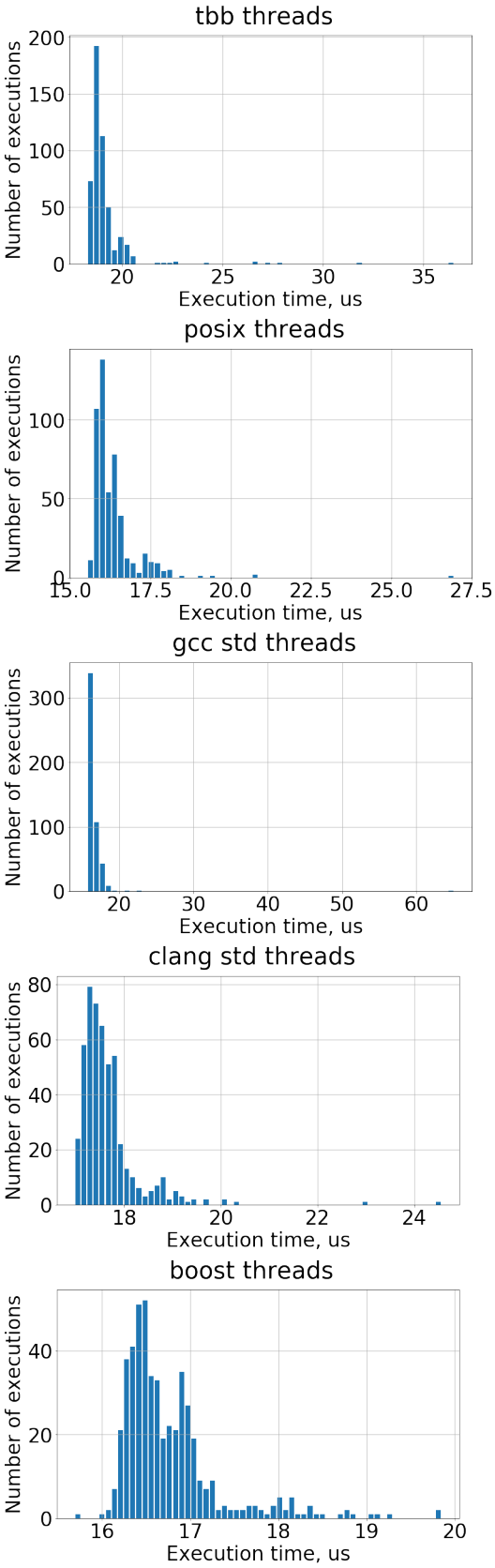


FIGURE A.2: Histograms of the received time results of one thread creation when creating 100 threads in one program on HP Probook 440 G5.

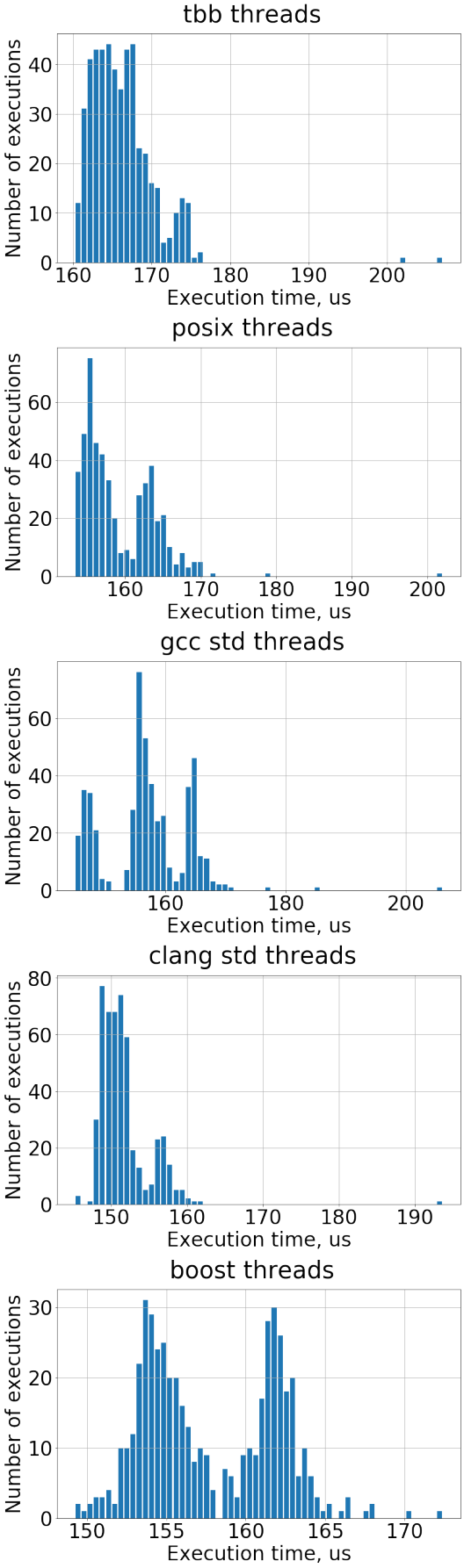


FIGURE A.3: Histograms of the received time results of one thread creation when creating 100 threads in one program on Raspberry Pi 3.

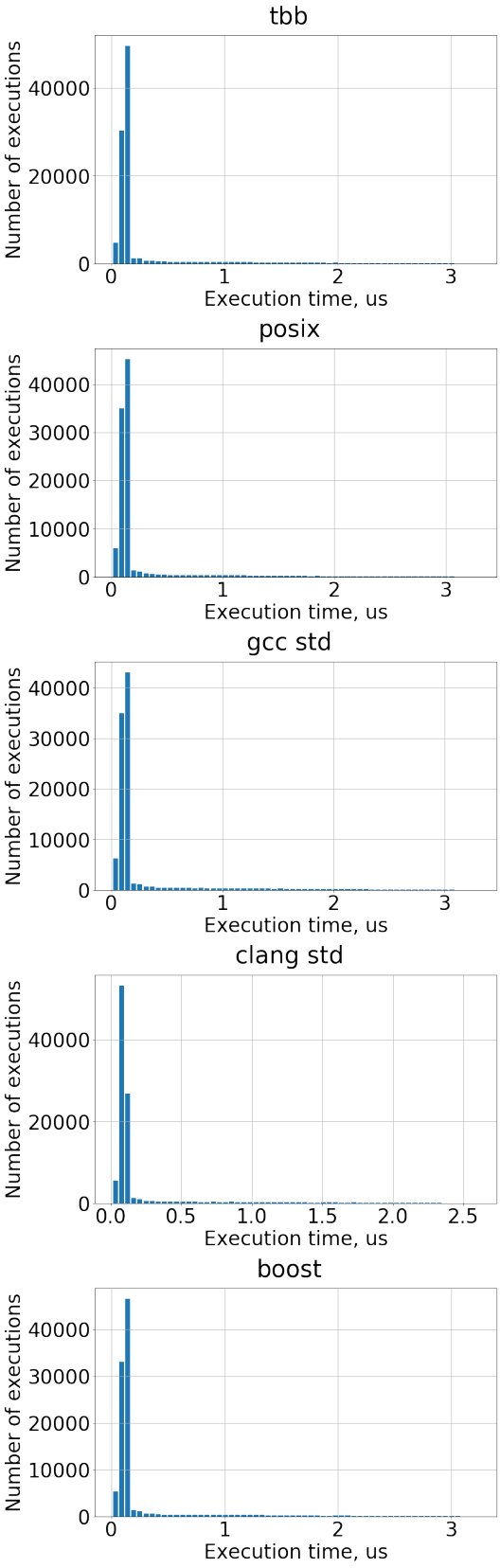


FIGURE A.4: Histograms of the received time results of same mutex request per one iteration by thread when creating 100 threads in one program on Dell Inspirion 5521.

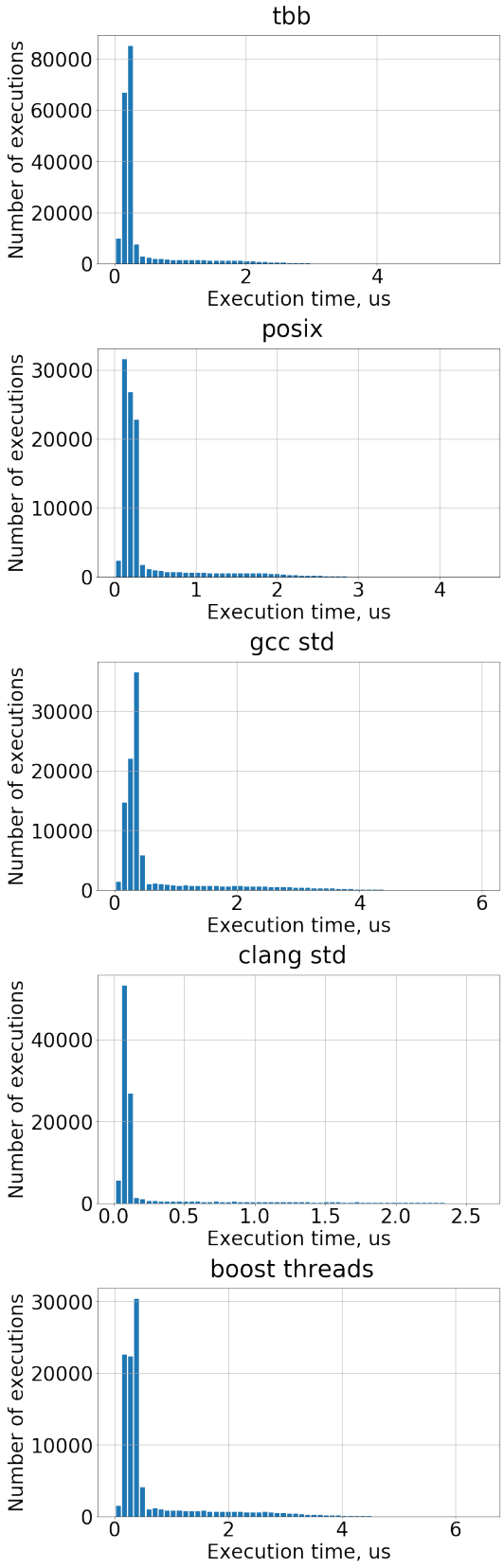


FIGURE A.5: Histograms of the received time results of same mutex request per one iteration by thread when creating 100 threads in one program on HP Probook 440 G5.

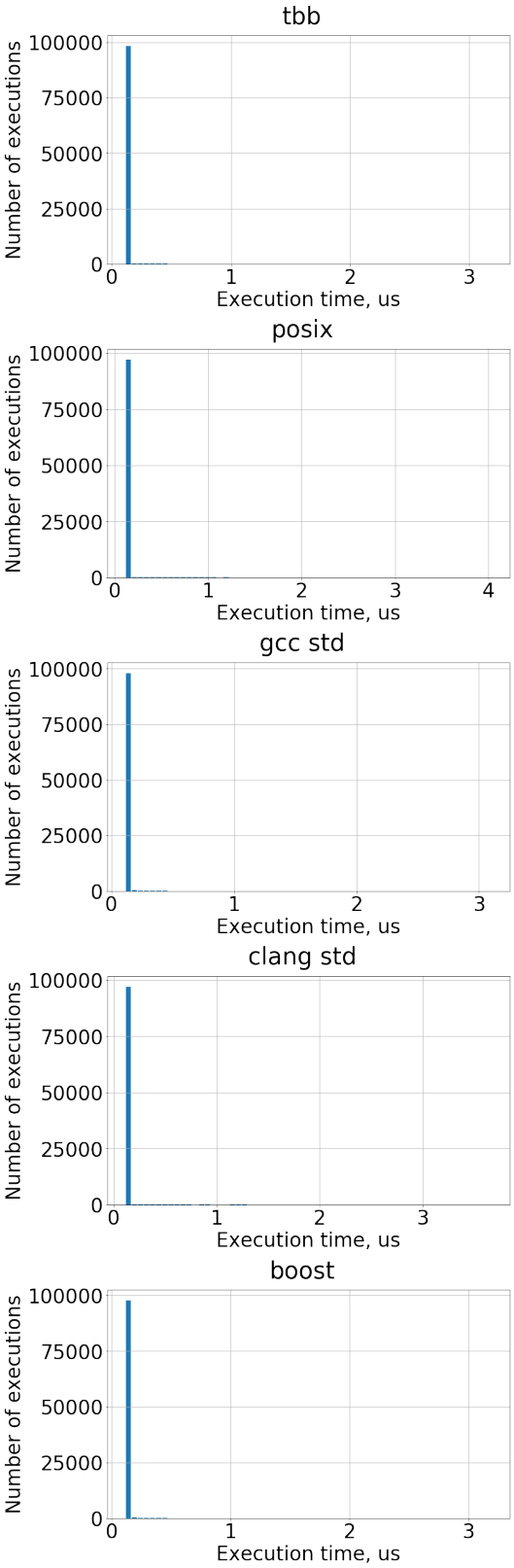


FIGURE A.6: Histograms of the received time results of same mutex request per one iteration by thread when creating 100 threads in one program on Raspberry Pi 3.

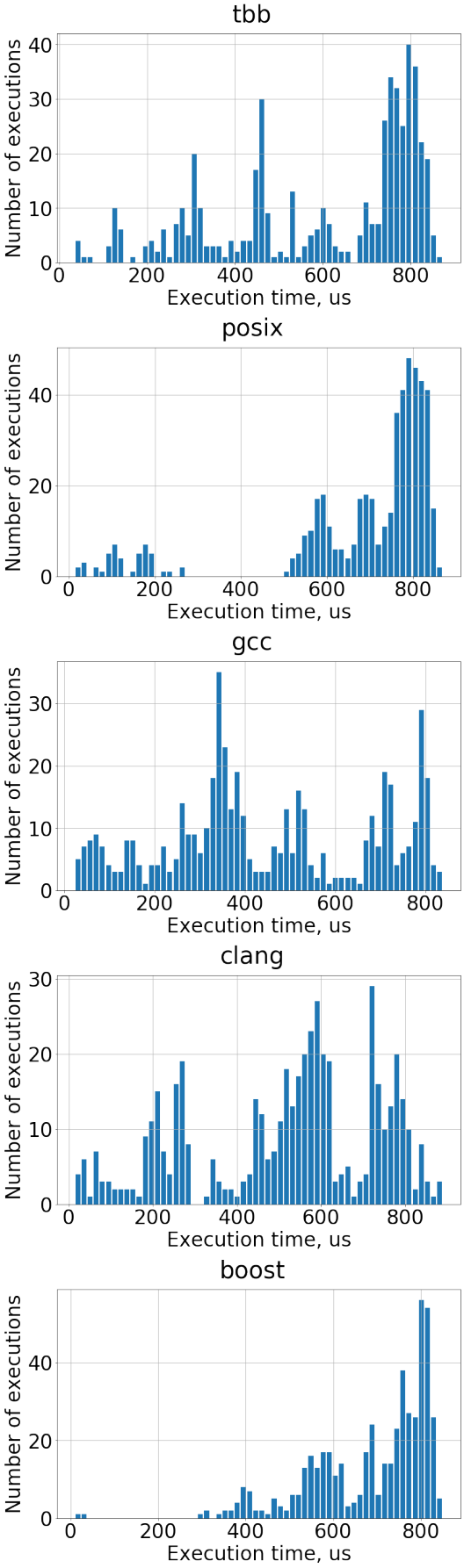


FIGURE A.7: Histograms of the received time results of the capture and release mutex by thread, when 100 capture and release mutex in one program on Dell Inspirion 5521.

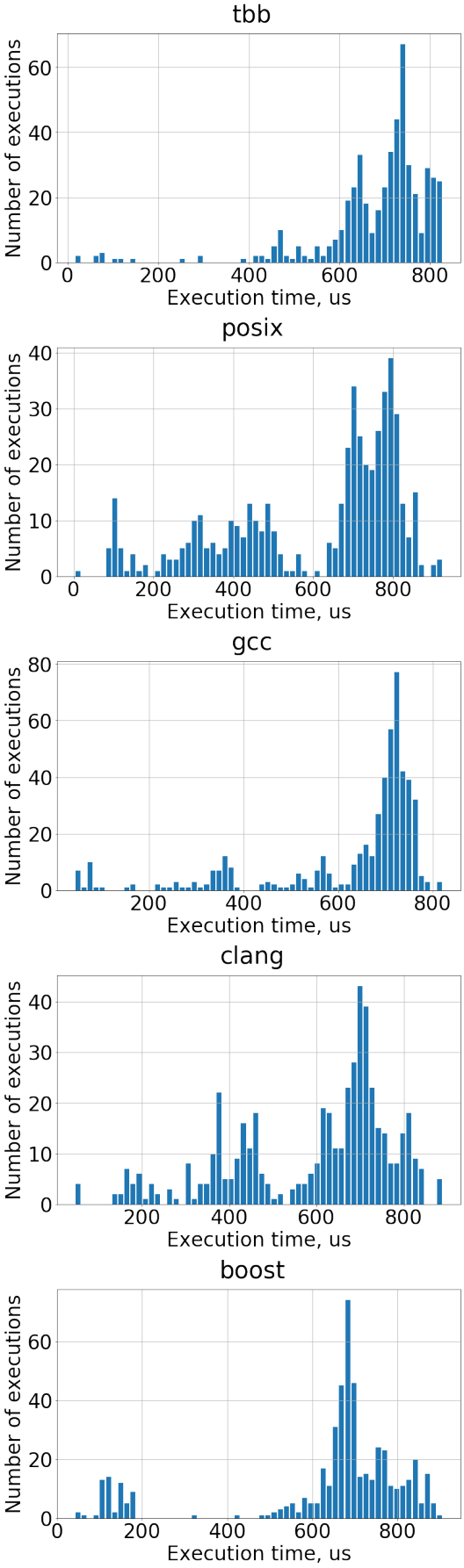


FIGURE A.8: Histograms of the received time results of the capture and release mutex by thread, when 100 capture and release mutex in one program on HP Probook 440 G5.

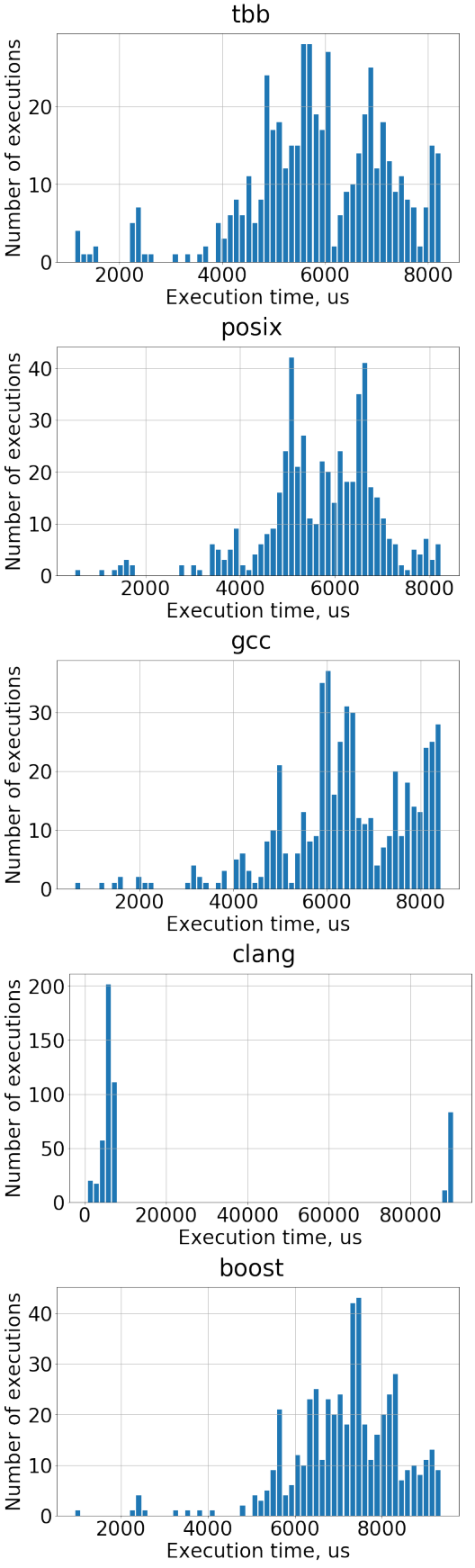
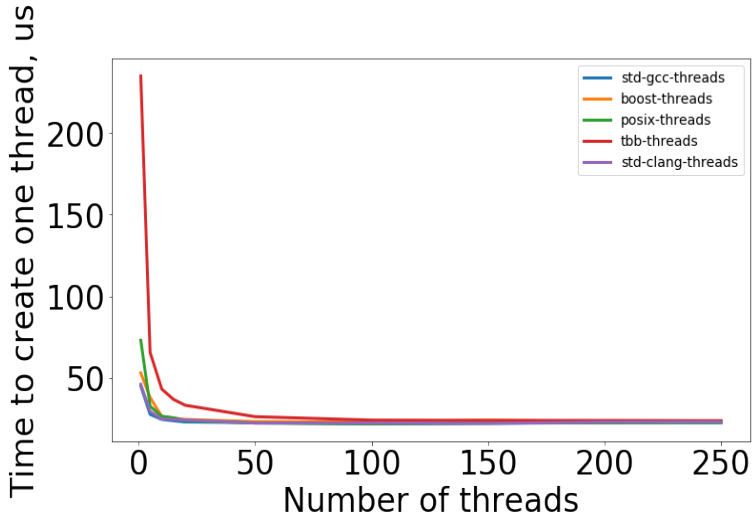
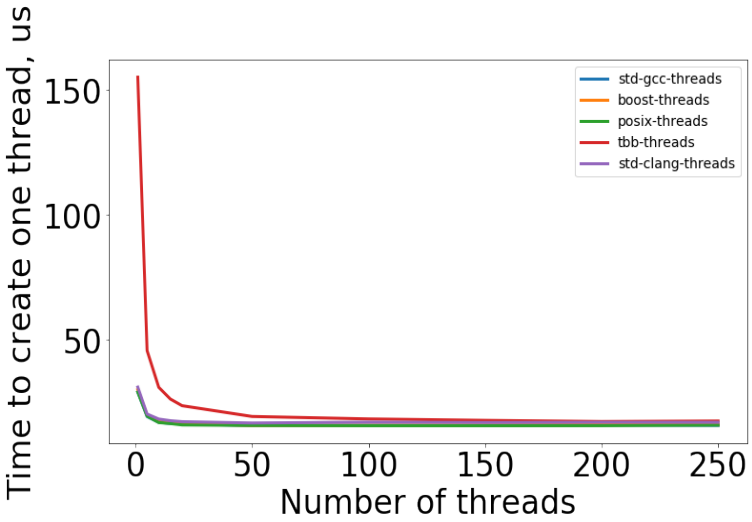


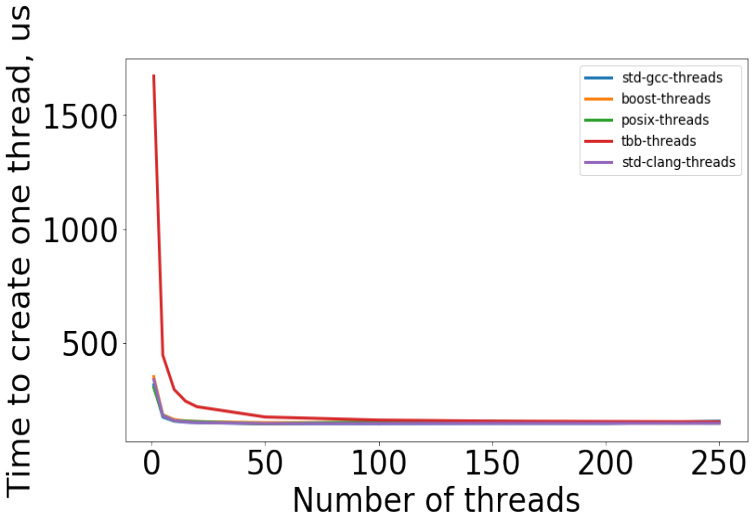
FIGURE A.9: Histograms of the received time results of the capture and release mutex by thread, when 100 capture and release mutex in one program on Raspberry Pi 3.



(A) Dell Inspirion 5521

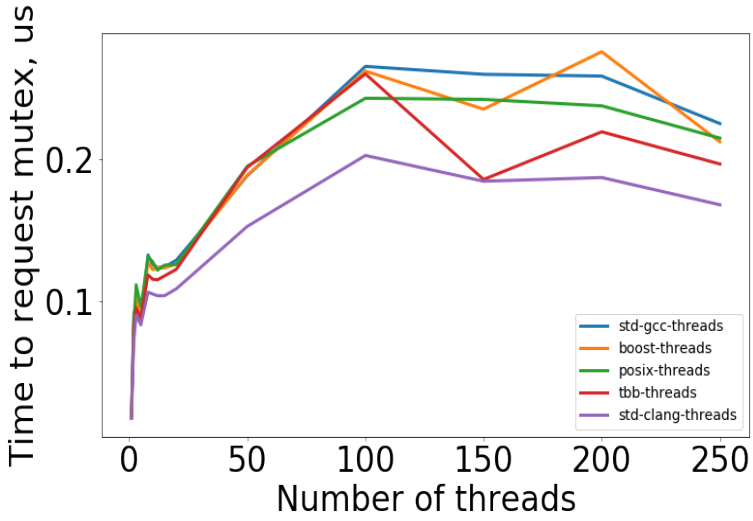


(B) HP Probook 440 G5

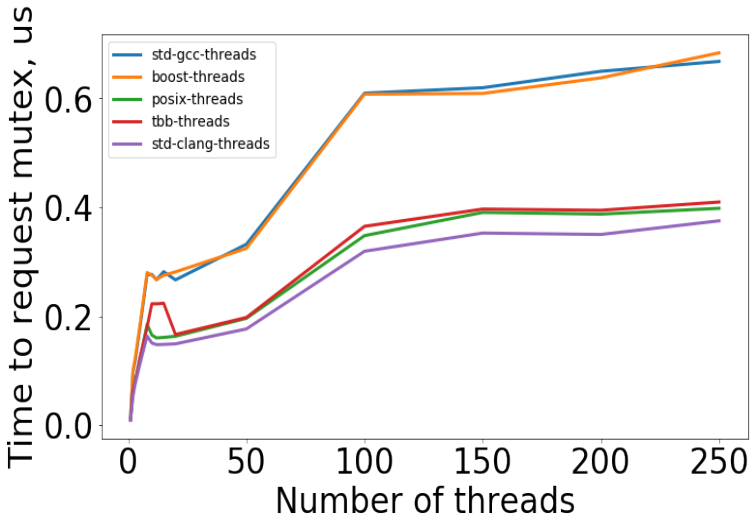


(C) Raspberry Pi 3

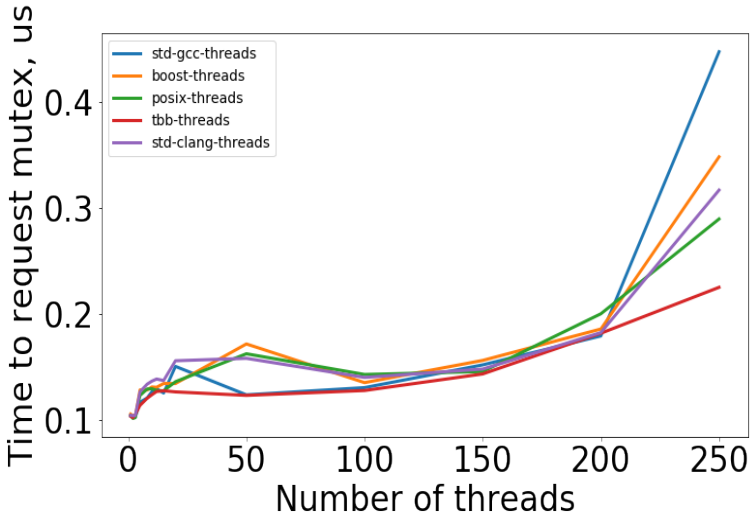
FIGURE A.10: Time to create one thread when creating different number of threads in one program, us.



(A) Dell Inspirion 5521

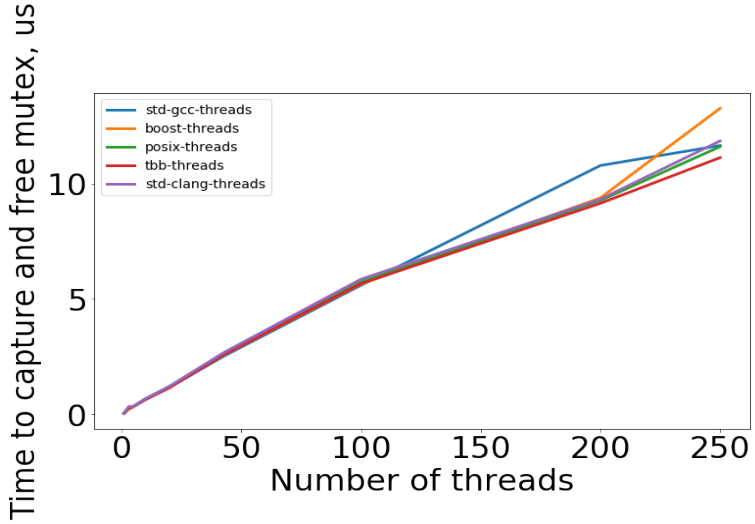


(B) HP Probook 440 G5

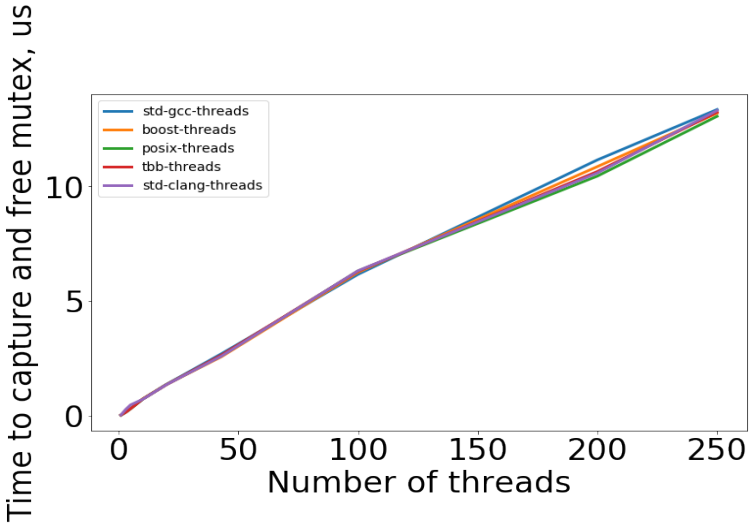


(C) Raspberry Pi 3

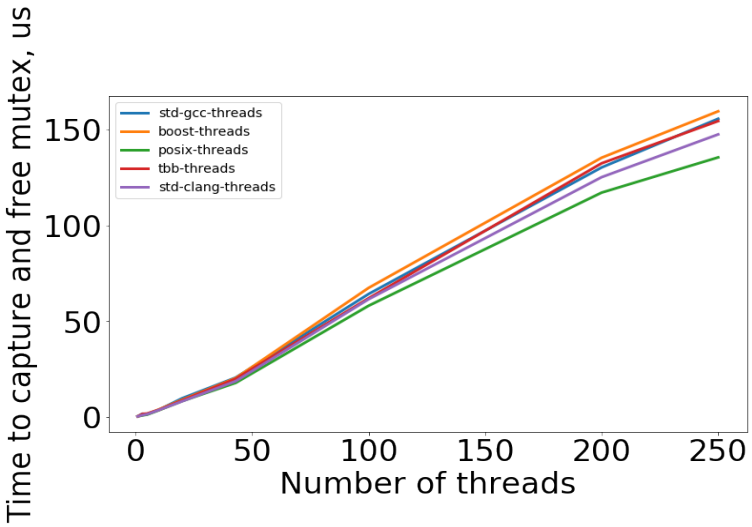
FIGURE A.11: Time to request mutex by thread, when different number of threads tries to request it, us.



(A) Dell Inspirion 5521



(B) HP Probook 440 G5



(C) Raspberry Pi 3

FIGURE A.12: Time to capture and release mutex by thread, when different number of threads tries to request it, us.

Bibliography

1. Clock Frequency from CPU database, Stanford VLSI Group http://cpudb.stanford.edu/visualize/clock_frequency. Accessed: 2019-04-30.

10. Official page of Google benchmark library https://github.com/google/benchmark. Accessed: 2019-04-30.

12. Official cite of MOSBENCH benchmark https://pdos.csail.mit.edu/archive/mosbench/, note = Accessed: 2019-04-30.

13. Official SPEC CPU2006 benchmark https://www.spec.org/cpu2006/, note = Accessed: 2019-04-30.

14. Kolmogorov-Smirnov Table http://www.real-statistics.com/statistics-tables/kolmogorov-smirnov-table/, note = Accessed: 2019-04-30.

15. Wilcoxon Signed Rank Table https://math.ucalgary.ca/files/math/wilcoxon_signed_rank_table.pdf, note = Accessed: 2019-04-30.

3. Official page of perf, Linux profiling with performance counters: https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2019-04-30.

4. Windows Performance Toolkit, https://docs.microsoft.com/en-us/windows-hardware/test/wpt/. Accessed: 2019-04-30.

5. Official page of Intel VTune Amplifier, https://software.intel.com/en-us/vtune. Accessed: 2019-04-30.

6. Performance Counters on Windows Dev Center, https://docs.microsoft.com/en-us/windows/desktop/perfctrs/performance-counters-portal. Accessed: 2019-04-30.

7. Official page of Processor Counter Monitor (PCM) https://github.com/opcm/pcm. Accessed: 2019-04-30.

9. Official page of Intel Turbo Boost https://github.com/opcm/pcm. Accessed: 2019-04-30.

A. Tanenbaum, H. Bos (2014). *Modern Operating Systems*. 4th. Pearson, pp. 97–115, pp. 148–165, pp. 111–146.

Boyd-Wickizer, Silas et al. (2010). "An Analysis of Linux Scalability to Many Cores." In: *OSDI*. Vol. 10. 13, pp. 86–93.

Castillo, Gilbert, Wendy Korn, and Patricia J Teller. "Just how accurate are performance counters?" In:

Diwan, Todd Mytkowicz Amer, Matthias Hauswirth, and Peter F Sweeney (2009). "Producing Wrong Data Without Doing Anything Obviously Wrong!" In:

Dongarra, Jack et al. (2001). "Using PAPI for hardware performance monitoring on Linux systems". In: *Conference on Linux Clusters: The HPC Revolution*. Vol. 5. Linux Clusters Institute.

Durbhakula, Suryanarayana Murthy (2018). "OS Scheduling Algorithms for Improving the Performance of Multithreaded Workloads". In: *arXiv preprint arXiv:1810.09442*.

Hennessy, D. Patterson J. (2013). *Computer Organization and Design*. 5th, pp, 28–51.

Maxwell, M et al. (2002). "Accuracy of performance monitoring hardware". In: *Proceedings of the Los Alamos Computer Science Institute Symposium (LACSI'02)*. Citeseer.

P. Fleming, J. Wallace (1986). *How not to lie with statistics: the correct way to summarize benchmark results*, pp. 218–221.

Papamarcos, Mark S and Janak H Patel (1984). "A low-overhead coherence solution for multiprocessors with private cache memories". In: *ACM SIGARCH Computer Architecture News*. Vol. 12. 3. ACM, pp. 348–354.

Snyder, Peter (1990). "tmpfs: A virtual memory file system". In: *Proceedings of the autumn 1990 EUUG Conference*, pp. 241–248.

Terpstra, Dan et al. (2010). "Collecting performance data with PAPI-C". In: *Tools for High Performance Computing 2009*. Springer, pp. 157–173.

Von Behren, Rob et al. (2003). "Capriccio: scalable threads for internet services". In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM, pp. 268–281.

Y. Laba, O. Farenyuk (2019). "Research and analysis of the effectiveness of creating threads". In: *International Scientific and Practical Conference "Innovative Technologies in the Development of Modern Society"*. NU "Lviv Polytechnic".