UNIVERSITY OF CALIFORNIA,
IRVINE

Real-time Software Execution Visualization: Design and Implementation

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Zizhao Han

Thesis Committee:
Professor James Jones, Chair
Professor David Redmiles
Professor Iftekhar Ahmed

2021

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

ACKNOWLEDGMENTS **vi**

ABSTRACT OF THE THESIS **vii**

1 Introduction **1**
1.1 Summary of Contributions 2
1.2 Thesis Structure 3

2 Related Works **4**
2.1 Visualizing Run-time Behavior, Post hoc 4
2.2 Visualizing Real-time Software Behavior 6
2.3 Visualizing Non-run-time Source Code 6

3 Motivation and Challenge **8**
3.1 Challenge 1: Showing all information 9
3.2 Challenge 2: Get Live Execution Traces 9
3.3 Challenge 3: Visualization Speed versus Program Execution Speed 9

4 Live Cerebro Visualization **11**
4.1 Data Collection 11
4.2 Visualization 12
4.3 Communication 17

5 Evaluation **19**
5.1 Communication Method 19
5.1.1 Experiment Setup 19
5.1.2 Result 20
5.2 Further Discussion 22
5.3 Highlight Method 25
5.3.1 Experiment Setup 25
5.3.2 Result 25

6 Conclusions and Future Work **28**
6.1 Conclusions 28
6.2 Future Work 29
6.3 User Study 29
6.4 Visualization 29

Bibliography **31**

LIST OF FIGURES

Page

4.1 System Overview 12
4.2 Layout 14
4.3 Highlight 16

5.1 Time Gap Difference (10,000 lines) 21
5.2 Total Running Time Difference (10,000 lines) 22
5.3 Total Running Time Difference in Profiler (1,000,000 lines) 24
5.4 How much time for each option to run different lines 24
5.5 Total Time Delay 26
5.6 Total Running Time 26

LIST OF TABLES

Page

5.1 Total Time Gap (10,000 lines) 21
5.2 Total Running Time (10,000 lines) 21
5.3 Total Running Time Difference in Profiler (1,000,000 lines) 23
5.4 How much time for each option to run different lines 23
5.5 Fade time gap vs Non-Fade time gap 25
5.6 Fade time gap vs Non-Fade total time 25

ACKNOWLEDGMENTS

First, I would like to thank Jim for every meeting we had. Though he is busy he always finds time to meet with me and give me insightful ideas. Also I want to thank Jim for inviting me to be a part of SpiderLab and show me how interesting doing research is. I wouldn't choose software visualization if he didn't show his amazing and cool project in class. I also want to thank Vijay for the support and help while doing the research. Even he has already graduated and be busy at work, he always helps and meets with me to give me his insightful thoughts.

Finally, I want to thank my parents, friend for the support they gave me. I also want to thank Michelle and Speedy for always be with me when I'm stressful. Thank you all for the love and support and I will always keep that in mind in the rest of my life. I hope after seeing my work, you will all be proud of me. Maybe except for Speedy since he is a cute little furry cat.

ABSTRACT OF THE THESIS

Real-time Software Execution Visualization: Design and Implementation

By

Zizhao Han

Master of Science in Software Engineering

University of California, Irvine, 2021

Professor James Jones, Chair

Software is invisible. In order to understand how the software works, Software Engineers invent many ways for software representation, such as Software Architecture Diagrams and Class Diagrams. These methods help us understand the software from a high-level perspective. However it is hard for the developers to relate the run-time behavior with the source code. With Object-Orientated Programming, the software can be decomposed into several modules. Every feature needs different modules to cooperate. It is even harder for developers to match the feature and the underlying source code. This work presents a visualization that includes a Seesoft view of the overall source code and a dynamic live execution view of the run-time software. In order to demonstrate the execution as live as possible, we used different techniques and test them with two programs (JPacman and JEdit) to validate the the design decisions we made to achieve the best performance. Furthermore, the results demonstrate our visualization is capable of visualizing program execution in real time with lower delay time than without my improvements.

Chapter 1

Introduction

Since the emergence of Software, understanding how software works has been a difficult challenge for all software developers and users. Software is invisible and unvisualizable [11]. In order to help us understand software, software engineers created many different representations that focus on different aspects of software. For example, we created the Use Case Diagram to represent the features of software and the Control Flow Graph to help us understand how software works. But we lack the ability of representing the run-time execution of the software. With the increasing size and complexity of software, software development always involves multiple teams and long-term maintenance. Nowadays we don't need to build software from scratch. We have open source software and tools to help us build our own software. But this requires us to understand the existing software and then we can modify it, especially the code itself. Understand the mapping between code and features can help us not only in software maintenance but also in software debugging.

However, modern software always consists of multiple modules or components. Each feature of a software always involves different components working together. From the code's perspective, Object-Orientated-Programming is becoming more and more popular. Each

feature requires function calls from one object to another. Besides, multi-threaded programs are difficult for developers to debug and understand. This paper represents LiveCerebro, a visualization that can represent all code in the project and highlight statements that are being executed. This paper explains how did we build the system and verified some design decisions we made to achieve the best performance.

1.1 Summary of Contributions

In this paper, we represent LiveCerebro — a software visualization that (1) reads all files from the root directory of the given project, (2) creates the Seesoft view of all files with the ability of addressing each statement, (3) gets the execution information of the given program from all kinds of program instrumenters (in this paper we use Blinky), and (4) visualizes the program execution by highlighting the statements that are being executed.

Our visualization provides these key advantages:

- The visualization contains all the information. The Seesoft view enables the user to have an overall view of the entire code itself. We can see the entire code from a single screen.
- The visualization is as live as possible. The Seesoft view is constantly changing as the program running. After the user interacting with the program, we can see which code are being executed immediately from the visualization.

1.2 Thesis Structure

This thesis is structured as following. In Chapter 2, we introduce some related works and technologies that we are using. In Chapter 3, we present the motivation of this project and the challenges that we are facing. In Chapter 4, we present LiveCerebro — our visualization to overcome the challenges, and Blinky — the back-end instrumenter we later use to help evaluate our visualization. In Chapter 5, we evaluate our visualization design decisions by choosing different implementation options. In Chapter 6, we conclude the work and discuss the future directions.

Chapter 2

Related Works

Multiple research efforts have been conducted that attempt to visualize run-time behaviors of software. Researchers have been trying to visualize such information to help developers better understand the software. Ball and Eick [5] categorized software visualization into four kinds: line representation, pixel representation, file summary representation and hierarchical representation. Bassil and Keller [6] researched on existing software visualization tools and found a gap between what developers need and what they have.

2.1 Visualizing Run-time Behavior, Post hoc

Palepu and Jones [21] visualized the execution traces of software by a Force-directed Graph Layout. Their work provides the basis upon which my research is built. In their work, they organized the software by different levels, such as instruction level and method level. The visualization used nodes to represent each instruction or method. The visualization can show the connections between instructions or methods. After running multiple test cases, the visualization will recognize a certain pattern of program execution. Their visualization

relies on running test suites to gather execution traces prior to visualization. In contrast, my visualization gathers and visualizes the traces as the program running so that we can see the relation between software features and instruction executions.

Cornelissen et al. [4] proposed Extravis. This visualization uses a circular view to represent the software structure and the connections between different components and a massive sequence view to show the consecutive calls in a chronological order. All visualizations above show the inner connections between different components in a software. However the visualization is not real-time and lacks the information about the mapping between software's behavior and its source code execution. Karran et al. proposed Synctrace [14], which focused on multi-thread programming and how the order of user interaction will effect the program. Deng et al. proposed Constellation visualization [8]. Much like our visualization, it is also line-level based. However it focused on the context of program execution while ours focused on the connection between software features and the code. Dietrich et al. [9] created a visualization focused on clusters and dependencies in Java programs. Their visualization helps developers to understand how different modules are related in the program. Kuhn et al. [16] use cartography to visualize clusters with different "topics" in the software. Krinke [15] developed a visualization that visualizes the program dependency graph by visualizing the program slice. Alimadadi et al. [3] created a mapping between low level events in JavaScript and high level behaviors. Ezzati-Jivan and Dagenais [10] also visualized trace data but based on the timeline and a hierarchical model. Zhang and Gupta [31] developed whole-execution traces visualization to visualize all kinds of traces in program execution such as Control Flow and Address. Jerding et al. [13] presents a visualization that can only visualize one execution at a time post hoc while ours visualize one execution real time.

2.2 Visualizing Real-time Software Behavior

De Pauw et al. [29] created Jinsight. Jinsight used a Histogram View to show the inner activity such as thread interactions, garbage collectors and deadlocks during the program running. It focused on the Object level. Reiss [24] proposed Jive — a visualization that consumes real-time Fava program traces. Jive focused on visualizing high-level information. It focused on multi-threaded program and visualizes how different threads enter different classes and packages. Since this visualization is programmed in Java, it used a monitor thread to monitor the program execution and get the traces. Later Reiss and Renieris modified Jive and created Jove [25]. Jove show more detailed information about how much time thread spent in different classes or packages. The result of this visualization is accumulated through time. On the contrary, my work focuses on which instructions are being executed at a certain time. Lo and Maoz [18] visualized trace data by a live sequence chart in specification mining.

2.3 Visualizing Non-run-time Source Code

Telea and Auber [26] created Code Flow to visualize the evolution of source code along the development and version changes. It focused on how code changes between different versions of software. Servant and Jones [23] also created an interactive visualization that allows users to search down to line level and show the historical change of the source code. Hanjalić [12], Voinea et al. [28] and Liu et al. [17] also focused their work on the evolution of source code. Chen et al. [7] and Marcus et al. [19] visualized traceability links between different artifact of the software. These visualizations help users to understand how different parts of the source code work together. Moreno et al. [20] created Jeliot to visualize data and control flows to help students learn procedural and object oriented programming. Winter et al. [30] created a Sextant, which visualizes software metrics for Java source code. These metrics can

help developers in many ways. In our work we build our visualization based on Seesoft View [22]. Seesoft View is a zoomed-away view of all source code in the project.

Chapter 3

Motivation and Challenge

Brooks [11] mentioned that:

> The software entity is constantly subject to pressures for change. ... In part, this is so because the software of a system embodies its function, and the function is the part that most feels the pressures of change. In part it is because software can be changed more easily — it is pure thought-stuff, infinitely malleable.

For modern software, the cost of this changeability is becoming more and more expensive, due to its large codebase and complex functions. In order to modify or maintain a software, developers have to fully understand its code. However, nowadays software using different third-party libraries or other open-source software as dependencies makes this process difficult and time-consuming. Palepu and Jones [21] created the Cerebro visualization to help developers to understand software's run-time behavior. The results show that software's execution contains certain execution patterns. In this paper, we want to show how these patterns relate to the code, and we want to make the visualization update in real time. To achieve this goal, we need to address three main challenges:

3.1 Challenge 1: Showing all information

The information we need to represent in this visualization is all of the source code of a given project. This visualization should be line-level based. Eick et al. [22] mentioned that because of the volume of code each project contains, it is difficult to represent line-orientated statistics in a single screen. Besides, the visualization itself is constantly changing due to the program execution. We need to find a good layout to show such information, and the layout itself should be easy to modified on the line-level.

3.2 Challenge 2: Get Live Execution Traces

Tikir et al. [27] provides code coverage that can dynamically insert or remove the instrumentation. JaCoCo [2] is a code-coverage library for Java. Both these code-coverage tools provide program-execution data *after* the running of test suites completes. Much of the prior work provides redundant information for the purpose of our visualization. In this work we need to capture and represent the execution data *as* the program is running. The execution traces are large and contain massive amounts of information. Our visualization only needs to know which line of which file is currently being executed. Hence, we need a code-coverage tool that can provide user-selected information in real time.

3.3 Challenge 3: Visualization Speed versus Program Execution Speed

In the code-coverage tool, the instrumentation is usually inserted after the actual instruction, which means we will always get the traces after the instruction being executed. Besides, the

communication between our visualization and the code coverage tool takes time. Moreover, the visualization needs to render the layout, which usually cost more time. The main goal for this visualization is to show the run-time behavior as live as possible without slowing the program. We need to minimize the communication time and the rendering time.

Chapter 4

Live Cerebro Visualization

In order to address three challenges we are facing, our visualization needs to

- Read all files from the given project and create a line orientated layout.
- Get real-time execution traces from the instrumenter.
- Update the layout as fast as possible to keep up with the program.

Figure 4.1 represents the basic logic of how our visualization works.

4.1 Data Collection

In this paper, we modified the dynamic instrumenter Blinky [21] to obtain execution traces in real time. We configured Blinky to collect line-level traces — full sequences of all source-code line-level execution events throughout the entire execution. The main information we need to gather is line number and file name. Blinky allows users to create their own profiler to collect specific data that they need. Program-execution traces are large. In order to minimize the

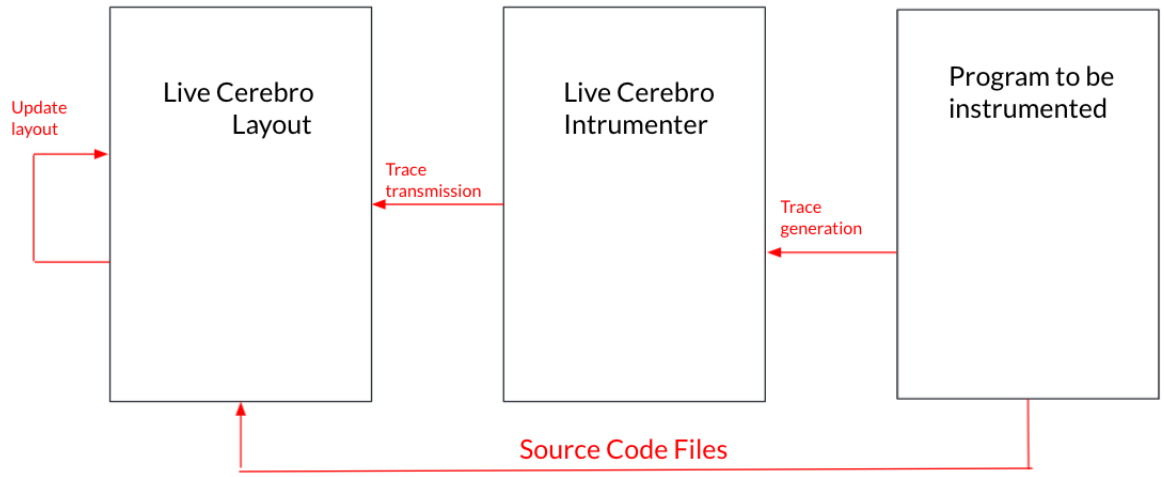

Figure 4.1: System Overview

traces we collect to save the communication time between Blinky and our visualization, we used an option in Blinky that only records line-level events. In Blinky, after each instruction gets executed, there will be a "Trace Event" that is generated. Since we are only collecting line-level traces, we can construct a line object from each trace event. In each line object, it contains its line number and file name.

Our visualization needs more data other than the traces. We need to show the entire source code. Our readFiles function needs to read all Java files recursively from the root directory of a project. Reading files using JavaScript is not fast. Besides, we have to draw each file's minimap, which is also expensive in time. To do this as efficiently as possible, we read the files asynchronously and draw the minimap after reading each file. After we draw the minimap, we also store each file's data, line by line, for future usage.

4.2 Visualization

Live Cerebro is targeted to Java applications. Thus, an implementation of the visualization written in Java was the first idea came into our minds. However, using Java libraries

for rendering is slower than other programming languages. Our visualization needs to be constantly changing, so the rendering speed is important. Additionally, we want our visualization to run not only on different platforms, but also on a website or in a web browser to ease installation. We chose the Electron platform [1]. The Electron platform supports developers to build cross-platform and web-based software using JavaScript.

The visual layout that we are using is the Seesoft View [22]. The Seesoft view can represent line-orientated statistics to give us an overview of the source code. Besides, VSCode and other editors provide Minimap on the side of the editor to provide an overview of the current file. We choose to combine all the minimaps of all files in a project to form a Seesoft-like layout. Moreover, our visualization needs to highlight each line of code. A two-layer layout can help us do this efficiently. The bottom layer (code layer) shows the actual code and the upper layer (decoration layer) draws white rectangles to perform the highlight function. Figure 4.2 is an example layout for the JPacman program.

In order to give the user an overview of all source code and help developers to understand the whole program by watching the highlighted part of the code, it is important to scale the size of the minimap so that the whole layout can fit in one screen. However, modern software is often large in terms of the size of the code base. A software project often contains more than a hundred source code files, and each file could have hundreds of lines of code. For some project, it is impossible to scale all files in a row because the width of each file could be smaller than a pixel. So we decide to create four containers which split the whole screen vertically into four parts. Each part will contain one quarter of the files. For each container, there will be a scroll bar for users to scroll down to see the rest of the lines. The Seesoft view's performance is limited by the screen resolution, screen size, and internet speed, if we demonstrate the tool via Zoom. Ideally, users should use one screen to show our visualization and another one to show the actual program to achieve the best use experience.

The Seesoft View itself cannot provide enough information due to the size of it. However, in

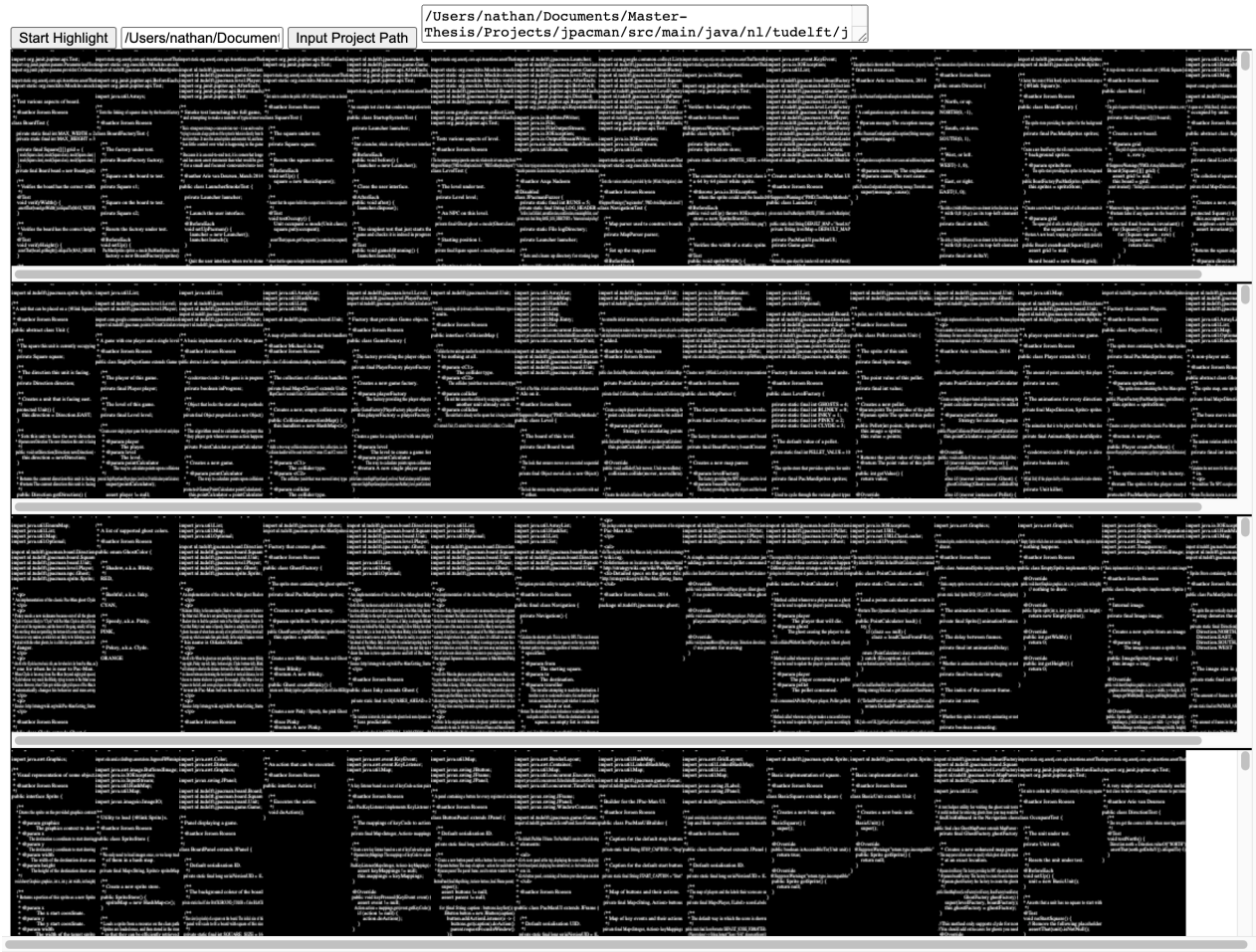

Figure 4.2: Layout

order to help users to see the connection between the source-code execution and the function of the software, users need to know which line of which file is being highlighted. To address this problem, we create a text area above the Seesoft view to show this information. The text area will respond to mouse-over events, which shows which line number and file name the mouse cursor is over. In this case the user can investigate areas of the program that were highlighted during program execution.

In order to show the program execution in our Seesoft layout, we draw small white rectangles on top of the actual code currently being executed to represent that this line of code is being executed. This highlight function could be done in two ways:

Non-Fading Highlight the instruction and clear the highlighting after a few milliseconds.

Fading Highlight the instruction with low transparency and increasing the transparency along the time until it's not highlighted.

The second option with fading could give users a better perception of the recently executed code by giving some time for the fade-out to occur to allow users to see these highlights. The executed instructions will become darker and darker over the time period of the fade-out. However, this fading effect needs us to draw the rectangle with different transparency levels multiple times. The rendering time of our visualization is important because we want our visualization to keep up with the program execution. Besides, normally some instructions will get executed multiple times in a short time slot. In this case the fading version will lose its purpose because some instructions will be highlighted all the time. The evaluation about these two highlight method will be in Chapter 5. Figure 4.3 is an example of highlighting in our visualization.

Figure 4.3: Highlight

4.3 Communication

After collecting the traces, we need to send that information to our visualization in real time. Because our visualization and the instrumenter are two separate processes, we decided to use sockets to communicate across the processes. The main challenge we are facing is that the speed of trace generation is much faster than the speed of our visualization. We identified three options to handle the much faster trace generation from the visualization:

- Option 1: Use a buffer in our visualization to store all the traces.
- Option 2: Slowing down the actual program to wait for the visualization.
- Option 3: Discard some traces and only visualize the latest ones.

The first option needs a buffer to store all the information and visualize them. However the size of the traces could be larger than the memory space. The buffer will eventually explode in size due to the size of traces that we are receiving. Besides, this would cause a time gap between the visualization and program execution, which is contrary to the goal of real-time visualization. It is very possible that we are visualizing instructions that were executed seconds or minutes ago.

The second option is that we slow down the actual program to make sure the program and visualization are executed in the same speed. One problem with this option is that it requires extra information to coordinate the speed. Moreover, slowing down the actual program is not user friendly: the user experience with a slowed-down program would be quite compromised.

The third option is discarding some traces and only visualize the latest one. In this way we will lose some information but increasing the speed of the program. The third option is best-effort provided, much like UDP servers. For UDP servers, there would not be any congestion control or packet resends, and it is also best-effort. If the socket got full then some packets

will be dropped. We created a UDP server in our profiler. This server will be initialized when the tracing starts. When each line gets executed, it will send JSON-format information containing the line number and file name to our visualization. In our visualization there is not any buffer to store the information. It will visualize the incoming information based on the receiving order. Since the UDP protocol does not guarantee packets arriving in order, our visualization cannot represent the order of program execution. But it can show which part of the code are being executed and build a connection between software features. The evaluation of these three options is in Chapter 5.

Chapter 5

Evaluation

5.1 Communication Method

5.1.1 Experiment Setup

We use JPacman for our experiment. JPacman is a Java game program that involves several user interactions and loops in the code. In this experiment, we measure two things:

- Total running time after executing 10000 lines of code.
- Average time delay.

We measure total time by setting a timer between the first line to get executed and the 10,000th line to get executed. In the profiler when we send the socket, we include a timestamp to represent the time of execution. In our visualization, we also record the timestamp after this line got highlighted. The average time delay is the average time difference between the execution time and the highlighting time.

Recall from Chapter 4 that Communication Option 1 requires us to create a buffer in the visualization to store all the traces and visualize them in order. In the experiment we create a TCP server in the profiler and send traces through the TCP socket to our visualization. In our visualization, we create a buffer to store all the traces. To avoid buffer explosion, we clear the buffer once the size of the buffer gets too large. The highlight method polls traces from the buffer and visualizes them.

Recall from Chapter 4 that Communication Option 2 requires us to slow down the instrumented program. We pause the program with a very small delay every time it sends out a trace. We attempted multiple different pause times. However for JPacman, setting a time out caused some bugs, and sometimes the program would crash. Besides, even with smaller timeouts, the program would run much slower because we are setting a timeout after every instruction. The user experience was awful. For JPacman, it takes more than 10 minutes for the GUI to show up with 10ms time delay. And based on the results of other two options, 10ms delay is not good enough for our visualization to keep up with the program. So this option was not deemed viable.

Recall from Chapter 4 that Communication Option 3 requires to discard some traces and only visualize the latest ones. We created a UDP server in the profiler that sends Datagram packets to our visualization. Since the UDP protocol itself does not guarantee every packet will arrive, we are discarding some packets. In our visualization we just visualize the incoming packets in the order that they are received, as fast as the visualization can keep up.

5.1.2 Result

We ran JPacman ten times for each option with the same user input. Table 5.1 and Table 5.2 show the data. Figure 5.1 and Figure 5.2 show the results.

	1	2	3	4	5	6	7	8	9	10	Average
UDP time gap	3563315	4287266	5622379	4442605	4023799	3568391	3455789	3888816	3490100	4429564	4076978.5
TCP time gap	1682722	4691617	1518839	2817145	3461722	3728027	1766091	1980672	1807256	2435793	2588988.4

Table 5.1: Total Time Gap (10,000 lines)

	1	2	3	4	5	6	7	8	9	10	Average
UDP total time	4732	5220	5807	5650	4868	4824	4592	4853	4706	5193	5065.4
TCP total time	2044	2858	1935	1985	2350	2438	1873	2110	1796	2237	2162.6

Table 5.2: Total Running Time (10,000 lines)

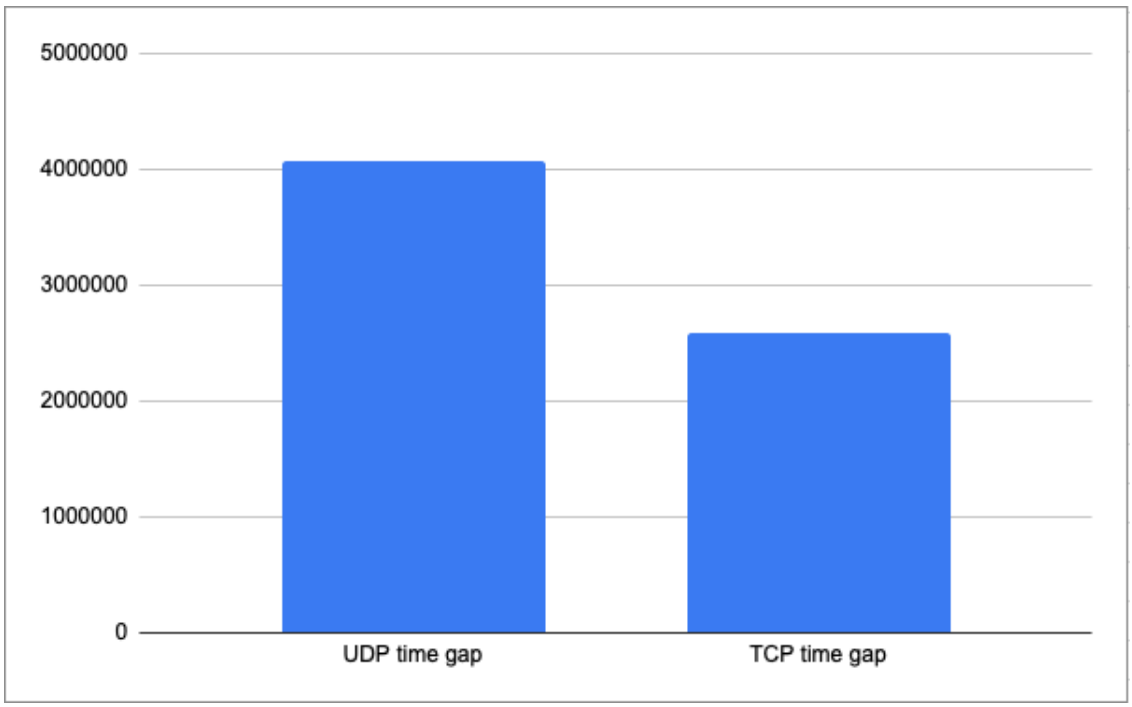

Figure 5.1: Time Gap Difference (10,000 lines)

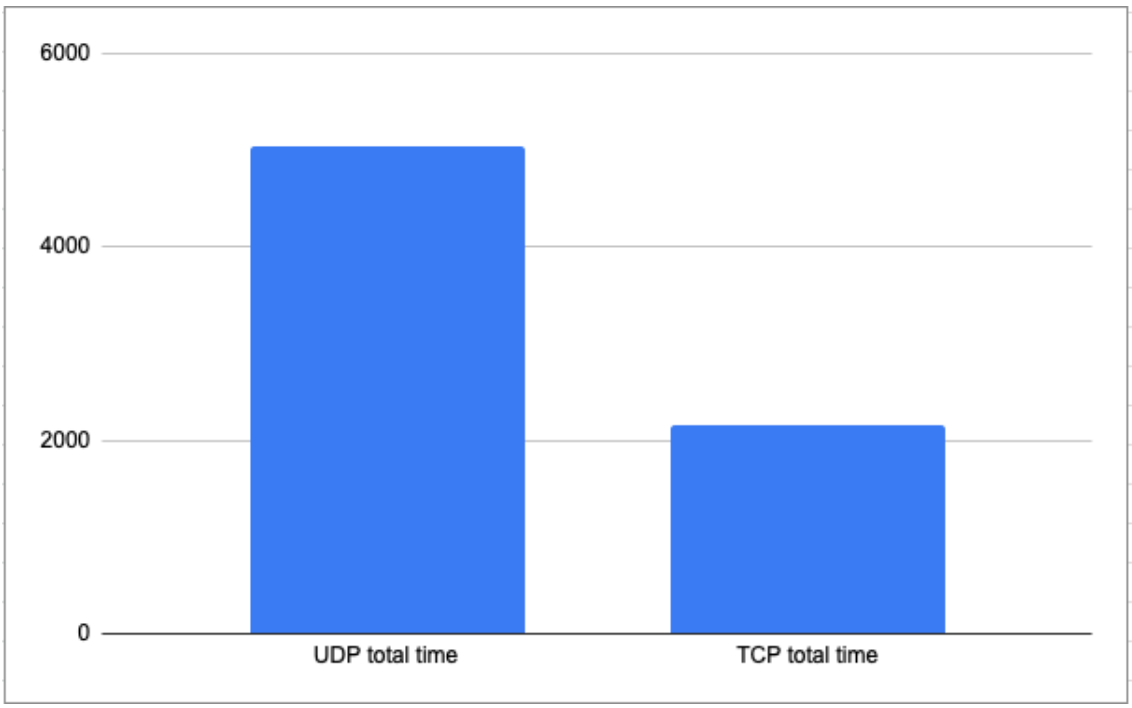

Figure 5.2: Total Running Time Difference (10,000 lines)

These results show that Option 1 has a lower delay and faster speed than Option 3. For time delay, because TCP builds a connection between the visualization and the profiler, the transmission rate will be faster than the UDP protocol, since each UDP packet needs to route through the internet to find the visualization's IP address. Besides, the time of constructing the Datagram packet also spends more time than write stream in TCP. For the total time of running 10000 lines of code, Option 1 spends less time than Option 3. This is reasonable because in UDP we are discarding some traces, and we only count how many instructions we are visualizing. It takes longer time for UDP version to reach 10000 lines.

5.2 Further Discussion

However, Option 1 makes JPacman run slower than Option 3. We first observed this by watching the Pacman character moving. In Option 1 the Pacman blinks slower than Option 3. We conduct another experiment to verify our finding. We move the line counter to the

	1	2	3	4	5	6	7	8	9	10	Average
UDP total time	9045	9118	8434	8464	8666	11521	10472	10018	9873	9783	9539.4
TCP total time	50559	49279	55242	60586	60708	57984	57981	61257	53465	55137	56339.8

Table 5.3: Total Running Time Difference in Profiler (1,000,000 lines)

Lines	100	1000	10,000	100,000	1,000,000	5,000,000
UDP total time	35	47	1389	4558	10964	48657
TCP total time	68	86	1523	8775	50425	802648

Table 5.4: How much time for each option to run different lines

profiler. We record the total time for instrumenting 1,000,000 lines of code. Table 5.3 shows the data. Figure 5.3 shows the result.

The results shows with the TCP socket, JPacman runs slower than using UDP socket. We believe the reason is the TCP connection gets slower when more traces are sent to the socket. TCP requires congestion control and packet retransmission. When there are too many traces sent through the socket, the chance of packet loss and errors also increases. So the connection gets slower. The profiler will wait until it can send more traces.

Table 5.4 and Figure 5.4 show how much time it takes for running different lines. The result confirms our theory. Obviously Option 1 takes longer time than Option 3, which means Option 1 will make JPacman run more slowly.

Since Option 1 has lower delay but will make the actual program run slower. We believe Option 1 is not suitable for program like JPacman. The JPacman program is constantly running without any idle time. The traces it generates are large and involves many loops in the code. If we visualize every instruction that gets executed, some part in the Seesoft view will always be highlighted. Option 3 handles this by discarding some traces, which not only makes JPacman run faster but also shows a better view in the visualization. We can actually see instructions become bright and then dark.

However for other projects that are idling most of the time waiting for user input, such as

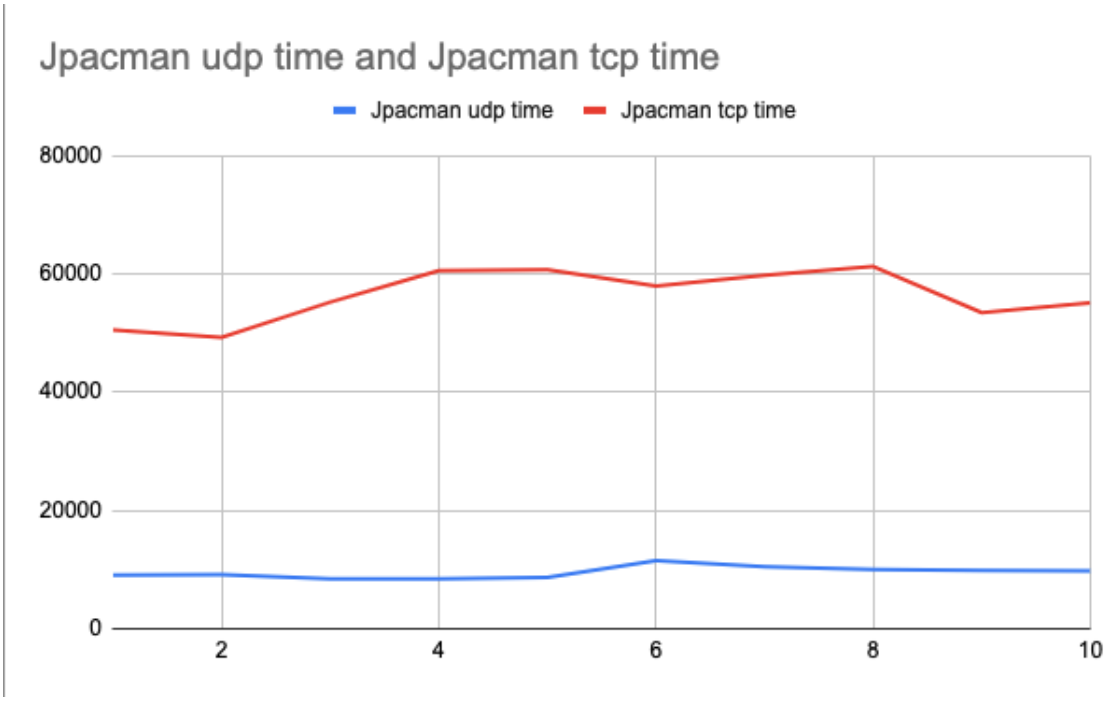

Figure 5.3: Total Running Time Difference in Profiler (1,000,000 lines)

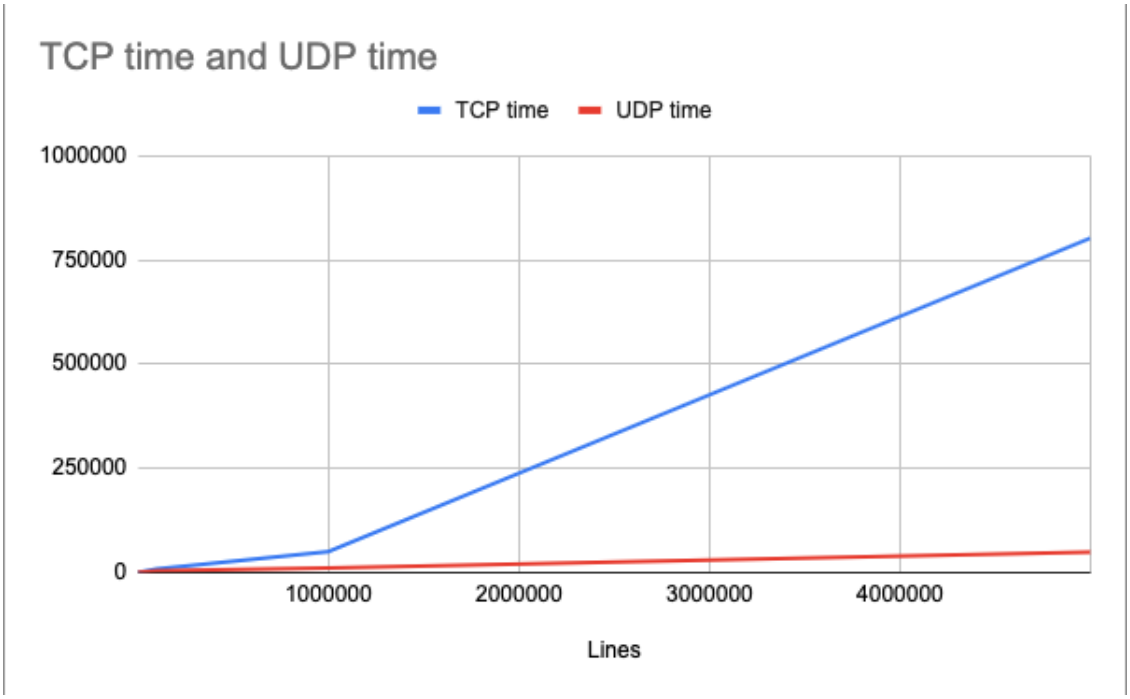

Figure 5.4: How much time for each option to run different lines

	1	2	3	4	5	6	7	8	9	10	Average
Fade time gap	1417826176	1418699309	1420684627	1415693487	1413678546	1421986745	1436549072	1409760450	1410357069	1426980450	1419221593
Regular time gap	4077202.4	3968725.1	3846925.6	4529871.5	4469310.8	3583164.5	3397438.8	4976385.8	3878918.8	4680604.5	4140854.78

Table 5.5: Fade time gap vs Non-Fade time gap

	1	2	3	4	5	6	7	8	9	10	Average
Fade total time	350821	335431	394685	305798	315294	369875	334962	346810	316482	365789	343594.7
Regular total time	5044.5	5233.8	5178.4	4912.9	5967.8	5576.2	4079.5	5897.7	4578.5	4439.2	5090.85

Table 5.6: Fade time gap vs Non-Fade total time

JEdit, we may consider Option 1. JEdit idles most of the time until a user clicks some buttons. This kind of program generates less trace data, which will not cause socket congestion, and as such is suitable for Option 1. Since Option 1 has lower delay time than Option 3, it is more live.

5.3 Highlight Method

5.3.1 Experiment Setup

The fading effect was implemented by recursively calling the highlighting function with different levels of transparency. In our experiment for each line of code, it calls the highlighting function six times. In this study, we are still measuring total running time and average time gap for 10,000 lines. Since the highlight method takes more time, it is more likely to cause TCP socket congestion. So we are using the UDP version in our experiment.

5.3.2 Result

Table 5.5 shows the total time gap for 10 experiments we conduct. Table 5.6 shows the total running time for 10 experiments we conduct.

Figure 5.5 shows the result of total time delay after running 10,000 lines of code, and Figure

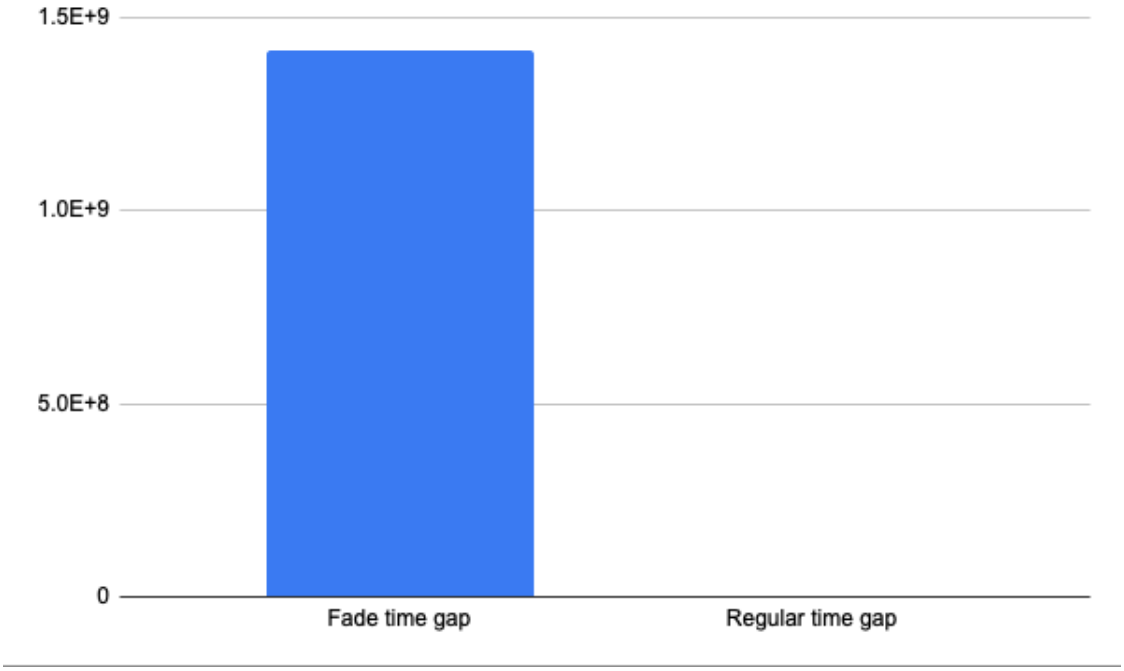

Figure 5.5: Total Time Delay

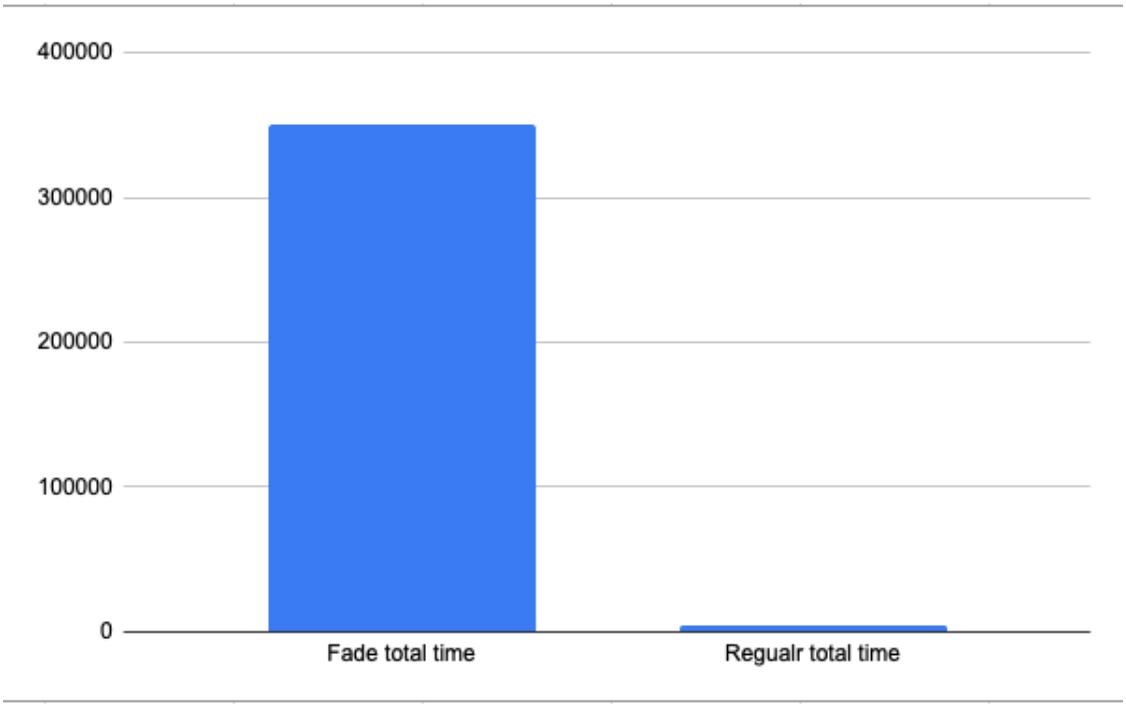

Figure 5.6: Total Running Time

5.6 shows the total time after running 10,000 lines of code. In the experiment, we found out that sometimes the highlight function would stop because there are too many recursive methods that have already been called. The visualization will continue visualizing the remaining recursive functions before it visualizes newly incoming traces. This is the reason why the total time delay and total time for the fading effect highlight method take much longer time than the regular one. Part of the reason is that JPacman generates traces constantly so the generation speed is far faster than our visualization's consuming speed with the fade effect highlight function. It would achieve better performance if we use other programs such as JEdit, which would have fewer, smaller bursts of trace data rather than JPacman's constant stream of trace data.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this paper, we presented Live Cerebro, a visualization that visualizes the code execution inside the running program and is intended to help developers to understand the software. Our visualization works with program instrumenters. It reads all files from the root directory of a project and draws the Seesoft View of all files to give the user an overview of all the source code. Then it receives program-execution traces from the instrumenter in real time and visualizes the execution by highlighting the instruction on the Seesoft view. The user can move their mouse onto any instructions to see the line number and file name. Since this happens in real time, users can see a clear mapping between the software's features and its code.

We also do some experiments to validate some design decisions that we made. In the profiler, we use servers with different protocols to send out execution traces to our visualization. The UDP server is faster than the TCP server, which can help our visualization to keep up with the program execution. The information loss is barely noticeable due to the speed of program

execution and the speed of our visualization. We compared how different highlighting functions effect the visualization speed. The results show that without fading the visualization can run faster, which allows our visualization to keep up with the program execution.

6.2 Future Work

6.3 User Study

In the future, we want to conduct user studies to evaluate how much time and effort we save with Live Cerebro to understand a program. In our evaluation we only evaluated the performance of our visualization but not how useful it is. We want to know if the developer actually understands the software more easily using our visualization. This user study can be conducted by dividing users into two groups, one using our visualization and one without. Then we could ask the users to perform some task, such as identifying which code implements some features of the software. We could measure the time that it takes to give an answer from both groups, as well as their accuracy.

6.4 Visualization

Our visualization still has some limitations. The profiler sends out a Datagram packet when an instruction get executed. This will slow the program down since sending a packet takes too much time. Slowing the actual program will cause a bad user experience. It is reasonable to try other inter-process communication methods to reduce the communication delay.

Besides, Live Cerebro doesn't support multi-thread program. Our visualization treats all incoming traces equally regardless of which thread is executing the code. However multi-thread

programs are difficult to understand and debug if we don't know the thread information. Users can better understand multi-thread programs if we use some mechanism to differ different threads such as using different color to highlight the code.

There are also some limitations in our Seesoft View. The canvas we are using is set to a fixed size. If the canvas can adjust its size based on the number of total files and the length of each file, the loading time of our visualization would be reduced. Besides, if there are too many files in a project, the minimap for each file could be shrunken to a smaller size. It would lose more information since the Seesoft view can not represent what the actual source code looks like. We can investigate other visualization approaches to address these issues of scale.

Bibliography

[1] Electron Official Site. `https://www.electronjs.org/`.

[2] Jacoco. https://www.jacoco.org/jacoco/.

[3] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding javascript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 367–377, New York, NY, USA, 2014. Association for Computing Machinery.

[4] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. 15th IEEE International Conference on Program Comprehension (ICPC '07), 2007.

[5] T. Ball and S. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[6] S. Bassil and R. Keller. Software visualization tools: survey and analysis. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 7–17, 2001.

[7] X. Chen, J. Hosking, and J. Grundy. Visualizing traceability links between source code and documentation. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 119–126, 2012.

[8] F. Deng, N. DiGiuseppe, and J. A. Jones. Constellation visualization: Augmenting program dependence with dynamic information. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–8, 2011.

[9] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow. Cluster analysis of java dependency graphs. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, page 91–94, New York, NY, USA, 2008. Association for Computing Machinery.

[10] N. Ezzati-Jivan and M. R. Dagenais. Multiscale navigation in large trace data. In *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–7, 2014.

[11] Frederick P. Brooks. No Silver Bullet, Essence and Accidents of Software Engineering, 1987.

[12] A. Hanjalić. Clonevol: Visualizing software evolution with code clones. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, 2013.

[13] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of the 19th International conference on Software engineering*, ICSE '97, pages 360–370, New York, NY, USA, 1997. ACM.

[14] B. Karran, J. Trümper, and J. Döllner. Synctrace: Visual thread-interplay analysis. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–10, 2013.

[15] J. Krinke. Visualization of program dependence and slices. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 168–177, 2004.

[16] A. Kuhn, D. Erni, P. Loretan, and O. Nierstrasz. Software cartography: thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):191–210, 2010.

[17] C. Liu, X. Ye, and E. Ye. Source code revision history visualization tools: Do they work and what would it take to put them to work? *IEEE Access*, 2:404–426, 2014.

[18] D. Lo and S. Maoz. Mining hierarchical scenario-based specifications. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 359–370, 2009.

[19] A. Marcus, X. Xie, and D. Poshyvanyk. When and how to visualize traceability links? In *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '05, page 56–61, New York, NY, USA, 2005. Association for Computing Machinery.

[20] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, page 373–376, New York, NY, USA, 2004. Association for Computing Machinery.

[21] V. K. Palepu and J. A. Jones. Revealing Runtime Features and Constituent Behaviors within Software. 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT), 2015.

[22] S.C. Eick, J.L. Steffen, E.E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. IEEE Transactions on Software Engineering, vol. 18, no. 11, pp. 957-968, 1992.

[23] F. Servant and J. A. Jones. Chronos: Visualizing slices of source-code history. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, 2013.

[24] Steven P. Reiss. Visualizing Java in Action. SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization, 2003.

[25] Steven P. Reiss and Manos Renieris. JOVE: Java as it Happens. SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization, 2005.

[26] A. Telea and D. Auber. Code flows: Visualizing structural evolution of source code. *Computer Graphics Forum*, 27(3):831–838, 2008.

[27] Tikir, Mustafa M. and Hollingsworth, Jeffrey K. Efficient Instrumentation for Code Coverage Testing. SIGSOFT Softw. Eng. Notes 27, 4 (July 2002), 86–96., 2002.

[28] L. Voinea, A. Telea, and J. J. van Wijk. Cvsscan: Visualization of code evolution. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, page 47–56, New York, NY, USA, 2005. Association for Computing Machinery.

[29] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of Java programs. In Revised Lectures on Software Visualization, International Seminar, pages 151–162, 2002.

[30] V. Winter, C. Reinke, and J. Guerrero. Sextant: A tool to specify and visualize software metrics for java source-code. In *2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 49–55, 2013.

[31] X. Zhang and R. Gupta. Whole execution traces. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 105–116, 2004.