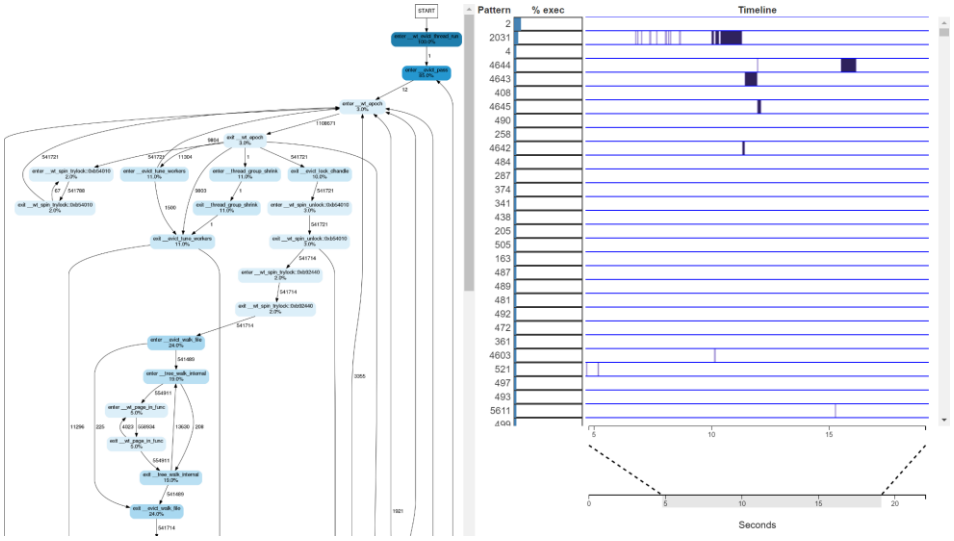
ThreadViewer: Visualizing a Thread's Behavior in a Program Execution

Augustine Wong



Abstract—This paper presents ThreadViewer, a software performance debugging visualization tool for analyzing the behavior of a single thread in a multi-threaded execution. ThreadViewer borrows heavily from our prior work in creating a suite of tools for software performance debugging. Prior to ThreadViewer, we created two visualization tools designed to help engineers analyze execution traces: FlowViz and TimeSquared. The intention was for engineers to use both of these tools together when doing performance debugging. Therefore, the goal of ThreadViewer is to incorporate the visual encoding styles of both FlowViz and TimeSquared so that engineers do not have to use two tools just to debug the same execution trace. The biggest challenge of this project was figuring out a method of reducing the cardinality of execution traces so that engineers can smoothly transition between the FlowViz and TimeSquared visualizations. Although ThreadViewer is still in development, it has proved useful in helping us critique the way we are reducing the cardinality of our execution traces.

Introduction

Software performance debugging is a challenge problem often encountered by computer engineers. A recent survey of 308 engineers revealed that 92% of them faced performance issues in the preceding year, with each issue taking on average 80 hours to solve. However, the survey also found that some engineers encountered issues which took longer than a month to solve [1].

To assist engineers with performance debugging, my research group developed a suite of tools called DINAMITE: a tool kit for Dynamic Instrumentation and Analysis for Massive Trace Exploration. DINAMITE includes tools for collecting and visualizing execution traces. Two of the visualization tools available with DINAMITE are FlowViz and TimeSquared [9]. FlowViz and TimeSquared are designed to visualize execution traces that capture the function calls made by a multi-threaded execution. FlowViz provides an overview of each thread's behaviour while TimeSquared provides a detailed view of the threads' executions.

ThreadViewer is a software performance visualization tool born out of the realization that both FlowViz and TimeSquared should be used together to debug software performance. The motivation behind creating ThreadViewer is to integrate FlowViz and TimeSquared together into one tool so that engineers can easily switch between them when doing performance debugging. However, because FlowViz and TimeSquared work on the extreme opposite ends of the data scale, ThreadViewer needs to provide an intermediate visualization step between the two visualizations; this step is essentially the "zoom" part of the mantra of "Overview First, Zoom, Details on Demand". This project focused on the development of the intermediate visualization step between FlowViz and TimeSquared. The biggest challenge was coming up with a method of reducing the cardinality of our execution traces so that they can viewed at a scale in between the overview and detail level.

Ideally, ThreadViewer would be able to assist engineers in analysing all threads in a program execution at the same time. However, as a starting point for development, I chose to design ThreadViewer so that it can only be used to analyse one thread.

The remainder of this paper is organized as follows:

Section 1 provides details about DINAMITE and discusses why without ThreadViewer, engineers cannot use FlowViz and TimeSquared together to do performance debugging. Section 2 discusses the dataset used to develop ThreadViewer and the technique I employed to reduce the dataset's cardinality. Section 3 lists the domain tasks which ThreadViewer supports. Section 4 describes the design of ThreadViewer. Section 5 outlines how ThreadViewer was implemented. Section 6 evaluates the effectiveness of ThreadViewer. Section 7 discusses the insights that we gained from using ThreadViewer. Section 8 lists the future work. Section 9 compares ThreadViewer to related work. Section 10 ends with a summary of my work.

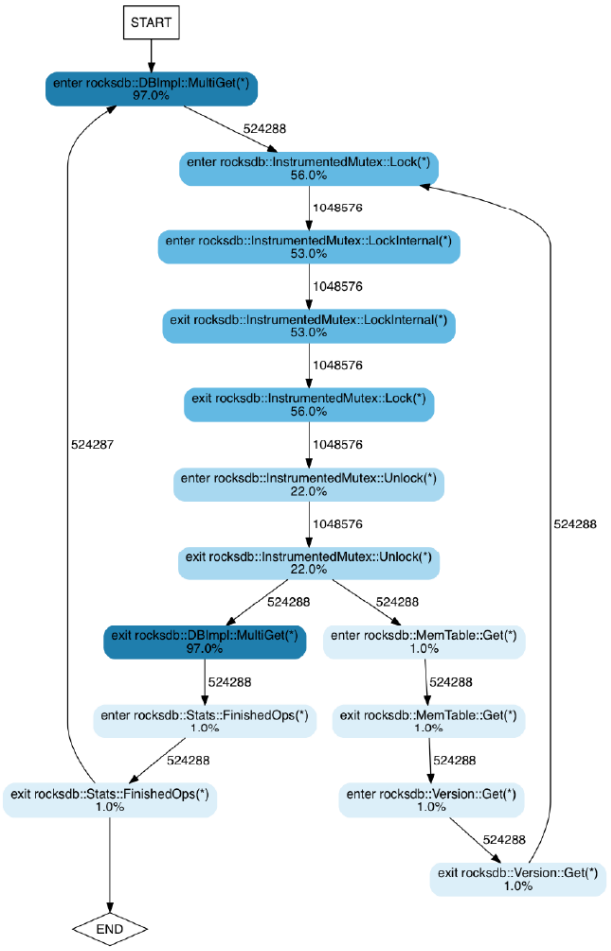


Figure 1. FlowViz execution flow diagram

1. PRIOR WORK

This section describes the data abstraction of DINAMITE traces as well as the FlowViz and TimeSquared visualizations. This section also details the problems that my research group encounter when trying to use FlowViz and TimeSquared together to analyze program executions.

1.1 Data Abstraction of DINAMITE Traces

When configured to trace functions, DINAMITE generates two records each time a thread in a multi-threaded program makes a function call. One record contains information about when the thread entered into the function call while the other record contains information about when the thread exited the function call. The attributes contained in each record are shown in Table 1 below:

Attribute	Attribute Type	Description
Function	Categorical	The name of the function being called
Direction	Categorical	Indicates if the record represents a function entry or exit
Thread ID	Categorical	Identifies the thread which either entered or exited the function call
Time	Ordered, quantitative	The time when the thread either entered or exited the function call

Table 1. The attributes of a record generated by DINAMITE

1.2 FlowViz

For each thread in a multi-threaded program, FlowViz generates an execution flow diagram capturing that thread's behavior. The execution flow diagram is a state transition graph, where the states represent function entries and exits. Figure 1 shows an execution flow diagram of a thread in RocksDB, a key-value store used by Facebook.

Each function call is represented by two nodes: one capturing the state of the thread entering the function call and the other node capturing the state of the thread exiting the function call. The nodes' color saturation level encodes the percentage of the thread's execution time that the corresponding function occupied. For instance, Figure 1 shows that the pair of nodes which represent the call to the function rocksdb::DBImpl::MultiGet(*) have full color saturation, meaning that the thread spent the majority of its execution time in rocksdb::DBImpl::MultiGet(*).

The directed edges between the nodes show transitions between the function call entries and exits that the thread made throughout its execution time. The edges are weighted according to how many times the transitions occurred.

My research group uses FlowViz diagrams to gain an overall understanding of the behavior and performance of a program execution. We have successfully used FlowViz diagrams to help engineers at Facebook discover a scalability bottleneck on RocksDB [9]. However, because we cannot use FlowViz diagrams to explore an execution over time, DINAMITE also provides the TimeSquared visualization tool.

1.3 TimeSquared

TimeSquared is a visualization tool which displays the threads' callstacks on a horizontal timeline. Figure 2 shows an example of a timeline of a program execution created by TimeSquared. Each thread's callstack is shown on a separate row in the timeline. TimeSquared uses line marks to represent function calls. The color hue channel encodes the different function attribute levels. The horizontal spatial positioning channel encodes the start time of the function calls while the horizontal length channel encodes the durations of the function calls. Compared to the FlowViz execution diagrams, TimeSquared provides a detailed view of an execution. For example, engineers using TimeSquared can see precisely when threads have entered or exited function calls.

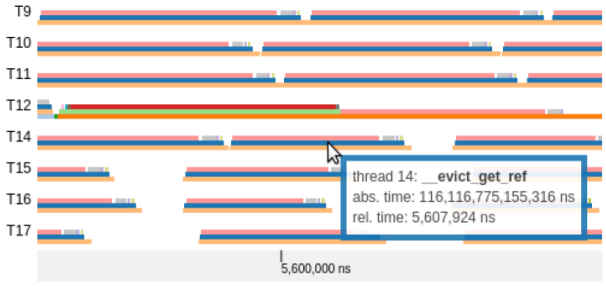


Figure 2. Timeline view of an execution trace provided by TimeSquared

Original trace:

load, displayDir, selectFile, renderChar, renderChar, renderChar, renderChar, insertChar, renderChar, insertChar, renderChar, select, copy, paste, renderChar, renderChar, select, copy, paste, renderChar, renderChar, renderChar, saveFile, displayDir, selectFile, load, displayDir, selectFile, renderChar, renderChar, renderChar, renderChar, renderChar, renderChar, renderChar, renderChar, renderChar

Mappings:

load=a, displayDir=b, selectFile=c, renderChar=d, insertChar=e, select=f, copy=g, paste=h, saveFile=i

Recoded trace:

Figure 3. Example of string representation of an execution trace [7]

1.4 Using FlowViz and TimeSquared Together

FlowViz and TimeSquared exist as separate tools so engineers cannot conveniently switch between them when doing performance debugging. Additionally, because FlowViz execution diagrams do not encode the time attribute, engineers cannot use FlowViz to determine which parts of an execution they should examine more closely with TimeSquared. For instance, suppose an engineer using FlowViz sees that a function is taking up an unexpectedly large percentage of a thread's execution time. The engineer would like to examine the calls made to that function in TimeSquared but FlowViz does not show the times when the function calls occurred in the execution. Consequently, the engineer does not know which time intervals in the program execution to examine with TimeSquared.

To help engineers switch from looking an execution trace with FlowViz to looking at an execution trace with TimeSquared, ThreadViewer needs to provide an intermediate visualization step between the FlowViz and TimeSquared visualizations. This intermediate visualization step requires that I develop a method of reducing the cardinality of our execution traces.

2 DATA

The dataset I used for this project is a DINAMITE trace containing 22 seconds of one thread's activity in WiredTiger, a key-value store used by MongoDB. This dataset contains approximately 11 million function entry and exit records, as well as 20 different function attribute levels. Additionally, the time attribute has a resolution measured in nanoseconds.

The following sections discuss how I reduced my dataset's cardinality.

2.1 Reducing Dataset Cardinality

To reduce the cardinality of this dataset, I exploited the fact that threads tend to execute the same sequences of functions repeatedly. For example, a thread could be repeatedly trying to acquire the same lock on a shared resource. Thus, it is possible to summarize a thread's entire execution as a finite set of "execution patterns", sequences of function call entries and exits that a thread makes repeatedly throughout its execution time.

2.1.1 Finding Execution Patterns with Sequitur

To find these execution patterns, I borrowed an idea proposed in a paper written by Walkinshaw et al. This paper suggested that an algorithm called Sequitur could be used to find phases of repeating behavior in a single thread [7].

The Sequitur algorithm was developed by Nevill-Manning and Witten to infer compositional hierarchies in a string of discrete symbols. Sequitur produces context free grammar. A context free grammar is defined as G = (Σ, N, S, P), where

- Σ is a finite set of terminals, which are the set of symbols belonging to the underlying language
- N is a finite set of non-terminals, where each non-terminal is a set of sequences of terminals
- S is a single non-terminal which represents the starting point of the language
- P is a set of production rules that map a non-terminal to a string of zero or more terminals and non-terminals.

Walkinshaw et al suggested that an execution trace containing an ordered list of function call entries could be represented as a string of discrete symbols and fed into Sequitur. Figure 3 is an example taken from Walkinshaw et al demonstrating how they converted a toy execution trace into a string.

The Sequitur output of the execution trace in Figure 3 is shown in Figure 4 below:

Production rules
0 → 1 2 2 2 3 3 4 i 5 1 4 4
1 → a 5 4 4
2 → e d
3 → f g h 4
4 → d d
5 → b c
Uncoded production rules
0 → 1 2 2 2 3 3 4 saveFile 5 1 4 4
1 → load 5 4 4
2 → insertChar renderChar
3 → select copy paste 4
4 → renderChar renderChar
5 → displayDir selectFile

Figure 4. Sequitur output of the trace in Figure 3 [7]

In Figure 4, Sequitur produced 6 production rules, with rule 0 being the initial rule S. Walkinshaw et al treated rule 0 as a representation of the complete execution trace, with the other rules representing patterns and sub-patterns within that trace.

2.1.2 Applying Sequitur to the Dataset

For this project, I used an open source implementation of the Sequitur algorithm [8]. However, one problem I encountered with using Sequitur to find execution patterns is that threads often make thousands of consecutive calls to the same function which causes Sequitur to produce patterns with un-necessarily deep hierarchies. I therefore modified the Sequitur algorithm such that the patterns generated from consecutive calls to the same function have minimal depth.

Because Sequitur is designed for finding patterns in strings of discrete symbols, I needed to create a string representation of my dataset. I created a string representation of my dataset by applying two steps:

1) Order the records by the time attribute in ascending order

2) Remove the time attribute from each record, leaving an ordered list of records which only contained the direction and function attributes. The combinations of the direction and function attribute values form the symbols. For example, "enter function __wt_evict_walk" is one symbol while "exit function __wt_evict_walk" is another symbol.

Using Sequitur, I was able to identify 7919 patterns in my dataset which contained 11 million items. The patterns also had a maximum depth of 116, despite the modification I made to the Sequitur algorithm to minimize the pattern depth.

I confirmed with the members of my research group that they were uninterested in exploring the hierarchy of patterns down to the terminals, especially because the patterns had such high depth. Consequently, I decided to flatten the hierarchy of the Sequitur output by "expanding" all production rules apart from rule 0 to their terminals; instead of having patterns consisting of sub-patterns, all patterns simply became sequences of function call entries and exits.

2.1.3 Time Attributes of the Execution Patterns

Although I now had a reduced dataset consisting of execution patterns, the reduced dataset had no time attribute; ThreadViewer users therefore would not have been able to use the TimeSquared visualizations to examine the patterns in detail. To address this issue, I wrote a Python script which found the times when the patterns appeared in my original dataset.

The script works by traversing through the sequence of terminals and patterns making up rule 0. Whenever the script finds a pattern, the script calculates the start and end times of that pattern based on its position within rule 0 and the number of terminals forming that pattern. For example, suppose that the first item in rule 0 is a pattern with five terminals. The start time of that pattern is the time attribute value of the first record in the dataset while the end time is the time attribute value of the sixth record in the dataset.

3 TASKS

I consulted with members of my research group to determine the domain tasks which ThreadViewer will support. We decided that ThreadViewer will support the following tasks:

1) Discovering which patterns dominated a thread's execution time
2) Discovering when the thread executed the patterns
3) Browsing the function call entries and exits which make up a given pattern

4 SOLUTION

ThreadViewer is divided into two panels. The TimeView panel uses a bar graph and timeline to show which patterns dominated the execution time and when the thread executed these patterns. The FlowViz panel is used to display the FlowViz execution flow diagram and show the function call entries and exits which make up a given pattern.

4.1 The TimeView Panel

Figure 5 provides a screenshot of the TimeView panel. ThreadViewer employs a vertical bar graph to display the percentage of a thread's execution time that each pattern occupies. The bar graph is always sorted in descending order so that users can immediately see which patterns dominated the execution time. To the right of the bar graph are the timelines showing when the thread executed the patterns; ThreadViewer provides each pattern with its own timeline because it would not have been practical to show all 7919 patterns together on the same timeline. Each pattern's timeline is aligned next to that pattern's bar in the bar graph so that users can see all of the information about a pattern in the TimeView panel with minimal eye movement.

To ensure that the timelines could fit into the TimeView panel, ThreadViewer sets the timelines' default unit of distance to be one second. However, as mentioned in section 2, the time attribute has nanosecond resolution. Thus, ThreadViewer provides users with the ability to navigate and zoom into the timelines. The purpose of providing users with the ability to zoom into the timelines is so that they can precisely locate the time intervals when patterns of interest occur. The idea is that once users have navigated to a time interval they are interested in examining more closely, they can pull up TimeSquared to get a detailed view of the time interval. However, at the time I wrote this paper, I have not yet integrated TimeSquared visualizations into ThreadViewer.

All of the timelines share two axes: the top axis displays precisely the time interval that the users have zoomed into while the bottom axis provides orientation. Users can zoom into the timeline either by brushing the timelines themselves or by manipulating the bar on the bottom axis.

4.2 The FlowViz Panel

The FlowViz execution diagram appears by default in the FlowViz panel because users will first wish to look at the execution flow diagram to get an overall understanding of how the thread is behaving. However, when users click on a pattern's row in the vertical bar graph, the FlowViz panel displays the function call entries and exits making up that pattern.

FlowViz execution flow diagrams are a way of visualizing DINAMITE traces without the time attribute. Execution patterns are just subsets of these traces without the time attribute. Thus, ThreadViewer displays the execution patterns as subsets of the full FlowViz execution flow diagram. Figure 6 shows the full execution flow diagram of my dataset while Figure 7 shows an execution pattern as a subset of the same execution flow diagram. When users click on a row in the vertical bar graph, the FlowViz panel removes color from the nodes which are not a part of the pattern. Additionally, the FlowViz panel highlights in red the transitions which occurred within a pattern. Each highlighted transition has two weights: the first weight represents the number of times that transition occurred in the execution pattern and the second weight represents the number of times that transition occurred throughout the entire thread execution.

When displaying the function call entries and exits within a pattern, ThreadViewer could theoretically remove the irrelevant nodes and edges from the FlowViz panel to save screen real estate. However, the members of my research group stated that if they are using ThreadViewer, they would likely switch between viewing the full execution flow diagram and viewing the patterns. Therefore, to maintain mental context, ThreadViewer does not alter the layouts of the nodes and edges in the FlowViz panel.

The advantage of using the FlowViz visual encoding style to display the function call entries and exits of each pattern is that the visualizations on the left panel remain the same size. Regardless of how big the patterns are; the execution patterns will always be smaller than the full execution flow diagram.

5 IMPLEMENTATION

Because our research group developed FlowViz and TimeSquared visualizations to be viewed on a web browser, I also designed ThreadViewer to be viewed on a web browser. I built ThreadViewer using html, Javascript, and the D3 library. I constructed the timelines in the TimeView panel out of a single SVG. The FlowViz execution diagrams are stored on ThreadViewer as PNG files.

6 RESULTS

I presented ThreadViewer to members of my research group so that they could provide feedback on ThreadViewer's effectiveness. My research group said that visualizing the execution patterns as subsets of the FlowViz execution flow diagram helped them easily understand what the thread was doing within each pattern. Additionally, my research group found the link navigation of the

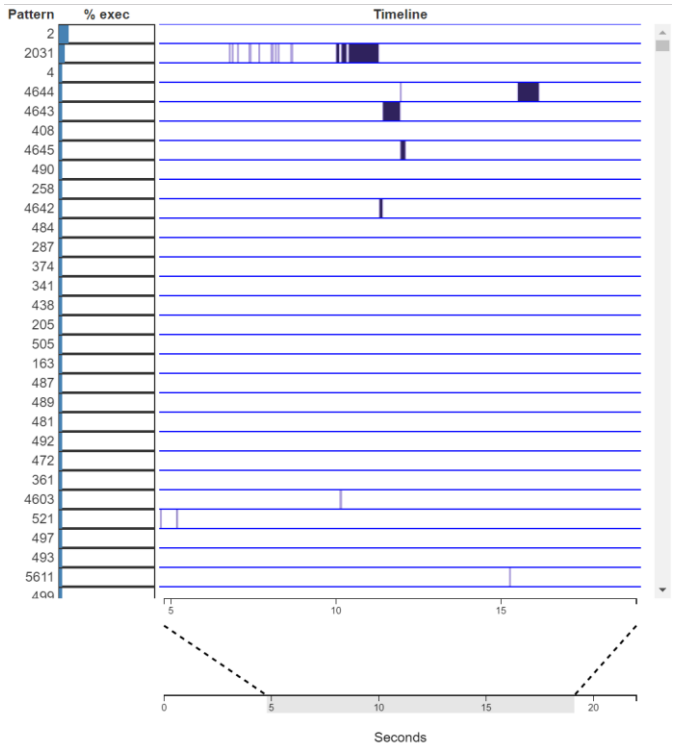


Figure 5. The TimeView panel of ThreadView

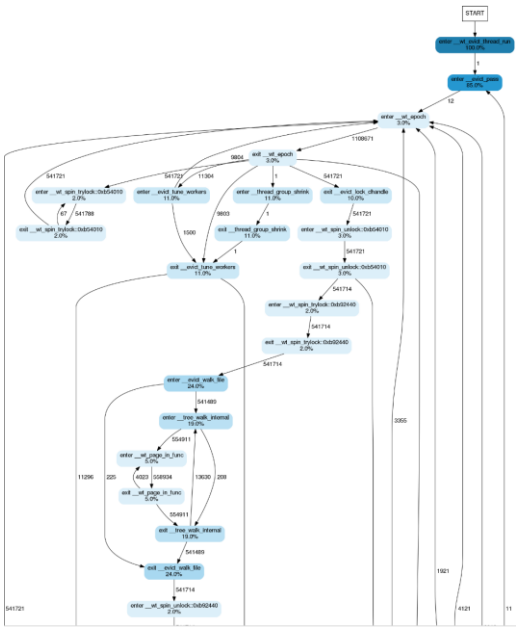


Figure 6. A thread's full execution flow diagram

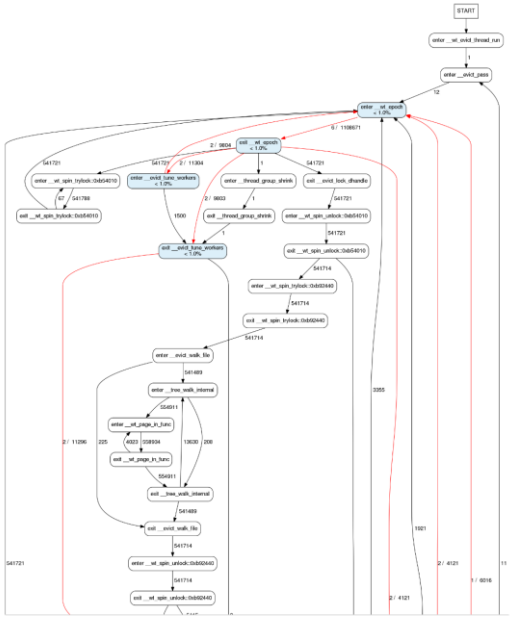


Figure 7. A pattern which the thread executed

timelines to be very convenient to use. However, my research group also noted that without having the patterns sharing the same timeline, it was difficult to see the order in which the thread executed the patterns. My research group also stated that ThreadViewer needs to support more domain tasks if it is to be an effective software performance debugging tool. For example, after viewing the execution patterns in the FlowViz view, my research group wanted to know which patterns were most and least likely to appear sequentially together in the thread's execution.

6.1 Scenario Walkthrough

Jim is a computer engineer who is using ThreadViewer to analyze the behavior of a thread in his multi-threaded program. He first looks at the full execution flow diagram in the FlowViz panel to get a general understand of how the thread is behaving. He then looks at the bar graph in the TimeView panel to see the percentages of the thread's execution time that each pattern occupied. He decides that he is interested in studying the pattern which took up the most execution time. Thus, he selects the first row in the bar graph and inspects the highlighted nodes and edges in the FlowViz panel. From studying the execution pattern in the FlowViz panel, he observes that the thread is behaving abnormally. He therefore looks at the pattern's timeline to see when the thread is executing this pattern. He zooms into one part of the timeline which shows the thread executing that pattern. He decides that he wants to view this part of the thread execution in detail so he clicks a button on ThreadViewer to bring up the TimeSquared visualization on a separate window.

7 DISCUSSION

ThreadViewer is in early development and is not ready to be used for performance debugging. However, my research group was able to use ThreadViewer to discover problems with the way the Sequitur algorithm detects execution patterns. These discoveries excited vigorous discussions within our research group about how to modify Sequitur to address these problems and even if we should create our own pattern detection algorithm. This section describes the problems with the Sequitur algorithm that members of my research group found using ThreadViewer.

7.1 Execution Patterns with a Small Number of Nodes

Viewing the execution patterns in the FlowViz panel showed that some execution patterns detected by Sequitur would be of no interest to engineers because they contained an extremely small number of function call entries and exits. For example, Figure 8 shows a pattern which only captured a single call to the function __wt_spin_trylock::0xb54010.

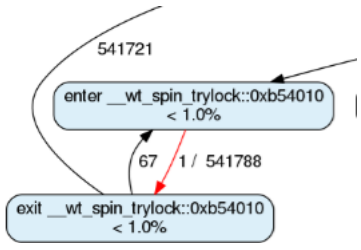


Figure 8. An execution pattern containing only one function call

7.2 Execution Patterns Capturing Incomplete Function Calls

Viewing the execution patterns in the FlowViz panel also showed that some patterns do not contain complete pairs of function call entries and exits. Figure 9 shows an execution pattern in which a thread exited a call to the function __wt_evict_walk without entering it. These execution patterns would be very confusing to ThreadViewer users.

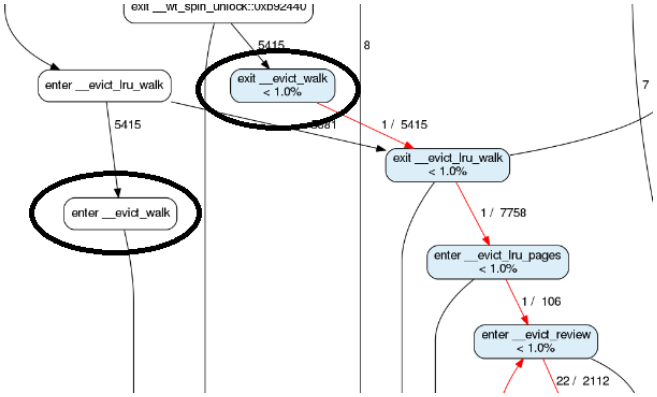


Figure 9. An execution pattern showing the thread exiting a function call to __evict_walk without entering it

7.3 Low Percentages of Execution Time

The bar graph in the TimeView panel shows that the vast majority of the execution patterns occupy less than 1 percent of the thread's execution time. However, it is likely that ThreadViewer is under-reporting the execution time percentages because I flattened the hierarchy of the Sequitur output and only looked at pattern occurrences in rule 0; it is probable that patterns only appear a few times in rule 0 but appear numerous times as sub-patterns within other patterns.

8 FUTURE WORK

Because of the discoveries my research group made with ThreadViewer, I intend to either modify the Sequitur algorithm to locate more meaningful execution patterns or develop my own pattern detection algorithm. Improving our pattern detection strategy will also help us reduce the dataset cardinality even further. For example, constraining Sequitur to only create patterns with complete pairs of function call entries and exits will reduce the number of patterns that Sequitur can detect. If we are able to reduce the cardinalities of all DINAMITE traces to a small number of patterns, it may be possible to fit all of the patterns onto a single timeline. Thus, ThreadViewer users will be able to see the order in which the thread executed the patterns. Reducing the dataset cardinality also has the benefit of making the navigation of the timeline more responsive.

The visual encoding styles used in FlowViz also needs improvement. For instance, FlowViz should use line width channels to communicate edge weights as opposed to number labels. FlowViz also communicates the direction attribute of each state using labels. I would like to explore using node shapes to encode the direction attribute: nodes with one shape represent function call entries while nodes with another shape represent function call exits.

9 RELATED WORK

Visualization of execution traces is a well-researched field. Most approaches for serial trace visualization involve assigning one of the axes the time variable while the other axis is used to represent different processes, classes, instructions, or methods [6] Tools like TimeSquared which adopt this visualization scheme can only display fractions of the total program execution duration. Other visualization tools provide an overview of the execution trace and allow users to zoom in on parts of the trace for more details. Extravis is a tool which uses a "massive sequence view" to

show an overview of an execution and a "circular view" to show the interactions between the components of a program [4]. Synctrace is another tool which draws a serial timeline overview of a selected thread and shows the call stacks of different threads in the context view [3]. However, the visualization approaches adopted by both Extravis and Synctrace are not scalable for large traces.

There is also research into how to detect and visualize execution trace patterns. Knüpfer et al used complete call graphs to detect trace patterns and proposed how the Vampir GUI performance tool could visualize these patterns [2]. However, Knüpfer et al focused primarily on creating the pattern detection algorithm and did not perform any analysis on how feasible their visualization encoding ideas were.

Our approach to visualizing execution patterns is similar to the visual encoding schemes used by visualize genomic data. From a data abstraction perspective, execution traces and genomes are very similar datasets. Genes are long sequences of nucleotides while execution traces are long sequences of execution events. Moreover, genomes have larger cardinality than our execution traces. The human genome, for instance, contains approximately 3 billion nucleotides; in contrast, our execution traces only contain millions of events. Therefore, the visualization techniques employed by genome browsers can accommodate the typical sizes of our execution traces.

Rather than simply visualizing the entire genome, genome browsers allow users to study individual genes [10]. Genes are approximately 10,000 nucleotides in length, and there are approximately 20,000 genes in the human genome. Thus, visualizing a single gene is a more tractable infovis problem than visualizing an entire genome at once. Some visualization tools for analyzing genomic data also use derived attributes to guide users to the most relevant genes they are interested in studying [5]. Similar to these tools, we chose to visualize individual execution patterns instead of displaying the entire execution trace. Additionally, ThreadViewer also has derived attributes which guide users to the most interesting execution patterns.

10 CONCLUSION

I present ThreadViewer, a visualization tool for analyzing a thread in a program execution. This tool uses the FlowViz visualization to provide an overview of a thread's behavior and presents users with the different patterns of behavior that occurred in the thread's execution. A bar graph shows which patterns dominated the thread's execution time while a timeline provides users with the ability to see precisely when the patterns occurred in the execution. Once users have identified sections of the execution that they would like to examine more closely, they can view those sections in detail using TimeSquared. However, TimeSquared has yet to be integrated into ThreadViewer.

Although ThreadViewer is still in the early stage of development, it has already revealed problems with the way we are detecting execution patterns. I will continue to use ThreadViewer to evaluate the effectiveness of any future pattern detection strategies that I create.

ACKNOWLEDGEMENTS

I would like to thank Tamara Munzner for mentoring me throughout this project. I would also like to thank the members of my research group for providing me with feedback about ThreadViewer, in particular Sasha Fedorova and Ivan Beschastnikh.

REFERENCES

[1] A Survey on Performance Tuning by Plumbr. https://plumbr.eu/blog/performance-blog/java-performance-tuning-survey-results-part-i.

[2] Andreas Knüpfer, Bernhard Voigt, Wolfgang E. Nagel, and Harmut Mix, **Visualization of repetitive patterns in event traces**, in International Workshop on Applied Parallel Computing. Springer Berlin Heidelberg, 2006.

[3] Benjamin Karran, Jonas Trumper, and Jurgen Dollner, **Synctrace: Visual Thread-interplay Analysis**, in IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), 2013.

[4] Danny Holten, Bas Cornelissen, and Jarke J. Van Wijk, **Trace Visualization Using Hierarchical Edge Bundles and Massive Sequence Views**, in IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), 2007.

[5] Joel A. Ferstay, Cydney B Nielsen, and Tamara Munzner, **Variant View: Visualizing Sequence Variants in their Gene Context**, in IEEE Trans. Visualization and Computer Graphics (Proc. InfoVis), 19(12):2546-2555, 2013.

[6] Katherine E. Isaacs, Alfredo Giménez, Todd Gamblin, and Abhinav Bhatele, **State of the Art of Performance Visualization**, in EuroVis, 2014

[7] Neil Walkinshaw, Sheeva Afshan, and Phil McMinn, **Using compression algorithms to support the comprehension of program traces**, in Proceedings of the Eighth International Workshop on Dynamic Analysis. ACM, 2010.

[8] Sequitur, http://www.sequitur.info/

[9] Svetozar Miucin, Lyuyu Ye, Augustine Wong, Derek Chan, Ivan Beschastnikh, and Alexandra Fedorova, **Blasting Performance Bugs with DINAMITE**, submitted to the USENIX Annual Technical Conference, 2017

[10] UCSC Genome Browser, https://genome.ucsc.edu/