# Critical Section Investigator: Building Story Visualizations with Program Traces

Michael D. Shah Samuel Z. Guyer
Tufts University
Medford, MA
{mshah08,sguyer}@cs.tufts.edu

***Abstract*—Detecting performance problems that infrequently occur can be very difficult with traditional profilers. Most profilers only show the average time of execution or the total time a method contributes to the overall program's execution time. Most profilers do not explain or show why different control paths within a method executed may have resulted in variable execution times. When debugging concurrent programs for performance problems, the complexity and variability in execution time can potentially be even greater.**

**In this paper we take a first step in visualizing individual method's different execution paths within multithreaded Java programs. We restrict our domain to looking at critical sections for this initial analysis, as variability in critical sections may cause more noticeable performance variation. Our software visualization tool, Critical Section Investigator (CSI), builds on the visualization and interaction techniques in previous works like KCachegrind with several enhancements. The result of our work is the first tool to our knowledge that visually shows potential performance differences in synchronized methods in Java programs using a profiling and storytelling structure.**

***Index Terms*—Java, Critical Sections, Concurrency, Program Comprehension, Software Visualization.**

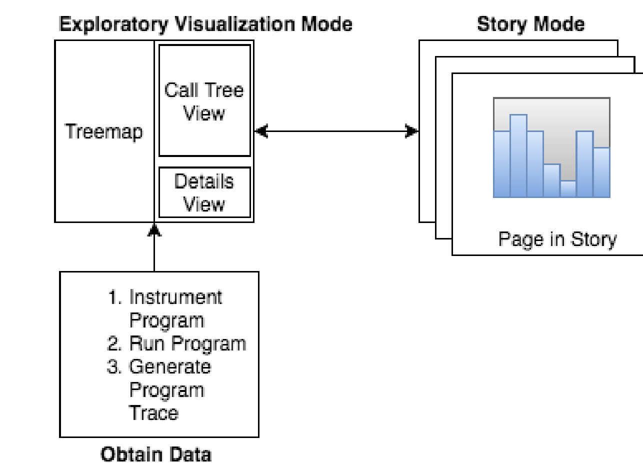


Fig. 1: The architecture of our visualization allows users to switch between two modes: Exploring Profiling Data and Story Mode. While switching views, all visualizations remains coordinated with each other to aid in investigation tasks.

## I. Introduction

Programmers spend large amounts of time writing multithreaded software, often with the goal of achieving greater throughput and better execution times. In order to monitor performance, profilers are used to identify where code can be optimized. Most profiling tools at a minimum display the total amount of time each individual method consumes of a programs total execution time. The level of granularity may also be broken into the individual time spent in each execution of the method and displayed as an average. If there exists an execution or several executions of a method that took significantly longer time to execute, this information is lost from profilers that only present averages or totals. If a method has a high standard deviation of execution time, this may correlate with the control paths taken, and developer can use a methods call tree to help determine this. Thus, visualization tools that can show the control paths taken may assist in teasing out software performance problems that are outliers.

In our work, we have chosen to work on a specific type of performance problem that could occur with locking primitives. We have gathered execution traces on methods in the Java programming language that are marked as *synchronized*. A method that is synchronized is known as a critical section or critical method. This means that when this synchronized method executes, a lock is held that impedes the progress of other threads when contending for a shared resource. Because these methods exist in the presence of multiple threads, it is possible that long software hangs can occasionally occur when threads are not able to make progress. This can disrupt the user experience in real time (e.g., streaming video) and interactive applications (e.g., video game).

Our main contributions in this paper are the following:

- A software visualization tool focused on visually showing the variability of call trees for critical sections in the Java programming language.
- A storytelling view that allows users to quickly tag and sift through profiling information that a user has marked as important.
- A use case on discovering performance bottlenecks in critical sections.

## II. Our Tool - CSI

Our tool the Critical Section Investigator (CSI) provides three multiple coordinated views for analyzing synchronized methods (which we will also refer to as critical sections) in an exploratory visualization mode. In the following subsections, we describe how we obtain the input to our visualization, how

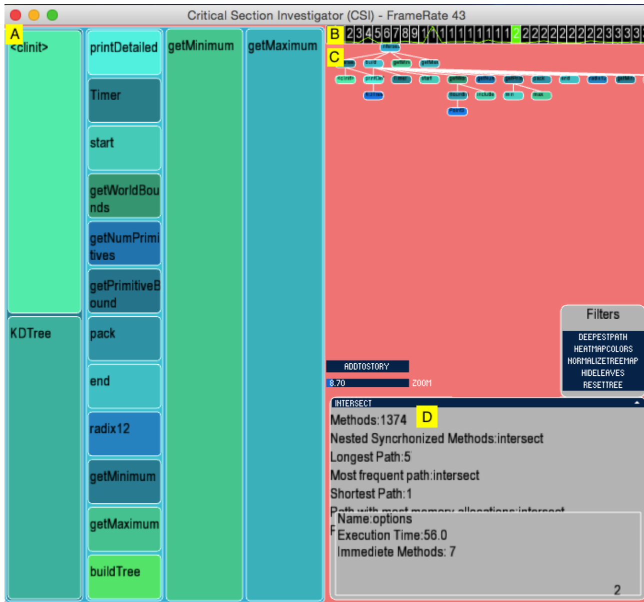


Fig. 2: Our visualization tool has multiple coordinated views. A details panel is on in the bottom right, and each item that is highlighted will highlight corresponding nodes in both the treemap and node-link diagram. The node-link diagram represents a critical sections call graph. The treemap pictured is using time spent in a method to encode the size of each rectangle

we visualize the input data, and how our visualization is used to find performance problems.

### A. Input

The input to our visualization is a call tree that is annotated with profiling information. In order to obtain the call tree, we built a custom java agent that inserts probes into Java programs to obtain a call tree with profiling information and code metrics using the Java Assist framework [8]. In order to make our instrumentation more efficient, we built a static analysis with the Soot framework to identify which methods were marked as synchronized so we only instrument a subset of all of the Java methods [15]. The subset of methods we instrument are the synchronized method themselves, and then only the methods reachable from the synchronized method, so that we can build the call tree and perturb our program in the most minimal way when gathering data.

### B. Exploratory Visualization Mode

In the exploratory visualization mode, we present three coordinated views that display information to the user. Each of the views role in the visualization is described below.

*1) Call Tree View (Figure 2-C):* The Call Tree View shows a directed acyclic graph of all of the methods called from a synchronized method. The layout of the call tree is based on the graph layout by Wetherell and Shannon for drawing a minimum width tree [3]. We made a slight modification such that child nodes would be centered under the parent, while still showing a compact tree. If the tree is too wide or deep, filters exist that allow users to hide leaves. Additionally, if we are only interested in the longest path in the tree, all other paths can be hidden immediately.

*2) Treemap View (Figure 2-A):* The treemap view is coordinated with the call tree view showing the equivalent call tree of a synchronized method. A treemap is a space-filling visualization that visualizes hierarchical structures to a user [1]. The treemap uses the area of a rectangle to encode attributes, and rectangles embedded within another rectangle represent a parent-child relationship in the call tree. We use a simple slice and dice algorithm to build the treemap. This preserves the order in which methods are called in the call tree, laying them out left to right and top to bottom. This design decision was made so that the treemap view and call tree view would align with each other. In our current implementation, we map by default the execution time of each method to the size of each node in the treemap. If the aspect ratio of the nodes in the treemap is not visually pleasing, users can normalize the size of the nodes to make selection and interaction easier by pressing a normalize button on our interface.

*3) Details View (Figure 2-D):* As nodes are moused over in the call tree and treemap, a details view is updated showing information about each method. For each method, we display the following code metrics on demand:

- Execution time - The time it took for this single instance of the method to execute
- Average execution time - The average time this method takes to execute (regardless of caller).

### C. Interaction

When we start our visualization tool, the entire program call tree is visible. The entire call tree is displayed so a user can view how linear the program is, and spot synchronized methods that appear to be on the critical path of execution. In order to view individual methods, we select which synchronized method we are interested in viewing from a drop down list sorted in alphabetical order by method name. A new call tree will become visible, showing the first execution of the selected method.

For the selected method, a line graph (Figure 2-B) is displayed at the top of our call tree view. The line graph is divided into blocks that represent each execution of that method. The line drawn between each block indicates a user selected criteria (e.g., execution time), and we can view how that criteria changes each execution of the currently selected method. As we hover our mouse over each block, we can quickly preview the call tree of each instance that the method was called. Spikes in the line graph indicate to us a change during that methods execution, and are indicators of where to investigate what may have caused that spike. Left-clicking any of the blocks in the line graph will hide them.

When we begin exploring the call tree, each node in the Call Tree View is equivalent to the color of each node in the treemap view for consistency. Highlighting a node in the treemap will highlight all of the nodes in the call tree path

along that view. Highlighting a node in the call tree view, will highlight only that individual node in the treemap view.

When one node is highlighted, the details pane will update to show code metrics about that individual node. If several nodes are highlighted (i.e., when using the Treemap view to select), then code metrics along that path will be aggregated and displayed in the details pane.

Our visualization also supports several mouse and keyboard interactions in order to support our exploratory visualization.

- Pressing the up and down arrows show and hide different layers of the treemap to reduce visual clutter. Highlighting a node in the call tree view and pressing shift also brings a user to that nodes depth in the treemap so the node can be analyzed without being obstructed.
- Left-clicking on a node hides that node and any of the nodes children in both the treemap and the call tree view so users can prune away unnecessary nodes.
- Right-clicking on a node sets that node as the new root, and hides every other node. This allows us to focus on analyzing one path in the call tree.
- A zoom widget allows users to inspect the call tree for more fine grained selection.

When we have a call tree that needs to be further investigated (whether it has been filtered or not), we can click the *Add Story* button to add the current view of the call tree to the story mode, which is described in the next section.

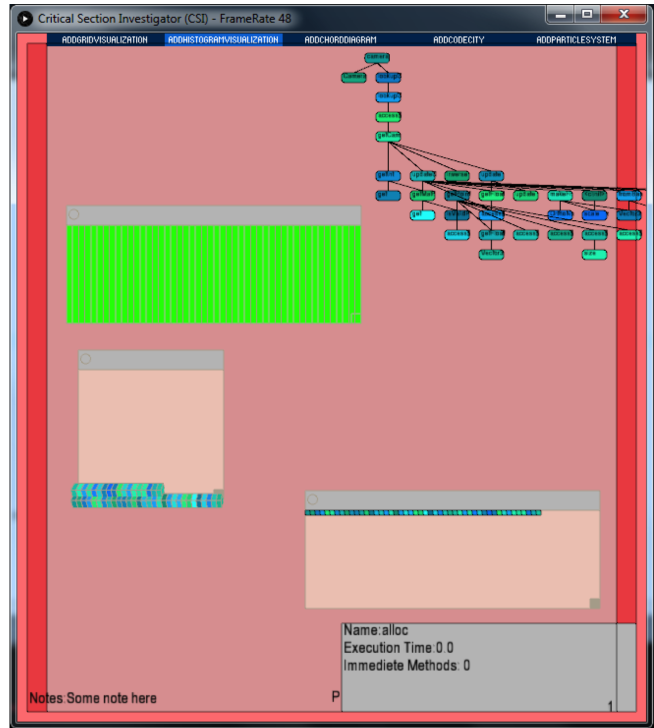


Fig. 3: Users can add notes to each instance of the call tree view that they want. They may then scroll through in a linear manner the stories they have recorded.

## III. Story Telling

Recent work by Boy et al. in using story telling in visualizing has suggested it may be possible to engage users in understanding data [10]. We provide a linear narrative structure in this visualization, and hope that this will inspire other types of story structures that can be told for working with profiling data as described by Segal and Heer [6]. Storytelling using data visualizations has previously been demonstrated by Lee et al., but not in the context of a profiling tool [2].

### A. Storytelling Mode

We can toggle our visualization into storytelling mode (shown in Figure 3) at any time from the exploration mode. When in storytelling mode, all of the call trees that we have marked to add to the story are displayed. Each of the call trees that we have added, are put sequentially into *pages* which can be scrolled through.

In storytelling mode, we investigate each call tree in a more targeted manner by annotating each page in the story. The annotations act as sticky notes that are placed on a page, where each sticky note takes the form of a data visualization for the call tree on the current page. Each of these visualizations that we annotate a page with, are also coordinated with the call tree in view. When annotations are added, they can be resized, dragged around, and this information is preserved.

The following annotations can be added in any combination to each page in a story:

- Histogram - A histogram showing executions times of each of the methods in the call tree.
- Grid Visualization - A grid visualization where each method is a cell and can be colored by a program attribute creating a heat map (e.g., we use time spent in method for our use cases). The purpose is to show an overview of all of the called methods and which ones might be of interest immediately.
- Chord Diagram - Allows the user to demonstrate highly or sparsely connected components within a call tree [5].
- Note - Text notes can be added to annotate the visualization. This is useful for crafting part of the story, and adding in comments about the call tree.
- Code City - A custom 3D representation of the call tree can be displayed using a city as a metaphor as done by Wettel and Lanza [16]. Our view is much more simplified then the authors work, but can help give context to the radical differences between different call trees by viewing a 3D structure.

## IV. Evaluation

### A. Use Case

In order to evaluate our tool, we report on finding where variable performance exists in a multithreaded Java program. The benchmark we are using is called Sunflow, which is a Java multithreaded raytracer [17]. The objective is to find synchronized methods where performance is highly variable and could potentially be a performance problem because of lack of scalability. When we have found these potentially problematic

| Method Name | Performance Problem | Information Uncovered | Resolution Steps With Our Visualization |
|---|---|---|---|
| GetMinmum | Single Instance of longer execution | Of 38 method calls, one takes significantly longer than the others. | 1) We add two instances of the execution to our story mode so we can quickly flip between them. 2) We visited our line graph and saw a spike, and noticed the execution time jump to 42ms from 1ms. 3) We investigated the source code to see why this may occur. |
| Intersect | High variability and time spent | The range of execution varies between 1ms and 1162ms | 1) We notice in most cases where the call tree expands, a call to intersect is made when browsing in exploration mode. We notice few Java Standard Library calls. 2) Next we normalize our treemap, so we can easily select nodes buried deep in our tree. 3) We see then investigate the source code to see where any shortcuts can be taken. |
| Options | Amortized Data Structure | Hashmap has to resize | 1) First we see there are three methods called, with the second method spiking. 2) We see the treemap for the second instance, and notice a function in the call tree called resize that takes most of the time. 3) We believe changing the initial size of the hashmap can mitigate this problem. Not mitigating this problem will make this oscillation in execution time occur more frequently. |

TABLE I: The table displays some of the results to the questions we asked during our use case problems investigated with our tool.

sections of code, we will then make recommendations for how to refactor this software.

While undergoing this investigation, we asked the following questions to guide us:

- Are the majority of method calls to standard libraries?
- Do amortized algorithms play a large role in the variability in call trees?
- Do any call trees oscillate with any regular pattern in execution time?

### B. *Discussion of Results*

We started our investigation by picking out synchronized methods in our tool that had multiple runs. The results of our investigation can be found in Table I.

### C. *Implementation and Performance*

Our software visualization is a standalone application built in Java using the Processing 3.0 framework and runs on Mac, PC, and Linux [14]. Our visualization runs at interactive framerates (30 frames per second) on our test system which was a a Dell Precision T3610 with an Intel Xeon CPU E5-1620 at 3.7 GHz, 48 GB of RAM and an NVIDIA Quadro K600 graphics card.

## V. Discussion

The problem of adding instrumentation to record call trees means that the values returned for execution time could be perturbed. We attempt to mitigate this by mapping keys(Integers) to method names when logging our trace. We then do post-production to recreate the trace with method names. This helps minimize I/O latency when the actual program is running, and can help prevent garbage collections that could occur in Java if we were to use bigger key names (e.g., a String primitive).

Our current instrumentation does not deal with code that is dynamically loaded, as our instrumentation is all done statically before runtime when we collect a trace. Thus, Java programs that make heavy use of reflection (i.e., code that is dynamically loaded at runtime) may not be good candidates for using our tool, as many potential paths could be missed in each call tree. The Java Assist framework does allow classes to be reloaded during runtime using a hotswapper, thus it may be possible to mitigate this problem in future work.

For our visualization, it is important to ask how well will our visualization scale to be useful for programs that execute for long periods, or for methods that execute substantially more times than others. It is possible that extremely large call trees can also clutter our visualization. In order to mitigate this, we plan to add additional views and layouts of the call tree. The addition of more filters to prune nodes could also help. When selecting which of the call trees to investigate, filtering to only view call trees which are different could eliminate many of the different execution paths.

## VI. Related Work

Our tools interface was inspired by KCachegrind which is a call graph viewer which makes uses of a node-link diagram and a treemap [12]. Our tool differentiates itself by presenting new interaction techniques between the coordinated views, supplies unique built-in filters, works on Java instead of C++ programs, and a storytelling mode with several additional visualizations.

*1) City Metaphor Visualizations:* The work by Waller et al. presents Synchrovis which uses the software as a city metaphor for program comprehension using static and dynamic program traces [9]. Four different types of synchronization concepts are encoded and displayed: Threads, Monitor/Semaphore, Wait/Notify, and Thread Join. The 3D layout allows for large software systems to be displayed and navigated, which is different from our system which fits to the screen space. While both systems look at synchronization, our

system focuses on performance variations between call trees, rather then correctness issues.

*2) Thread Cities:* The Thread Cities Tool by Hahn and et al. presents a visualization technique that allows users to look at multiple levels-of-detail of threads in a multithreaded software system [7]. The metaphor of a traffic system is used where different lanes of traffic represent threads. The tool uses a visual analytics approach to perform analysis and zoom into the important details on demand for each node. Multiple views of a threads attributes can then be zoomed into for performing program understanding and performance analysis tasks. Our tool takes a similar visual analytics perspective for trying to understand why problems may occur, but uses a combination of different visualizations and interactions to achieve this.

*3) Concurrency Execution:* The tool SPIN evaluates how to visualize multi-threaded programs using UML sequence diagrams with concurrency information [4]. The challenge their work addresses is representing the individual threads. Our work primarily focuses on the performance of each critical section, regardless of which thread executes it.

The tool BEAT by Johnson and Marsland uses a visualization to mimic video editing software for following program traces of Java programs [13]. BEAT integrates directly with the Eclipse IDE and gives programmers a step by step trace of the concurrent execution, while our tool operates external to an IDE.

*4) Profiling Tools:* Other profiling tools like Java VisualVM provide call trees and real time monitoring of threads [11]. However, the visual interface is primarily related to CPU load and how much time is spent in a function. While this profiler provides thread information, it does not focus specifically on the critical sections.

## VII. Conclusion and Future Work

In this paper we have presented our software visualization tool the Critical Section Investigator (CSI) for analyzing performance in synchronized methods. Our system was tested on a use case to demonstrate its effectiveness for finding where performance bottlenecks may occur in a critical section.

In our future work we would like to enhance our visualization in the following ways:

- We would like to stream call tree data from programs executing in real time in our visualization. This would also allow us to analyze dynamically loaded code for the Java language, and present new research challenges into how to best prune relevant data.
- We would like to add more graph layout algorithms to help the user explore data. There exists a wide body of alternations to the Treemap and call tree layouts that might increase usability of our tool which we would also like to study.
- We would like to build on the storytelling mode, and add additional data visualization abstractions useful for debugging performance problems. Our initial data visualizations are also very simple, and can be expanded upon in the future.
- A more in-depth evaluation with a full user study on how our tool helps developers find performance problems. This will allow us to understand what data developers need to find problems, and how our tool compares to traditional profilers in increasing overall program performance.

## Acknowledgment

The authors would like to thank the Tufts University Redline Research group for their feedback during the development of this system.

## References

[1] Ben Shneiderman. 1992. Tree visualization with tree-maps: 2-d space-filling approach. ACM Trans. Graph. 11, 1 (January 1992), 92-99.

[2] Bongshin Lee, Rubaiat Habib Kazi, and Greg Smith. 2013. SketchStory: Telling More Engaging Stories with Data through Freeform Sketching. IEEE Transactions on Visualization and Computer Graphics 19, 12 (December 2013), 2416-2425.

[3] C. Wetherell and A. Shannon. 1979. Tidy Drawings of Trees. IEEE Trans. Softw. Eng. 5, 5 (September 1979), 514-520.

[4] Cyrille Artho, Klaus Havelund, and Shinichi Honiden. 2007. Visualization of Concurrent Program Executions. In Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02 (COMPSAC '07), Vol. 2. IEEE Computer Society, Washington, DC, USA, 541-546.

[5] Danny Holten. 2006. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. IEEE Transactions on Visualization and Computer Graphics 12, 5 (September 2006), 741-748.

[6] Edward Segel and Jeffrey Heer. 2010. Narrative Visualization: Telling Stories with Data. IEEE Transactions on Visualization and Computer Graphics 16, 6 (November 2010), 1139-1148.

[7] Hahn, S.; Trapp, M.; Wuttke, N.; Dollner, J., "Thread City: Combined Visualization of Structure and Activity for the Exploration of Multi-threaded Software Systems," in Information Visualisation (iV), 2015 19th International Conference on , vol., no., pp.101-106, 22-24 July 2015

[8] Java Assist - http://jboss-javassist.github.io/javassist/

[9] J. Waller, C. Wulf, F. Fittkau, P. Dohring, and W. Hasselbring, Synchrovis: 3d visualization of monitoring traces in the city metaphor for analyzing concurrency, in VISSOFT. IEEE, 2013, pp. 14.

[10] Jeremy Boy, Francoise Detienne, and Jean-Daniel Fekete. 2015. Storytelling in Information Visualizations: Does it Engage Users to Explore Data?. In Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15). ACM, New York, NY, USA, 1449-1458.

[11] JVisualVM - https://visualvm.java.net/

[12] Kcachegrind.github.io, 'KCachegrind', 2015. [Online]. Available: https://kcachegrind.github.io/html/Home.html. [Accessed: 26- Nov-2015].

[13] Paul Johnson and Stephen Marsland. 2010. Beat: a tool for visualizing the execution of object orientated concurrent programs. In Proceedings of the 5th international symposium on Software visualization (SOFTVIS '10). ACM, New York, NY, USA, 225-226.

[14] Processing - https://processing.org/

[15] Raja Valle-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, Vijay Sundaresan, Soot - a Java bytecode optimization framework, Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, p.13, November 08-11, 1999, Mississauga, Ontario, Canada.

[16] Richard Wettel, Michele Lanza, and Romain Robbes. 2011. Software systems as cities: a controlled experiment. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 551-560.

[17] Sunflow - http://sunflow.sourceforge.net/