CLC_____ Number_____

UDC_____ Available for reference Yes ☐No ☐

Undergraduate Thesis

Thesis Title: Python Performance Profiling and Visualization Design

Student Name: Ao Jiao

Student ID: 12013016

Department: Computer Science and Engineering

Program: Computer Science and Technology

Thesis Advisor: Yuxin Ma, Associate Professor

Date: **06 / 07 / 2024**

Commitment of Honesty

1. I solemnly promise that the paper presented comes from my independent research work under my supervisor's supervision. All statistics and images are real and reliable.

2. Except for the annotated reference, the paper contents no other published work or achievement by person or group. All people making important contributions to the study of the paper have been indicated clearly in the paper.

3. I promise that I did not plagiarize other people's research achievement or forge related data in the process of designing topic and research content.

4. If there is violation of any intellectual property right, I will take legal responsibility myself.

Signature: 焦傲

Date: 06 / 07 / 2024

Python Performance Profiling and Visualization Design

[Abstract]: With the development of software engineering and the rapid expansion of the scale of software, developers need to understand the runtime details of the program more urgently. A large amount of information is generated during the running of software, including function call information, running time information and other statistical information. Visual analysis helps people understand complex information and data faster, and helps people quickly grasp the key information. Comparing with raw numbers or tables, charts and graphs can more intuitively reflect the internal relationships and trends between data.

This project implements a statistical performance analysis profiler based on cProfile and a deterministic performance analysis profiler based on *sys.setprofile*, and the above profilers are encapsulated by a Python package. After obtaining the program performance data, a browser tool chain based on React is used to visualize the obtained data. Comparing with the existing visualization tools, this tool has richer functions and more interactive interface, and it combines both statistical performance analysis and deterministic analysis to make results more reliable.

[Keywords]: Performance analysis; Visualization; Software engineering; Visual analysis; Call stack visualization

Python 性能分析与可视化设计

[摘要]： 随着软件工程领域的发展软件的规模急剧扩大， 开发者更加迫切地需要深入了解程序运行时的运行细节. 软件运行过程中会产生大量信息, 包含函数调用信息, 运行时间信息以及其它统计性质的信息. 可视化分析可以帮助人们更快地理解复杂信息和数据, 迅速抓住关键信息. 相比于原始的数字或表格, 图表和图形能更直观地体现数据之间的内在联系与趋势.

本项目实现了基于 cProfile 的统计性能分析分析器和基于 *sys.setprofile* 的确定性性能分析分析器, 将以上分析器使用 Python package 进行封装. 在获取程序性能数据后使用浏览器工具链对得到的数据进行可视化, 与现有的可视化工具相比, 本工具拥有更加丰富的功能, 交互性更强的界面, 并且结合了两种性能分析方式使得结果可信度更高.

[关键词]：性能分析；可视化；软件工程；可视分析；调用栈

Table of Content

1. Introduction

1.1 Research Background and Significance

With the rapid development of computer science and software engineering, software engineering projects become more and more complex. This complexity manifests itself in many ways: the need for scalability, the complexity of dependencies, collaboration across teams and geographies, and so on. This complexity also brings many challenges, such as the performance of the software becomes more and more difficult to optimize, the internal running logic of the program becomes more and more difficult to understand, and the maintainability and security challenges come with it.

As an important means of software analysis, performance analysis has a very important position. Performance analysis includes deterministic performance analysis and statistical performance analysis. Deterministic performance analysis involves code-level performance analysis and system-level monitoring, which can accurately record all the details of function execution. Statistical performance analysis attempts to analyze code through sampling and probabilistic methods. Due to its small performance overhead and wide application scope, it is widely used for monitoring in production environments.[17] In 2024, a security vulnerability of xz utils was exposed. xz utils 5.6.0 and 5.6.1 versions were backdoor implanted into several stable Linux distributions including Debian, Fedora, etc. Fortunately, a security researcher at Microsoft called Andres Freund reported these suspicious conditions. He found that the program ran with an unnoticeably high CPU footprint. Following this lead, he finally discovered the security breach. This is a good example of the importance of the profiler.

1.2 Overview of Visual Performance Analysis and Related Works

Performance analysis generates a large amount of data that is a natural fit for visual analysis. Using graphs and charts to present performance data can help developers and analysts more intuitively understand the performance of the software. The call stack generated by the function execution can be combined with the timeline to draw precise

call details, and other execution data can also be used to obtain useful information through statistics and data mining and display by means of charts.

There are a variety of visual performance analysis tools for different platforms and programming languages. Among them, there are not only industrial-grade system monitoring complete solutions, automated testing and monitoring platforms, but also small tools for individual developers. The following sections provide a brief overview of various small performance profiling tools for individual developers, analyze their functionality.

1. vprof[11]

vprof is a visual performance analysis tool for Python. It has a built-in web server, which can be used from the command line to visually display CPU usage, memory usage and code internal running data.

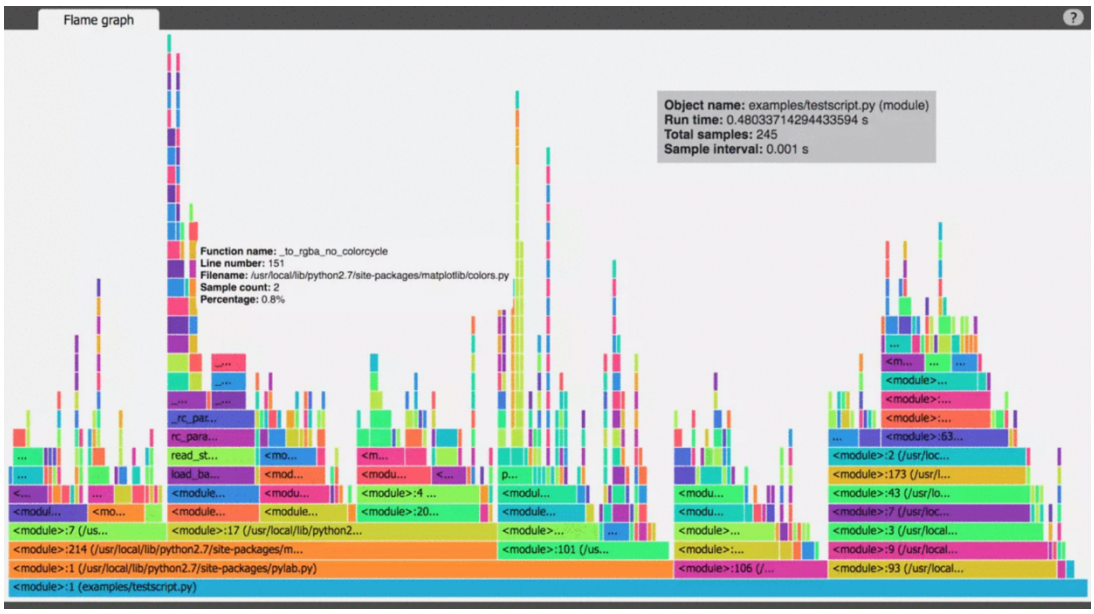

Figure 1-1 vprof flame graph screenshot

2. gprof2dot[15]

gprof2dot is an open-source script implemented in Python, which is used to convert performance data generated by various performance analysis tools into a visual call graph. The nodes in the call graph are in the structure of a directed acyclic graph and can display the call relationship between functions and the proportion of

execution time of each function.

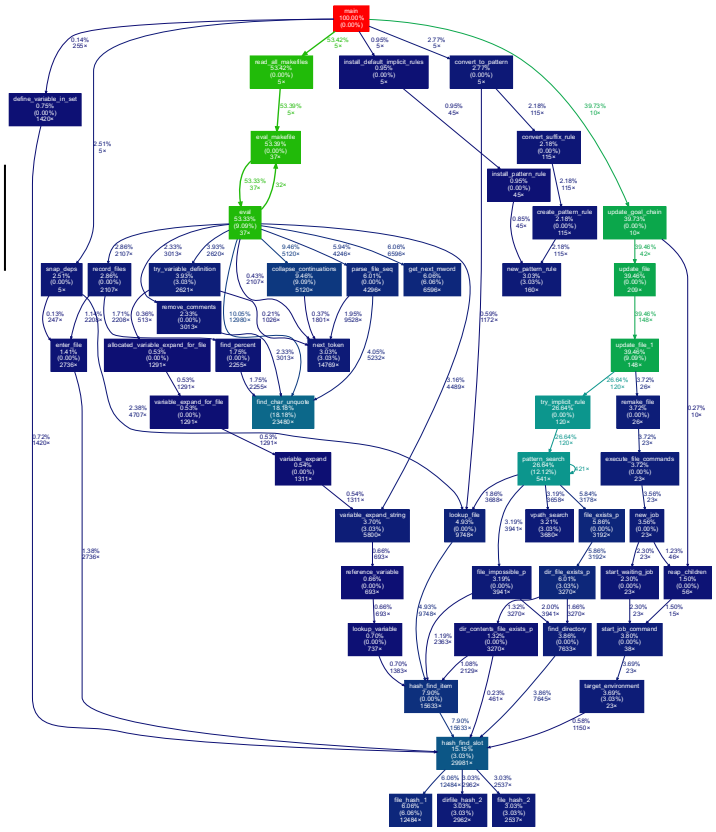

Figure 1-2 gprof2dot screenshot

3. JetBrain Profiler[18]

JetBrain IDE provides a convenient performance analysis tool for developers. These tools can help developers to easily detect the CPU and memory usage in the program during the development process and provide a chart to facilitate developers to view the call relationship between functions.

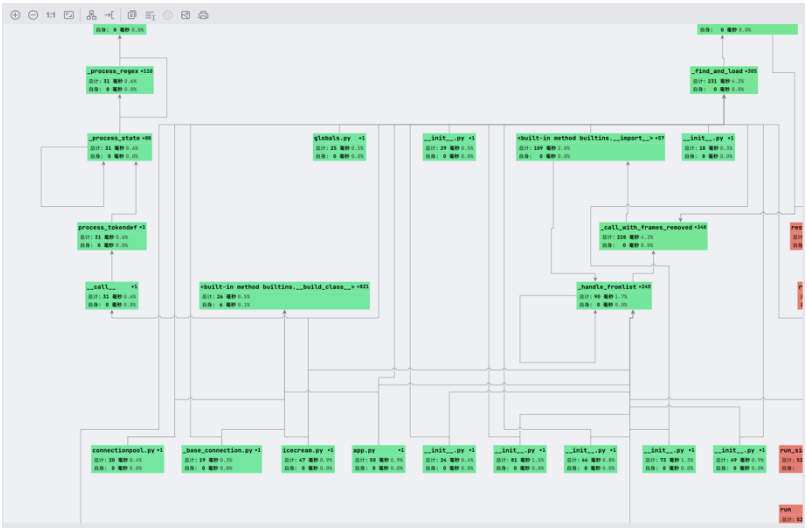

Figure 1-3 JetBrain Profiler screenshot

The logical hierarchy of the above projects is different, and it can be seen from various angles: from the data fetching side, vprof has a built-in performance analyzer, which is directly passed to the display side through the built-in web server; gprof2dot relies on external profiling tools to parse the data in a predefined .prof format.[19] JetBrain Profiler gets the data directly from the IDE platform. While vprof is a lightweight tool with a simple interface for each view, gprof2dot focuses more on the visualization of directed acyclic graph (DAG).

1.3 Browser-based Visualization Technology Overview

1.3.1 Basic Flow of Visualization

The basic flow of the visualization is shown in Figure 1-4.

1. The user selects the Python source file for performance analysis through the interface and uploads it.
2. RESTful server receives the file uploaded by the frontend and invokes the analyzer module.
3. The profiler module runs the code, analyzes and aggregates the execution data.
4. Pass back the processed data through the RESTful server.
5. The frontend receives the data and draws the visual interface.

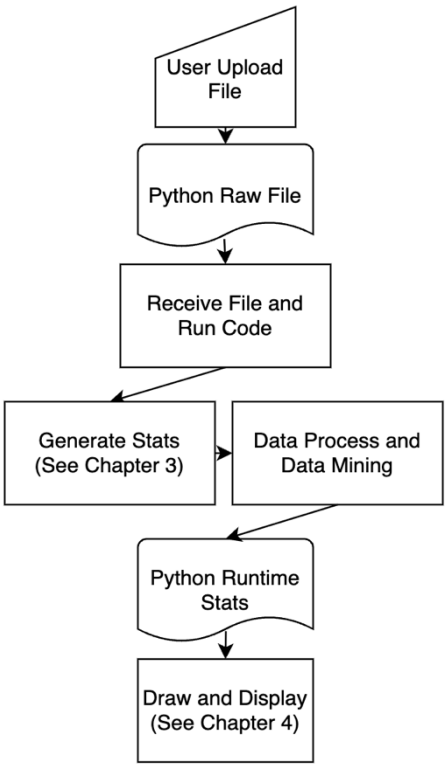

Figure 1-4 Visualization flow

1.3.2 Browser Drawing Technology

Browser technology is one of the cornerstones of modern Internet and digital communication.[2] With the development of browser technology, various specifications and tool chains tend to mature.[1] As the carrier of visualization technology, the importance of charts is self-evident. In the early days, drawing in the browser usually relied on plugins such as Flash and Silverlight. These toolchains were gradually replaced by HTML toolchains due to their security, compatibility, and performance advantages. At present, the mainstream drawing technology of the browser is SVG, Canvas, WebGL and so on.

SVG (Scalable Vector Graph) became a W3C recommendation standard in 2003. SVG graphics are vector graphics, which can guarantee no distortion when scaled, and can be used for high-precision graphics. SVG has good compatibility and fully supports DOM. HTML5 Canvas is also one of the W3C standards. Unlike SVG's standard, Canvas's drawing is based on pixels, which will cause aliasing when scaled. Canvas has high drawing efficiency, and it has better performance in drawing with a large amount

of data and fragmentary and heavy images. The detailed information is shown below.

	canvas	svg
Render	Based on pixels.	Based on shape.
Scaling	Zigzag on image edges.	No zigzag.
Control	Via script language.	Via script language or CSS.
Minimal unit	The entire canvas is part of the DOM, base graph is manipulated by script.	Every base graph is part of DOM and can be manipulated independently.
Efficiency	Efficient, especially when elements are complex.	Not very efficient.
3D support	Support 3D by specifying Canvas3D.	Do not support.

Table 1-1 Canvas and svg comparison

Visual charts contain many basic graphs. As the underlying browser drawing technologies, SVG and Canvas are not usually used directly to visualize the drawing of charts. This task is usually performed by various visualization frameworks. Visualization framework is the encapsulation of SVG and Canvas. Different from SVG and Canvas, which are based on drawing a single graph, all kinds of visualization frameworks provide the encapsulation of various levels of logic/interaction/information display from the perspective of the displayed data, the relationship between graphs and the significance of images. D3,[5] Highcharts, antv.X6 and others are based on SVG implementation, and ECharts and others are based on Canvas implementation.

1.3.3 Hierarchy of Visualization

As an intuitive way to display data, visual charts help users understand and analyze the internal relationship between data. From the perspective of chart implementation and construction, the construction of a visual chart is divided into the following levels.

1. Data layer.

Data layer is the foundation of visualization and includes the process of collecting,

organizing and preparing data. At this level, more attention will be paid to the authenticity, accuracy and completeness of the data. In practice, it is common to use a separate backend for data collection and preprocessing.

2. Graph mapping layer.

This layer is concerned with mapping data to visual elements. At this level, developers choose appropriate forms of visualization to effectively communicate the relationships and trends of the data.

3. Interaction layer.

This layer is concerned with the process of interaction between the chart and the user, including mouse operation, keyboard input, and so on. This layer is coupled with the graph-mapping layer, providing an interactive way for users to explore the data in greater depth and see more detailed information.

4. Annotation and explanation layer.

This layer provides the necessary text and images to understand the diagram, including explanatory text, legend, comment, etc.

1.4 Main Content and Organization Structure of the Project

This project takes Python to determine the performance analysis as the starting point, studies the underlying implementation and optimization of the visualization system, the implementation and encapsulation of the analyzer, and completes a full-process Python performance visual analysis system. The specific contents of each chapter are as follows.

This project takes Python to determine the performance analysis as the starting point, studies the underlying implementation and optimization of the visualization system, the implementation and encapsulation of the analyzer, and completes a full-process Python performance visual analysis system. The specific contents of each chapter are as follows.

The 1st chapter is the introduction part, which mainly introduces the research background, research significance, related research work at home and abroad and the

overall structure of the paper.

The 2nd chapter introduces key features and attributes of Python Virtual Machine and Python runtime data.

The 3rd chapter is a requirements analysis and tasks analysis.

The 4th chapter is a detailed introduction the visual analytics framework, including UI implementation, profiler and backend implementation. It shows the design of the UI interface, including every specific graph using several real case studies.

The 5th chapter is interview and some discussion about React framework,

2. Python Runtime Data

2.1 Python Virtual Machine Overview

Running a Python program involves a few steps, from source code to final execution, and this process is mainly done through the official implementation of Python, which is CPython.

When a Python program is running, the source code is lexically parsed, and the interpreter translates the source code into tokens. These tokens are the smallest elements of a programming language, like keywords, variable names, operators, and so on. Next, in parsing, the interpreter combines words into statements and checks to see if they are properly structured according to Python's syntax rules. This phase produces an abstract syntax tree that represents the structure of the program. Starting with an abstract syntax tree, the interpreter then compiles these constructs into bytecode.

The compiled bytecode is sent to the Python Virtual Machine (PVM) for execution. The Python virtual machine is a part of CPython, which is a stack-based interpreter.

2.2 Python Runtime Features

In Python, every function call creates a stack frame. This is a data structure that stores information about the environment in which the function executes, including local variables, parameters, return addresses, and so on. These frames are pushed onto

the call stack along with the function call, forming the context in which the program is executed. When a function returns from execution, its frame is popped off the call stack and returned to the state it was in before the function was called. This process of pushing and popping a frame is Python's context switch, which ensures the independence and safety of each function call and makes it easier to track down errors and debug your program. During context switches, we can log and monitor how the function is running by reading information from the frame object.

3. Requirement Analysis and Tasks Discussion

The above has introduced the basic process of visualization, the basic technology of browser drawing, the Python virtual machine and the running characteristics of Python programs. This section will analyze the needs of this project based on the above information.

3.1 Requirement Analysis

R1: Data Collection Detail

Python program execution contains many details, from the perspective of deterministic analysis, including the start and end time of the program function call, the pid (process id), tid (thread id), caller-callee relationship of the function, and from the perspective of statistics, including the average running time of the function event, the overall number of calls, and so on.

R2: Influence on performance

Logging function running events will inevitably affect the running performance. To avoid excessive data distortion, the following details should be considered when building the profiler: the amount of data collected should be as low as possible, and only the useful information should be collected; data collection should be separated from processing whenever possible, and time-consuming analysis should be deferred to information gathering.

R3: Frontend Interface Requirements

The frontend interface is the focus of the visualization project, and it can be

expected that a program run will involve multiple function calls, which will be reflected in many fine color patches and graphic elements.

3.2 Tasks Analysis

Based on the above three requirements, the following three tasks are proposed:

T1: Runtime Data Acquisition

This project needs to obtain program running data from two perspectives of deterministic analysis and statistical analysis, so it needs to develop two different performance analyzers. Among the problems are:

- *What kind of information needs to be collected and how will it be presented?*
- *What is the focus of deterministic and statistical performance analysis?*

T2: Control Performance Loss

Frame data is easy to obtain, but how to systematically obtain running data is a problem. The backend process needs to be carefully designed.

T3: Frontend Performance Optimization and Interaction

To ensure the performance of the program, it is necessary to optimize the frontend interface to avoid stagnation. Each function event contains a large amount of information that cannot be displayed in the main screen/main diagram at once, so it is necessary to design the corresponding interaction method so that the user can freely choose to view more detailed information. Issues to consider include:

- *What kind of interaction does the frontend interface need?*
- *What is the focus of deterministic and statistical performance analysis?*

4. Visual Analytics Framework

4.1 Data Processing and Modeling

4.1.1 Profiler Overview

Python program running data acquisition is an important part of this project. This project implements a total of two kinds of profilers, corresponding to deterministic

performance analysis and statistical performance analysis.

cProfile is a built-in performance analysis module for Python and is part of the Python standard library written in C. It is a wrapper for the profile module, which records the number and name of functions executed during the program execution, the total time and number of times each function is executed, and imports the above information into the .prof format file.

There are two ways to accurately record the start and duration of a function call. One way is to set up a hook function on the context switch during the function execution. Each time the function is called (the context switch), the current stack frame is logged, thus obtaining the entire call stack. Another approach is to set a timer during the execution of the function, sample the run of the function at a fixed interval, record the run information of the time slice, and finally restore the entire call stack through all the above slices. It is important to note that accurate performance analysis always affects performance (even greatly), so the goal in this process is to minimize distortion rather than eliminate it.

sys.setprofile is a function in Python's built-in sys module that sets up a global profiling function that allows tracking and profiling of events at the Python interpreter level. While setting up an profiling function with *sys.setprofile*, the function will be called every time a specific Python event (such as a function call, function return, exception, etc.) occurs. When the profiling function is called, three parameters are passed: frame, event, and argument, representing the current stack frame, the event type, and the event's parameters, respectively. All the data can be retrieved that the function is running.

Corresponding to the two performance analysis methods, using cProfile profiler has less impact on performance, but only partial information can be recorded. Using *sys.setprofile* records every details of each call, but it has poor performance because it requires extra Python code to be executed for each context switch.

4.1.2 Call Stack Recovery

To avoid the profiler having too much negative impact on performance, the profiler

usually collects as little information as possible only at runtime. In this case, the profiler only collects the start and duration of each function call, so we need to devise an algorithm to retrieve the call stack from this information. Given that only one function can be executed at a time, the following algorithm is proposed and implemented.

Algorithm 1: Call Stack Recovery

Input: Stack event array.

Output: The returned stack event array, event contains calculated "children" attribute.

```
1:  Initialize min-heap minh based on invocation time of each event.
2:  Initialize max-heap maxh based on finish time of each event.
3:  Initialize stack stk
4:  level ← 0
5:
6:  i ← arr[1]
7:  j ←arr[end]
8:  while size(minh) != 0 do
9:      temp_event = minh.pop()
10:     temp_event.level = level
11:     level += 1
12:
13:     if size(stk) != 0 then
14:         stk[-1].children.append(temp_event)
15:     end if
16:     stk.append(temp_event)
17:
18:     while maxh.peak() == temp_event do
19:         maxh.pop()
20:         stk.pop()
21:         if size(stk) != 0 then
22:             break
23:         end if
24:         level -= 1
25:         temp_event = stk[-1]
26:     end while
27: end while
28: return stk
```

The restored call stack has the same structure as the call stack itself. We can see from the JSON file that each event object has an additional children property that

represents the function events directly called in that event.

4.2 UI Implementation

This section will introduce the design of UI interface and its flow of use. The program is divided into a main interface and three diagram display interfaces, the user enters the corresponding graph display interface by uploading the code in the corresponding position of the main interface.

The main interface provides the necessary guidance for the user with clear titles and instructions. There are three entry points for submitting Python code, corresponding to three views, which will be covered next.

The third submission provides an interactive form to help the user enter a customized application runtime, auto-add cells dynamically, and a JSON preview window at the bottom to preview the user's input.

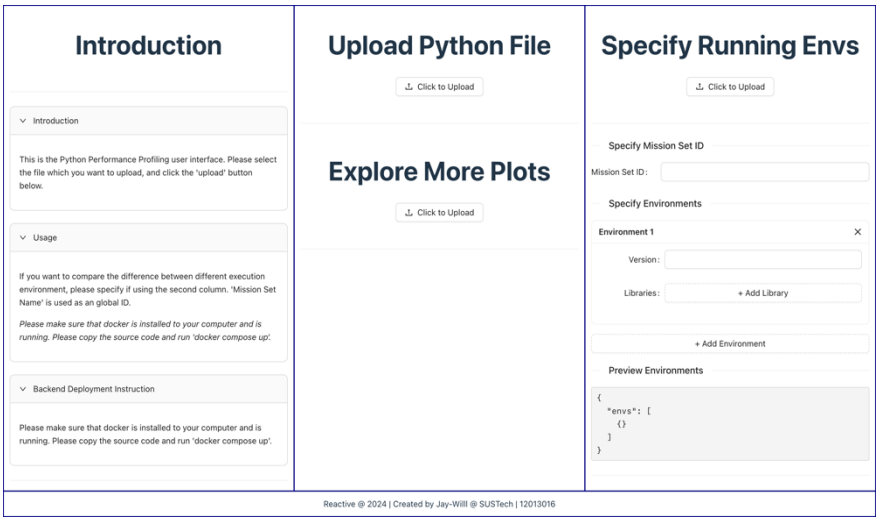

Figure 4-1 Homepage screenshot

4.2.1 Inspecting the Call Stack

In a typical Python program, the root function call event entry is *builtin.exec*, other entries are the offsprings of this root entry. It forms a tree with an indefinite number of branches.

4.2.1.1 Icicle Graph

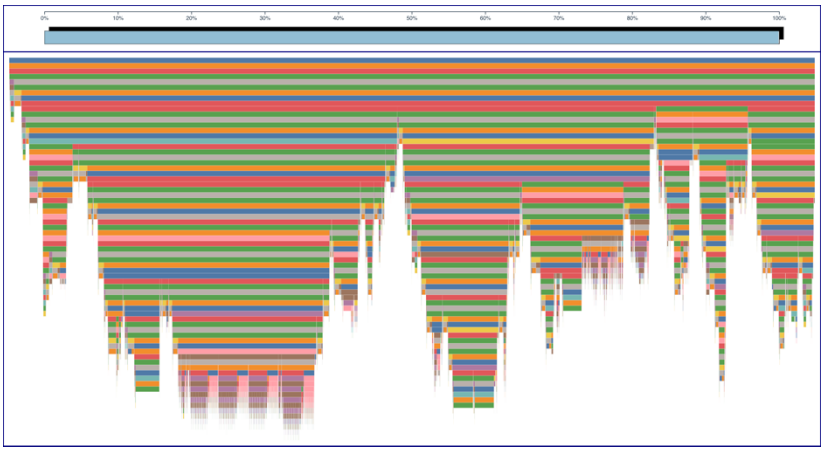

Figure 4-2 Icicle plot screenshot

Icicle graph is the very fundamental graph of program performance profiling, and it provides a both intuitive and accurate way to visualize the structure. It directly reflects the quantitative and caller-callee relationship between function call event entries.

Chart Elements: Every color block in the chart represents a function call entry. The width of the block represents the duration of the specific function call. It is worth noting that the length relationship between each function call block is accurate. If one color block is placed below the other, it means that the longer one is the caller of the shorter one. The color blocks are arranged in order of rank to clearly show which level they are in the call stack.

Interaction: Icicle graph is the very fundamental graph of program performance profiling, and it provides a both intuitive and accurate way to visualize the structure. It directly reflects the quantitative and caller-callee relationship between function call event entries.

4.2.1.2 Radial Tree Graph

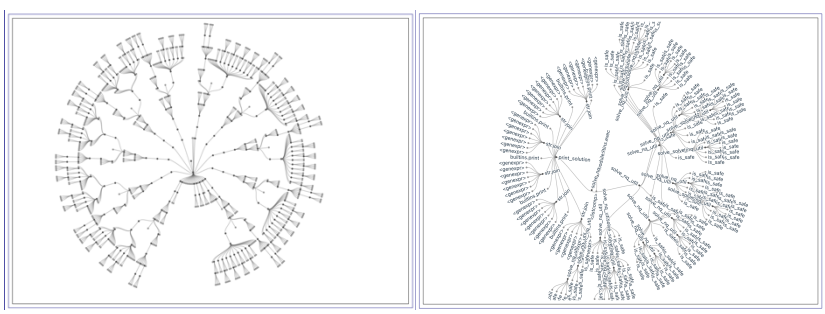

Figure 4-3 Radial plot screenshot

Radial tree graph is a great way to represent a tree structure. Generally, it is a morphed minified version of the icicle graph. It extracts the important generation information in the tree structure and can be fully displayed in a smaller screen size.

Chart Elements: Every node in the chart represents a function call entry. The node in the center represents the entry of the Python program. The node are arranged in order of rank to clearly show which level they are in the call stack, and the N+1 layer is the callee of layer N. The lines between nodes represent direct calling relationships.

Interaction: The radial tree graph implemented in this project supports mouse hover interaction. When users hover mouse over a color block, a pop-up windown will pop up automatically, used to display detailed runtime information of that node.

4.2.2 Inspecting Statistical Operating Data

Statistical operating data of a Python program contains runtime information, more importantly, it records important information about the relationship between function calls, including the number of recursive calls, the number of non-recursive calls, the percentage of function runs, the average time per run, and so on.

4.2.2.1 Scatter Plot

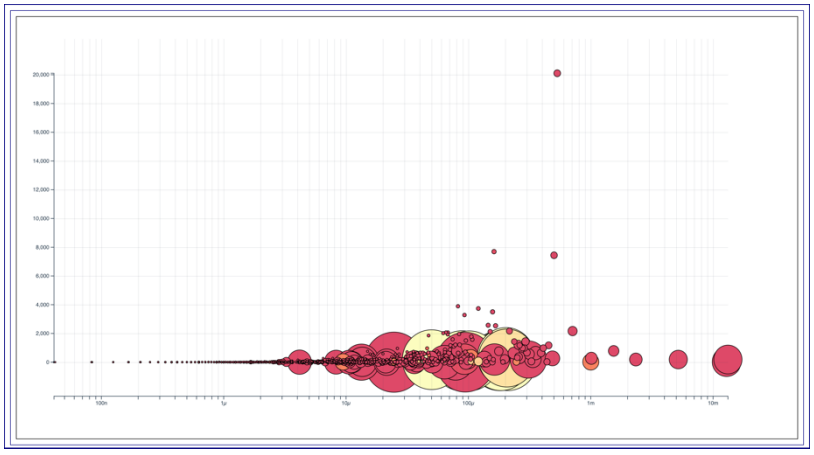

Figure 4-4 Scatter plot screenshot

Given the runtime data characteristics described above, scatter plot is a good vehicle for visualizing these data. This chart flexibly combines the various information

of the function running, which can easily help users to understand the bottleneck of the function running, and the number of events between each function.

Chart Elements: The X-axis represents the average time taken to run each function, and the Y-axis represents the number of times the function was run. Each block has a different color and size, where the color represents the ratio of recursive calls to the total number of calls, and the size represents the total run time of the function.

Interaction: The scatter plot implemented in this project supports mouse hover interaction. When users hover mouse over a color block, a pop-up window will pop up automatically, used to display detailed numerical data.

4.2.3 Inspecting Multi-environments Comparison Data

The user uploads the code file after selecting the environment that needs to be executed, and the system will automatically detect it.

4.2.3.1 Bar Chart

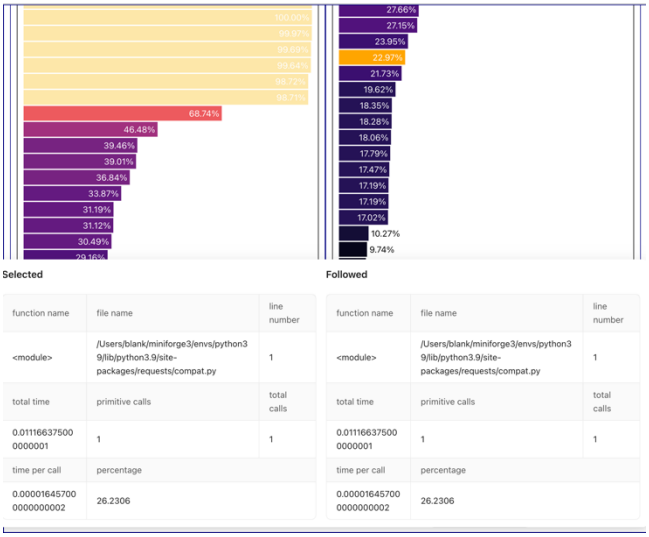

Figure 4-5 Bar chart screenshot

As a basic form of chart, bar chart can also provide important information.

Chart Elements: The X-axis represents the time consume percentage of each function, and the Y-axis sorts the functions by percentage. The bars are colored proportionally to help the user get a sense of the proportion of events they run.

Interaction: The bar chart implemented in this project supports mouse hover interaction. When users hover mouse over a color block, a pop-up window will pop up automatically, used to display detailed numerical data.

4.2.3.2 Cascade Treemap

A Cascade treemap is a special type of treemap that is used to effectively display hierarchical data in a cascade on a two-dimensional plane. It was proposed by Ben Schneiderman in the early 1990s to display tree-structured data by nesting rectangles.

Chart Elements: Each rectangle represents a node in the tree, and its area is proportional to the size of the represented data. The chart not only shows the relationship between the execution of the function, but also reflects the size of its execution event. The chart is colored by function nesting depth to help users understand the time composition of the entire function run.

Interaction: The cascade treemap implemented in this project supports mouse hover interaction. When users hover mouse over a color block, a pop-up window will pop up automatically, used to display detailed numerical data.

It's worth noting that in the multi-context comparison scenario, multiple charts are linked. When the mouse hovers over a function event graph, the corresponding function event graph in the other context is also highlighted, and the data from both are displayed in the popover component.

4.3 Implementation

4.3.1 Encapsulation of Profiler

In this project, to facilitate the call of the analyzer and improve the system integration, all the functions of the analyzer are integrated in a Python package. The structure of a Python package is shown below.

Profiler Python Package Structure

```
1: .
2: ├── __init__.py
3: ├── __main__.py
4: ├── frame.py
```

```
5:  ├── runner.py
6:  ├── profiler
7:  │   ├── __init__.py
8:  │   ├── base_profiler.py
9:  │   ├── event_profiler.py
10: │   └── model
11: │       ├── __init__.py
12: │       ├── base_model.py
13: │       └── calc_model.py
14: └── view
15:     ├── __init__.py
16:     ├── list_view.py
17:     └── stack_view.py
```

Figure 4-6 Profiler structure

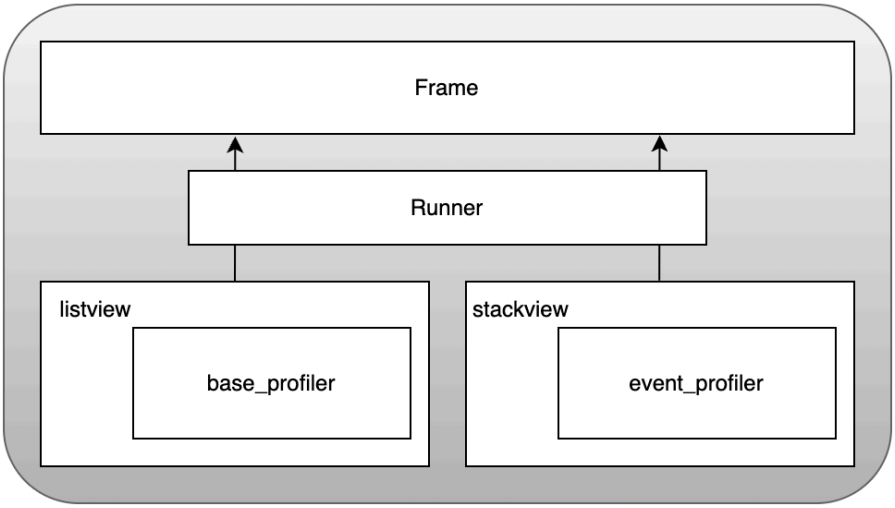

Figure 4-7 Profiler execution flow

In the profiler, *base_profiler* wraps the profiler based on the cProfile implementation and *event_profiler* wraps the profiler based on the *sys.setprofile* implementation. `model` stores the objects and classes used by profilers.

frame and *view* define interfaces that are exposed to the user, receive user arguments, and return processed results. Used to further encapsulate the profiler.

runner implements a code runner to invoke new threads to execute code, creating an isolated runtime environment.

4.3.2 Cluster and Server

In order to realize the function of frontend uploading code and backend running and returning data, this project implements a backend server cluster system. The system has the following features: allows users to customize the execution environment of Python code, including the main version of Python, the version of dependencies. The execution data for each environment is returned after the runtime data is obtained. Its architecture diagram is shown below.

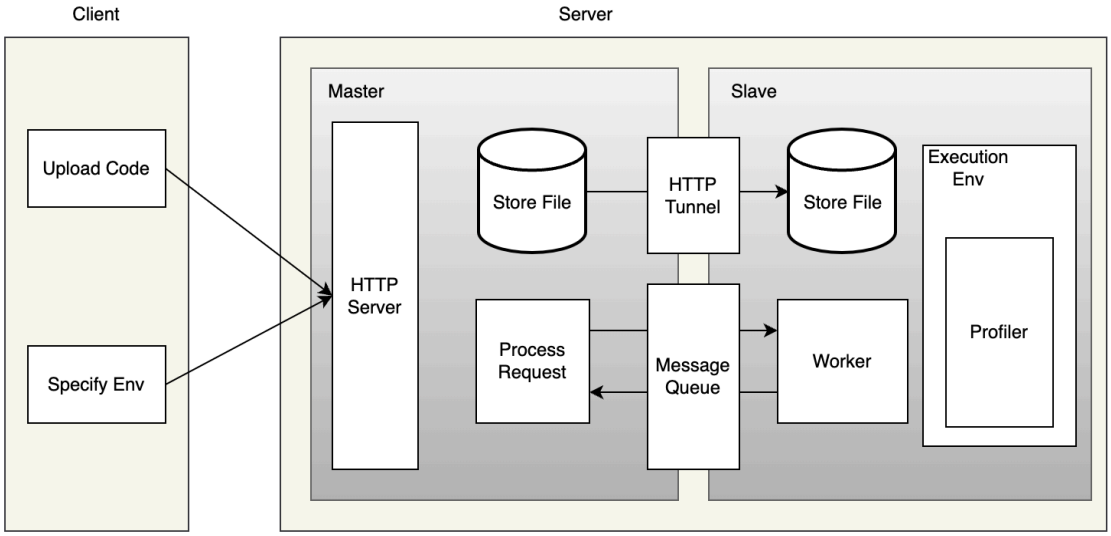

Figure 4-8 Server cluster architect

To implement the above functions, this project uses Docker to implement two processing nodes: master and slave. In a normal request flow, the master node receives the code uploaded by the user and the special environment configuration, and attaches a global id to the input, which is easy to store and track. After receiving the uploaded code, the master node will back up the code file locally and forward it to the slave node, which will not run directly after receiving the code file but wait for the master node to process it. The slave node will use conda to create a virtual environment for each run, run the profiler described above, and return the results via the message queue.

The above architecture ensures the independence, robustness, security and reliability of the whole system.[14] Using docker to isolate the master and slave nodes

makes the whole system structure and division of labor clear and ensures that the analysis process will not be interfered by other factors. Using the conda virtual execution environment not only ensures the purity of the runtime environment, but also avoids the dependency problem between python libraries as much as possible, making it possible to use the runtime comparison between different versions of the library.

5. Case Study and Interview

5.1 Case Study

In this section, we will explore how the system works in real life and how it helps us understand the inner workings of Python programs.

5.1.1 Simple NumPy Example

NumPy is an open-source Python library that provides high-performance multidimensional array objects and a wide range of tools and functions for working with these arrays. In this section, the project analyzes a simple NumPy example. Please refer to Appendix C Code Block 1 for the code.

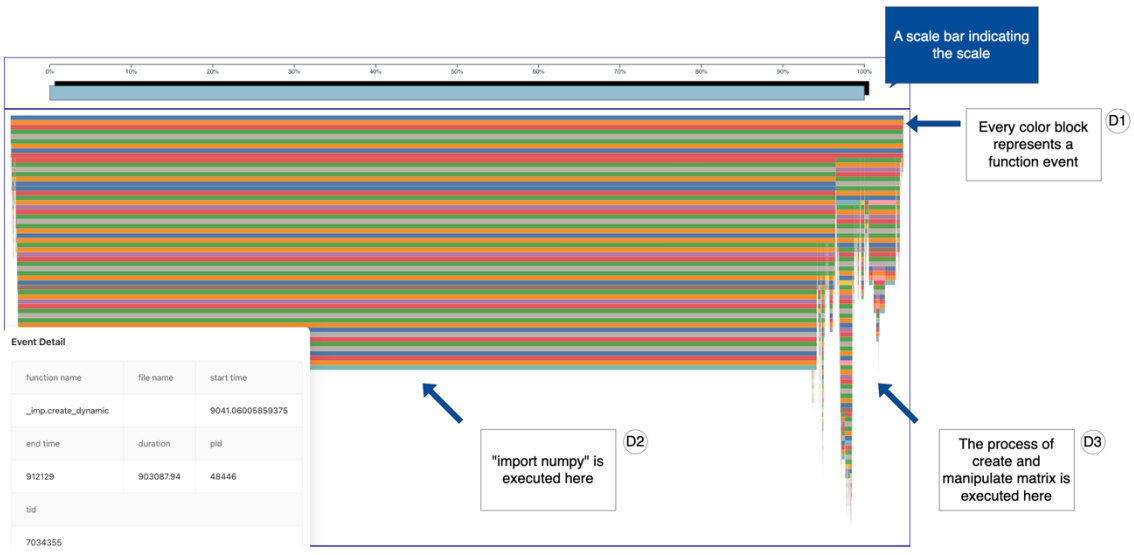

Figure 5-1 NumPy case study screenshot

As shown in the figure ***D1***, every color block delineates the start and end time. In ***D2*** part of the graph, it shows a plateau structure. Looking at it using an interactive

window, we can see that the *_imp.create_dynamic* function takes a lot of time to run. That is because in the Python extension module, *_imp.create_dynamic* is a low-level function for dynamically loading shared libraries. This function is part of the built-in Python module *_imp* and is primarily used to import compiled binary extension modules, such as.so files, at runtime. These files are written in C and compiled into a DLL, a process that is usually quite time-consuming to load. While the extension are ready, it'll take relatively shorter time to execute the main part of the module, like creating or manipulating matrix, which is clearly shown in ***D3***.

5.1.2 Simple Recursive Example

Recursive call is an important program structure, which is obviously reflected in some classical programs such as the Eight Queens problem. Recursive structures also occur in other general-purpose programs. Since the recursive structure usually involves many numerical calculations, finding this structure can help us to understand the performance bottleneck of the program and optimize the performance of the program better.

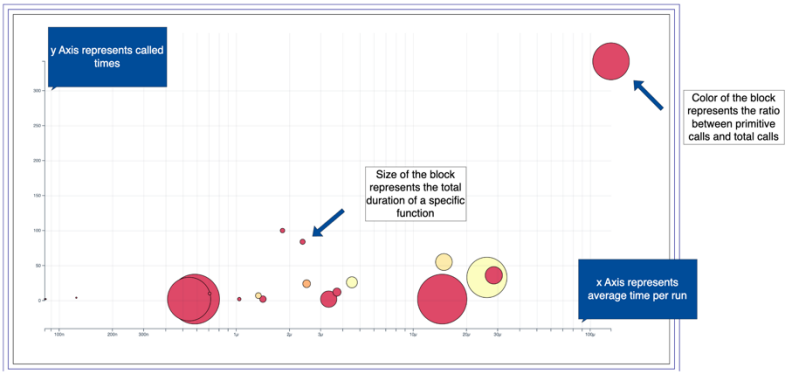

Figure 5-2 Recursive case study screenshot

As it can be seen from the figure, some functions take a long time to execute but are rarely executed; they are usually low-level or near-low-level functions that serve as necessary constructs for the program. Of more concern are the colors near the Y-axis or the lighter colors, which are often the performance bottleneck of the program because they are called more often or iterate more often.

5.2 Summary and Interview

Compared with other open-source projects (shown in Section 1.2), this system makes significant improvements in the following aspects.

1. Security and effectiveness of data acquisition

In terms of data acquisition, users can completely customize the Python interpreter version and the version of dependent class libraries, and the code analysis process is strictly limited in the virtual environment (docker container and conda virtual environment), which ensures the security of the system and avoids the impact of interference factors on the performance analysis process.

2. Diversity of interactions

The charts of this system have more interactive ways and can provide more intuitive and diverse information. The details are shown in the table below.

	vprof	**gprof2dot**	**JetBrainProfiler**	***This project***
Acquisition	Built-in	External	Built-in	Built-in
Mode	Deterministic	Statistically	Deterministic	Both
Usability	Command line	Command line	GUI	GUI
Environment	Non-isolated	N/A	Isolated	Isolated
Interactivity	Little	N/A	Little	Abundant

Table 5-1 Comparison between projects

In the project, we distributed the system to undergraduate students of science and engineering universities for evaluation, including students of computer science and engineering, other science and engineering students with programming experience, and zero-foundation students who are currently learning Python programming. During the evaluation process, invited students can freely use the system and compare different systems using the same example. Most of the invited people gave very positive evaluation of the system, and they fully affirmed the advantages of the system. Some students from the Department of Computer Science and Engineering gave the following

comments: "This system combines deterministic and statistical performance analysis, so I can get a very good idea of how precise sampling is affecting performance." "The comparison between different versions of the library is very useful, it helps me understand the impact of different versions on specific programs." Other students were also satisfied with the ease of use of the system and felt that using Docker brought noticeable benefits to the system, including near zero cost deployment and out-of-the-box features. We also received suggestions from students on the improvement of the system, including suggestions on the aesthetic degree of the system itself and suggestions on the interactive logic of the frontend interface of the system.

6. Discussion and Conclusion

6.1 Discussion about UI Interface

6.1.1 Discussion on React Framework

There are many browsers and JavaScript-based stacks and ecosystems. React was chosen as the main framework for this project because it implements a set of virtual DOM and update rules to better manage state and improve performance.[8] JSX helps developers organize their code better. On top of that, due to the complex relationship between the different components of the visual interface, a more efficient way to centrally manage state and events was needed, so the project introduced Redux as a global state manager. To better manage style sheets,[9] this project introduces LESS.[10] To facilitate faster packaging and development, this project uses Vite instead of the official scaffolding to initialize the project.

React embraces functional programming by codifying its logic into hook functions, hooks functions are provided by official or third-party libraries and can also be customized.

They come in a variety of categories, but they can perform a specific task related to the internal implementation of the framework, such as updating state to trigger redrawing, or providing an interface to handle side effects during program execution. Provides memorization and dataflow dependency logic, such as memorizing functions

that have been built and data that has been computed.

Logically, the system is very clear, but there are many problems in the actual implementation process. First, in terms of code organization, an application based on MVC architecture usually has a clear logic architecture. From the organization of React, its MVC should look like this: state perform as M, render and return logic performs as V, which C is *setState* hook function. But the role of M and C is designed to be too weak, and V in this situation can be too complicated. In real world situation there are so many *useCallback* and *useMemo*, which enervate the robustness and scalability of the whole application. Things can get even worse when it comes to third-party components that need to be managed like svg, canvas or other external components.

It's also worth saying that while embracing functional writing is perfectly compatible with the nature of the JavaScript language, it deters clear component lifecycle management by experimenting with hooks like *useEffect*, *useLayoutEffect* to simulate component creation, mounting, and In some special cases, the system can become very unstable, because the real update operation logic is hidden like a black box, and the problem becomes difficult to solve.

6.1.2 Discussion on Layout Control

Layout control in React projects is essential for creating user-friendly, performant, and maintainable web applications. It impacts user experience by ensuring intuitive navigation and responsive design across devices. Optimizing component arrangement boosts performance by improving load times and reducing rendering issues. Good layout practices enhance maintainability and scalability, making it easier to update the application and add new features, but the layout control is very difficult, because it is related to both the content presented and the logic of the program, so it is difficult to decouple the content and the layout. The following principles were followed for layout control in this project.

First, this project fully uses flex layout control, which allows the application to be displayed on computers as well as mobile phones or tablets, because the layout control system automatically detects the screen size and arranges the appropriate size and

position for each component.

The application is divided into several parts. Each part of the application is completely independent, and each part manages its child components separately, resulting in a top-down structure for layout control and data flow.

The code for each component to acquire its layout information is shown below.

```
function Example() {
  const divRef = useRef(null);
  const [dimension, setDimension] = useState({width: 0, height: 0});

  useUpdateLayoutEffect(() => {

  }, [dimension]);

  useLayoutEffect(() => {
    if (divRef.current) {
      setDimension({
        width: divRef.current.clientWidth,
        height: divRef.current.clientHeight
      })
    }
  }, []);

  return (
    <div
        ref={divRef}
        style={{
            width: '100%',
            height: '100%'
        }}
    >
        <>COMPONENT HERE</>
    </div>
  )
}
```

Figure 6-2 Layout control logic

In general, this project gets the size and position information of the component through a combination of *useState* and *useLayoutEffect*. By defining the size of the component in JSX, the component will have a chance to get its own size information before being rendered to the screen. This information is then saved to the state using the *useLayoutEffect* hook for later use.

It's worth noting that when a component has two levels of child components, this way of getting the position and size of the component by itself doesn't work. Since the rendering of the child component is always completed before the rendering of the parent component, and the position and size information of the child component is inherited from the parent component, if we continue in the previous way, the child's size will be set to a default size, usually 0 for both length and width. This is where the case it is necessary to pass layout information from parent to child via props.

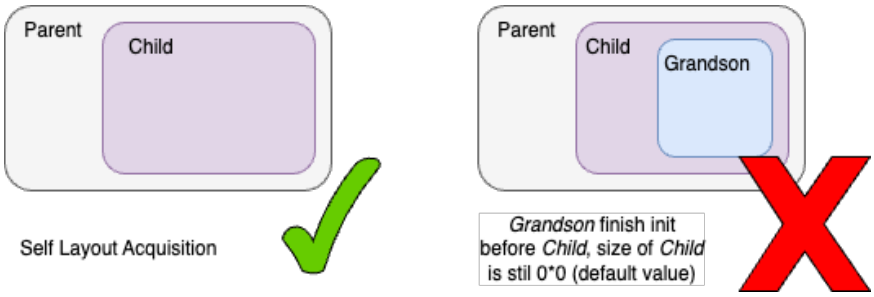

Figure 6-3 Layout management pitfall

Through the above layout control methods, the application can run on various types of devices, including mobile phone, tablets and ordinary computer screens.

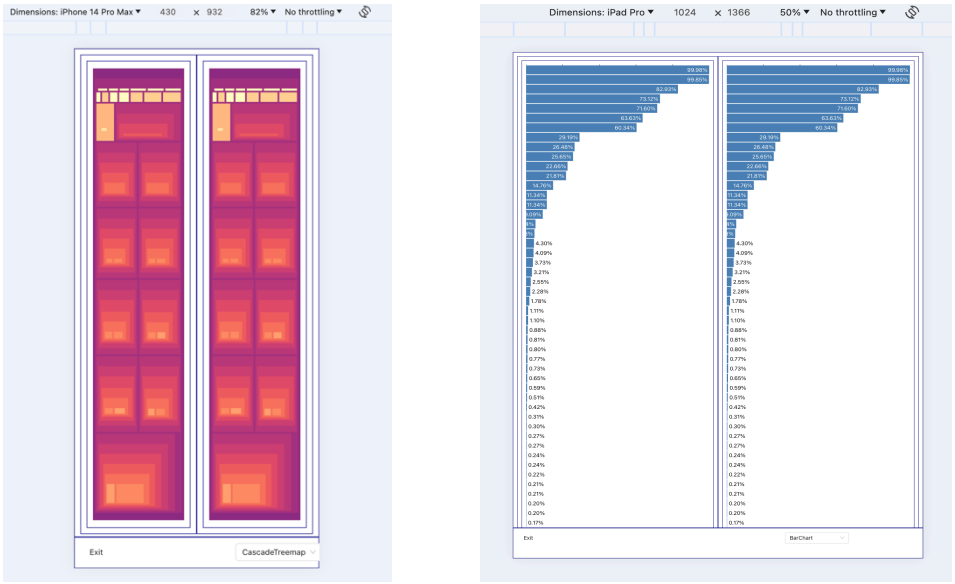

Figure 6-4 Application on iPhone and iPad

6.1.3 Discussion on SVG Graph Zooming

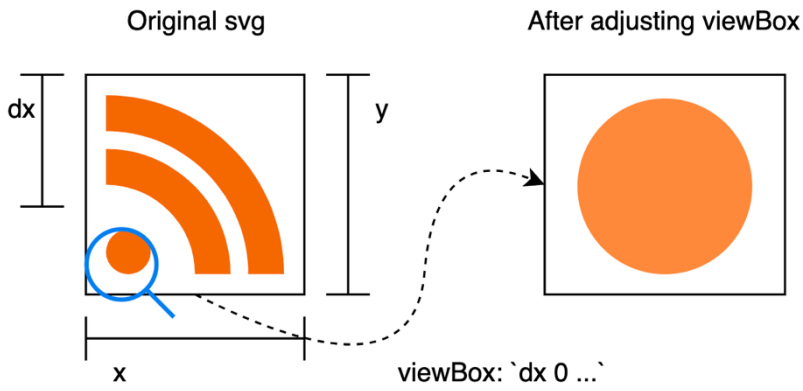

Figure 6-5 SVG viewBox explanation

Compared with other open-source projects (shown in Section 2.2), this system makes significant improvements in the performance of svg zooming. This system uses the *viewBox* attribute of svg to control the zoom. Compared with other svg graphics zoom implementation, this implementation method has obvious performance advantages. All graphs are created at initialization time and only change the position of the view window during interaction. This project tested other off-the-shelf third-party zooming libraries including d3.zoom[21], panzoom[20]. They perform poorly in scenarios involving such many graph fragments, and re-traversing and plotting the location of each event consumes a lot of time.

6.2 Limitation and Future Work

This project encountered great difficulties in the implementation of the data display side, because this project involved many custom components, and many components were designed with the original intention of "building from scratch". However, due to the complexity of the frontend component itself, the layout control, interaction logic and implementation logic of the component are often mixed, and the component itself is coupled with routing, global state management, data acquisition and other parts, which leads to a high degree of coupling of the whole system and the development is very difficult. The completion of some components is also not as expected, such as the progress bar component, which uses the underlying event responder of the browser without further encapsulation and boundary conditions, and in special cases will

produce special problems such as inconsistent feel and behavior that does not meet expectations.

This project realizes the acquisition and display of Python program running data. To facilitate data flow, a custom JSON format is used to encapsulate and transmit data. In the future, the project will consider supporting additional languages by uploading .prof files directly.

Sometimes the results of the analysis can be very large, which makes it necessary to abandon some possibilities more interesting results for the project, because all data are retrieved synchronously. Even so, the minified data is sometimes too large, causing browser engine heap overflow issues at runtime. In this case, part of the chart cannot be viewed.

In terms of system ease of use and other aspects, this project considers adding a database in the future to record the code upload records of users. This gives the freedom to select the results that have already run and compare them.

Supplementary Explanation

This project is an open-source project, please refer to https://github.com/Jay-Willl/reactive. The backend uses rprof as a wrapper of profiler, which is developed for this project. Some design structures and ideas refer to the vprof project.[11]

Acknowledgement

I would like to express my deepest appreciation to all those who provided me the possibility to complete this thesis. A special gratitude I give to my advisor, professor Yuxin Ma, whose contribution in stimulating suggestions, helped me to coordinate my project especially in writing this thesis. I would also like to acknowledge with much appreciation the crucial role of the professor Zhuozhao Li, who provides valuable suggestions to my thesis and works.

I am also immensely grateful to my parents, who have supported me firmly and spiritually throughout my undergraduate life.

To Austrian composer Gustav Mahler, French composer Maurice Ravel and German composer Johannes Brahms, thanks for your wonderful music.

Appendix

Appendix A

Appendix A1: Frontend Technology Stack

Technology Stack	Version	Description
react	18.2.0	Frontend UI framework.
redux	5.0.1	Global state management.
ahooks	3.7.10	Reusable customized hook.
antd	5.15.4	Component library for react.
D3.js	7.9.0	SVG management.
Highlight.js	11.9.0	Code syntax highlight.

Appendix A2: Backend Technology Stack

Technology Stack	Version	Description
flask	3.0.0	HTTP server framework.
flask-cors	3.0.10	Flask CORS support.
zeromq	4.3.5	Message queue.
requests	2.31.0	Make and send HTTP request.

Appendix B: Hooks Function

Function	Source	Description
useState	Official library	Add local state to function components, causing the component to re-render whenever the state changes.
useEffect	Official library	Perform side effects in function components.
useLayoutEffect	Official library	Read layout and perform synchronous updates before the browser repaints.
useUpdateLayoutEffect	ahooks	Perform updates after layout calculations but before paint.
useMemo	Official library	Memoizes a computed value.
useCallback	Official library	Memoizes a callback function.
useCreation	ahooks	Ensures a value is created only once and memorized.
useDispatch	Redux toolkit	Returns a reference to the dispatch function from the Redux store.
useSelector	Redux toolkit	Extract data from the Redux store state.
useRef	Official library	Returns a mutable ref object which persists across re-renders.
useMouse	ahooks	Acquire mouse position.
useGetState	ahooks	Access the current state value from within a callback function.
useSateState	ahooks	Ensures updates are only made if the component is still mounted.
useNavigate	React router	Programmatically navigating between routes.

Appendix C: Examples Used

Code Block 1: A numpy sample

```
import numpy as np

a = np.array([1, 2, 3, 4, 5])
b = np.array([10, 20, 30, 40, 50])
print("Array a:", a)
print("Array b:", b)
c = a + b
d = a * b
e = a ** 2
print("\na + b:", c)
print("a * b:", d)
print("a ** 2:", e)
mean_a = np.mean(a)
sum_b = np.sum(b)
max_a = np.max(a)
min_b = np.min(b)
print("\nMean of a:", mean_a)
print("Sum of b:", sum_b)
print("Max of a:", max_a)
print("Min of b:", min_b)

matrix_1 = np.array([[1, 2], [3, 4]])
matrix_2 = np.array([[5, 6], [7, 8]])
matrix_product = np.dot(matrix_1, matrix_2)
matrix_transpose = np.transpose(matrix_1)
print("\nMatrix 1:\n", matrix_1)
print("Matrix 2:\n", matrix_2)
print("Matrix product:\n", matrix_product)
print("Matrix transpose:\n", matrix_transpose)

a_filtered = a[a > 2]
print("\nFiltered array a (elements > 2):", a_filtered)
```

Code Block 2: A recursive sample

```
N = 6
board = [[0] * N for _ in range(N)]


def print_solution(board):
    for row in board:
```

```
        print(' '.join('Q' if x else '.' for x in row))


def is_safe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True


def solve_nq_util(board, col):
    if col >= N:
        return True
    for i in range(N):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_nq_util(board, col + 1) == True:
                return True
            board[i][col] = 0
    return False


def solve_nq():
    if not solve_nq_util(board, 0):
        print("Solution does not exist")
        return False
    print_solution(board)
    return True


solve_nq()

call_count = {}


def count_calls(func):
    def wrapper(*args, **kwargs):
        key = func.__name__
```

```
        call_count[key] = call_count.get(key, 0) + 1
        return func(*args, **kwargs)

    return wrapper


@count_calls
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)


@count_calls
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)


@count_calls
def sum_recursive(n):
    if n == 0:
        return 0
    else:
        return n + sum_recursive(n - 1)


@count_calls
def sum_iterative(n):
    total = 0
    for i in range(n + 1):
        total += i
        sum_recursive(0)
    return total


@count_calls
def direct_call(n):
    return n

```

```
factorial(5)
factorial(5)
fibonacci(6)
sum_recursive(10)
sum_iterative(5)
direct_call(10)

print("Function call statistics:")
for func, count in call_count.items():
    print(f"{func}: Total Calls = {count}")
```

References

[1] 尤雨溪. 开源前端框架纵横谈[J]. 程序员, 2013(03).

[2] Matt Frisbie. Professional JavaScript for Web Developers, 4th Edition[M]. Indianapolis, Indiana. John Wiley & Sons, 2020.

[3] Allen Wirfs-Brock and Brendan Eich. 2020. JavaScript: the first 20 years. Proc. ACM Program. Lang. 4, HOPL, Article 77 (June 2020), 189 pages. https://doi.org/10.1145/3386327.

[4] J. Calleya, R. Pawling, C. Ryan and H. M. Gaspar, "Using Data Driven Documents (D3) to explore a Whole Ship Model," 2016 11th System of Systems Engineering Conference (SoSE), Kongsberg, Norway, 2016, pp. 1-6, doi: 10.1109/SYSOSE.2016.7542947.

[5] D3: Data-Driven Documents, [online] Available: https://d3js.org/what-is-d3.

[6] J.Heer and M.Agrawala. Software Design Patterns for Information Visualization. IEEE Trans Vis and Comp Graphics, 12(5):853–860, 2006.

[7] J.Heer and M.Bostock. Declarative language design for interactive visualization. IEEE Trans Vis and Comp Graphics, 16(6):1149–1156, 2010.

[8] React: The library for web and native user interfaces, [online] Available: https://react/dev.

[9] H. W. Lie. Cascading Style Sheets, [J] PhD thesis, University of Oslo, 2005.

[10] Less: Linear Style Sheet [J] Available: https://lesscss.org.

[11] vprof project, [online] Available: https://github.com/nvdv/vprof.

[12] M. Bostock, V. Ogievetsky and J. Heer, D³ Data-Driven Documents, in IEEE Transactions on Visualization and Computer Graphics, vol. 17, no. 12, pp. 2301-2309, Dec. 2011, doi: 10.1109/TVCG.2011.185.

[13] Flask: The Python micro framework for building web applications. [online] Available: https://flask.palletsprojects.com.

[14] 刘思尧,李强,李斌.基于 Docker 技术的容器隔离性研究[J].软件,2015,36(04):110-113.

[15] gprof2dot project, [online] Available: https://github.com/jrfonseca/gprof2dot.

[16] FlameGraph, [online] Available: https://www.brendangregg.com/flamegraphs.html.

[17] Brendan Gregg, Systems Performance: Enterprise and the Cloud, 2nd Edition[M]. (2020).

[18] JetBrain Profiler, [online] Available: https://www.jetbrains.com/pages/intellij-idea-profiler.

[19] GNU Profiler, [online] Available: https://ftp.gnu.org/old-gnu/Manuals/gprof-

2.9.1/html_mono/gprof.html.

[20] panzoom, [online] Available: https://github.com/anvaka/panzoom.

[21] d3-zoom, [online] Available: https://github.com/d3/d3-zoom.