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

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

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

The goal of this work is to extend the open-source Performance Version Control System — Perun by implementing a module that is able to profile web pages programmed in TypeScript or JavaScript. The profiler can collect metrics such as website loading time, responsiveness during interactions with the website, or memory consumption. At the same time, the module supports visualization of the measured results using a heatmap, a pairplot or a call graph.

Abstrakt

Cieľ tejto práce je rozšíriť verejne dostupný verzovací systém – Perun implementáciou modulu, ktorý je schopný profilovať webové stránky naprogramované pomocou TypeScriptu prípadne JavaScriptu. Profiler je schopný zbierať metriky, ako napríklad dobu načítania webu, odozvu pri interakciách s webom čí spotrebu pamäti. Zároveň je modul schopný vizualizovať namerané výsledku pomocou heatmapy, pairplotu alebo grafom volaní.

Keywords

Typescript, profiling, Perun, continuous monitoring, Node.js, OpenTelemetry

Klíčová slova

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

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,

Rozšířený abstrakt

Každý vývojár chce aby jeho aplikácia bola bezchybná. Avšak, vývojári sa často stretávajú s rôznymi problémami, ktoré im v tom bránia. Jeden z najčastejších problémov je práve *výkon*. Častý výkonnostný problém je napríklad tzv. *bottleneck* (úzke miesto). Bottleneck označuje časť systému, ktorá obmedzuje jeho celkový výkon a spomaľuje beh programu. Tento faktor môže spomaľovať alebo brániť efektívnemu fungovaniu celej sústavy. Riešenie tohto problému vyžaduje optimalizáciu programu, ktorá začína identifikáciou miesta v systéme, kde sa bottleneck nachádza. Najčastejší spôsob, ako získať údaje o výkone a výkonnostné metriky je *profilovanie*. Hlavným cieľom profilovania je získavanie tzv. kľúčové ukazovatele výkonnosti (KPI), ako napríklad oneskorenie webovej stránky, vstupné/výstupné operácie za sekundu (IOPS), utilizácia (ako sú zdroje vyťažené) alebo zaťaženie CPU. Pri webových aplikáciách sa môžu vyskytnúť ďalšie KPI, čo môže zvýšiť zložitosť analýzy programu. Pre lepšie porozumenie nazbieraných dát sa využívajú vizualizácie, ktoré uľahčujú interpretovanie výsledkov. Medzi bežné spôsoby vizualizácií výsledkov profilovania (profilov) patria *heat mapy*, *flame grafy* alebo rôzne stromové štruktúry.

TypeScript je jeden z populárnych programovacích jazykov na vytváranie webových aplíkácii. TypeScript je *open-source*, silne typovaný jazyk vyvíjaný Microsoftom. TypeScript je nadstavba JavaScriptu a na jeho správne fungovanie je potrebné ho transpilovať do čistého JavaScriptu. Na rozdiel od JavaScriptu podporuje statické typovanie a podporuje objektovo orientované programovanie.

Hlavný cieľ tejto bakalárskej práce je vytvoriť nový modul, ktorý rozšíri funkcionalitu nástroja Perun, vyvíjaný výskumnou skupinou *VeriFIT* na Fakulte informačných technológií v Brne na VUT. Nový modul by mal byť schopný profilovať webové aplikácie naprogramované v TypeScripte a byť schopný vizualizovať výsledky v podobe rôznych grafov.

V práca sa podarilo splniť všetky stanovené ciele, čo bolo potvrdené testovaním profileru na sérii rôznych experimentov. Profiler podporuje web stránky programované pomocou TypeScriptovej knižnice *Express* a dokáže tvoriť a vizualizovať profily. Modul podporuje vizualizácie čiarovými grafmi, heat mapy, ale aj tzv. *pairplot*. Modul je zároveň integrovaný priamo do nástroja Perun, čo bolo nad rámec zadania. Keďže je to len prvá verzia profileru, je tu ešte veľa možností, ako ho rozšíriť o nové možnosti, napríklad o podporu rôznych *frameworkov*, ako *React*, či *Vue*.

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
June 17, 2024

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 developer wants to create a perfect application. However usually encounters many different problems and bugs that are in the way to do so and one of the most common issues is the *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.“ [33]

To optimize such bottlenecks away, one must first detect and locate them. *Profiling* is one of the most common approaches for gathering performance data about programming. The main goal of profiling is to identify and measure key performance indicators (KPI) 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) or CPU load. However in e.g. the web application there can be more suitable KPIs making the analysis even more difficult. For a better understanding of the gathered data, performance engineers rely on visualizations to help them interpret the overall program performance. Common ways of visualizing the results of profiling (called *profiles*) are heat maps, flame graphs, tree maps, and others.

TypeScript is one of the most popular languages for creating web applications as the strongly typed dialect of JavaScript. TypeScript is also a free, transpiled, 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 [39]. Unlike JavaScript, TypeScript supports object-oriented programming concepts in the vein of classes, interfaces, and inheritance. [37]

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

This thesis is organized as follows. Program analysis and profiling are introduced in Chapter 2. Perun, the version control system, is introduced in Chapter 3. A comparison

of existing open-source TypeScript profilers is presented in Chapter 4. Analysis of requirements for profiler for this thesis discussed in Chapter 5. The design and architecture of the module are described in Chapter 6. All other information about the implementation is presented in Chapter 7. Experiments conducted with the programmed application on a test project with a real use case will be illustrated in 8.

Chapter 2

Performance Analysis

This chapter will introduce the concept of software analysis (and its categories) profiling, and other concepts related to profiling.

Performance analysis is a critical aspect of software development and system optimization. Whether you are a performance engineer, a developer, or a system administrator, 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 Static Analysis vs. Dynamic Analysis

Program analysis can be divided into two groups based on when the analysis is performed — static analysis and dynamic analysis. These two approaches provide different perspectives on evaluating software quality and performance, each with its advantages and limitations. Understanding both approaches is crucial for effectively assessing and improving software projects. This section is based on the following sources [18, 5, 8, 38].

2.1.1 Static Analysis

Static analysis involves analyzing the machine or source code of a program without actually running it. Static analysis is usually carried out in the early stages of development to detect program bugs as soon as possible. Static analysis is performed, for example, by *compilers* during type checking or analyses for optimization, which identify valid performance-improving transformations. Some tools can also visualize the code in addition to identifying bugs. Static analysis can easily uncover certain bugs that cannot be reliably detected through dynamic analysis. Purely static analysis tools are a nice alternative to dynamic analysis tools because they consider all program paths. However, there are many bugs, such as memory-related issues, that static analysis may not be able to uncover.

> Static analysis can handle very large systems and does often not need a model of the environment but it can produce many false alarms. [38]

2.1.2 Dynamic Analysis

Dynamic analysis does not have a clear definition. A lot of authors define dynamic analysis in a different way. One of the definitions by *Michael D. Ernst* states:

> Dynamic analysis operates by executing a program and observing the executions. Testing and profiling are standard dynamic analyses. [5]

Tools performing dynamic analysis are usually able to instrument the target code (except for testing). The analysis code runs as part of the program's normal execution, without disturbing the program's execution. The analysis must maintain some kind of analytical state, which we can call metadata. These are crucial for dynamic code analysis. [18] An example of dynamic analysis could be *profiling*.

2.2 Profiling

Profiling creates a picture of the program's performance as a whole. Profilers are tools used by developers and performance engineers to identify problematic parts of a program. In the context of profiling, the problematic parts refer to source code that causes I/O bottlenecks, CPU bottlenecks, suboptimal database queries, memory or, power consumption issues, etc. Profilers are capable of obtaining information about a project such as the number of calls to individual functions, the execution time of functions, memory usage, and (de)allocation, or simply finding *hot paths* in your code. Some profilers can profile a project without directly modifying the project's code. However, there are profiling techniques, based both on sampling and tracing approaches, where direct code instrumentation is required. [10, 34]

2.3 Continuous Monitoring

Continuous monitoring is a real-time method used for the analysis and management of (typically deployed) production applications, identifying issues such as performance, user experience, CPU utilization, disk I/O consumption, network usage, and function call time, while also tracking resource utilization and detecting performance degradation at the cost of higher overhead. Continuous profiling provides developers with additional insights into the code that is being expected. Monitoring can take place at all layers of the software system, from hardware to application state and it often involves some form of dynamic analysis.

Continuous monitoring can be used to profile long-running reactive systems and applications such as web pages or cybersecurity intrusion detection systems. This section is based on the following sources: [13, 11].

2.4 Sampling

Sampling creates so-called *samples* (statistical data) with a certain sampling frequency during program execution, forming a coarse picture of the program's performance. These samples can contain data, e.g. about hardware counters state, memory usage, or time since the application started. The lower the sampling frequency, the less data sampling captures, while a higher sampling frequency increases overhead. The advantage of samplers is that their performance overhead can be lower than that of tracing tools. [9] Tracing will be described in more detail in section 2.6.

Sampling is less accurate in terms of individual function calls, but generally incurs low overhead and has less impact on how the program under profiling runs.

2.5 Instrumentation

Instrumentation creates instrumentation points or so-called 'hooks' after the program starts by modifying instructions in memory and inserting instrumentation routines. However, it's important to note that some instrumentation methods like compile-time instrumentation insert code during the compilation process. Instrumentation creates instrumentation points or so-called *hooks* after the program starts by modifying instructions in memory and inserting instrumentation routines. A similar principle applies to inserting breakpoints during program debugging. Unlike traditional debugging, dynamic instrumentation offers flexibility by allowing the insertion of instrumentation at any point in the program's execution, including third-party libraries. This capability facilitates the creation of custom performance statistics for any software. Instrumentation thus enhances the level of observability of the target program. This approach facilitates uncovering issues, profiling performance, and gaining deep insights into the runtime characteristics of a software system. [10]

2.6 Tracing

Tracing an application involves capturing events within the target program. It records and stores various data that can be further processed and utilized for application analysis. The main difference between *tracing* and *sampling* lies in the fact that sampling collects data periodically at certain intervals, while tracing records specific events at the time they occur, such as system calls, process creation and termination, CPU utilization, or custom events for debugging and optimization. [10, 9]

Tracers utilize various event sources, including static and dynamic instrumentation or technologies like BPF (Berkeley Packet Filter). The Linux operating system has several built-in tracing tools, such as strace (for system calls), tcpdump (network packets), Linux Ftrace, BCC, and bpftrace, which are general-purpose tracing tools. [10, 9]

Compared to the method to sampling-based techniques, tracing methods typically result in a higher overall overhead.

2.7 System Performance Metrics

In profiling, we can measure different *metrics*, *also known as Key Performance Indicators (KPI)*. Metrics can be defined as measurable quantities that describe a performance aspect of the program 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. We are going to list some of the most common performance metrics. The format of this section was inspired by [12].

- **Execution Time Analysis** – 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.
  - **Total elapsed time** – amount of time required to execute the entire program. This metric is irrelevant for measuring web applications because here we focus on continuous monitoring.
  - **Function time** – refers to the amount of time spent executing a specific function within a program.

  - **Basic block time** – is continuous code segment that has no branches. You can read more about basic blocks here: [7].

- **Time Measurement Types** – Profiling also distinguishes different types of time that can be measured.
  - **Wall time** – „is real-world time or elapsed time that program or process takes to run from start to finish as measured by a person“ [36].
  - **CPU time** – CPU time is the duration during which the CPU processes data for a program or process, or when the program actively engages the CPU in tasks, such as performing arithmetic and logic operations. The processor is typically not utilized at 100% during the execution of a program or application. Part of this time is spent on fetching and storing data in RAM or on an external storage device. [35]
  - **Kernel time** and **User time** – CPU time is primarily divided into two groups: time spent at the user level, *user time*, and time spent at the kernel level, *kernel time*. Other groups of CPU time include Idle Time, Interrupt Time, and Context Switch Time. Kernel time encompasses activities such as system calls, kernel threads, and interrupts. The user time/kernel time ratio, measured across the entire system, indicates the type of work being performed. Computation-intensive applications may spend most of their time executing user-level code, resulting in a user/kernel ratio of up to 99:1. On the other hand, applications that are I/O-intensive make many system calls, increasing the proportion of kernel time. This ratio depends on many other factors. [9]

- **Functions calls** – The number of calls of each function in the program can be helpful in finding non-optimal parts of the code. A simple can be the caching results of the function so that the developer does not have to call it more than it is needed. For example, if a function is loading data from a database caching queries, and speeding up the program as a result can save many function calls.

- **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, we are going to introduce Perun, **Per**formance **Un**der Control. Perun is an open-source lightweight Performance Version System that serves as a wrapper over Version Control Systems like Git. Perun mainly focuses on profiling the duration of functions and programs. Perun tracks performance profiles linked to various project versions while managing version control systems like Git. This tool provides a versatile suite for automating performance regression tests, processing profiles (including creating performance models), and interpreting results. Essentially, Perun streamlines version management and offers automation capabilities for performance testing, ensuring efficient tracking and analysis of a project's performance history.[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 automates the process of data collection and profile creation for individual minor versions, eventually condensing the entire process into a single command. It can generate profiles with each new commit in the supported version control system to ensure no new minor or major version of the tracked program is missed. The exact specification of tasks for this process is stored and loaded in YAML format. YAML files serve as a natural format for defining task automation. The task specification is inspired by continuous integration systems.

3. **Genericity** – The format of performance profiles is based on the JSON format, it has no major restrictions and requirements. Perun includes a set of visualization methods and several advanced options for visualization, postprocessing, and modules used as building blocks for task automation. At the same time, Perun is easily expandable. It does not impose great requirements for adding new modules such as new data collectors, postprocessing, customized visualization, or another version control system.

[1] This chapter is based on Perun documentation [6, 28, 29]

4. **Easy to use** – Perun's workflow, interface, and storage are significantly inspired by the git system, 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 still in development. [28]

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

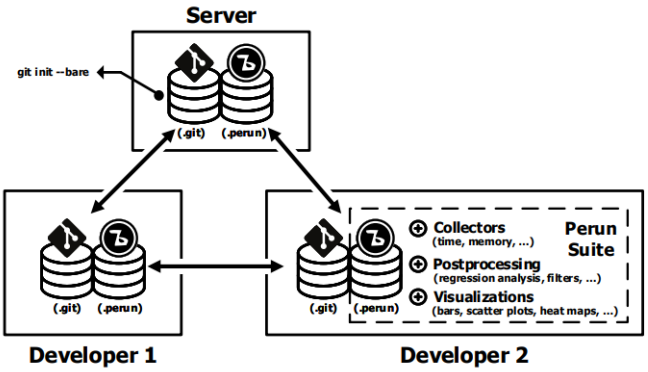

Figure 3.1: Comparison of the git and Perun systems [28].

3.2 Architecture and Technical Details

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.

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 profilers that create profiles, a collection of postprocessors for transformation, and a variety of visualization approaches and wrappers for graphical and command-line interfaces. [28]

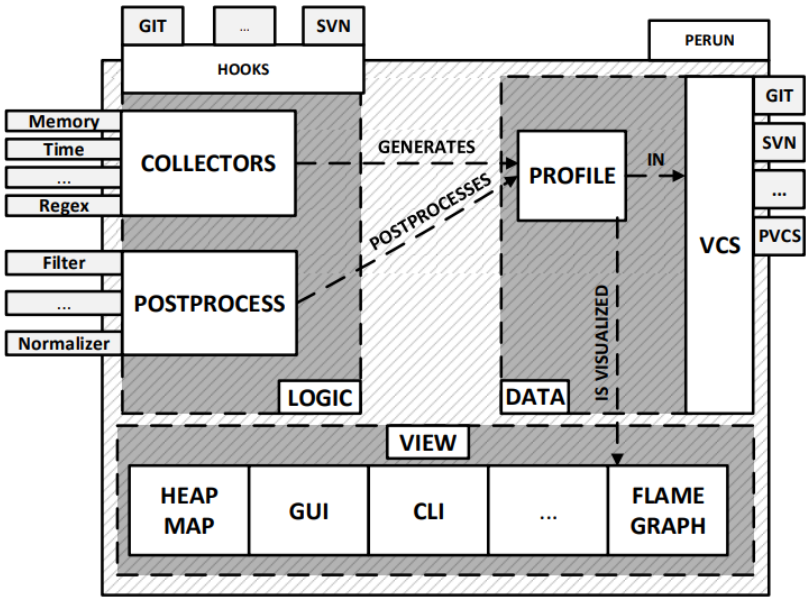

Figure 3.2: The scheme above shows the basic decomposition of Perun suite into sole units [28]

3.2.1 Architecture of Tracer Profiler

This subsection briefly introduces one of the profilers (collectors) within Perun, called *Tracer* as an illustration of Perun's architecture.

- Potential instrumentation locations are sought in the input executables unless specified manually. Subsequently, an instrumentation program and instrumentation handlers are generated. The instrumentation program guides the instrumentation tool on how to proceed with instrumentation, and the handlers specify what should happen at individual instrumentation points.
- The produced instrumentation program and handlers are used to instrument the input executables.
- The executables are started and generate raw performance statistics.
- Finally, the raw performance statistics are then turned into a performance profile. [6]

Figure 3.3 shows the workflow of the Tracer profiler is divided into four steps. [6]

3.3 Detecting Performance Changes

Performance degradation of the program can be caused by even a small change in the program, a minor version, or every release. Manual degradation evaluation is too complex, often requiring the examination of multiple use cases. Perun allows the process of performance degradation checking to be automated by assigning a performance profile to each version and release. Perun can automatically check performance degradation between any

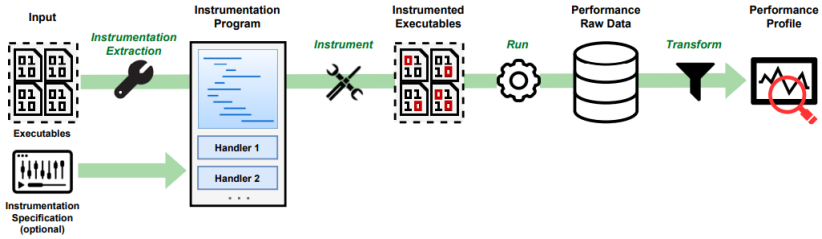

Figure 3.3: Comparison of the git and Perun systems [6]

minor versions with the same configuration in history, thus protecting the project from potential performance degradation. Potential performance changes are then reported along with more precise information (such as location, and speed) for specific pairs of profiles compared by Perun. The result of Perun's version comparison could be, for example:

- *No Change,*
- *Total Degradation* or *Total Optimization,*
- *Not in Baseline* or *Not in Target,*
- *Severe Degradation* or *Severe Optimization,*
- *Degradation* or *Optimization,*
- *Maybe Degradation* or *Maybe Optimization,*
- *Unknown.*

This section is inspired by [28]. Refer to section *8.1 Results of Detection* for more information.

Chapter 4

Existing TypeScript Profiler Frameworks

In this chapter, we will briefly introduce 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 such as Walltime, CPU time utilization, and Memory consumption[1]. Figure 4.1 shows an example output of a Node application profiling.

Figure 4.1: Example of Node.js profiling output after post-processing.

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) [4]. Node.js supports profiling of TypeScript applications using

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

TS-Node that is based on *V8*. The output of the profiling is a *.log* file that needs to be post-processed to be in a human-readable format[2].

4.1.2 Chrome Built-in Profiler

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

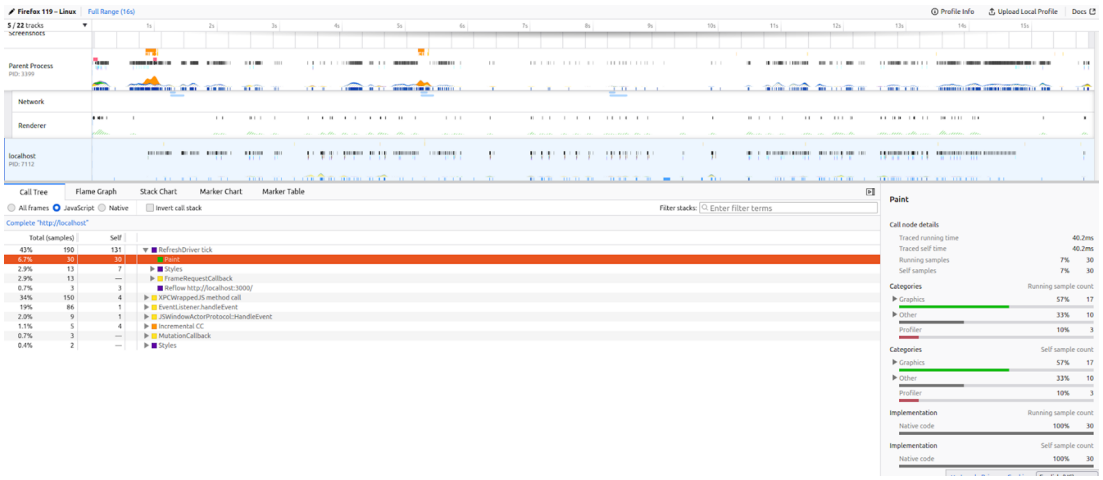

Figure 4.2: Chrome Performance tab – output after recording.

4.1.3 Deno

Deno is a runtime environment for JavaScript, TypeScript, and WebAssembly. Deno isn't primarily a TypeScript profiler, but it can profile TypeScript to some extent. We can think of Deno as the server-side profiler, for example for profiling Node. Deno UI resembles the built-in Chrome profiler as Figure 4.3 and Figure 4.4 below show. This means Deno may not be the optimal tool for handling the expected use case. You can find more about installation and usage here [2].

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

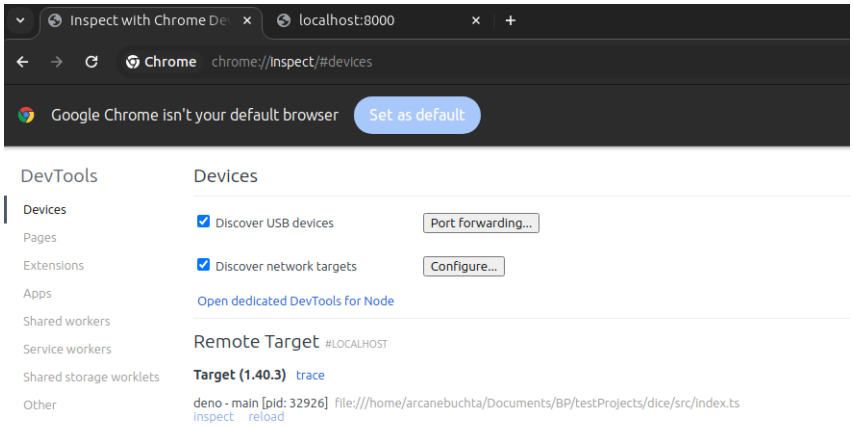

Figure 4.3: Deno tab after running it on project – ready to record metrics.

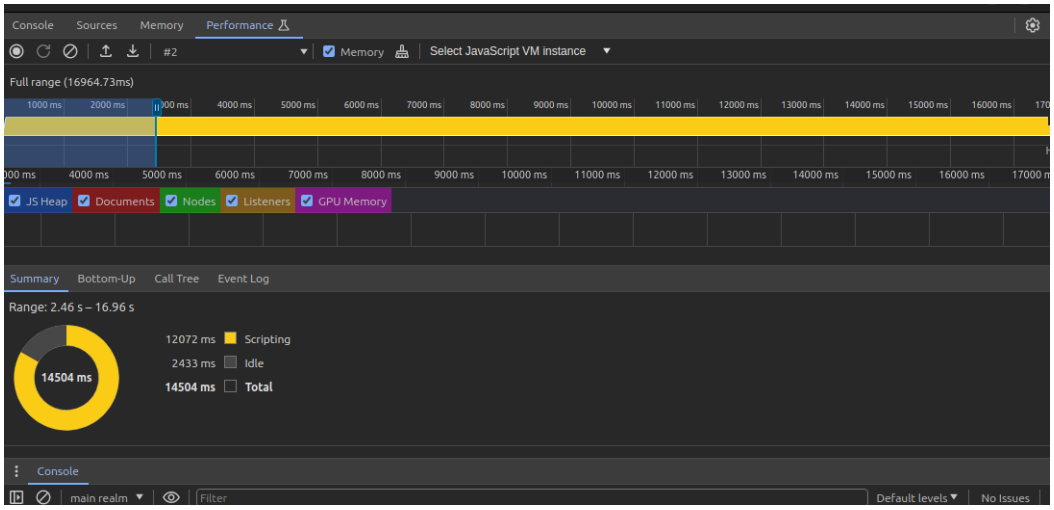

Figure 4.4: Example of Deno profiling output after recording.

4.2 JSC

JSC or *JavaScriptCore* is a Javascript profiler also supporting TypeScript. For instance, the *Bun* package management for Node uses it. A sample profiler from JavaScriptCore powers the JavaScript and Events timeline in Web Inspector[3]. JSC's previous tracing profiler has been replaced with the sampling profiler. In addition to being orders of magnitude quicker than the previous tracing profiler, it offers more precise information about where time is spent inside the running application. [40]

4.3 Other Profilers Supporting TypeScript

In this section, we will shortly introduce other possible ways to profile TypeScript. We aim to emphasize that these profilers can profile *Node.js* and *Typescript*, but it is not their primary focus.

- Gecko profiler – like Chrome, Firefox also has a built-in profiler for web applications. It can be used the same way as the one in Chrome we described in section 4.1.2. Gecko Profiler is basically a C++ component that is used for instrumenting Gecko. „*Gecko is Mozilla's rendering engine for the web. It is made up of HTML parsing and rendering, networking, JavaScript, IPC, DOM, OS widget abstractions, and much much more.*“ [17]. 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 profiler in order to collect a sample. [16] Figure 4.5 shows an example of what the output can look like.

- OProfile – OProfile is an open-source project containing a statistical profiler for the Linux operating system. OProfile can profile entire code with minimal overhead. The profiler uses performance counters of the CPU to acquire many different metrics. OProfile also supports creating a call graph of the profiled program. Several post-processing tools transform the data into a human-readable format. [26] OProfile does not profile during the runtime of the web. OProfile is part of Linux command *operf*[4].

- Intel VTune – 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. [31, 15] Figure 4.6 shows an example output of the *Intel VTune*.

[3] For an overview of Web Inspector, you can explore further details here https://developer.apple.com/documentation/safari-developer-tools/web-inspector

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

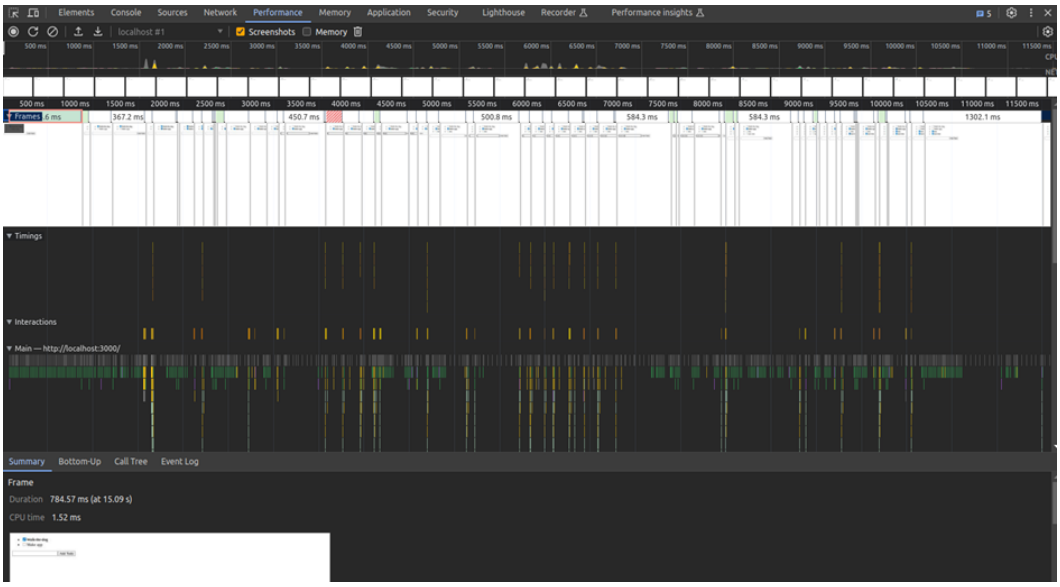

Figure 4.5: Example of Firefox profiling output after recording.

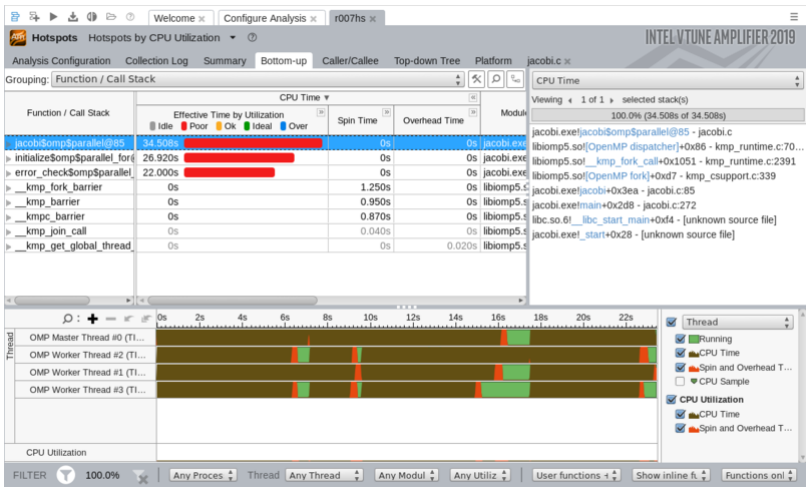

Figure 4.6: Illustrative example of the profiling output from Intel VTune [14].

4.4 Summary

This section provides a comparison of various profilers discussed in this chapter. This summary serves as a reference point for comparing the features and characteristics of each profiler, aiding in the selection of the most suitable tool for specific profiling needs. Some of them are only sampling profilers, and some have more options of observing the code. The important characteristic for this thesis is instrumentation. This columns contains both automatic and and custom instrumentation. The main distinction between automatic and custom instrumentation lies in how the profiler gathers data: automatic instrumentation applies predefined monitoring methods, while custom instrumentation allows users to define their own monitoring criteria.

	Sampling + Tracing	Instrumentation	License
V8	Yes	Yes	Open-source
JSC	Yes	No	Open-source
Gecko profiler	Yes	Yes	Open-source
OProfile	Yes	No	Open-source
VTune	Yes	Yes	Proprietary

Table 4.1: Profilers summary

Chapter 5

Analysis of Requirements

In this chapter, we are going to summarize the intended functionality of the TypeScript profiler and list its basic requirements. The profiler, in general, should meet certain criteria. The criteria are divided into functional and non-functional requirements. The format of this chapter was inspired by [20].

5.1 Analysis of Requirements for a TypeScript Profiler

This section describes the intended requirements for the TypeScript profiler. Our focus centers on three core elements:

Metrics – Firstly, the most crucial metric to measure is the time taken for the initial pixels of the website to load and appear on the screen during the first loading of the website. Secondly, *latency* while interacting with the website, and last but not least measuring the memory consumption to gauge the overall resource utilization.

Overhead in general is expected to be as minimal as possible, since this is only the first version of the profiler, there is a possibility of encountering challenges. The overhead for now is expected to be below 50%.

Visualizations – metrics collected by the profiler need to be visualized. For this purpose, the module will use Python library *Pandas* to work with data as needed, and Python visualization libraries such as *MatPlotLib* to visualize metrics like requests over time in a line graph, *Seaborn* and *Holoview* to visualize the correlation between metrics like heatmap or pairplot. There is also an option to make a call graph using an open-source library for TypeScript projects[1].

5.2 Functional Requirements

Functional requirements define the specific functionalities that a system should be able to perform to meet the needs of its users.

- **F-IU (Independent Usability)** – The profiler is usable as an independent tool, and also as part of Perun.

[1] https://github.com/whyboris/TypeScript-Call-Graph

- **F-IP (Integration to Perun)** – The solution is integrated into Perun.
- **F-RO (Runtime Overhead)** – The profiler keeps runtime overhead as low as possible to provide the user with the most relevant data.
- **F-DP (Data Post-processing)** – The raw data collected by the profiler are post-processed to be easily understandable for the user. The processing is quick.
- **F-LM (Latency Measuring)** – The profiler is capable of measuring the elapsed time from the moment it starts loading content of the web application until it loads completely, essentially capturing the latency of the web application.
- **F-RR (Request Rate)** – The solution tracks the number of requests for each route.
- **F-MM (Memory Measurement)** – The solution measures used heap memory.
- **F-FS (File System Read/Write)** – The solution measures both the number of file system reads and writes performed by the application.
- **F-UST (User/System CPU Time)** – The solution measures both user CPU time and system CPU time of the application
- **F-USU (User/System CPU usage)** – The solution measures both user CPU usage and system CPU usage.
- **F-VI (Visualization)** – The solution is able to visualize post-processed data in graphs or a call graph.

5.3 Non-functional Requirements

Non-functional requirements specify the attributes that characterize the operation of a system, rather than its specific behaviors. While functional requirements describe the behaviors and features of the system, non-functional requirements describe the qualities that determine how well the system performs those behaviors and features.

- **NF-EU (Easy to Use)** – The solution is easy to understand and use even for new user.
- **NF-RP (Result Portability)** – The profiler generates the same or very similar results regardless of the runtime environment.
- **NF-MD (Minimum Dependencies)** – The solution uses only a bare minimum of mandatory dependencies to lower the entry barrier for new users to use it.
- **NF-SS (Storage Space)** – The resulting profile data should be stored efficiently while allowing for flexibility in storage requirements.
- **NF-RC (Reporting Comprehensibility)** – The reporting tool provides comprehensive outputs with easy-to-understand names, legible formatting, and easy-to-read graphs.

Chapter 6

Profiling Architecture

Several other methods are available for profiling TypeScript web applications beyond the profilers discussed in Chapter 4. These are tools to assist us in profiling rather than actual profilers for TypeScript. In this chapter, we will shortly describe what tools we considered to use for profiling, their pros and cons, and what we chose to build our own implementation on for this thesis. We will also introduce the architecture of the solution proposal.

6.1 Existing Tools to Make Profiling Easier

In this section, we are going to introduce some of the tools we experimented with while searching for the ideal tool for this thesis, one that is easy to use and capable of providing profiling data. Whey are not standalone profilers, but they make it possible to observe web applications in some ways.

6.1.1 OpenTelemetry

OpenTelemetry is an open-source observability framework and toolkit for creating, managing, and exporting telemetry data such as metrics, traces, and logs. OpenTelemetry can be used with various programming languages, e.g. C#, Java, Rust, PHP, but also Node.js (that includes both JavaScript and TypeScript). OpenTelemetry is in most cases used with observability backends, for instance, Jaeger, Prometheus, or Zipkin. Various exporters provided by OpenTelemetry enable data transmission to these backends or to the console. The key purpose is to provide a simple instrumentation regardless of programming language, environment, or infrastructure. Storing data and visualization are intentionally left to other tools. [25]

Metrics in OpenTelemetry are collected using *counters*, *gauges* or *histogram*. They are created using the OpenTelemetry meter. There are several types of counters and gauges. Each of them has its specific use case. You can see a list of them in Figure 6.1. We will describe more about how to use them in the section 7.1. After initializing the counter/-gauge/histogram, it is necessary to create middleware to fill it with data and specify an exporter. Exporters are used to export data to the observability backend, console or it is possible to create a custom exporter.

Instrument Kind	Selected Aggregation
Counter	Sum Aggregation
Asynchronous Counter	Sum Aggregation
UpDownCounter	Sum Aggregation
Asynchronous UpDownCounter	Sum Aggregation
Gauge	Last Value Aggregation
Asynchronous Gauge	Last Value Aggregation
Histogram	Explicit Bucket Histogram Aggregation, with the `ExplicitBucketBoundaries` advisory parameter if provided

Figure 6.1: List of OpenTelemetry counter used for processing data. You can read more about them here: [24].

Traces are made out of spans. Spans are obtained by automatic or custom instrumentation of the target project using *NodeTraceProvider*, *SpanProcessor*, and *exporter*. The exporter can be again one of the predefined ones or a custom one. You can create multiple spans for various purposes. Observing part of the target project or button clicks. Figure 6.2 shows an example of a trace generated by OpenTelemetry.


```
{
  "name": "hello",
  "context": {
    "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "0x051581bf3cb55c13"
  },
  "parent_id": null,
  "start_time": "2022-04-29T18:52:58.114201Z",
  "end_time": "2022-04-29T18:52:58.114687Z",
  "attributes": {
    "http.route": "some_route1"
  },
  "events": [
    {
      "name": "Guten Tag!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
}
```


Figure 6.2: A trace generated by OpenTelemetry containing timestamps, trace_id and span_id and other data [21].

Logs – Traces logs are shown above, more important for this thesis are metric logs. Metric log entries contain several parts as Figure 6.3 shows. **Descriptior** is a fully editable part of the log entry. It is defined when creating a new metric using a counter, gauge, or histogram. **Labels** are a fully customizable part, where you can specify any additional information that you need to log, e.g. the route where the request was made. You can add as many more as you need. **Aggregator** shows a kind of metric, that is in numerical value. Next, it shows the actual value that was measured or aggregated (depending on the type of metric type, see Figure 6.1) and the timestamp when was the request made. **AggregationTemporality** indicates the temporal aggregation behavior of a metric. **Resource** displays OpenTemetry

attributes. It specifies how the values of a metric should be aggregated over time. Last but not least **instrumentationLibrary** shows the meter name specified in the meter definition.


```
{
   "descriptor":{
      "name":"request_latency_summary",
      "description":"Latency of requests in milliseconds",
      "unit":"ms",
      "metricKind":1,
      "valueType":1
   },
   "labels":{
      "route":"/"
   },
   "aggregator":{
      "kind":0,
      "_current":2434.75192,
      "_lastUpdateTime":[
         1713452815,
         27031103
      ]
   },
   "aggregationTemporality":2,
   "resource":{
      "attributes":{
         "service.name":"unknown_service:/usr/local/nodejs/bin/node",
         "telemetry.sdk.language":"nodejs",
         "telemetry.sdk.name":"opentelemetry",
         "telemetry.sdk.version":"0.24.0"
      }
   },
   "instrumentationLibrary":{
      "name":"OpenTelemetry-metrics"
   }
}
```


Figure 6.3: Example of log generated by OpenTelemetry after formatting it to JSON.

OpenTelemetry is the tool I have chosen to implement the TypeScript profiler for this thesis.

6.1.2 Sentry

Sentry is partly open-source and partly proprietary software. Sentry offers a free developer plan but also paid ones. Paid features are in the proprietary part of the code. Sentry is both instrumentation software and observability backend. Sentry provides code-level observability that makes it easy to diagnose issues and learn continuously about your application code health. Sentry provides the following features to help you keep your app running smoothly. Sentry does *Error Monitoring* (also automatically creating an issue with the error), *Performance Monitoring* (collecting traces, metrics, monitoring your application performance in production), *Release Health Monitoring* (managing releases), *Recurring Job Monitoring* (managing crontasks), *Visibility Into Your Data Across Environments* (provides dashboard with graphs, transactions, and their traces), *Code Coverage*, and much more. [32]

The reason, why Sentry was not chosen for implementing this thesis is the problematic extracting of data before sending it to Sentry. The *beforeSend* method in Sentry allows you to modify or inspect the event data before it's sent to the Sentry server. This can be useful for adding custom data, filtering sensitive information, or performing any other necessary modifications to the event payload before it's processed by Sentry. However, the Sentry transaction containing the data in this method does not include the data we need. Also, in the free version of Sentry, it exports less data. Most of the data, e.g. the actual traces, are created directly in Sentry's backend so accessing them before exporting is impossible. For this thesis, it is crucial to get those data to create profiles and visualization later on.

6.1.3 Puppeteer and Playwright

Puppeteer, a Node.js library, provides an advanced interface for managing headless Chrome or Chromium via the DevTools Protocol. Additionally, it offers the flexibility to utilize full (non-headless) Chrome or Chromium. Puppeteer can generate screenshots and PDFs of pages, traverse a Single-Page Application, test Chrome extensions, capture traces, diagnose performance issues, and automate actions on web pages e.g. form submissions, or other actions. This can be used, e.g. to generate Selenium tests. Therefore, Puppeteer is mainly used for testing purposes and not profiling. [1]

Playwright was developed specifically to address the requirements of end-to-end testing. It offers comprehensive support for all modern rendering engines, including Chromium, WebKit, and Firefox. Whether on Windows, Linux, or macOS, Playwright can be utilized in both headless and headed modes. Additionally, it provides native mobile emulation capabilities for Google Chrome on Android and Mobile Safari. Playwright was inspired by Puppeteer, so its the main purpose is testing and not profiling. [30] The Figure 6.4 shows an example of capturing timestamps using Playwright.

```
Time info: {
  timeOrigin: 1706203651824,
  timing: {
    connectstart: 1706203651831,
    navigationstart: 1706203651824,
    secureConnectionStart: 0,
    fetchstart: 1706203651824,
    domContentLoadedEventStart: 1786203651968,
    responseStart: 1706203651834,
    domInteractive: 1706203651845,
    domainLookupEnd: 1706203651831,
    responseEnd: 1706203651835,
    redirectStart: 8,
    requestStart: 1706203651832,
    unloadEventEnd: 0,
    unloadEventStart: 0,
    domLoading: 1706203651838,
    domComplete: 1786203651968,
    domainLookupStart: 1786203651831,
    LoadEventStart: 1786203651968,
    domContentLoadedEventEnd: 1706203651968,
    loadEventEnd: 1706203651968,
    redirectEnd: 0,
    connectEnd: 1706203651831,
    navigation: {type: 0, redirectCount: 0 }
  }
}
```

Figure 6.4: An example of time measurement output of Playwright used for profiling web application.

Puppeteer and Playwright comparison

While Puppeteer is being developed by Google, Playwright is being developed by Microsoft. Playwright is a more recent tool and its creation was inspired by Puppeteer, therefore Puppeteer has more active users. While Playwright harnesses languages like Python, .NET, Java, and JavaScript/TypeScript, Puppeteer predominantly utilizes JavaScript. [27]

Since both Puppeteer and Playwright are mainly for testing purposes, and compared to OpenTelemetry, are not as helpful for this thesis's use case, I decided to not use either of them and use OpenTelemetry instead.

6.2 Architecture

In this section, we will introduce the design of our profiler, data storage, and how different modules communicate with each other while creating a profile.

Perun modules utilize actual profilers to get profiling data, such as metrics and traces. The TypeScript web profiler we are creating in this thesis is working on a similar principle. There are three key parts of this architecture for profiling TypeScript web applications: Perun, the profiler using OpenTelemetry, and the target project to profile.

It is first necessary to initialize *Perun* by creating *.perun* directory in the project directory using *perun init* in a virtual environment. Now we can run Perun commands. This thesis uses two of Perun's commands: *collect* for running the profiler and *view* for running visualizations. Figure 6.5 illustrates how Perun, communicates with the target project and the profiler.

When *perun collect* is called, Perun runs the profiler of choice. After profiling is one, the data are processed and parsed into a *profile*. Perun saves the data to the *.perun* folder in the project.

When *perun view* is called, the Perun module loads the created profile given by a parameter and processes it to make a number of different visualizations.

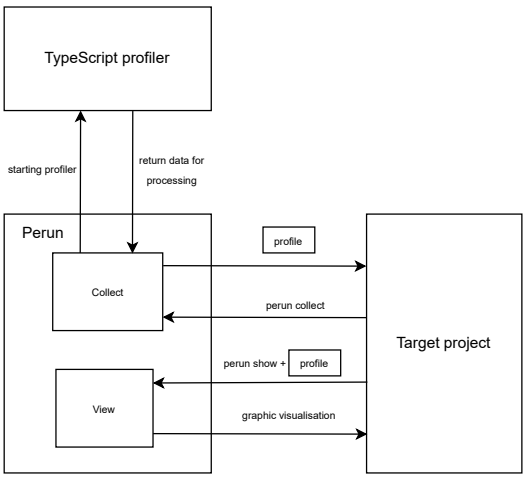

Figure 6.5: An architecture schema of TypeScript web application profiler, including communication between modules from the moment the user runs the command to create and return profile. The view module for creating visualizations is included as well.

Figure 6.6 illustrates our design of the TypeScript profiler module using OpenTelemetry packages and how classes should communicate with each other.

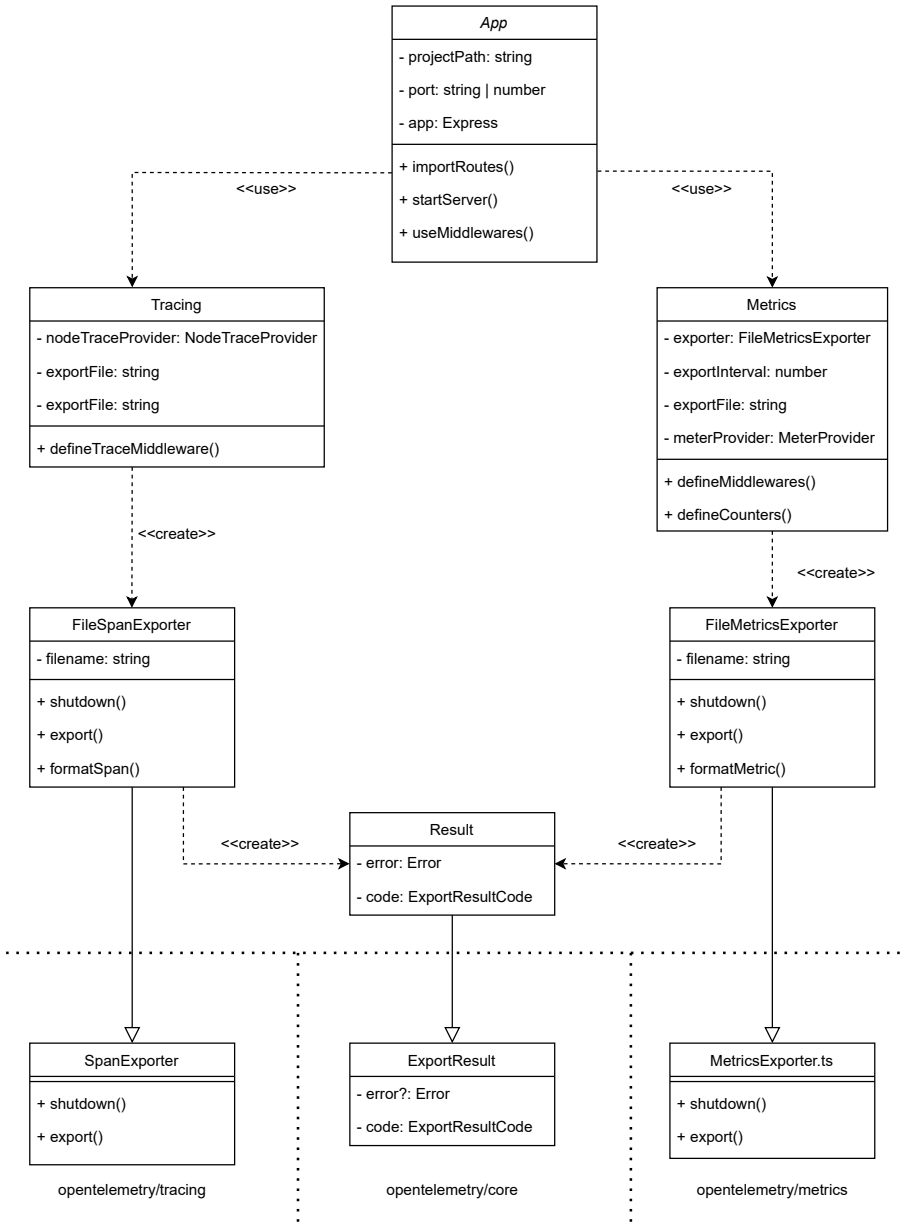

Figure 6.6: A class diagram showing the design and expected interactions between classes in the TypeScript profiler.

The *App* class is used for creating the Express application and including tracing and metrics middlewares. The *Tracing* class creates middlewares for trace spans. The *FileSpanExporter* class is used to parse and export trace spans to the log file. The *Metrics* class creates counters and middlewares for metrics. The *FileMetricsExporter* is used to parse and export metrics to the log file. The *Result* is a class that exporters create to pass the return code and, if an error occurs, the error itself.

Chapter 7

Implementation

In this chapter, we will summarize everything about the implementation, describe the methods used to make the TypeScript web application profiler, write about integrating the module into Perun, and the visualization itself. Also, we will explain how to use the module using the Perun interface.

7.1 Collecting profiling data

Figure 6.5 illustrates that for profiling TypeScript web applications, we have the option to utilize both the *OpenTelemetry profiler* and *Perun* modules to profile TypeScript web applications. The profiler can also be used independently as the requirement **F-IU** suggests. In this section, we will describe both of those modules in detail and how were the modules actually implemented compared to the design.

7.1.1 TypeScript Profiler

OpenTelemetry can be used to observe code in multiple languages, but in order to observe TypeScript applications, it is a good practice to use the language of the profiled application to set up OpenTelemetry as well. We started by initializing a new *Node.js* project with the start script **app.js**, initializing an *Express* application with one or more routes and listening on a port (we can choose any non-reserved port for initializing an Express application). **Express** is a framework that simplifies the process of building web applications and APIs with Node.js, and it is one of the most straightforward methods to develop TypeScript web applications [3]. Then we tried to reproduce the example from the documentation for both metrics and traces. The Express application in the TypeScript profiler uses port 9000, so it is reserved and cannot be used by the target project while profiling.

Metrics

It is important to choose which metrics to measure. In our case, it was the **Number of Requests** made for every route to satisfy **F-RR** requirement. First, we need to create the so-called *Meter* to use OpenTelemetry *Metric API*. The Meter is responsible for creating *instruments*. Instruments are used to report measured values. [23]

OpenTelemetry provides a *MeterProvider* class to obtain meters. To create an instance, we can provide an object to configure the provider. The configuration can contain the *exporter* we want to use and other parameters described in the documentation. There are

multiple types of exporters in OpenTelemetry as we mentioned in Subsection 6.1.1. For the first measurements, we used **ConsoleMetricExporter** to see data directly in the console. To obtain *Meter* from the MeterProvider, it is necessary to call a getter and give the Meter its name.

```
const meterProvider: MeterProvider = new MeterProvider({
    exporter: exporter,
    interval: INTERVAL,
});

const requestMeter: Meter = meterProvider.getMeter(METER_NAME);
```

The next step is to create an instrument or other kind of metric. The kinds of instruments were listed in Figure 6.1. Each instrument has the *name* of the instrument, *kind* of the instrument (counter or one of the other kinds). Optionally, it can also have a *unit* and a *description*. For the remainder of the chapter, we will refer to the instrument as *metric*. [22]

```
const requestCount = requestMeter.createCounter('page_requests', {
    description: 'Request count',
});
```

Last but not least, to be able to measure a metric, we need to create a middleware to obtain the metric and let OpenTelemetry process it. Our middlewares are all made as arrow functions that return another arrow function with parameters *Express request* and *Express response*. Instead of the function keyword, arrow functions use a fat arrow => to separate the parameters and the function body. This makes the code shorter and more readable. Metrics have in most cases their own middleware. We created a *Map* to save the route and the number of requests on the route. This metric was created as a *Counter* and it uses an *add* function to add requests to the Map. The middleware needs to be used by the Express app so it can start measuring.

```
export const countAllRequests = () => {
    const boundInstruments: Map<string, any> = new Map();

    return (
        req: express.Request, res: express.Response, next: NextFunction
    ): void => {
        if (!boundInstruments.has(req.path)) {
            const labels: { route: string } = { route: req.path };
            const boundCounter = requestCount.bind(labels);
            boundInstruments.set(req.path, boundCounter);
        }

        boundInstruments.get(req.path).add(1);
        next();
    };
};
```

At this point, we can measure the number of requests for each route, but there are two problems. The first one is that routes are defined in the profiler, which does not make sense, since we need to be able to profile arbitrary projects. For this, we need to have the other project prepared. The project needs to use Express routing as well and export the Express app. We support both default export as well as named export with the name *app*. The profiler imports this script and performs a dynamic import of the project. Then the Express app in the profiler uses the imported app to get its routes with their implementation. A path to the file with exported Express is supplied as a parameter **--path**. The path parameter is mandatory. The second problem is that metric log entries do not contain the route, where the request was made. The solution to this problem is to bind custom labels to the entries. A label is an object containing the route that is extracted from the Express request parameter.

To get metrics, requests need to be made. There is a bug in OpenTelemetry or Express we noticed, so even if there are requests in the background and they are logged successfully to the log file, they all have the same timestamp. Although, we are not sure if this is intended or not. It is the timestamp of the last visit or manual reload of the route. So if we run the profiler, only load a route once, and let it make background requests, then all the entries in the log will have the same timestamp. This is why in order to test a web application it is necessary to visit the routes or refresh the web repeatedly to get more data.

Traces

To set up a tracing we need *NodeTraceProvider* from OpenTelemetry. Additional configuration is again optional. It can contain parameters such as *spanLimits* (configuration for span limits), *serviceName* (name of the service), and *sampler* (a configuration for the sampling strategy). Then we need to register the provider, add *SpanProcessor*, and specify an exporter. I again specified a console exporter named *ConsoleSpanExporter*. It is also needed to create middleware with a similar structure as in the case of metrics. Using the trace provider, we can obtain a Tracer object by using a getter and specify its name. Then we need some spans to observe with our tracer. The spans can be created for observing requests, button clicks, etc. It is necessary to name them so we can distinguish them in logs. Also, end spans in the middleware. Now after using the middleware in the Express app, we have a functional tracing.

```
const provider: NodeTracerProvider = new NodeTracerProvider({
    spanLimits: {
        attributeCountLimit: 512,
        linkCountLimit: 512,
        eventCountLimit: 512,
    }
});

provider.register();
provider.addSpanProcessor(
    new SimpleSpanProcessor(
        new FileSpanExporter(FILENAME),
    ),
);
```

```
const tracer: Tracer = provider.getTracer('tracer');
const span: Span = tracer.startSpan('request');

// Continue with the middleware implementation

span.end();
next();
```

Restructuring Project and Exporting Data to Log Files

To make the code more readable and clear, we split the application into more files. The setup of metrics was moved into **metrics.js** and the set up for traces into **tracing.js**. Since we decided to make the profiler written in TypeScript at the beginning of the chapter, we initialized TypeScript by creating *tsconfig.json* and added build script to *package.json*. We filled in missing types to variables and return types for functions until the build was successful.

Exporting metrics and traces to the console is not ideal for the profiler because of future data post-processing. This is why we decided to export log entries to a file. The problem is that OpenTelemetry does not support this kind of exporter. On the other hand, we can use inheritance to create our own custom exporters.

We used *MetricExporter* to create *FileMetricsExporter* and *SpanExporter* to create *FileSpanExporter*. Both exporters have similar structures but also their own purposes. Exporting metrics and spans is a bit different and they do not have the same data types. Both of them have a constructor for path and filename and they also create the log directory if it is not present. The extended classes have methods we need to implement, such as *export*. This method logs all entries provided by OpenTelemetry if they are not empty strings. The entries are converted to JSON and finally, it creates *Result* with the status code. Exporters log to the console information about what entries were added to log files (the traces or metrics entries).

The Result class is also extended from the OpenTelemetry *ExportResult* class, containing an error message, if provided, and a status code. Now metrics are logged into *data/metrics/metrics.log* directory and traces are logged into *data/tracing/tracing.log* directory. There is a lot of data being logged even when there are no interactions with the web (requests in the background) so console logs can be disturbing if we are not debugging something. That is why we added the **--silent** parameter to silence all console logs from exporters.

Process Object and Adding More Metrics

To add a new metric we need to find a way how to obtain the values before passing it to OpenTelemetry. OpenTelemetry supports both automatic and custom instrumentation but they are not meant to collect metrics. A `process` object provides information about, and control over, the current Node.js process [19]. The `process` object can provide a lot of data about the application we can use to get various metrics. The `process` object contains these data whether we use OpenTelemetry or not, so using the process object does not increase runtime overhead. We can say that the profiler satisfies **F-RO** requirement.

We added support for more metrics by creating new meters. Every meter is used for a specific metric or a group of metrics, for example, error metrics.

We have three *Error Metrics* now: error rate, error codes, and error messages. They use the same meter but they have different counters, so they have their own entries in the log file. Errors are detected using the response code. The response code is supposed to be 400 or higher to be detected as an error. The */favicon.ico* route is skipped intentionally because it is not strictly necessary for a website to have the file, but it is considered a good practice.

Other important metrics we need to mention are *Memory* and *Latency*. If we want to log the actual values of memory consumption and latency for each request, we have to change how we use their counters in their middleware. Values of both of them can increase or decrease over time, depending on how dynamic the web is. So the right option is to use the *UpDownCounter*. As Figure 6.1 shows the UpDownCounter also makes sum aggregation. This caused a lot of trouble. If the latency of the first request is around 20ms, then the latency of the one-thousandth request on the same route should still be around 20ms, not the sum of them (around 20000ms). We tried storing the last value and subtracting it before passing it to the OpenTelemetry counter. However, as Node.js applications have multiple processes it kept overwriting the data incorrectly, resulting in negative values[1]. This was fixed by resetting the counter before logging the data and making the middleware asynchronous. Then the negative values were no longer reported, but to be entirely sure we added post-processing to limit the lower bound of the record to 0. This satisfies the requirements **F-MM** and **F-LM**.

The last group of metrics we want to mention are **File System Reads**, **File System Writes**, both **System** and **User CPU Time**, and both **System** and **User CPU Usage** to satisfy the requirements **F-FS**, **F-UST**, and **F-USU**. The *UpDownCounter* is used in those cases as well but in this case, we want that sum aggregation. So the only thing we must do in the middlewares for each of them is to extract the data from the *process object* and fill the counter.

Comparison of the Desing and the Implementation

Figure 7.1 shows the real structure of the profiler. There were some changes compared to the original design, since **app.ts**, **metrics.ts**, and **tracing.ts** are scripts, not classes, we decided not to show them in the class diagram.

[1]We also tried to define the latency and memory counters inside the middlewares but that resulted in obtaining a constant value. After changing the route, it stopped measuring latency and memory completely.

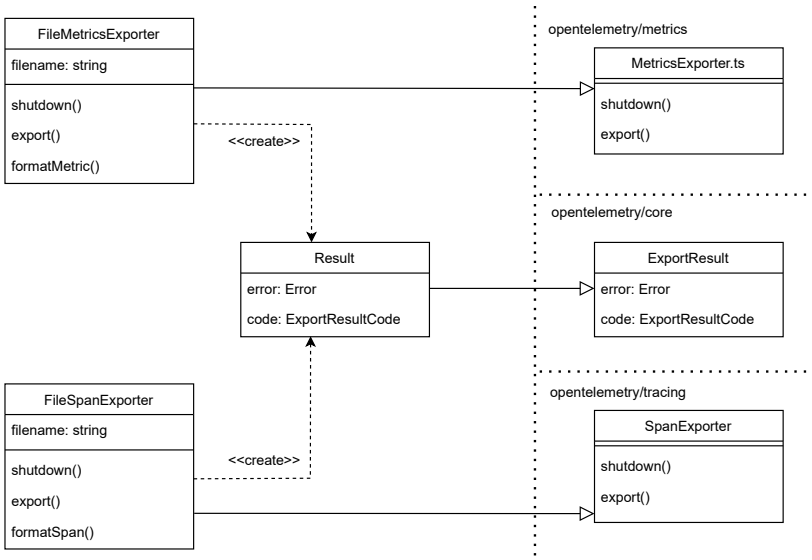

Figure 7.1: A class diagram of the implemented TypeScript profiler. The original design was introduced in Figure 6.6.

7.1.2 Integration to Perun and Usage

Perun contains several commands that serve as useful tools for various purposes. The command **collect** is used to profile projects in different programming languages and collect data (e.g. metrics and traces). Each collector has its purpose. We must add a new collector to integrate the TypeScipt profiler into Perun. After registering a new module in the collect directory and creating the necessary scripts, we created functions that all collectors need to contain. The *before* function specifies what should happen before the start of the profiling. The *collect* function starts the profiler. The *after* function post-processes the data and gives them to the profile factory of Perun to create a profile. And finally, the *teardown* function handles resource cleanup, which occurs after both successful and unsuccessful executions of the collect command.

In our collector, the *before* function was not needed. The *collect* function runs the profiler using the Perun's *non-blocking process* function and waits for the web application to start. It also uses a timeout to terminate the profiled program if the application is up. After the process is stopped, processes on the used ports are killed. Those ports are 9000 – the port of the TypeScript profiler, and the port, on which the profiled project is running. The *teardown* function also calls the function to kill all processes on ports that were used. Finally, the *after* function creates an instance of *Parser* class to parse the resulting log files. We decided with the thesis supervisor to not use the tracing log for now and to create a profile only from the metric log. We cannot merge both metrics and traces into a single profile simultaneously as it would result in inconsistency and we currently lack clarity on how to effectively leverage this log. The metric log is parsed line by line using a native JSON parser for Python. The parser returns tuples of data for each line and after parsing all data, the *after* function returns a profile. After metrics are processed it moves the metric log file into the *done/* folder and appends the current timestamp to the filename for debugging purposes. Now after another run of the collector, it creates a new metrics log and data are not mixed up with the previous runs of the collector.

Limitations

There are three known limitations of our TypeScript profiler we are going to describe in the list below.

- Exporting the Express application from the target project is necessary for the profiler to obtain the routes with their content to reproduce it.
- Projects that run through a *Docker container* have proven to be problematic for the profiler execution. The profiler runs the project's server and then tries to run it on its port (9000), but we had some problems trying to install Perun in the container the same way we did in the local environment.
- The profiler supports TypeScript Express routing, but it was not tested with other JavaScript and TypeScript routes like React.js or Vue.js routers. Because they have a completely different structure, we are confident it would not work without modifications of the profiler source code.

Usage and Summary

Figure 7.2 shows a screenshot of options for *web* collector with a caption.

```
Options:
  -o, --otp TEXT          Path to the OpenTelemetry profiler script.
                          [required]
  -p, --proj TEXT         Path to the project to be profiled. If it's not given
                          actual directory is used.
  -e, --express TEXT      Path to file in your project containing express() app
                          with export. The export must be default or named
                          'app' Examples: 'src/app', 'src/app.ts'  [required]
  --port INTEGER          Port on which project run.  [required]
  -t, --timeout INTEGER   Timeout for the runtime of profiling in seconds.
                          Default is seconds.
  --help                  Show this message and exit.
```

Figure 7.2: Options for web collector in Perun. The web collector uses the TypeScript profiler to profile web pages.

Examples of usage can be:

```
perun collect web --otp ~/OpenTelemetry/ --express src/express.ts
--port 8000 -t 600
```

This would run the collector using the OpenTelemetry TypeScript profiler for 10 minutes (as the timeout is set to 600 seconds). The profiler runs the target project on port 8000 because the project has an Express application configured to this port. At the same time, the profiler runs on port 9000 as an identical web page, because it uses the routes from the target project. To profile the project we need to use the profiler's Express app on localhost with port 9000 to generate metric and trace logs. When it reaches the timeout, the profiling ends and the metrics log is processed to a profile. The profile is saved to *.perun/jobs/* directory of the profiled project. We can consider requirements **F-IP** and **F-DP** as satisfied. We can consider the requirement **NF-SS** as satisfied because the profiler does not take much storage space and it stores only JSON entries in two log files.

7.2 Visualization

As mentioned in the last section, Perun contains multiple commands. The command to run visualizations is named **show**. As in the case of a collect module, it is also necessary to create and register the submodule. We named it *web* as in the previous case.

Data are parsed from a profile, given by the argument, into a list and passed to functions to create various graphic visualizations. There are three types of visualizations in this submodule. Each of them uses the **Pandas**[2] library to convert data from the profile into a Pandas `DataFrame` making it easier to transform the data into graphs.

1. **Line graph** – this graph is mostly used to visualize metrics without units, e.g. request rate for different routes. By default, it generates one graph for every route, but it also has a parameter to make a combined graph for all routes. There is a function for obtaining labels for graphs. If it is not defined it throws an exception. As a parameter, it takes the metric name and returns the graph label for the metric. The graphs are made using the **Matplotlib** library. Figure 7.3 illustrates a line graph example[3].

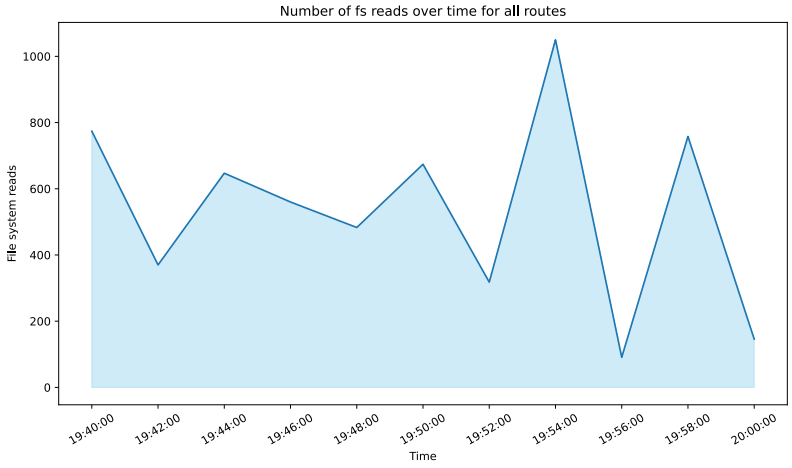

Figure 7.3: An example of a line graph visualizing file system reads for all routes. The plot is generated using Matplotlib.

2. **Pairplot** – This graph is used to visualize correlations between metrics. One of the parameters for this function is an array of metrics. We can use as few as two or three, and as many as tens of metrics to make this pairplot dynamic. Pairplot is a matrix of scatter plots for visualizing relations between different metrics. The main diagonal contains graphs where it should compare the metric to itself, but to not make something useless, pairplot instead makes histograms of the metric. We also split different routes by color so the data makes more sense. A bigger pairplot can still be confusing, so we decided to make an option to make more pairplots, each for a different route again. The pairplot is generated by the **Seaborn** library in Python. There is also a method for obtaining labels for different metrics. It will throw an

[2] https://pandas.pydata.org/getting_started.html

[3] Matplotlib documentation: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html

exception if the label is not defined. The Matplotlib is used also here to show the graph. This is the only graph, where the more data we have, the longer it takes to show and save the graph. If we look carefully we can see that some metrics have linear relationships and some metrics for some routes hint at quadratic relationships. Figure 7.4 shows an example of the pairplot generated for 9 metrics[4].

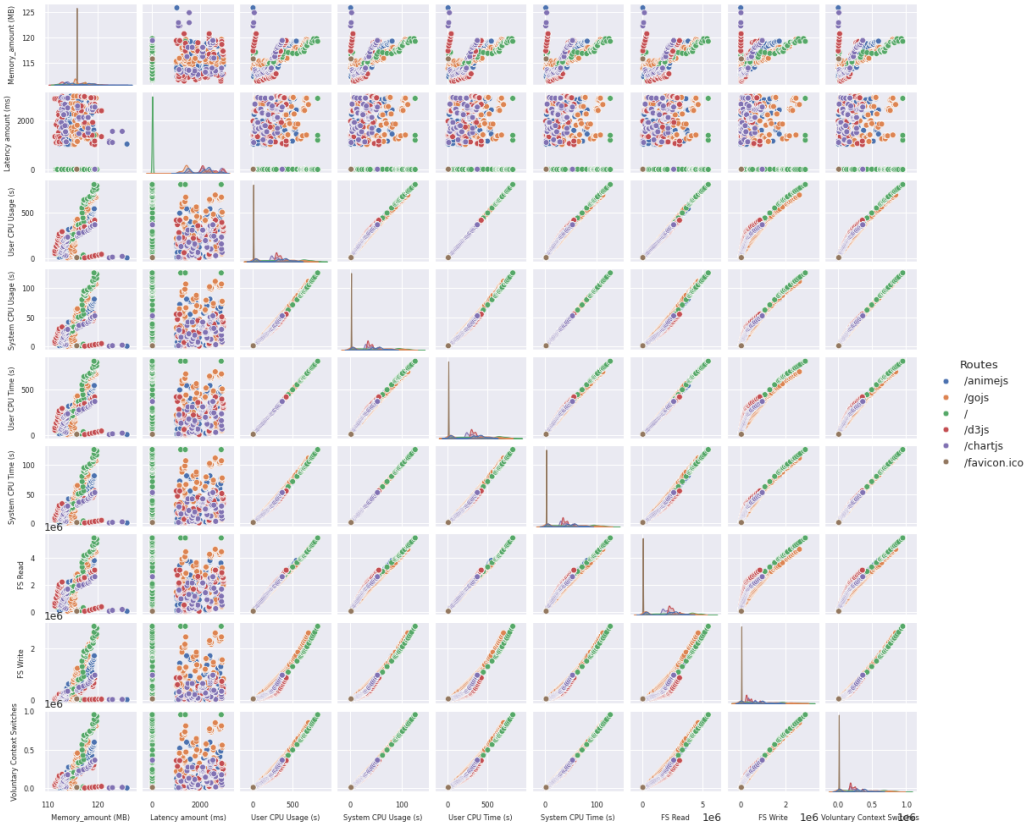

Figure 7.4: Pairplots for all generated by Seaborn to visualize correlations.

3. **Heatmap** – There are two types of Heatmap that we made. The first shows metric values in time and the second uses the X-axis for time, the Y-axis for routes, and the tiles in the graph show the amount of memory used over time on the route. What they have in common is that both use **Holoview** library for Python and **Bokeh** extension for interactivity with the graph. Missing data are filled with zeros to remove most of the white spaces in the Heatmaps, but there are some exceptions where the *Holoview* still left some spaces white without any values regardless of the color pallet. Heatmaps are generated as HTML files because Holoview does not support SVG graphs. Figure 7.5 shows an example of the heatmaps generated for all routes together and divided by route[5].

[4]Seaborn documentation: https://seaborn.pydata.org/generated/seaborn.pairplot.html
[5]Holoview documentation: https://holoviews.org/reference/elements/plotly/HeatMap.html

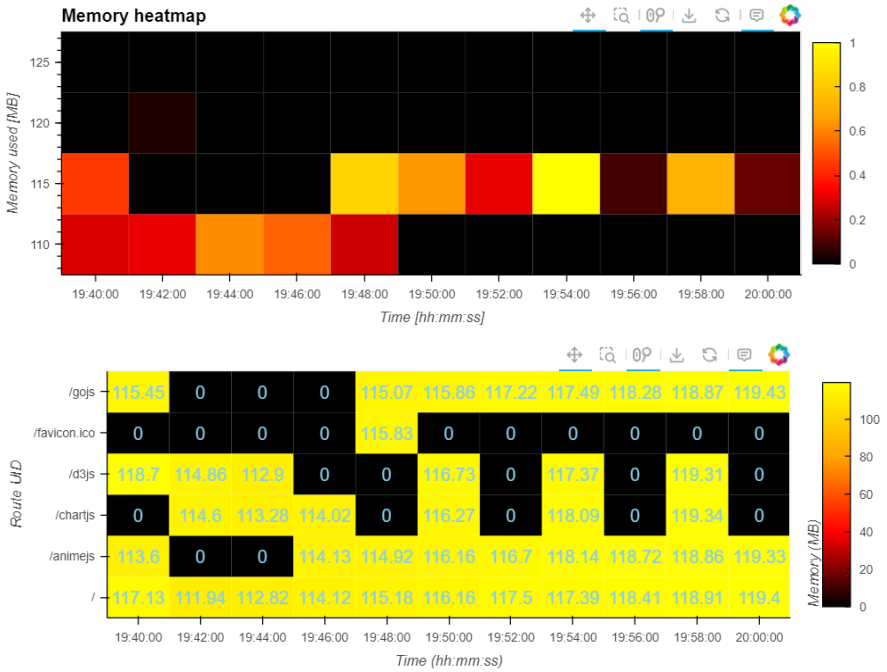

Figure 7.5: Heatmaps generated by Holoview to visualize memory used while running a web application and to visualize memory used on different routes.

After all graphs are generated, the *show* command runs a callgraph if the *show* option is given. The call graph library requires to be installed globally before running the visualizations using:

```
npm install -g typescript-call-graph
```

It statically parses the typescript files in the target project and creates 3 types of callgraphs. If we include a JavaScript or a TypeScript library it will not be included in the call graph. Figures 7.6 and 7.7 show examples of the callgraph types.

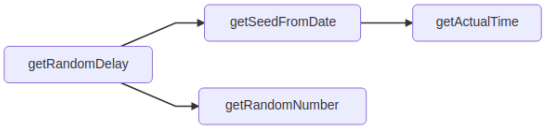

Figure 7.6: Type of TypeScript callgraphs generated by https://github.com/whyboris/TypeScript-Call-Graph

Figure 7.7: Type of TypeScript callgraphs generated by https://github.com/whyboris/TypeScript-Call-Graph

7.2.1 Usage and summary

Figure 7.8 shows a screenshot of options for *web* visualizations with a caption.

```
Options:
  -g, --group-by TEXT  Group by values in graphs by period. The default value is 1 minute. For example group
                       values by:
                       5s      - 5 seconds
                       1min    - 1 minute
                       1h      - 1
                       hour 1D - 1 day
  -s, --show           Show generated graphs and call graph. Graphs will be
                       saved in any case to the current directory.
  --help               Show this message and exit.
```

Figure 7.8: Options for web visualization command in Perun. The *web* option supports only profiles generated by *web* collector.

Examples of usage can be:

```
perun show .perun/jobs/web-\[_\]-\[_\]-2024-04-18-18-01-21.perf web --show
    -g 2min
```

This uses the selected profile for the visualization. The *show* option shows all the graphs made. Since the heatmaps are saved as HTML files, it shows them in the default web browser. The *group by* option specifies how data are grouped by time axes in both line graphs and heatmaps. The default is 1 minute. The correctness of the chosen interval is intentionally left to the user. The recommended interval is 1/10 of the total time of measuring. This command generates *Matplotlib* line graphs as SVG files, *Seaborn* pairplot as SVG files, and *Holoview* heatmaps as HTML files.

There are not many dependencies in the profiler and we added only 1 dependency to Perun, so we can say that **NF-MD** requirement is satisfied. The requirements **NF-EU** and **NF-RC** partly rely on a subjective evaluation, but, from our perspective, have also been satisfied.

Chapter 8

Experiments

This chapter will describe the environment used for the experiments and also demonstrate this thesis's results with a series of experiments to evaluate the TypeScript profiler in a non-trivial use case for analyzing real TypeScript web applications. Experiments compare different scenarios that can occur when using the target project. This is why we made changes to the testing project for every experiment.

8.1 Configuration of the Experimental Setup

This section briefly describes the environment used for both developing the profiler and the experiments that were made.

All experiments were made on a single laptop with the following parameters listed in Table 8.1.

Model	Lenovo IdeaPad L340-15IRH
Architecture	x86_64
CPU	Intel i5-9300H (4) @ 2.400GHz
Kernel	6.2.0-39-generic
OS	Ubuntu 23.04
RAM	5366MiB

Table 8.1: Parameters of a laptop on which the experiments were made. The memory allocation is unconventional because the system is running within a VirtualBox environment.

For experiments, we made a simple TypeScript project containing 5 routes. The root route includes a button to redirect to the other routes. Each of these 4 routes contains a different JavaScript library demonstrating its usage. The following sections will provide a more detailed description of the experiments. Since the pairplot images are too huge, we decided to focus on heatmaps for both the memory consumption used and the latency of the requests.

8.1.1 The JavaScript Libraries

This subsection will shortly introduce the project for experiments and the JavaScript libraries that were used. Footnotes will include URLs to the library's examples.

As we mentioned before, each route uses a different JavaScript library. Every route demonstrates an example usage of the library. The browsers are caching every page to make the loading smoother. Therefore, running the same test multiple times is unnecessary.

- **D3.js** is used to create various graphs and plots. In this case, we created a histogram from randomly generated values[1].
- **Chart.js** is used for creating interactive and customizable charts and graphs on web pages, providing a straightforward way to visually represent data through various chart types such as line charts, bar charts, pie charts, and more[2].
- **Anime.js** is an animation library that enables developers to create smooth and expressive animations for web projects[3].
- **Go.js** is a library for building interactive diagrams and graphs on the web[4].

8.2 Experiment 1: Profiling the Unmodified Project

This section describes an experiment on the introduced project without any modification of the original code. That means every route is defined as presented in Section 8.1. Therefore, this experiment assesses the actual functionality of the module. Every route, except the root route, has its own HTML file containing a TypeScript script of a demo showcasing the respective library. The HTML is the only thing sent to the response of the Express route. a set of selected images generated during this experiment will be included in Appendix A.

Figure 8.1 shows the memory consumption of the scenario described above in both combined heatmap and heatmap split by routes. Figure 8.2 shows how the request latency behaves in the same scenario.

[1] D3.js examples: https://observablehq.com/@d3/gallery?utm_source=d3js-org&utm_medium=hero&utm_campaign=try-observable.
[2] Chart.js examples: https://www.chartjs.org/samples/2.6.0.
[3] Anime.js examples: https://codepen.io/collection/XLebem.
[4] Go.js examples: https://gojs.net/latest/samples.

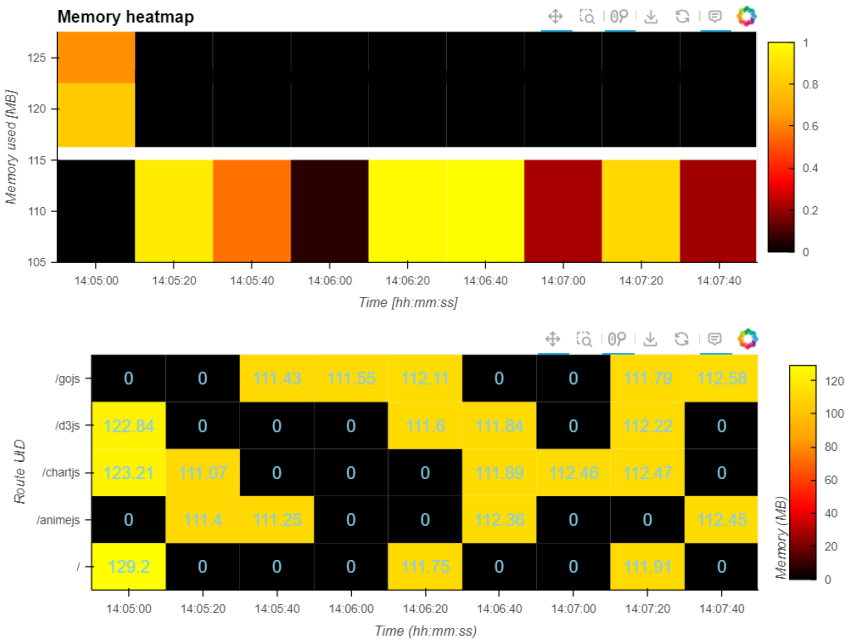

Figure 8.1: Memory consumption heatmaps generated in experiment 1. The first heatmap represents combined consumption for all routes together, and the second represents the consumption split by routes.

Figure 8.2: Request latency heatmap split by routes generated in experiment 1.

Both the latency and the memory usage demonstrate low measured values, indicating efficient performance and minimal resource consumption. This suggests that the libraries operate efficiently, with minimal delays in processing requests, and don't have a heavy demand on the system memory.

8.3 Experiment 2: Injecting Delay

The goal of the second experiment is to experiment with increased latency by intentionally slowing down the loading. Before sending an HTML file to Express, two nested *for loops* were implemented for each route (except the root route), each with 10000 iterations. Each nested loop iteration calculates the sum of the results obtained by applying the Pythagorean Theorem to each combination of indices of the for loops (i and y). A set of selected images generated during this experiment will be included in Appendix B.

Figures 8.3 and 8.4 show how the measured values of memory consumption and page latency have changed after this modification of the code.

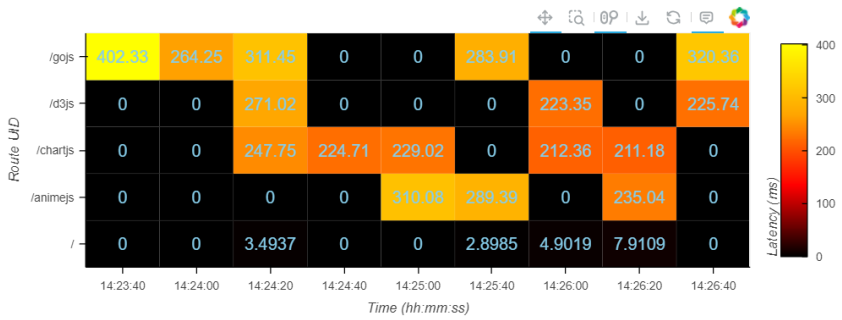

Figure 8.3: Request latency heatmap split by routes generated in experiment 2.

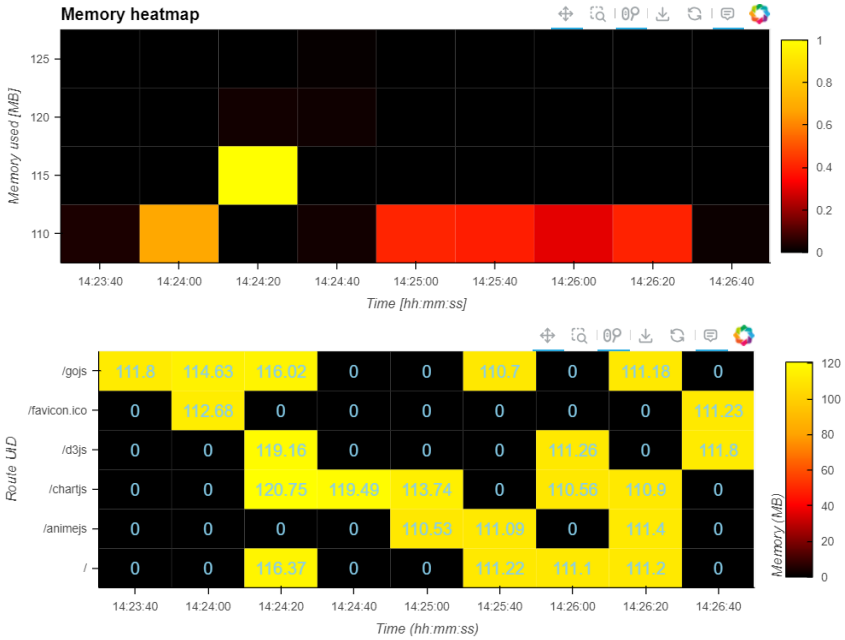

Figure 8.4: Memory consumption heatmaps generated in experiment 2. The first heatmap represents combined consumption for all routes together, and the second heatmap is split by routes.

The figures show that while the heap memory consumption stays almost the same, or even lower, page latency on the other hand increases almost 100 times after doing a lot of non-trivial computations compared to the first experiment. The non-trivial computation proved to not be that demanding in terms of memory consumption of the web page, but it can slow down the loading of the web page.

8.4 Experiment 3: Allocating Large Arrays

In the last experiment, we edited the code of the target project such that it tries to allocate large arrays of data to use as much heap memory as possible to evaluate how well the

profiler and visualization behave in this situation. The code from experiment 1 was edited so that before sending HTML content to the Express response, every route (except the root route) does memory-demanding operations. Before the routes are defined, a variable for a sum is defined. At first, every route creates an array with 1 million items filled with random values. To prevent the *Garbage Collector* (GC) from freeing all memory before using it, we store 2 of the elements from the array into variables and add them to the sum variable. At last, we used a console log to log the variable sum. This prevented the GC from interfering before the loading of the web routes. A set of selected images generated during this experiment will be included in Appendix C.

Figures 8.5 and 8.6 show how measured values of page latency and consumption heap memory have changed.

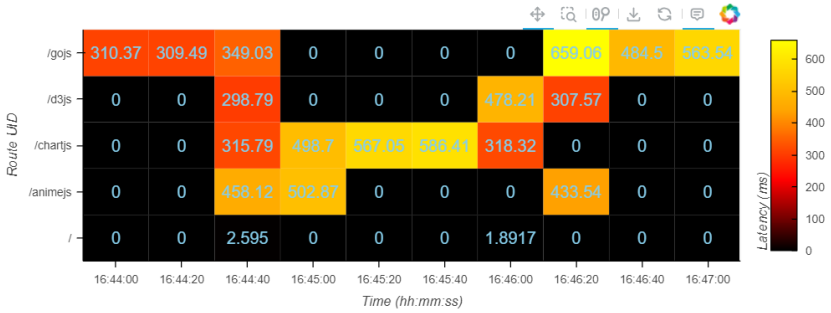

Figure 8.5: Request latency heatmap split by routes generated in experiment 3.

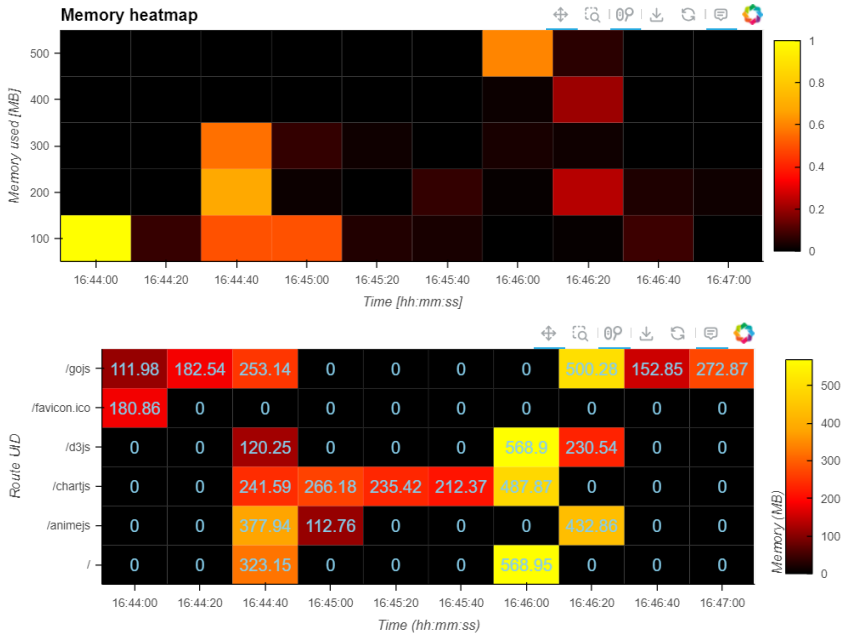

Figure 8.6: Memory heatmaps generated in experiment 2. The first heatmap represents combined consumption for all routes together, and the second heatmap is split by routes.

The figures show that both metrics have increased after allocating a lot of memory on the heap and additional computation steps. The page latency has increased 100 times compared

to the first experiment. On the other hand, the computation was not that demanding in terms of memory consumption of the web page, and increased only 5 times compared to the first experiment.

Chapter 9

Conclusion

The main goal of this thesis was to implement a new module for the Perun project. The module should be able to profile web applications written in JavaScript and TypeScript, and post-process the collected data to make reasonable and suitable visualizations. Reaching this goal was successful and the profiler works for Express-based web applications. Using Perun, we are able to post-process the data to create *Line graphs*, *Pairplots* of any size, and different *Heatmaps*. The solution was evaluated on both trivial and non-trivial examples and demonstrated that measuring the memory consumption and latency works as expected. The results correctly manifested in the visualizations with increasing allocations and delays caused by artificially injected computations. As suggested in Subsection 7.1.2, the solution is successfully integrated into the Perun, which was not part of the original assignment.

Currently, the solution supports web applications made using the Express library. However, there are many other ways how to make web applications in TypeScript and many other routers from different JavaScript and TypeScript frameworks. One possible expansion of this thesis is to extend support for these other routing options, such as Vue router or React router. Since every router supporting Typescript has a completely different structure (for example a Vue router needs at least 2 parameters for each route: an endpoint and a Vue component), it can become quite challenging and time-consuming to make this profiler support multiple routing methods.

At this point, the profiler can collect traces to some extent but does not post-process them for now. As Perun collectors support creating one kind of profile for one collector, we decided to not mix traces with metrics into one profile. This opportunity presents an avenue for further expansion of the bachelor's thesis, such as enhancing the collection of traces for specific elements identified by their *id* or *class name*. Processing and visualizing traces can be another opportunity to expand this thesis and improve the TypeScript web profiler and the Perun module.

Bibliography

[1] CHROME.COM. *Overview of Puppeteer* [online], 11. january 2018 [cit. 2024-04-20]. Available at: https://developer.chrome.com/docs/puppeteer.

[2] DENO.COM. *Deno Runtime Quick Start* [online]. [cit. 2024-04-11]. Available at: https://docs.deno.com/runtime/manual.

[3] DEV.TO. *TypeScript Express: Building Robust APIs with Node.js* [online], 07. june 2023 [cit. 2024-05-04]. Available at: https://dev.to/wizdomtek/typescript-express-building-robust-apis-with-nodejs-1fln.

[4] EMBREY, B. and BRADLEY, A. *TS-Node: Overview* [online]. June 2022 [cit. 2023-10-30]. Available at: https://www.npmjs.com/package/ts-node#overview.

[5] ERNST, M. D. Static and dynamic analysis: Synergy and duality. In: *WODA 2003: Workshop on Dynamic Analysis*. May 2003, p. 24–27.

[6] 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.

[7] GEEKSFORGEEKS.ORG. *Basic Blocks in Compiler Design* [online]. October 2019 [cit. 2024-02-08]. Available at: https://www.geeksforgeeks.org/basic-blocks-in-compiler-design.

[8] GEEKSFORGEEKS.ORG. *Difference between Static and Dynamic Testing* [online]. May 2019 [cit. 2024-02-02]. Available at: https://www.geeksforgeeks.org/difference-between-static-and-dynamic-testing.

[9] GREGG, B. BPF Performance Tools, Linux system and Application Observability. In:. Addison-Wesley Professional, 2019. ISBN 9780136554820.

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

[11] HOORN, A. van, ROHR, M., HASSELBRING, W., WALLER, J., EHLERS, J. et al. In: *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework* [online]. November 2009.

[12] HÁJEK, V. *Analýza výkonu programů v jazyce C#*. 2023. Available at: https://www.vut.cz/studenti/zav-prace/detail/148641.

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

[14] Intel. *Intel VTune* [online], 01. july 2022 [cit. 2023-12-04]. Available at: https://hpc-wiki.info/hpc/Intel_VTune.

[15] Intel. *Profiling JavaScript* Code in Node.js* [online], 11. january 2024 [cit. 2024-04-11]. Available at: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2024-1/profiling-javascript-code-in-node-js.html.

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

[17] mozilla.org. *Gecko* [online], 28. july 2023 [cit. 2024-04-11]. Available at: https://firefox-source-docs.mozilla.org/overview/gecko.html.

[18] Nethercote, N. *Dynamic Binary Analysis and Instrumentation or Building Tools is Easy*. 2004. Dissertation. University of Cambridge. Available at: https://nnethercote.github.io/pubs/phd2004.pdf.

[19] nodejs.org. *Node.js v22.0.0 documentation, Process* [online], 24. april 2024 [cit. 2024-04-25]. Available at: https://nodejs.org/api/process.html#process.

[20] Ondřej, M. *Analysis of Software Resource Consumption*. 2023 [cit. 2024-02-10]. Available at: https://www.vut.cz/studenti/zav-prace/detail/146746.

[21] opentelemetry.io. *Metrics SDK, Default Aggregation* [online], 04. june 2022 [cit. 2024-04-20]. Available at: https://opentelemetry.io/docs/concepts/signals/traces.

[22] opentelemetry.io. *Metrics API, Instrument* [online], 21. may 2023 [cit. 2024-04-25]. Available at: https://opentelemetry.io/docs/specs/otel/metrics/api/#instrumen.

[23] opentelemetry.io. *Metrics API, Meter* [online], 21. may 2023 [cit. 2024-04-25]. Available at: https://opentelemetry.io/docs/specs/otel/metrics/api/#meter.

[24] opentelemetry.io. *Metrics SDK, Default Aggregation* [online], 01. june 2023 [cit. 2024-04-19]. Available at: https://opentelemetry.io/docs/specs/otel/metrics/sdk.

[25] opentelemetry.io. *What is OpenTelemetry?* [online], 08. april 2023 [cit. 2024-04-19]. Available at: https://opentelemetry.io/docs/what-is-opentelemetry.

[26] OProfile. *About OProfile, Overview* [online], 15. july 2019 [cit. 2024-04-11]. Available at: https://oprofile.sourceforge.io/about/.

[27] oxylabs.io. *Playwright vs Puppeteer: The Differences* [online], 31. january 2021 [cit. 2024-04-20]. Available at: https://oxylabs.io/blog/playwright-vs-puppeteer.

[28] 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.

[29] 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.

[30] playwright.dev. *Getting started, Introduction* [online], 11. august 2022 [cit. 2024-04-20]. Available at: https://playwright.dev/docs/getting-started-vscode.

[31] RESEARCHCOMPUTING.PRINCETON.EDU. *Profiling with Intel VTune Profiler* [online], 04. december 2021 [cit. 2024-04-11]. Available at: https://researchcomputing.princeton.edu/faq/profiling-with-intel-vtun.

[32] SENTRY.IO. *What is Sentry?* [online], 08. september 2020 [cit. 2024-04-20]. Available at: https://docs.sentry.io/product.

[33] 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.

[34] STACKIFY.COM. *What is Code Profiling? Learn the 3 Types of Code Profilers* [online]. August 2016 [cit. 2024-01-31]. Available at: https://stackify.com/what-is-code-profiling.

[35] TECHOPEDIA.COM. *CPU Time* [online]. June 2014 [cit. 2024-02-08]. Available at: https://www.techopedia.com/definition/2858/cpu-time.

[36] TECHTARGET.COM. *Basic Blocks in Compiler Design* [online]. May 2022 [cit. 2024-02-08]. Available at: https://www.techtarget.com/whatis/definition/wall-time-real-world-time-or-wall-clock-time.

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

[38] VOJNAR, T. *Static Analysis and Verification* [online]. 2023 [cit. 2024-02-02]. Available at: https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-01.pdf.

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

[40] WEBKIT.ORG. *Introducing JSC's New Sampling Profiler* [online], 15. june 2016 [cit. 2023-12-04]. Available at: https://webkit.org/blog/6539/introducing-jscs-new-sampling-profiler.

Appendices

Appendix A

Experiment 1 Figures

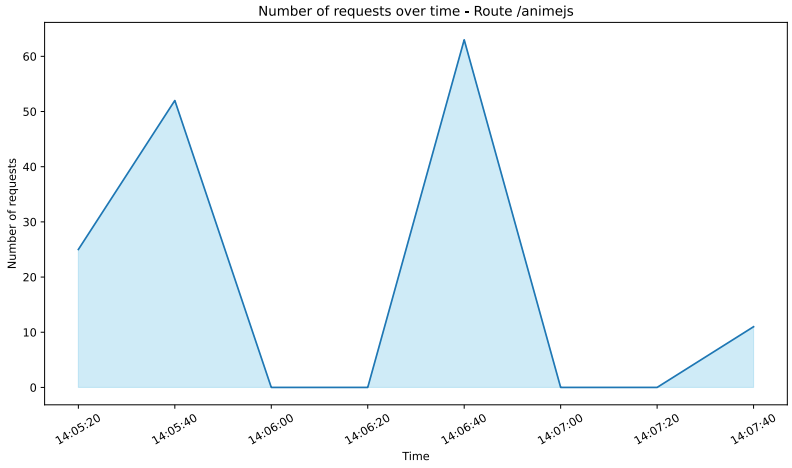

Figure A.1: A request rate graph for route */animejs* for experiment 1.

Figure A.2: A pairplot for metrics on route */animejs* for experiment 1.

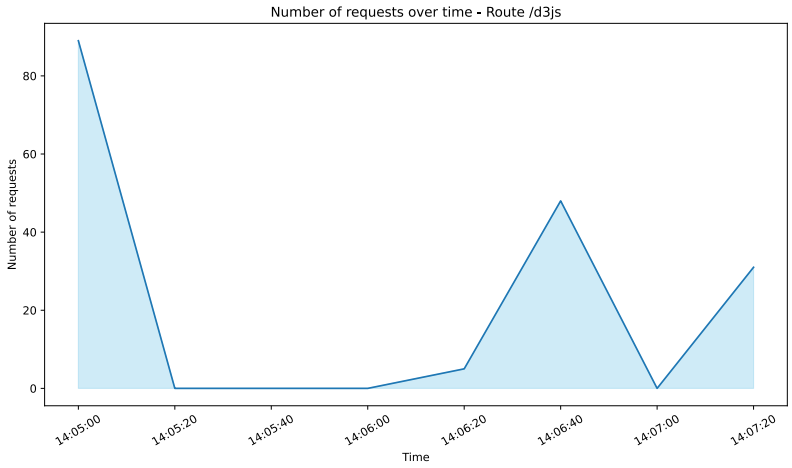

Figure A.3: A request rate graph for route */d3js* for experiment 1.

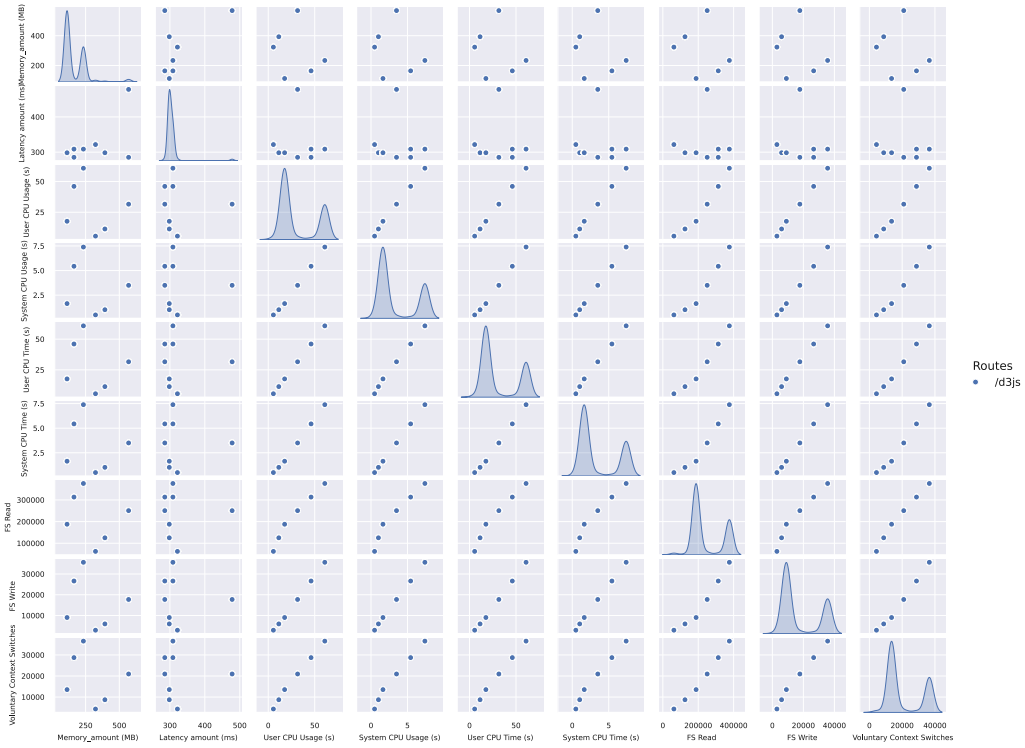

Figure A.4: A pairplot for metrics on route */d3js* for experiment 1.

Appendix B

Experiment 2 Figures

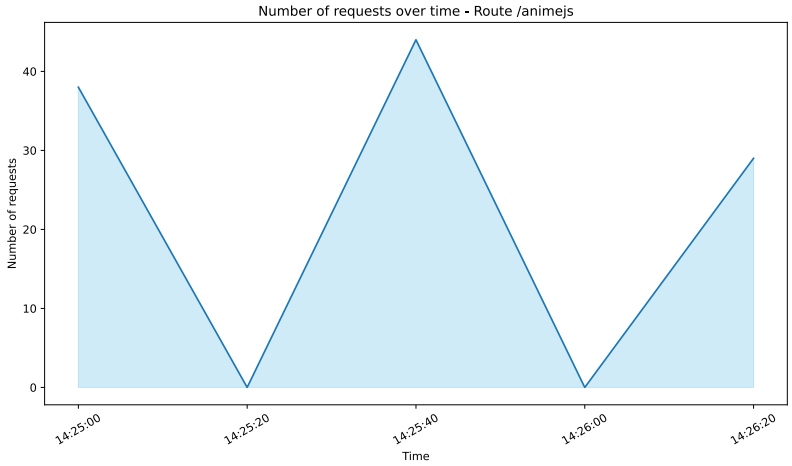

Figure B.1: A request rate graph for route */animejs* for experiment 2.

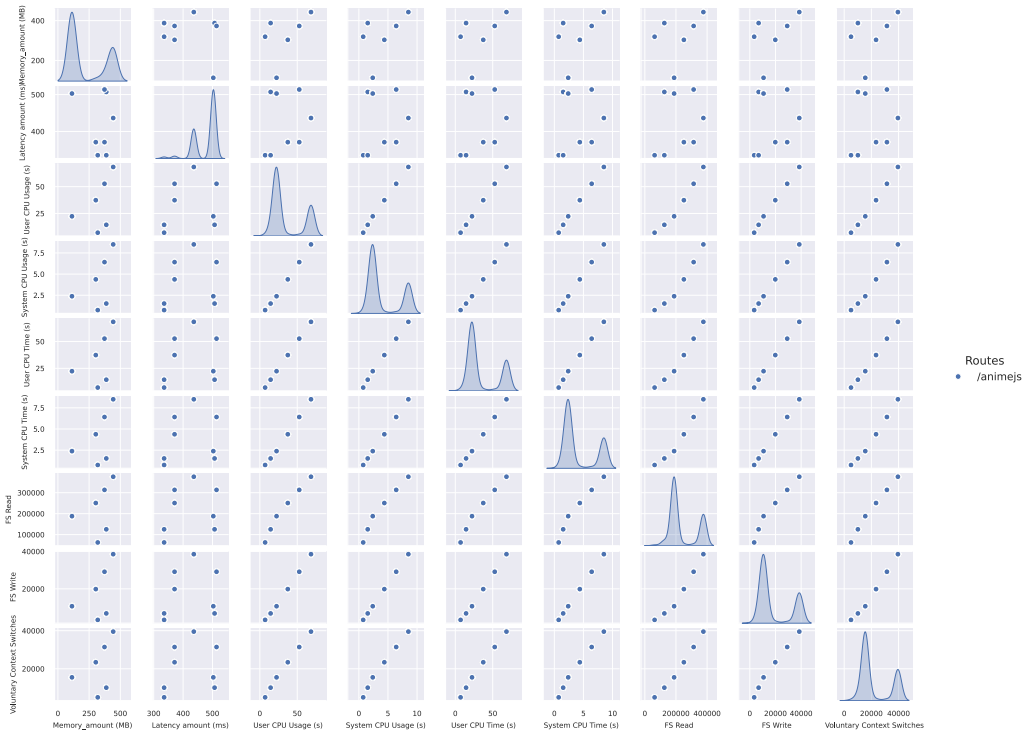

Figure B.2: A pairplot for metrics on route */animejs* for experiment 2.

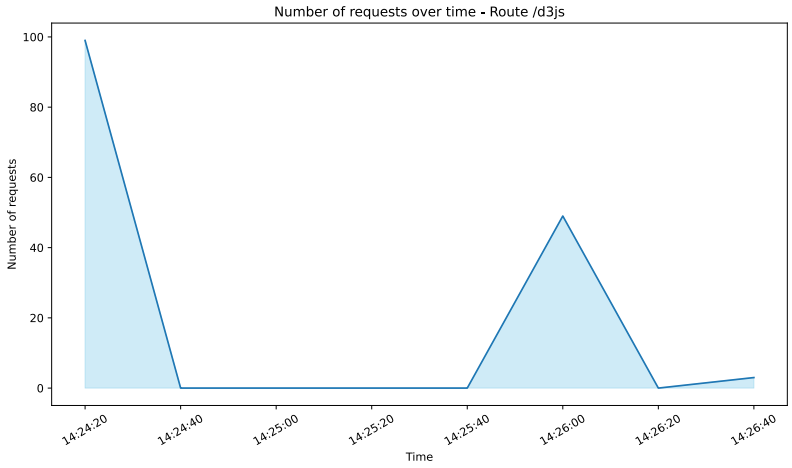

Figure B.3: A request rate graph for route */d3js* for experiment 2.

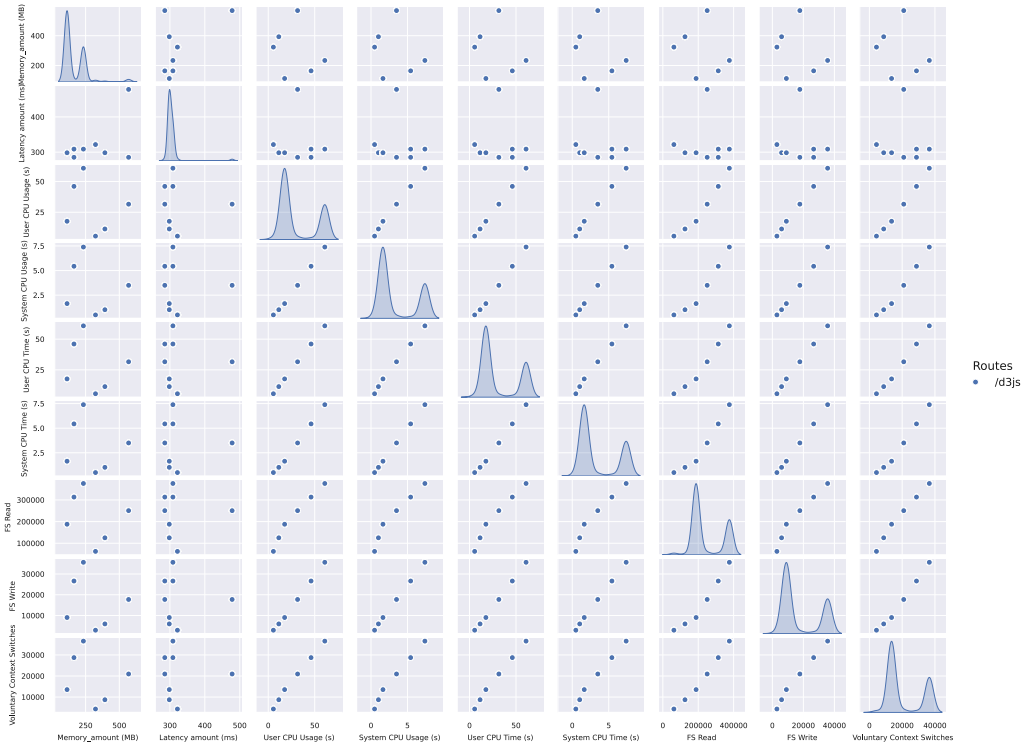

Figure B.4: A pairplot for metrics on route */d3js* for experiment 2.

Appendix C

Experiment 3 Figures

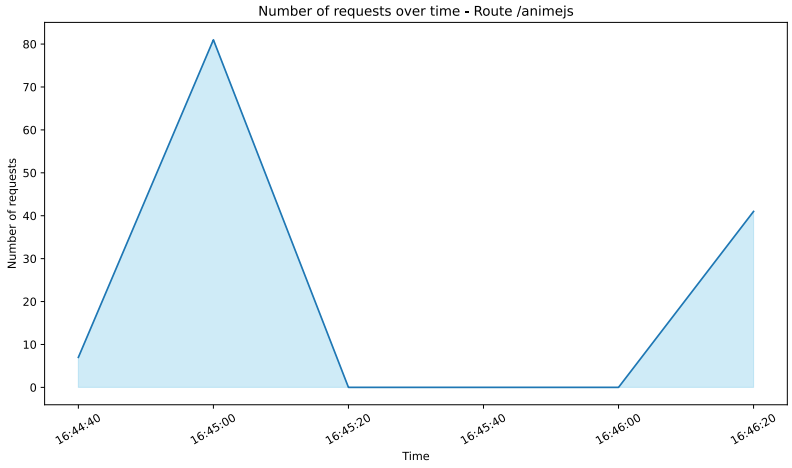

Figure C.1: A request rate graph for route */animejs* for experiment 3.

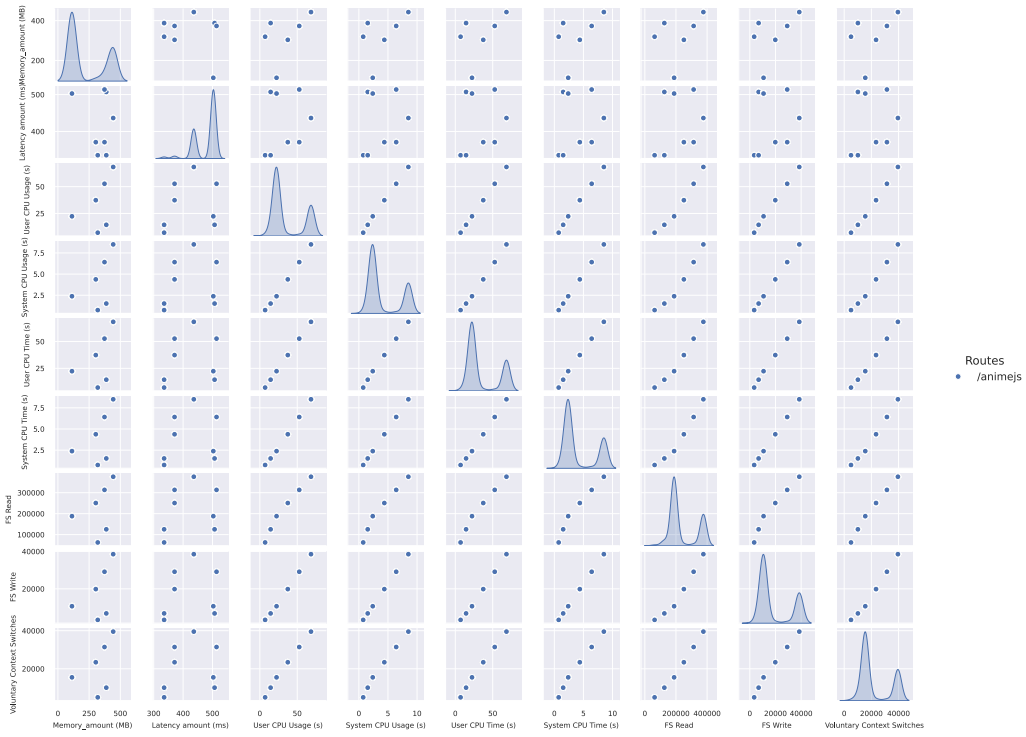

Figure C.2: A pairplot for metrics on route */animejs* for experiment 3.

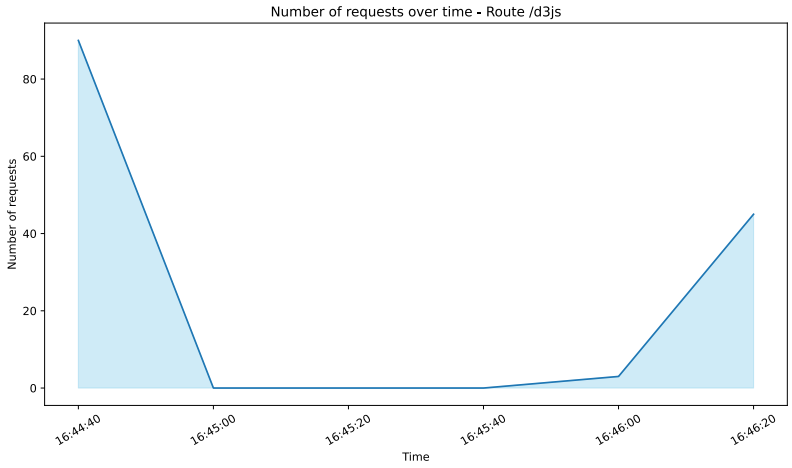

Figure C.3: A request rate graph for route */d3js* for experiment 3.

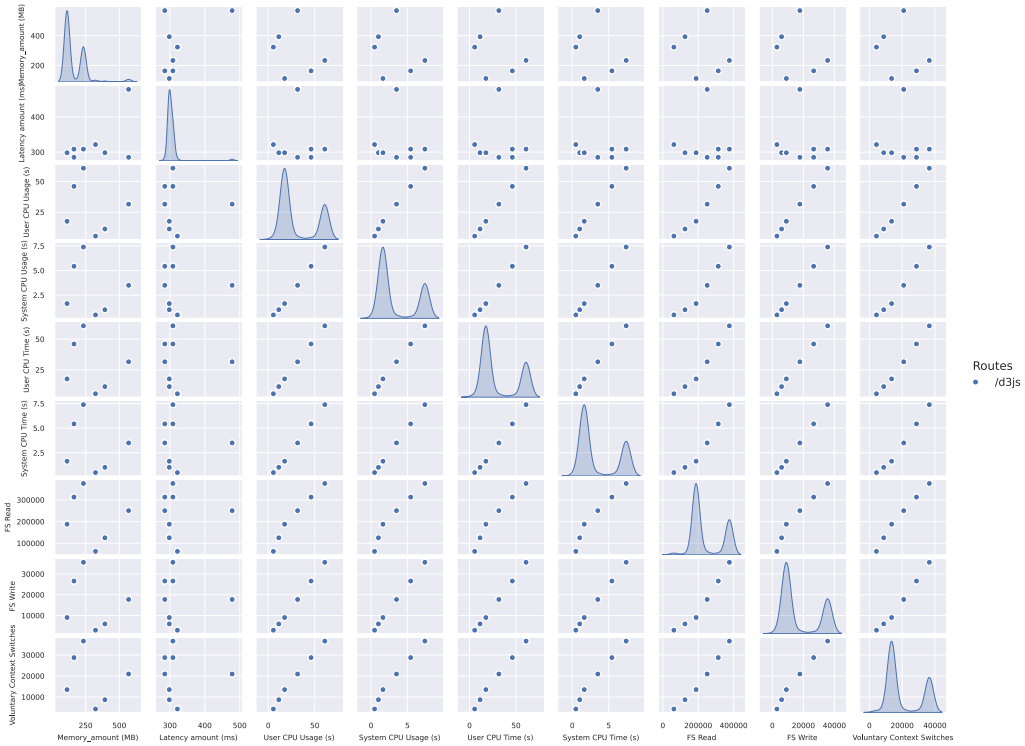

Figure C.4: A pairplot for metrics on route */d3js* for experiment 3.