Interactive Data Science Final Project Report: Profiler

Haonan Wang
haonanw@andrew.cmu.edu

Lichen Jin
lichenj@andrew.cmu.edu

Weiyi Zhang
weiyizha@andrew.cmu.edu

Yijie Zhang
yijiez2@andrew.cmu.edu

ABSTRACT

Programmers need to keep track of their programs' well-being and optimize the performance of their codes. Profilers are useful tools for experienced programmers to detect CPU bottlenecks and memory errors, but beginners may find it difficult both to use these tools and to interpret their raw text outputs. In this project, we build a user-friendly visualization for new programmers to conduct both CPU and memory analysis.

There are three major components. Call graph gives users an overview of the program's structure and also supports free exploration into a small group of functions through expanding and collapsing operations. CPU line-wise panel allows users to detect in detail which lines of the code consumes most time. Memory line-wise panel enables users to detect possible memory bugs in each line of codes. All components are highly interactive and easy to read. Our application is lightweight and end-to-end. Users can spare the trouble of calling profilers and interpreting raw text outputs. All they need to do is to upload their codes or input through the editor.

Project URL

INTRODUCTION

Profiling, namely performance analysis, has long been an important way for experienced programmers to find the performance bottlenecks for a program so that they can optimize their code. For example, if the program is not running efficiently or sometimes triggers errors, the programmer may want to check which part of the code takes the most execution time and where the exceptions exactly happen.

There have been a number of profiler tools analyzing program performance such as CPU and memory to serve these purposes. However, one problem with them is that they are mainly designed to generate text-format output; even if some visualizations are supported, they are not expressive enough. These tools are convenient for experienced programmers, but not the case for beginners who may have a hard time interpreting the complex lines of results. Better visualizations of the profiling results with more human-centered interactions are needed to help the beginners understand the performance data as well as their written programs.

Therefore, our main goal in this project is to build a user-friendly and lightweight profiling tool for CPU and memory performance analysis. This web application will provide effective and interactive visualizations and thus enable the user to easily understand the program's structure and find the bottleneck part in their code. To be more specific, our application does not build an independent profiler software; instead it uses existing tools to generate raw text results, and visualizes the text in a human-centered way. So far we support the CPU and memory profiling on C language. The application would perform an end-to-end job, taking only the source code as input and providing the visualizations on-the-fly. We will focus on detailed methods and design in the following sections.

RELATED WORK

Performance Data

There exist many tools that help evaluate programs' performance. Profiling tools, such as gprof [2], are useful techniques to analyze CPU usage. A profiler can pause execution of a program repeatedly and collect the current instruction pointer and the call stack. It then presents an analysis of the percentage of time spent in each part (both line-wise and function-wise) of the code based on these collected samples, and thus may help determine the bottleneck of the program. However, it requires linux machine and the raw text output is hard to read and interpret for freshman programmers.

Valgrind [4] is a programming tool for memory debugging and memory leak detection. It is a virtual machine using just-in-time (JIT) compilation techniques. It will first translate the codes into Intermediate Representations(IRs) and do some extra operations to these IRs. Then it will transfer the IRs to the machine code and run it on the processors. Same as gprof, it also outputs raw text results that can be challenging for new programmers to read.

Performance Visualization

[3] introduces a split into sub-goals of a successful performance visualization. The first step is global comprehension, which presents a big overview picture of how the program behaves and helps users compare their expected performance with the achieved one. The second step is problem detection, which helps programmers fix attention on certain anomalous behaviors and performance bottlenecks and allows them to explore in more detail freely to gain a deeper understanding. Thirdly, a visualization may present diagnosis and attribution, possibly through linking, correlation and tracking.

Call graph visualization is one of the most common performance visualization methods, which shows relationship among functions and usually encodes additional information

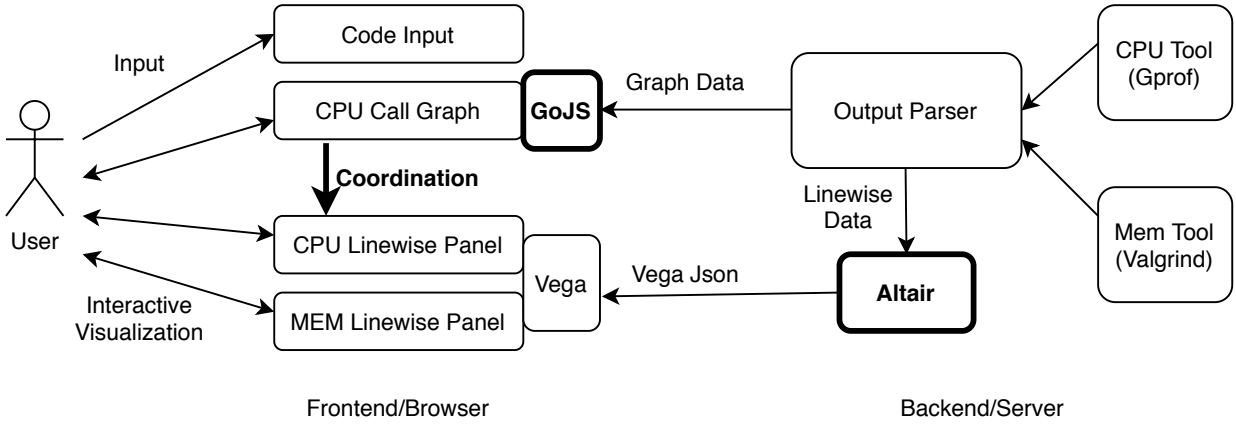

Figure 1. The overall architecture of the application. The data sets are parsed from the unstructured raw text outputs from the tools. To efficiently support different visualizations, two different libraries are used specifying visual encoding. GoJS directly run in the frontend while altair send the encoded representations back. The user inputs the code, the system performs the profiling and visualization, and the user can interact with the results.

related to each individual function as well. One common representation is node-link metaphor, which uses nodes to represent functions and links for function calls. Some tools use an indented tree layout [7], some use conventional tree drawings [5] and others may use treemaps [6] and sunbursts [1].

METHODS

Overview Design

The diagram of our system in illustrated in figure 1. To build a better interactive visualization application, we separate the frontend instead of using Streamlit. Upon user input, the application would perform profiling using software tools and parse structured data from the text-format outputs of these tools; the data sets include the CPU call graph, the line-by-line CPU time usage data, and different memory issues on exact source code lines. The visualization tools then generate interactive visualizations which are rendered in frontend and displayed back to the user.

Front/Backend Infrastructure

At front-end, We use Vue.js as our JavaScript framework and Bootstrap as our CSS framework. We provide two options for users to input their code, a file upload button and a code editor. Users can also first upload their code and then further edit the code using the editor.

We developed our back-end with Flask. Our back-end service exposes two RESTful APIs for CPU profiling and memory profiling respectively. After users submit their code, the front-end will consume the back-end APIs to retrieve the profiling results. At the back-end, our service will make calls to gprof or Valgrind to generate the profiler reports when receiving the code. For CPU call graph, we parse functions and function calls information into JSON; for CPU/memory linewise panel, we first make use of Pandas to convert the text output from profiler to DataFrame, then create our visualization with Altair. The back-end will respond the front-end request with the JSON specification for our plot, and the front-end will display it to user with Vega-Lite or GoJS.

Next, we would mainly focus on the highlighted parts of the diagram which are the key to the interactive visualizations. Technically we manage to visualize the CPU and Memory data with GoJS and Altair(Vega), and support meaningful interactions between different views.

CPU Call Graph

Call graph visualization intends to give users both a global view of calling relationship among functions and a local view of certain sub function groups. Following node-link design, we represent a function in a node and a function call in a link (directed edge in graph context), with nodes encoding function implementation time in color and links encoding time flowing through this call in width (size). Both nodes and links allow mouse hover to see more detailed information.

It begins as a complete call graph and supports cumulative expanding and collapsing operations, i.e., a collapsed node would show a '+' button and upon click it would expand the its direct children, and a expanded node would show a '-' button and upon click it would collapse all its children and grandchildren. Note a child may have multiple parents, and we allow users to collapse some of them to discover partial dependency. During expanding and collapsing, node positions would not change in order to maintain a global context and clear structure organization, but node colors would adapt dynamically.

Our call graph visualization is built on GoJS, a javascript package that takes lists of JSON-formatted node and edge information to establish interactive diagrams and graphs. Each time a button is clicked, we render new JSON lists directly in front-end and redraw the diagram without calling to back-end. Therefore, the interaction is immediate and prompt.

To support expanding and collapsing operations, for each node and edge, we maintain a constant json for its invariant information and a variable json for its current state. When a parent node is collapsed, we process each child by first recursively collapsing the child and afterwards setting the link time as 0, the child implementation time as 0 and converging that time to its parent implementation time. Expanding happens layer by layer, and therefore we only recover each child's time and the

link's time flow, but do not recursively expand grandchildren. In the case of multiple parents, a child is hided only if all its parents collapsed. Otherwise, we only hide the link and re-encode the child's in its remaining time and downwards update grandchildren nodes and links. For the latter, precise time figures are not available and we estimate them assuming same-time-per-call, i.e., a child with 50% valid time would lead to 50% valid implementation time in its own children.

CPU Linewise Panel

A line-wise panel shows in detail the time that each line of source code uses. The chart is implemented using Altair and sent back to the front-end in Vega. The chart is divided into two different views.

The first one is a statistical view of bar chart. The time spent is plotted as bars both in function level and line-number level. This chart gives the user a first impression on the numbers, without leveraging too much information on the semantic of the source code. The user can interact by clicking on a bar in the function-level plot, and the bars with line numbers corresponding to the function will be highlighted.

The other one is a realistic view of the source code. This plot displays the source code input by the user line-by-line, with the time percentage encoded as color of the background of the line. We implement this panel by layering a stacked rectangular plot with the source code lines as a text plot. The time percentage is encoded as opacity in the colored rectangles, so the more time spent in that line the deeper the color is. A mouse-over highlight interaction is introduced help the user concentrate on a certain line of code. To prevent color overlapping, another layer of non-filled rectangles are used to implement it.

In the realistic view, there is another transparent layer hiding a number of non-filled block covering the code block of each function. These rectangle function frames do not show themselves by default, and serve for the coordination with the call graph, which will be discussed in the next part.

Graph-Line Highlight Coordination

We design and implement a coordination between the CPU Call Graph and the realistic view of CPU Linewise Panel. If the user clicks on a node, the corresponding function block of the source code will be highlighted by a non-filled rectangle. It is a useful interaction for the user to navigate on the call graph and narrow down to a certain range of source code. As for implementation bridging GoJS and Vega, we register the event-listeners on nodes in the graph, which triggers a change of the color(opacity) in the corresponding rectangle covering that function block, noted in the section above. Considering the user may pay attention to callers and callees, we also provide the option to highlight callers and callees in different colors.

Mem Linewise Panel

A line-wise panel shows the detail that what memory bugs does each line of the source code has. The code line with error will be highlighted in the code. There are three different line-wise panels that refer to three kinds of memory errors respectively: forget to initialize the allocated memory, read/write

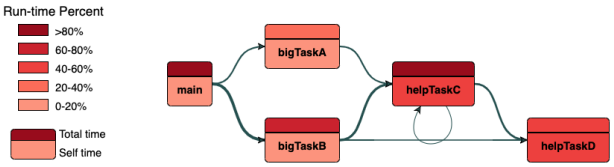

Figure 2. This complete call graph is the beginning view and acts as a global context. The upper color encodes total implementation time and the bottom color encodes self implementation time.

not allocated memory and memory leak. This is a useful visualization for the user to find their program's memory error. To help the users get more detailed information, user may also see the error type and leak memory bytes when hovering on the highlighted line. For the implementation, we wrote a text processing program to deal with the multiple types of outputs generated by Valgrind, and get the useful information from the output.Based on the information we get, a chart is implemented by Altair and sent back to the front-end in Vega.

RESULTS

Call Graph

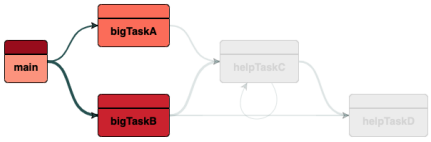

Figure 3. (a) First expansion. Hided nodes shown in light gray and collapsed nodes would possess all its children's time.

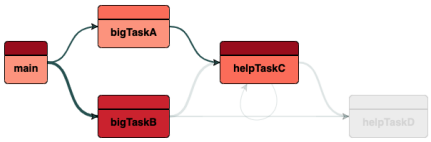

Figure 4. (b) Expand bigTaskA. Time partially flows into helpTaskC. BigTaskA's self implementation time is reduced, and the color is lightened.

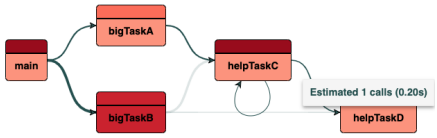

Figure 5. (c) Continue expanding helpTaskC. It only has partial valid time and therefore time flowed to helpTaskD is estimated.

As shown in Figure2, the time percentage of one function is encoded by color, and we split a node into two halves to encode both total time (including child and grandchild time) and direct time (self implementation). The bottom part is relatively larger, because we assume direct time is of more concern to users. Time flowed through a call is encoded in the width of the corresponding edge.

The expanding process is layer by layer, while the collapsing procedure pushes down for all its children and grandchildren. The hided nodes are still drawn on the canvas in order to maintain a global context. User may drag and move certain nodes,

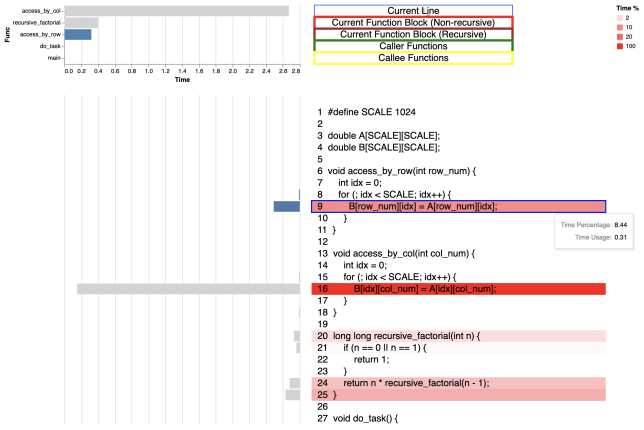

Figure 6. The concatenation of two different views of line-wise CPU usage, with interactions. The lines mainly contributing to the execution time can be easily found.

or freely zoom in and out, to explore into local details. Users may also see more detail information when hovering on nodes or edges. When no precise figure is available due to partial expansion, as shown in Figure 5, the tool tip information would remind that the time is estimated. Figure 3 to 5 show the process of successive expansions. From (a) to (b), only one parent for helpTaskC is expanded, so this node only has partial valid time and its child's valid time has to be estimated assuming a balanced time during different calls from helpTaskC.

CPU Linewise Panel and the Coordination with Graph

The CPU line-wise panel is shown in figure 6. The statistical view and the realistic view are aligned line-by-line. The tooltips and highlighting interactions aims the focusing on the certain lines.

Figure 7 illustrates the coordination of the call graph and code lines. When users click on a node in the call graph, the corresponding source code block and its callers and callees would be enclosed in the box. We handle recursive functions as a special case, and use a legend of colors to differentiate the highlighting boxes of different purposes.

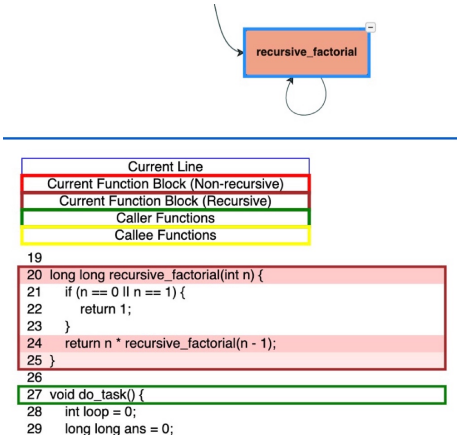

Figure 7. The result after clicking doTask in the call graph.

Mem Profiling Visualization

In this section, we will show the visualization result we get from a code snippet that has memory leak error(Figure 8). In this visualization code line 9 leads to a memory leak of 3600 bytes. By our memory profilings, the user can easily get which line does the error happen, the type of the error and the exact leak bytes.

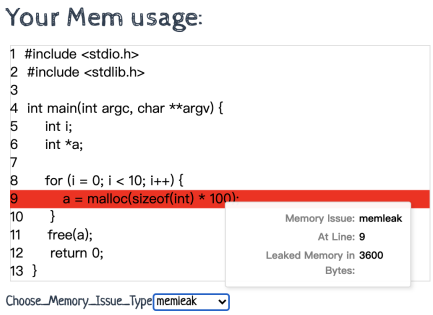

Figure 8. Memory Profile Example(Memory Leak)

DISCUSSION

In this section, we will make some discussion about how our system help beginner programmers to easily use the profiling result. In general, our application treats the results of traditional softwares as data, and interprets them in the way of data science.

The line profiling part is very straightforward and give the user a clear indication that what is the bottleneck for the code. The call graph visualization helps the beginner programmer to get a better understanding of structure for the code. And the collapse and expand technique allow the user to focus on some specific part of the program. The clear and easy to use interaction also avoid the users from getting lost.

The memory profiling part also helps the user to easily catch the error. The log generated by Valgrind is long, which is unfriendly to the beginner user. According to our observation to the beginners for programming(on course 11637 as TA), they are really bad at catch the bugs and meaningful information from long error logs. Instead, they only want to know which lines have bugs.

FUTURE WORK

Possible future work includes following ideas:

1. Support immediate code modification in the source code box of line-wise panel. This will improve the efficiency of the users.

2. Support more programming languages. E.g., we would like to support Python first.

3. Add tuning parameter as advanced option for users. For example, we will allow the user to define the profiling granularity of the profiling.

REFERENCES

[1] Andrea Adamoli and Matthias Hauswirth. 2010. Trevis: A Context Tree Visualization Analysis Framework and Its Use for Classifying Performance Failure Reports. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS '10)*. Association for Computing Machinery, New York, NY, USA, 73–82. DOI:http://dx.doi.org/10.1145/1879211.1879224

[2] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 120–126. DOI: http://dx.doi.org/10.1145/800230.806987

[3] Katherine E. Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. 2014. State of the Art of Performance Visualization. In *EuroVis - STARs*, R. Borgo, R. Maciejewski, and I. Viola (Eds.). The Eurographics Association. DOI: http://dx.doi.org/10.2312/eurovisstar.20141177

[4] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. DOI: http://dx.doi.org/10.1145/1273442.1250746

[5] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (May 2006), 287–311. DOI: http://dx.doi.org/10.1177/1094342006064482

[6] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. 2004. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *Computational Science - ICCS 2004*, Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 440–447.

[7] B. Wylie and M. Geimer. 2011. Large-scale performance analysis of PFLOTRAN with Scalasca.