BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PERFORMANCE ANALYSIS OF WEB APPLICATIONS

ANALÝZA VÝKONU WEBOVÝCH APLIKACÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TOMÁŠ VALENT**
AUTOR PRÁCE

SUPERVISOR **Ing. JIŘÍ PAVELA,**
VEDOUCÍ PRÁCE

BRNO 2024

Bachelor's Thesis Assignment

156868

Institut:	Department of Intelligent Systems (DITS)
Student:	**Valent Tomáš**
Programme:	Information Technology
Title:	**Performance Analysis of Web Applications**
Category:	Software analysis and testing
Academic year:	2023/24

Assignment:

1. Get acquainted with the Perun project (performance version system) and the field of software profiling.
2. Study available methods of program instrumentation, resource consumption measurement (e.g., function duration, memory consumption or CPU usage), and existing tools for performance analysis of web applications.
3. Design and implement a tool that measures consumption of at least one resource in web applications, or programming languages used in development of web applications (e.g., JavaScript, TypeScript or ASP.NET). The tool interface should respect the requirements of the Perun project.
4. Design and implement suitable visualisation of the resulting collected data (e.g., flame graph or tree view), or use and enhance at least two of the existing visualisations in Perun.
5. Demonstrate the solution on at least one non-trivial use-case.

Literature:

- Oficiální stránky projektu Perun: https://github.com/Perfexionists/perun
- V8 Profiler: https://v8.dev/docs/profile
- Gregg, B. (2020). Systems Performance, (2nd ed.). Pearson. ISBN: 9780136821694.

Requirements for the semestral defence:
First two points of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Pavela Jiří, Ing.**
Consultant:	Fiedor Tomáš, Ing., Ph.D.
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	6.11.2023

Abstract

Abstrakt

Keywords

Typescript, profiling, Perun, continuous monitoring, Node.

Klíčová slova

Typescript, profilovanie, Perun, nepretržié monitorovanie, Node.js

Reference

VALENT, Tomáš. *Performance Analysis of Web Applications*. Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Pavela,

Performance Analysis of Web Applications

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Jiří Pavela. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Tomáš Valent
December 4, 2023

Acknowledgements

I would like to thank the supervisor of this thesis, Ing. Jiří Pavela, and the technical consultant, Ing. Tomáš Fiedor Ph.D., for their supervision and patience.

Contents

Chapter 1

Introduction

Every programmer wants to have their application perfect. We usually encounter many different problems and bugs that are in our way to do so. One of the most common issues is *performance* of the program.

> Performance is part of the non-functional requirements of software. It can be defined as how efficiently a software can accomplish its tasks. A system is never more performative than its slowest part. And that part is what we call a *bottleneck* or *contention*. If you want to improve the performance of your system, you have to improve the performance of the slowest part. As all your processing is queueing in there, the rest of your system hasn't reached its peak yet. [10]

There are a lot of tools that can help us identify performance issues – profilers. Profiling is one of the software's forms of *dynamic analysis*. I will explain the types of program analysis in more detail in Chapter 2. The main goal of profiling is to identify performance issues such as throughput (either operations or data volume per second), IOPS (input/output operations per second), utilization (how busy a resource is, as a percentage), latency (operation time, as an average or percentile), CPU load and visualize them. Common ways to visualize the results of profiling are graphs and heat maps.

The main objective of this thesis is to develop a new module that will extend the system for versioning performance profiles *Perun*, which is developed by the *VeriFIT* group at the Faculty of Information Technology BUT, by a profiler that can profile web applications programmed in *Typescript*.

TypeScript is a free, strongly typed, compiled and open-source high-level programming language developed by Microsoft. TypeScript is a syntactic superset of JavaScript which adds static typing. This means that TypeScript adds syntax on top of JavaScript, allowing developers to add types [12]. Unlike JavaScript, TypeScript supports object-oriented programming concepts in the vein of classes, interfaces, and inheritance. [11]

The following chapters will deal with these topics: Introduction of program analysis and profiling in Chapter 2. Introduction of version control system Perun in Chapter 3. Comparison of existing open-source Typescript profilers in Chapter 4. Analysis of requirements for profiler for this thesis in 5. Design of the module in Chapter 6. All information about the implementation is in Chapter 7.

Chapter 2

Performance Analysis

This chapter will closely explain profiling as a whole, the types of software analysis, and some profiling concepts.

Performance analysis is a critical aspect of software development and system optimization. Whether you are a developer, a system administrator, or an IT manager, understanding and improving the performance of your applications is necessary. In a time when user expectations are higher than ever, and a fraction of a second can make a significant difference, it is essential to have the knowledge and tools to identify and remove performance bottlenecks.

2.1 Web application profiling

Profiling builds a picture of a target that can be studied and understood. In the field of computing performance, profiling is typically performed by *sampling* the state of the system at timed intervals and then studying the set of samples. The use of sampling provides a coarse view of the target's activity. How coarse depends on the rate of sampling. As an example of profiling, CPU usage can be understood in reasonable detail by sampling the CPU instruction pointer or stack trace at frequent intervals to gather statistics on the code paths that are consuming CPU resources. [4]

2.1.1 Static analysis

Static analysis is a sort of software testing approach that is used to look for errors in software without actually running the software application's code. Static analysis is performed in the early stage of development to avoid errors as it is easier to find sources of failures and it can be fixed easily [3]. Errors that cannot be uncovered using dynamic analysis can be easily discovered using static analysis.

Static analysis is for example code review. In code review, developers look at the code of each other with the goal of finding possible mistakes while they do not run the code to test it.

2.1.2 Dynamic analysis

Dynamic analysis is a sort of software testing that examines the dynamic behavior of the code. Unlike static analysis, dynamic analysis involves the actual running of the software. It comprises software testing for the input and output values that are evaluated.

Dynamic analysis of software can be for example profiling or unit testing. The point is to test the code with various inputs during the runtime.

2.1.3 Continuous monitoring

A real-time monitoring method called *continuous monitoring* is used to examine production applications in order to identify and resolve performance issues that impact application user experience. Identifying issues and opportunities for improvement, it entails collecting data on execution-related parameters such as CPU use, disk I/O consumption, and function call time. Continuous profiling provides developers with additional insights into functions that are performed. On the other side, engineers have the ability to gather CPU utilization data and note how to measure consumption for better performance. [5]

Continuous monitoring can be used to profile web pages, in cybersecurity, or in hospitals it can be „vital signs monitor".

2.2 Sampling

During profiling, sampling collects statistical data about an application's activities and is a useful beginning point for finding places to speed up your program. The sampling method collects information about the functions in your application at predefined intervals. Information about the application is gathered through data collection by sampling the data at a regular period or sampling frequency, such as once every millisecond. To build a model of where time was spent in the application, the gathered data is evaluated. Sampling could be a good option if you require precise call time data or are searching for performance problems in an application for the first time.

Sampling is less accurate in terms of calls, but it is inexpensive for the profiler and has less impact on how the program under profile runs. [6]

2.3 Instrumentation

During a profiling run, instrumentation profiling gathers comprehensive data about the work the tasks done by an application. Tools that either insert code into a binary file to record timing data or use callback hooks to gather and release precise timing and call count data while a program is operating are used for data collecting. Comparing the instrumentation method to sampling-based techniques reveals a large overhead. [6]

2.4 Sampling vs. Instrumentation

The benefit of sampling is that it requires less overhead and, as a result, is more likely to be statistically the most representative of reality in production. The benefit of instrumentation profiling is that you can obtain precise call counts on how many times your functions were invoked. This provides significantly more comprehensive information than standard sampling, which might distort time in particular cases. Functions that don't perform anything but are called frequently, for example, will seem more than they would in a real-world scenario.

Every function call in your application is annotated and instrumented with instrumentation so that when it is executed, it is included in the trace along with information about the caller. The current call stack is queried from the CPU at regular intervals with sampling, and each frame is added to the trace. [6]

2.5 Tracing

Tracing offers further information about how frequently a method was run. Tracing is useful if you require precise measurements of call numbers. Tracing has a greater influence on the speed of your code during collection, whereas sampling has a minor overhead. Furthermore, tracing might be slower to examine since it takes longer to view the data after it has been collected. [6]

2.6 Profiling metrics

In profiling, we can measure different *metrics*. Metrics show us how effectively the is program written and we can use them to visualize graphs, heat maps, and so on. Those metrics and visualizations can be used to analyze the performance of the program. Now I am going to list the most common performance metrics.

- **Total time** – The total time of a program is the easiest metric to measure while profiling. Profiling time involves measuring the amount of time for a specific piece of code or a program to execute. This can help identify bottlenecks and performance issues in the code.
  - **Wall time** – is time of a specific function. It is a difference between the start and the end of the function.
- **Functions calls** – The number of calls for each function in the program can be helpful in finding non-optimal parts of the code. simple solution can be the cached result of the function so that the developer does not have to call it more than it is needed. For example, if the function is loading data from a database it can save many function calls but also total performance time.
- **CPU** – CPU profiling focuses on monitoring the usage of the central processing unit (CPU). It helps identify how much CPU time is spent on various functions or code sections, aiding in optimizing code for efficiency.
- **Memory** – Memory profiling assesses the usage of system memory (RAM) by a program. It helps find memory leaks, inefficient memory allocation, and excessive memory usage, which can lead to performance problems or crashes.
- **Instructions** – Instruction profiling involves counting the number of machine instructions executed by a program. This can be useful for identifying code segments that may be expensive.

Chapter 3

Perun

In this chapter, I am going to describe what is Perun. **Per**formance **Un**der Control is an open-source lightweight Performance Version System that acts as a wrapper for current Version Control Systems while also managing performance profiles corresponding to different project versions [9]. It also includes a tool suite that allows you to automate performance regression test runs, post-process existing profiles, and understand the findings [8]. [1]

3.1 Overview

Perun has the following advantages over databases and sole Version Control Systems:

1. **Context** – every performance profile has a specific minor version associated with it, which fills in the gaps in your profiles with information about what was changed in the code base, when it was updated, who made the changes, and so on. The profiles themselves provide more details on the application setup or the performance regression run, in addition to the data that was gathered.

2. **Automation** – Perun allows one to easily automate the process of profile collection, eventually reducing the entire process to a single command, and can thus be hooked, for example, when one commits new changes, in the supported version control system to ensure that one never forgets to generate new profiles for each new minor or major version of the project.

3. **Genericity** – The supported format for performance profiles is based on JSON notation, with only minor needs and limits. The Perun tool suite includes a foundational set of general (as well as numerous particular) visualizations, postprocessing, and collecting modules that may be used as building blocks for automating activities and understanding results. Perun has only a few minimal requirements for developing and registering new modules.

4. **Easy to use** – Perun's workflow, interface, and storage are significantly inspired by git systems, with the goal of being easy to use (at least for the majority of prospective users). The current version has a Command Line Interface with git-like commands (e.g., add, status, and log). The Interactive Graphical User Interface is being worked on right now. [8]

Figure 3.1 shows a comparison of the git and Perun systems.

[1] This chapter is based on Perun documentation [8, 1]

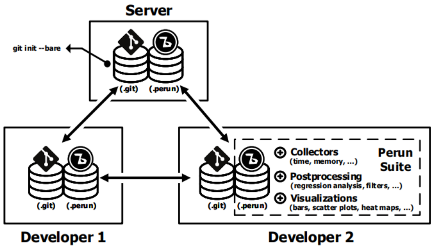

Figure 3.1: Comparison of the git and Perun systems [8]

3.2 Architecture

Perun is a tool package and a wrapper for Version Control Systems (VCS) like git that keeps track of performance profiles for different project versions. Figure 3.2 shows the workflow of the Tracer profiler is divided into four steps. [1]

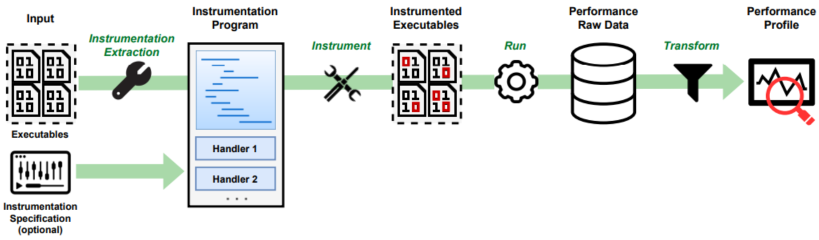

Figure 3.2: Comparison of the git and Perun systems [1]

- The input executables are then utilized to determine possible instrumentation locations (unless the user explicitly specifies them), and an instrumentation program and instrumentation handlers are built. The instrumentation program tells the instrumentation tool how to do the instrumentation, and the handlers tell the instrumentation tool what to do at specific instrumentation points.
- The produced instrumentation software and handlers are used to instrument the input executables.
- The executables received are started (in the manner chosen by the user) and generate raw performance statistics.
- Finally, the raw performance statistics are then turned into a performance profile.

Perun's internal architecture is organized into three sections: logic (commands, jobs, runners, and store), data (vcs and profile), and tool suite (collectors, postprocessors, and visualizers). The foundation of Perun is data, which includes profile manipulation and supporting wrappers (currently git and basic custom vcs) over existing version control systems. The logic is in charge of automation and higher-level logic operations, as well as profile development. Furthermore, the Perun suite includes a collection of collectors for profile creation, a collection of postprocessors for transformation, and a variety of visualization approaches and wrappers for graphical and command-line interfaces. [8]

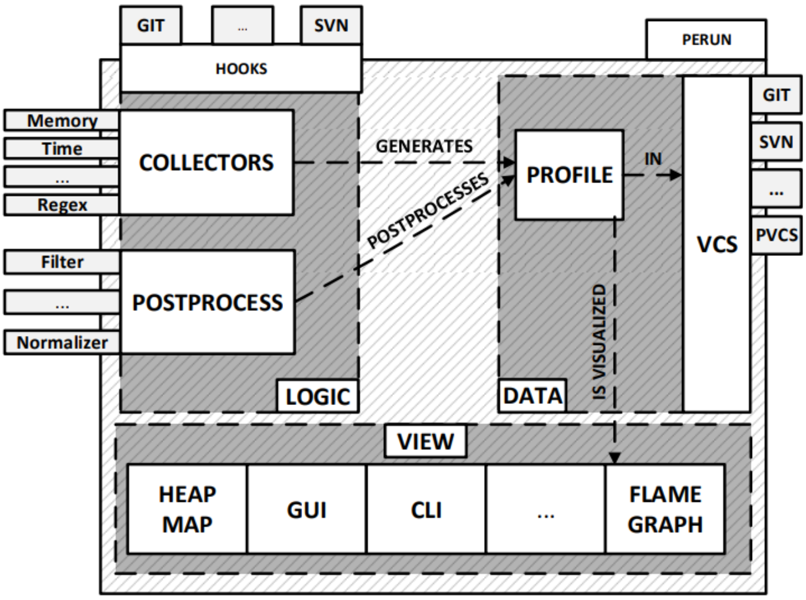

Figure 3.3: The scheme above shows the basic decomposition of Perun suite into sole units [8]

3.3 Detecting Performance Changes

Every change of the project, and every new minor version, can cause a performance degradation of the project. And manual evaluation of whether the degradation has happened is hard. Perun allows one to automatically check the performance degradation between various minor versions within the history and protect the project against potential degradation introduced by new minor versions. Potential changes in performance are then reported for pairs of profiles, together with more precise information, such as the location, the rate, or the confidence of the detected change. The detection of a performance change is always checked between two profiles with the same configuration (i.e. collected by the same collectors, postprocessed using the same postprocessors, and collected for the same combination of command, arguments, and workload). These profiles correspond to some minor version (so-called target) and its parents (so-called baseline). But baseline profiles do not have to be necessarily the direct predecessor (i.e. the old head) of the target minor version and can be found deeper in the version hierarchy. [8]

Chapter 4

Existing TypeScript profilers

In this chapter, I will briefly describe existing profilers that support analyzing the performance of Typescript applications.

4.1 V8

V8 is an open-source high-performance Javascript and WebAssembly engine made by Google that also supports profiling Typescript. V8 is also used in other third-party profiling tools such as *Node*, *Chrome*, *Deno*, and so on. V8 is able to collect data about Walltime, CPU and Memory profiling[1].

4.1.1 TS-Node

As the title suggests, TS-node is a TypeScript execution engine for Node.js and REPL (Read-Eval-Print Loop) [2]. Node.js supports profiling of Typescript applications using *TS-Node* that is based on *V8*. The output of the profiling is *.log* file that needs to be processed afterward to be in human-readable format Figure 4.1 shows an example output of profiling a Node application[2].

```
Statistical profiling result from isolate-0x5872c50-12236-v8.log, (1482 ticks, 0 unaccounted, 0 excluded).

 [Shared libraries]:
   ticks  total  nonlib   name
   1098   74.1%          /home/arcanebuchta/.nvm/versions/node/v14.21.3/bin/node
     96    6.5%          /usr/lib/x86_64-linux-gnu/libc.so.6
      4    0.3%          /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.32

 [JavaScript]:
   ticks  total  nonlib   name
      4    0.3%    1.4%  LazyCompile: *resolve path.js:1067:10
      2    0.1%    0.7%  LazyCompile: *normalizeString path.js:59:25
      1    0.1%    0.4%  LazyCompile: *realpathSync fs.js:1723:22

 [C++]:
   ticks  total  nonlib   name
    153   10.3%   53.9%  epoll_pwait@@GLIBC_2.6
     47    3.2%   16.5%  __write@@GLIBC_2.2.5
     16    1.1%    5.6%  pthread_cond_signal@@GLIBC_2.3.2
     13    0.9%    4.6%  fwrite@@GLIBC_2.2.5
     13    0.9%    4.6%  __mprotect@@GLIBC_PRIVATE
```

Figure 4.1: Node.js profiling output

[1]You can find out more about V8 here: https://v8.dev/docs

[2]You can find out more about TS-Node here: https://www.npmjs.com/package/ts-node

4.1.2 Chrome built-in profiling

Chrome like other browsers has its built-in profilers. We can find it by opening any browser tab and clicking *F12* or right-click select *Inspect*. After this, we can find it in *Performance* tab . In Figure 4.2 we can see an example output of Chrome.

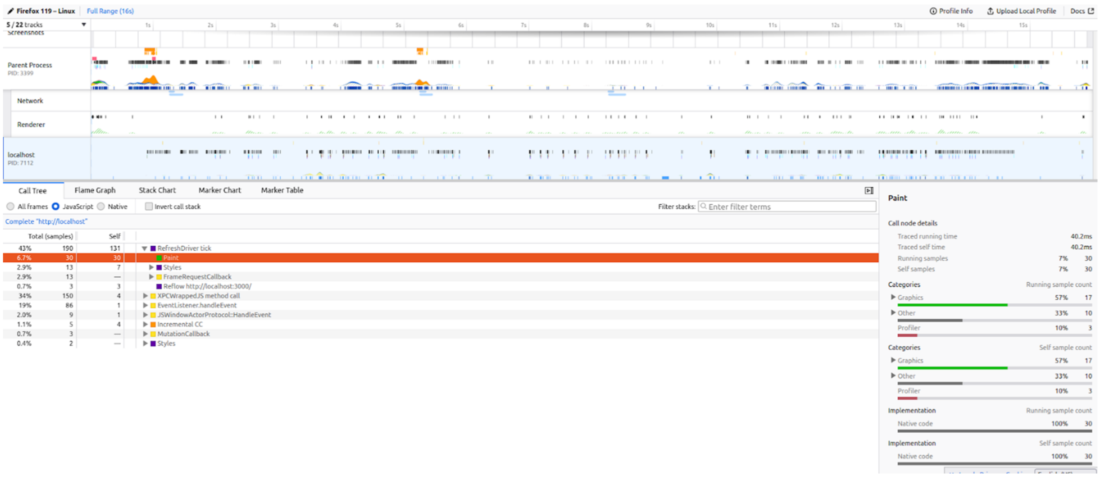

Figure 4.2: Chrome profiling output

4.1.3 Deno

Deno is primarily a Typescript profiler that runs the server that serves the website. We can think of Deno as the server-side profiler, for example for profiling Node. So it cannot be used to profile client site Typescript webs.

4.2 Bun

4.3 Other Profilers supporting Typescript

In this section, I will shortly introduce other possibilities to profile Typescript. I want to pinpoint that these profilers can profile *Node.js* and *Typescript*, but it is not their primary focus to do so.

- Gecko profiler - like Chrome, Firefox also has built-in profiling for web application. It can be used the same way as the one in Chrome as I described here 4.1.2. Gecko Profiler is basically a C++ component that is used for instrumenting Gecko. It is fully configurable and supports a wide range of data sources and recording modes. It is primarily used as a statistical profiler, pausing the execution of threads registered with the profile in order to collect a sample. [7]. In Figure 4.3 we can see how does the output can look like.

- OProfile - does not profile during runtime of the web. OProfile is part of Linux command *operf*[3]

[3]See more about *operf* here: https://man7.org/linux/man-pages/man1/operf.1.html

- Intel VTune - is primarily designed for profiling native code, including C, C++, and Fortran. While it can profile Node.js applications to some extent, its focus is not on TypeScript or JavaScript. In Figure 4.4 we can see how does the output can look like.

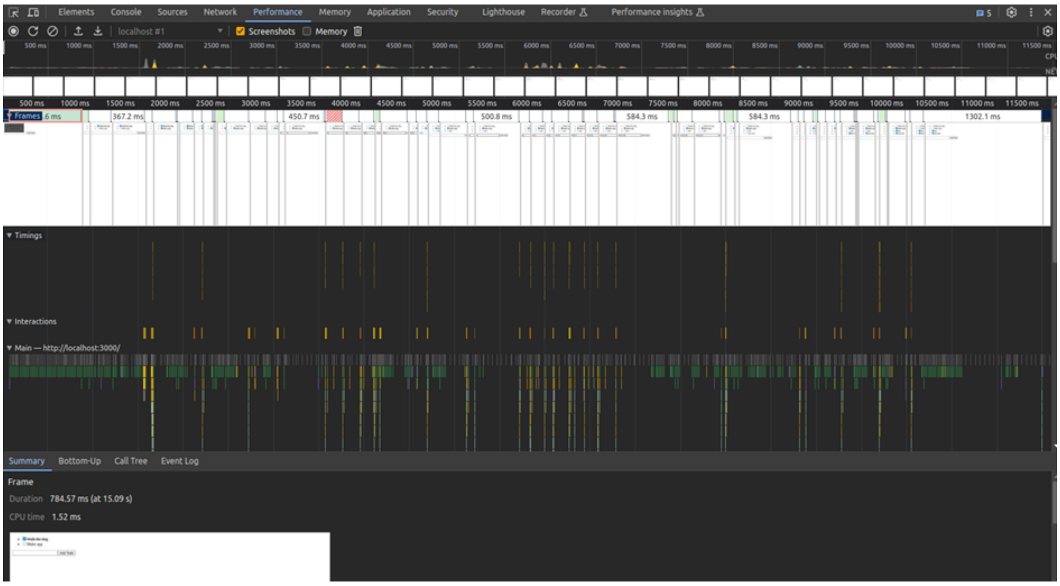

Figure 4.3: Firefox profiling output

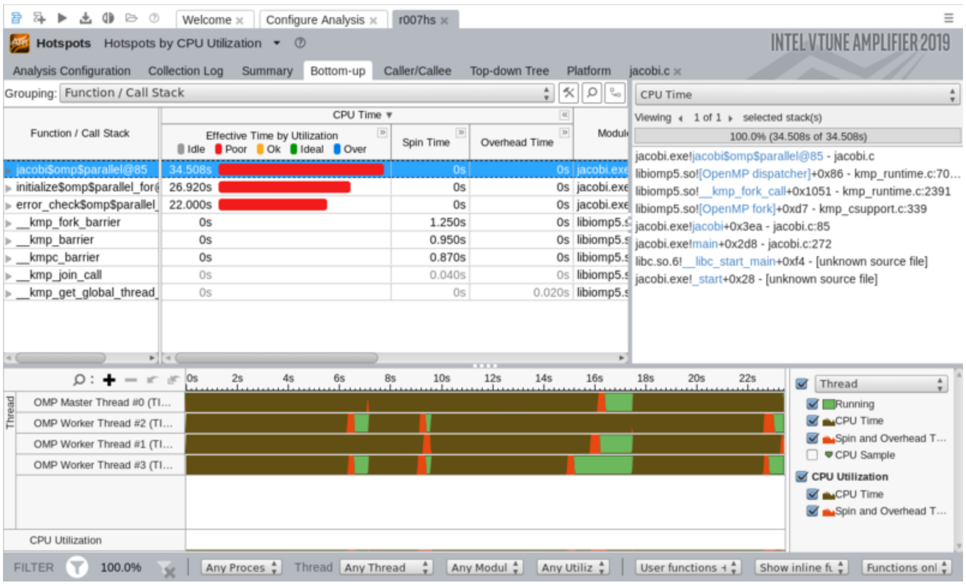

Figure 4.4: Intel VTune profiling output

Chapter 5

Analysis of requirements

Chapter 6

Architecture for Profiling

Chapter 7

About implementation

Chapter 8

Conclusion

Bibliography

[1] Fiedor, T., Pavela, J., Rogalewicz, A. and Vojnar, T. *PERUN: Performance Version System* [online]. June 2022 [cit. 2023-10-30]. Available at: https://arxiv.org/pdf/2207.12900.pdf.

[2] Fiedor, T., Pavela, J., Rogalewicz, A. and Vojnar, T. *PERUN: Performance Version System* [online]. June 2022 [cit. 2023-10-30]. Available at: https://www.npmjs.com/package/ts-node#overview.

[3] geeksforgeeks.org. *Difference between Static and Dynamic Testing* [online]. May 2019 [cit. 2023-10-15]. Available at: https://www.geeksforgeeks.org/difference-between-static-and-dynamic-testing.

[4] Gregg, B. System Performance: Enterprise and the Cloud. In:. 2nd ed. Pearson, 2020, chap. 2, p. 35. ISBN 9780136820154.

[5] influxdata.com. *Continuous Profiling* [online]. April 2023 [cit. 2023-10-18]. Available at: https://www.influxdata.com/glossary/continuous-profiling.

[6] microsoft.com. *Understand profiler performance collection methods* [online], 15. march 2023 [cit. 2023-10-23]. Available at: https://learn.microsoft.com/en-us/visualstudio/profiling/understanding-performance-collection-methods-perf-profiler.

[7] mozilla.org. *Gecko Profiler* [online], 21. november 2020 [cit. 2023-12-03]. Available at: https://firefox-source-docs.mozilla.org/tools/profiler/index.html.

[8] Pavela, J. and Fiedor, T. *Perun Documentation* [online]. June 2022 [cit. 2023-10-30]. Available at: https://raw.githubusercontent.com/tfiedor/perun/master/docs/pdf/perun.pdf.

[9] Pavela, J., Fiedor, T., Vojnar, T., Podola, R. and Míchal, O. *Perun: Lightweight Performance Version System* [online], 28. september 2023 [cit. 2023-10-24]. Available at: https://github.com/Perfexionists/perun/blob/devel/README.md.

[10] Sipriano, R. *Https://www.linkedin.com/pulse/what-actually-software-performance-rodrigo-sipriano* [online]. December 2021 [cit. 2023-10-14]. Available at: https://www.linkedin.com/pulse/what-actually-software-performance-rodrigo-sipriano.

[11] thenewstack.io. *What Is TypeScript?* [online]. July 2022 [cit. 2023-10-14]. Available at: https://thenewstack.io/what-is-typescript.

[12] W3SCHOOLS.COM. *TypeScript Introduction* [online]. March 2022 [cit. 2023-10-14]. Available at: https://www.w3schools.com/typescript/typescript_intro.php.