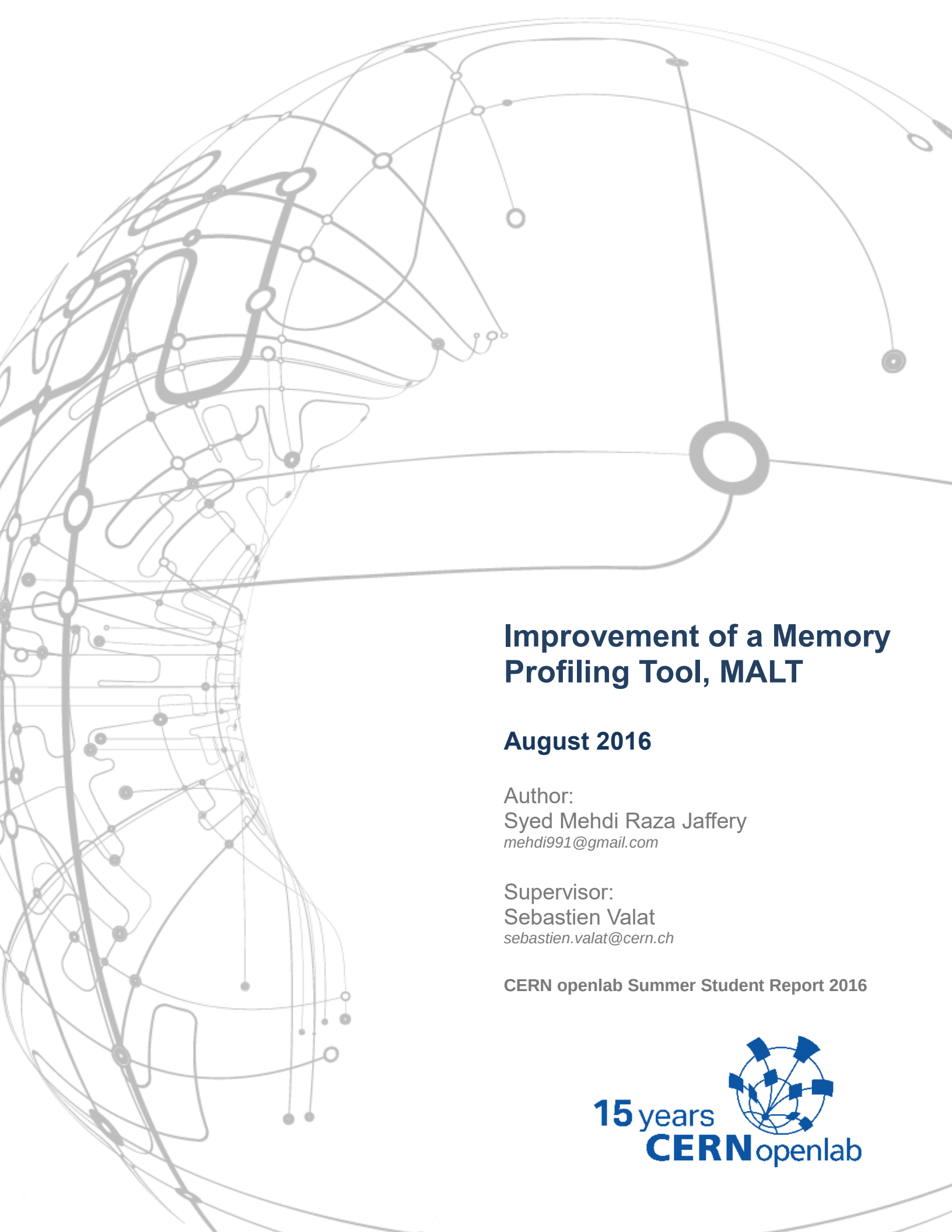

Improvement of a Memory Profiling Tool, MALT

August 2016

Author:
Syed Mehdi Raza Jaffery
mehdi991@gmail.com

Supervisor:
Sebastien Valat
sebastien.valat@cern.ch

CERN openlab Summer Student Report 2016

15 years CERN openlab

Project Specification

This project focuses on improving the web-based user interface for the *MALT* memory profiling tool. In particular, it introduces a new and improved source code viewer which shows annotations for memory statistics on memory allocating lines on the source code. Also, it adds a new call-tree viewer to the interface which allows the user to browse through the function calls in the profiled program to see where memory is being allocated and freed and exactly how much.

Abstract

MALT is a memory profiling tool designed for profiling programs on the Linux platform. MALT generated profile data can be viewed in the *MALT GUI* which is a web-based program for visualizing data and exploring code.

To further the development of MALT GUI, we improved its *Source Code Viewer* by redesigning the page and replacing the code editor. We compare several syntax highlighters and code editors and describe why we used Prism.js library for syntax highlighting plus its advantages such as being faster, leaner and easier to extend. We also discuss the performance gains and reduction in page load time gained by these changes.

Also, we discuss the addition of a new feature to the MALT GUI called a *Call Tree Viewer* to generate and view Call Graphs with memory statistics. We compare several graphing libraries for their suitability to make a call graph and why we selected Graphviz for MALT GUI. We discuss the implementation of call tree for transforming and filtering the data before it is rendered and shown to the user.

Table of Contents

1 Introduction

1.1 MALT

High Energy Particle (HEP) physics experiments use big software stacks to transport and analyze terabytes of data going out of the detectors. Most of these are written in C++ and use lots of memory. In this context, it is interesting to profile the memory usage of these applications to search for improvement. Hence, *MALT* (MAlloc Tracker) was developed as a memory profiling tool to track memory allocation and deallocation calls and map them onto the source code for any C++ application including those used in HEP experiments such as the LHC experiment. It provides the same approach as *Valgrind/KCachegrind* except for memory profiling.

MALT uses a C++ backend to instrument the program and track memory allocations and life cycle. MALT works by intercepting the calls the program makes to the standard library for allocating and deallocating memory to gather statistics about how much memory is being used and where and when.

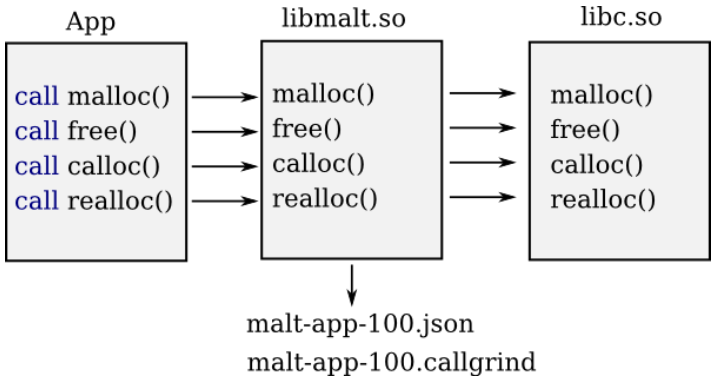

Figure 1 MALT intercepts allocation calls using Linux's LD_PRELOAD. [1]

The statistics regarding the profiling run is saved as a JSON file which can then be read into MALT's web-based *GUI* (Graphical User Interface) tool for visualization and exploration. The *JSON* (JavaScript Object Notation) file format is used as it is a common data exchange format and has wide spread support in libraries across different languages.

1.2 MALT Web-based GUI

MALT also provides a web-based graphical user interface tool to view the memory profile data created. The GUI tool reads from the JSON file and serves a website locally that the user can open in the browser to view the profile data. MALT GUI tries to present data as visually as possible using graphs and tables so that the user can get the most relevant information about where memory is being used. Global metrics are also provided as an indicator of the program's overall memory consumption along with memory timeline to provide temporal data about memory allocation and to show patterns.

The web server for the web-based GUI is written on *Node.js* and uses the *Express framework*. It exposes a *REST* API along with *HTTP Basic Auth* as a basic security mechanism for authentication. The front-end user interface is made with the *bootstrap* CSS framework and uses the *AngularJS*

framework. *D3.js* is used to provide interactive visualizations such as line charts, pie charts, etc. D3.js is used because it is the most versatile visualization library for JavaScript.

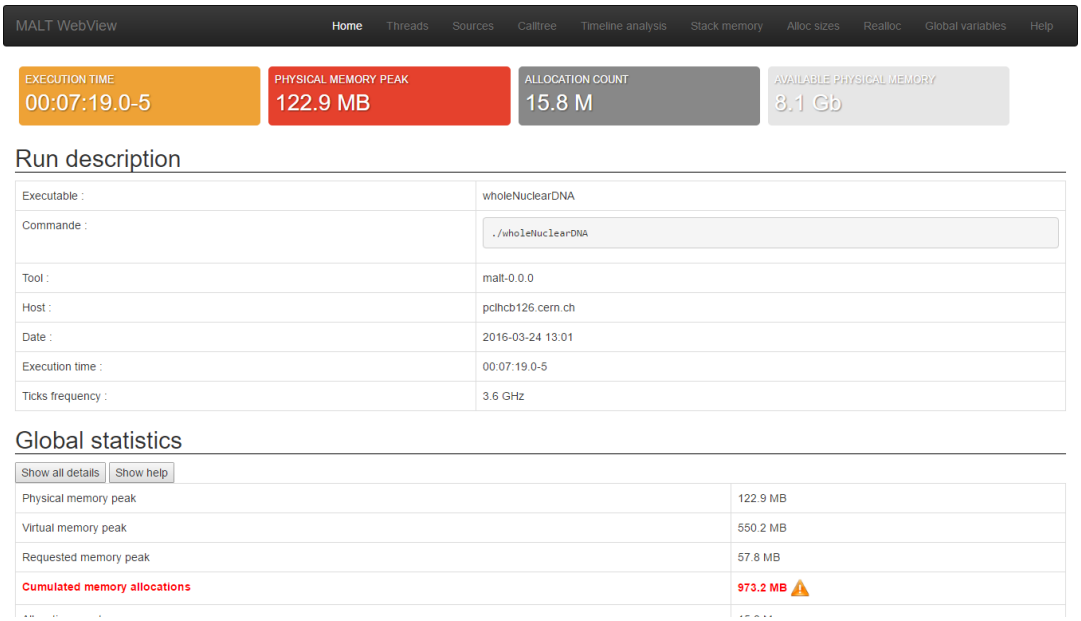

Figure 2 MALT Web GUI main page.

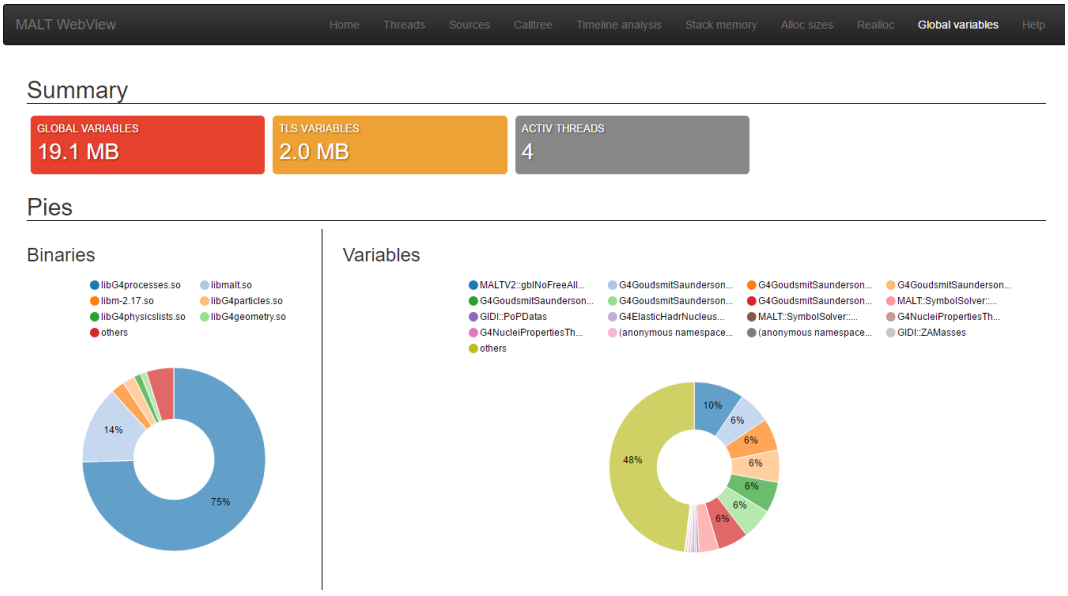

Figure 3 A page showing the memory consumed by global variables.

1.3 Data Visualization

The Malt GUI uses D3.js to provide interactive charts for visualizing data. D3.js provides dynamic binding of data to the charts that are actually SVG drawings. The programmer has full control over how the chart looks like. D3.js provides helping functions for laying out graphs and generating axes along with a large number of other utilities.

Data visualizations in MALT GUI include memory allocated per thread, memory allocated over time, most used allocation sizes, global variable memory usage and etc. The types of charts include line chart, bar chart, pie chart and histogram. To simplify code and to reduce boilerplate for some common types of charts, in some places, *NVD3.js* library was used to produce charts. NVD3.js still depends on D3.js but reduces the amount of code we have to write but at the cost of control over what features we have in the charts.

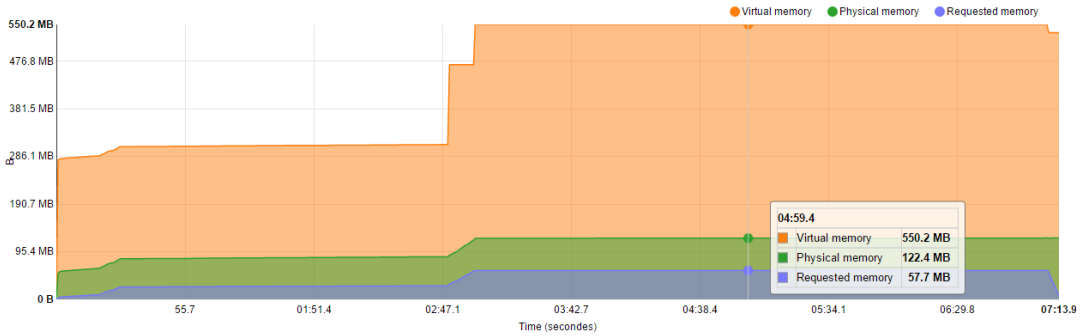

Figure 4 An example of a graph from MALT GUI: memory allocated over time.

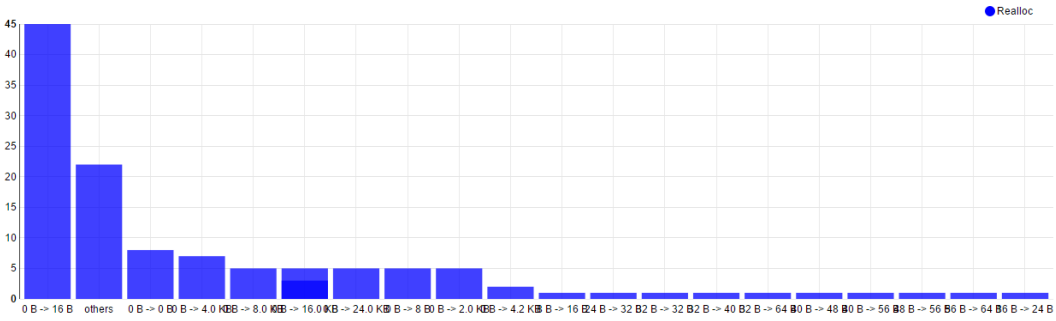

Figure 5 An example of a graph from MALT GUI: most used realloc sizes.

1.4 Improving the MALT GUI

The purpose of this project was to improve MALT's web-based GUI with new features or by fixing existing features and removing bugs. For this purpose, we explored several features that could use work and came up with three that could add immediate impact to the project. These three were:

- An improved source code viewer
- A call-tree viewer
- A trace visualization to show temporal data *(not covered by this report)*

The next few sections detail what these features do, how were they implemented and the problems faced while implementing them.

2 An Improved Source Code Viewer

2.1 Previous Solution

MALT GUI used the *CodeMirror* code editor to show the source code which was annotated with the memory allocation details using the markers feature provided by CodeMirror editor. CodeMirror is a very powerful open source code editor and has been used in popular projects such as in the dev tools for Firefox and Chrome, Light Table, Adobe Brackets, and many other projects. It comes with a lot of features such as syntax highlighting, auto completion, support for hundreds of languages, key bindings, etc.

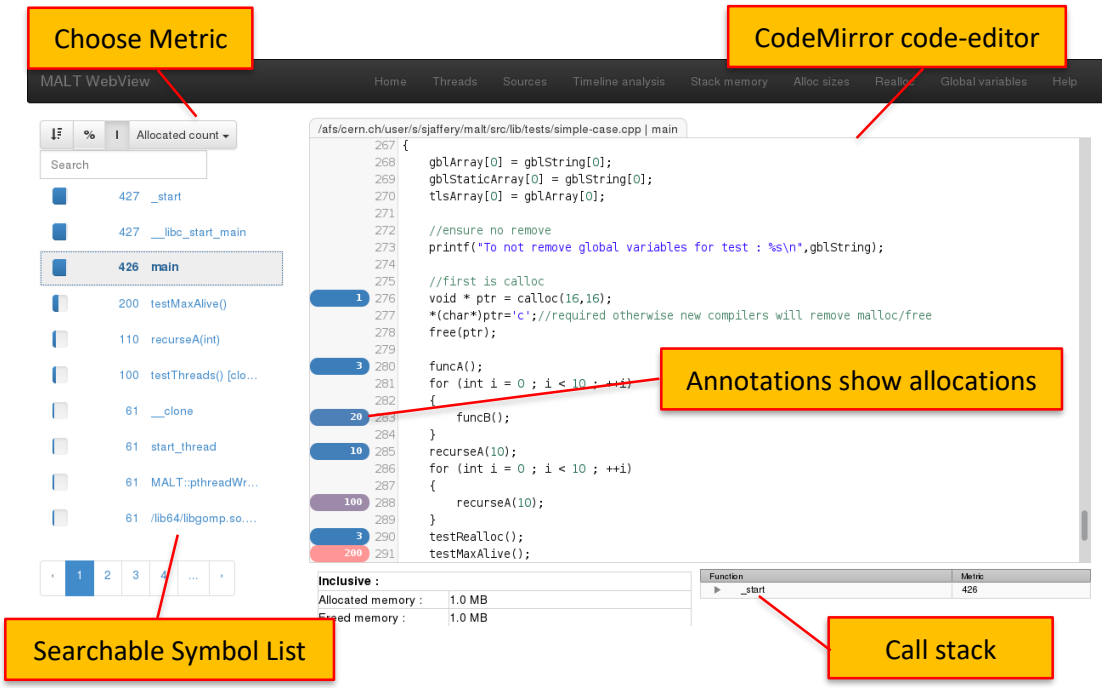

Figure 6 CodeMirror code editor used in MALT GUI Code Viewer page.

However, there were several problems with the existing integration of CodeMirror in the MALT GUI:

- The jump to line number feature did not work well with CodeMirror. MALT GUI uses this to navigate the user to the exact line where allocation is happening when the user clicks on a function name in the GUI. This saves the user's time by not requiring him to scroll manually all the way to the correct line. Note that files can be sometimes very large, running into thousands of lines. So such a feature is necessary to allow easier navigation to memory hotspots in source code.
- MALT GUI needed only a syntax highlighter for its source code viewer. Code is not allowed to be edited so all the code editing features that come with CodeMirror were actually all extra features that were never used anywhere in MALT GUI. This means that the MALT GUI loads a large amount of extra JavaScript and CSS for features that it never uses on the Source Code Viewer page, making the page slower and the initial loading time large. Indeed, even scrolling a large code file sometimes causing stutters in the browser windows, indicating some serious performance issues.

Hence, a need was felt to replace CodeMirror with something lighter, faster and better suited to our requirements. Need that the project only needed a syntax highlighter so both source code editors and syntax highlighting libraries were applicable.

2.2 Evaluating Alternate Solutions

We evaluated several alternate code editors and syntax highlighting libraries for JavaScript. Our focus was on speed, ease of integration and suitability to MALT's use case. Two candidate solutions that seemed appropriate were *ACE Editor* and *Prism.js* library. They are discussed in more detail below.

2.2.1 ACE Editor

ACE Editor is a full-featured Code Editor maintained by Cloud9 IDE and Mozilla. It has support for a large number of languages as well as features that are found in popular editors such as Sublime Text. One big advantage of this editor is that it is very performant, even with large files. Also, ACE Editor is very easy to integrate into an existing application, thanks to its easy to use API.

```
//declare metrics
var maltMetrics = {
    'alloc.sum': {
        name: 'Allocated mem.',
        extractor: function(x) {return x.alloc.sum;},
        formalter: function(x) {return maltHelper.humanReadable(x,1,'B',false);},
        defaultOrder: 'desc',
        ref: 'sum'
    },
    'alloc.count': {
        name: 'Allocated count',
        extractor: function(x) {return x.alloc.count;},
        formalter: function(x) {return maltHelper.humanReadable(x,1,'',false);},
        defaultOrder: 'desc',
        ref: 'sum'
    },
    'alloc.min': {
        name: 'Min. alloc size',
        extractor: function(x) {return x.alloc.min;},
        formalter: function(x) {return maltHelper.humanReadable(x,1,'B',false);},
        defaultOrder: 'asc',
        ref: 'max'
    },
    'alloc.mean': {
        name: 'Mean alloc size',
        extractor: function(x) {return x.alloc.count == 0 ? 0 : x.alloc.sum / x.alloc.count;},
        formalter: function(x) {return maltHelper.humanReadable(x,1,'B',false);},
        defaultOrder: 'asc',
        ref: 'max'
    },
    'alloc.max': {
```

Figure 7 Viewing a sample code in the ACE Editor.

However, we saw some issues with integrating ACE Editor as the source code viewer in MALT GUI:

- ACE Editor does not provide any support for adding markers to the editor gutter. We need this feature to be able to annotate the source code with memory allocation statistics. Writing a custom plugin for this is possible but would require extra effort and time.
- ACE Editor is one again a full-blown editor with lots of extra features. While some features might cover future use cases for MALT GUI but the fact remains that they are extra features and adds to the page load and hence the loading time. A simpler, lighter solution might be better.

2.2.2 Prism.js

Prism.js is a lightweight, extensible syntax highlighter that is with modern browsers in mind. The whole code including the style sheets just weight in under 5 KB. The syntax highlighting is done via regexes. The code for highlighting is very short so it can be edited to add new features if needed.

Prism.js is used as the default syntax highlighter in the Drupal CMS as well as many other famous blogs and websites.

Prism.js also provides a plugin approach to adding features so that the core remains small and all extra features such as line numbers, etc. are added as plugins. However, we could not find a plugin for adding markers to the gutter in the highlighted code. Also, there was no version of Prism.js for integrating with an existing AngularJS code base.

Figure 8 Prism.js syntax highlighter in action.

However, after weighing the benefits provided by the small size of the syntax highlighter over the missing features, we concluded that it would best suit our needs and the missing features can be added as new plugins by ourselves.

2.3 The New Source Code Viewer

MALT GUI now uses the Prism.js syntax highlighter. We have patched the library to add our plugin for adding markers or annotations to the highlighted code, which was only possible because of the small size of the Prism.js code base. We also use the line numbering plugin and a different theme, which can be downloaded from Prism.js official website. We use the D3 colour plugin to calculate colour for annotations. The annotations we add to the code viewer are colour coded to show the weight of the statistic. Red means a big value and blue means a small value.

We felt a visible difference in the speed with which the Source Code Viewer page loaded once we made the transition. Scrolling the page also felt snappier than before. This is all because how lightweight the new code for highlighting is. Jump-to-line feature now uses a smooth, animated scroll to the line, further adding to user's perceived page performance.

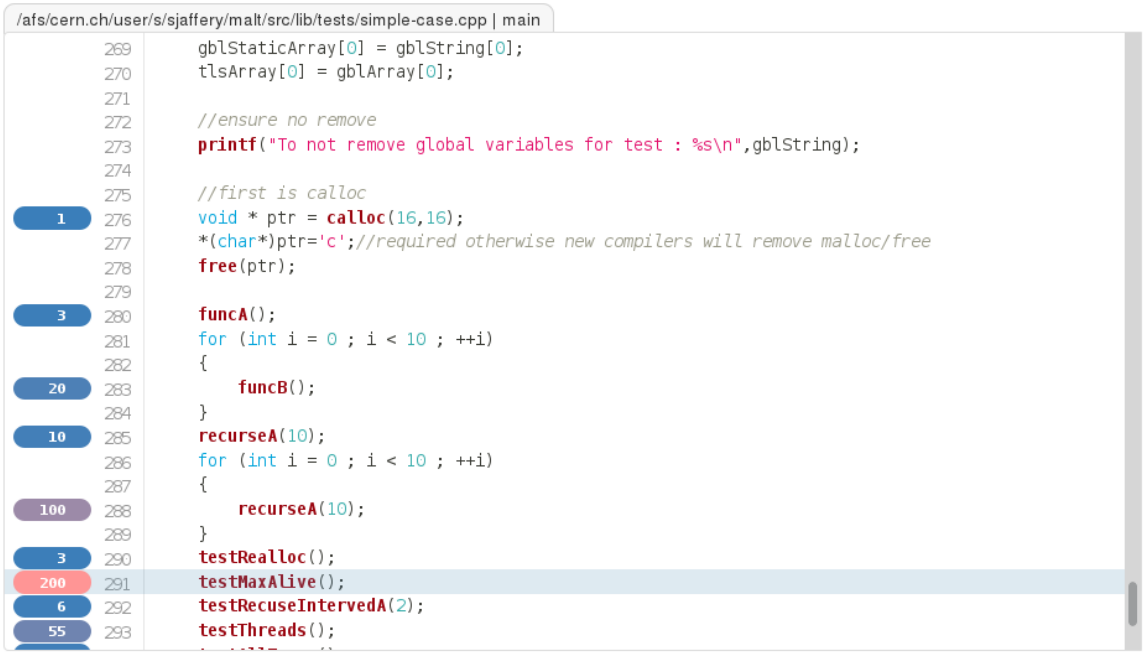

Figure 9 MALT GUI's new Source Code Viewer based on Prism.js.

2.4 Redesigning the Symbol List

The Symbol List on the source code viewer page is a way of navigating through interesting functions as the Symbol List allows ordering the functions by the number of allocations, size of memory allocated, memory freed, etc. The metric for ordering the list can be selected from a drop-down box and results in updating of the annotations on the Code Viewer too. User can quickly navigate to the function making the most number of memory operations or causing the largest amount of memory leak quickly.

However, the existing Symbol List had several design issues:

- The function names did not fit within the width of the symbol list and hence were truncated. This made identifying functions difficult.
- The list design was such that it wasted too much space for the metric value and the metric progress bar. This space can be used in other places.
- The code for the Symbol List was repeated on two pages as they also use the list.

Hence, to fix the issues, we refactored the code, first separating the logic for the Symbol List into a new directive for AngularJS to allow reusability. Secondly, we redesigned the list into a more compact form, stacking the metric value and metric progress bar over each other and by reducing the font of the text. Also, we now shorten the function names, hiding arguments and type parameters if the name is too long and if that does not work, we resort to truncating the name but we show a popup with the full name if the user hovers mouse over the function.

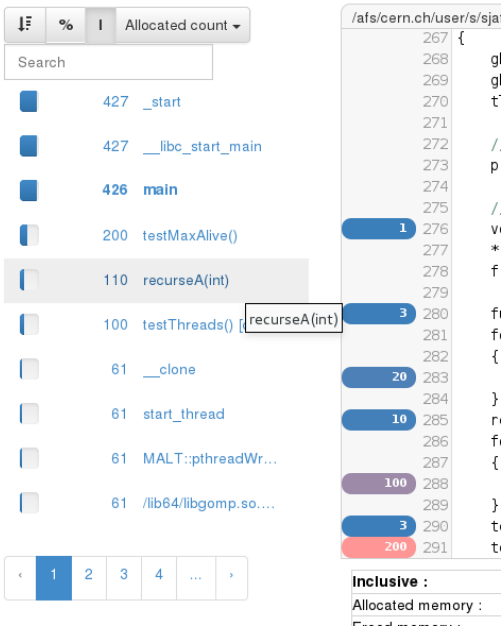

Figure 10 The old Symbol List.

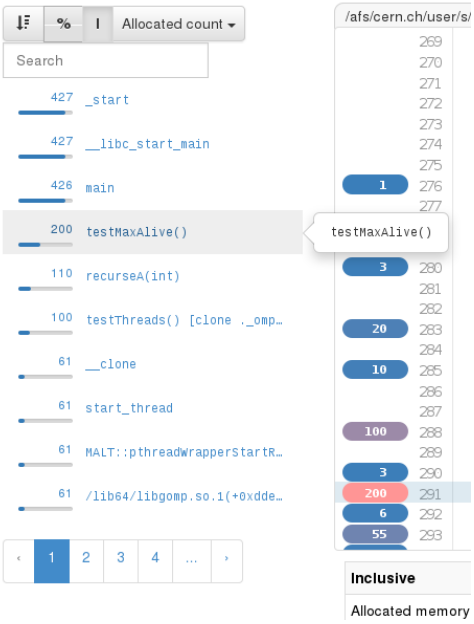

Figure 11 The redesigned Symbol List.

2.5 The Final Page

The final Source Code Viewer page looks like this after the syntax highlighter was changed and the symbol list was redesigned.

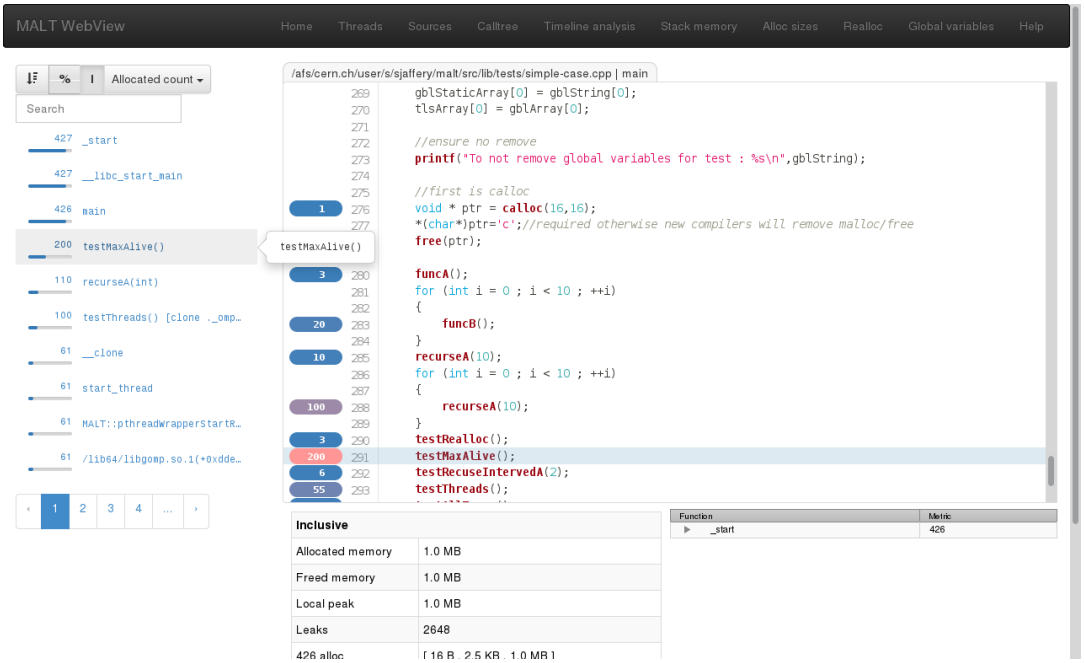

Figure 12 Final version of the Source Code Viewer page.

3 Call-Tree Viewer

3.1 Introduction

MALT provides users with the ability to explore stack traces and see the allocations that are happening with each function. This feature is part of the Source Code Viewer page where a stack trace is provided when a particular line or function is selected. Stack trace shows a hierarchical view of function calls showing the history of callers for a particular function.

However, what if a user wants to explore the hierarchy in a top-down approach, navigating from the first function called in a program down to the last where memory allocation is happening? Such a case can be handled by making a *Call-tree* (also known as a *Call-graph*), a graph showing function calls in the form of a hierarchical *network graph*, with each node representing a function call and edges between the nodes representing a call. If annotated with memory statistics, this graph would make it easier for the user to discover new unexpected places where allocation is happening and see the whole hierarchy for the call.

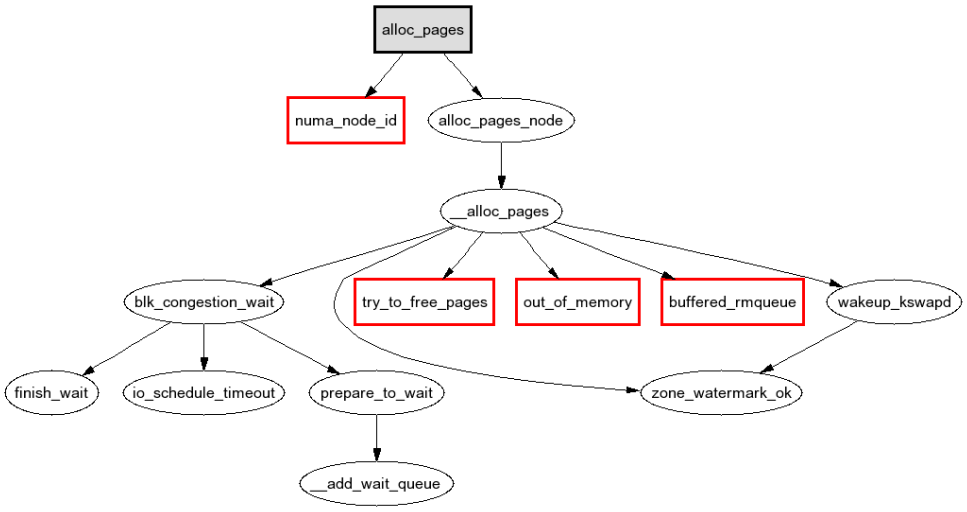

Figure 13 A sample Call graph of alloc_pages() in the Linux Kernel 2.6.12-rc2 made using CodeViz [2]

3.2 Requirements from a Memory Profiling Tool's Perspective

A useful call tree tool from a memory profiling perspective would have the following features:

- Ability to see memory statistics for a particular function in the call tree. Statistics can be memory allocations calls, memory usage, allocation rate, etc. Useful options would be to see inclusive and exclusive memory statistics for a function.
- Ability to visualize large graphs because call graphs for long running programs or for large code bases can often run into thousands of nodes.
- Ability to navigate or view a large call graph while retaining information usability.
- Ability to visualize graph in such a way that the time to find memory allocation hotspots is minimized. Also, user should be able to follow a hot region to find the real source of allocations or the original caller.

- The call tree tool should handle hierarchies where there are more than one roots, which can be caused by multithreading in a program. Other issues include recursive calls, and function calls from multiple sources and from multiple threads.
- Ability to filter data to see only the most useful information.

3.3 Existing Solutions

We evaluated existing tools for memory profiling that can generate call graphs.

The most popular tool currently is *Memgrind* which when used with KCachegrind can create call graphs for memory data. KCachegrind is a GUI tool that visualizes data generated from Valgrind based tools such as Callgrind and Memgrind. This is the closest to the solution that MALT GUI hopes to achieve but KCachegrind misses out on many points due to its support for only a limited set of statistics.

Other profiling solutions such as *Parasoft Insure++* and *PurifyPlus* do not have any free versions that we can evaluate. They do not provide screenshots or documentation of the call graph feature on their websites either. The lack of tools for memory use visualization only show how important this feature is for MALT GUI.

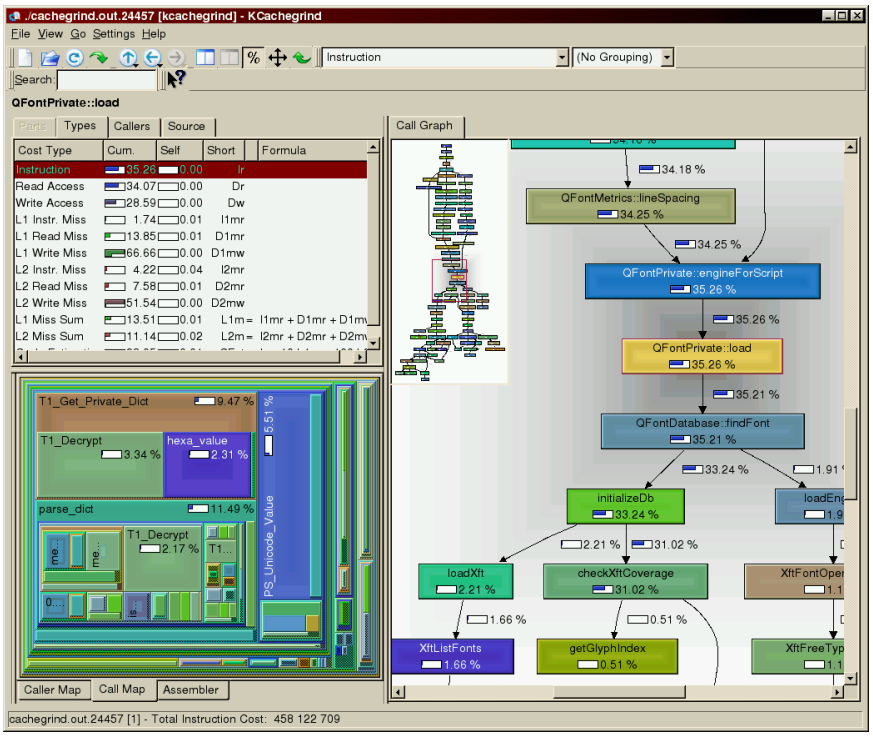

Figure 14 Screenshot from KCachegrind showing the call graph besides other graphs. [3]

3.4 Evaluating Alternate Solutions

In this section, we evaluate graphing solutions for implementing the call tree in MALT GUI. Our focus was on a graphing tool or library that would integrate easily with MALT GUI and fulfil the requirements that we have mentioned earlier.

3.4.1 D3.js Force Directed Layouts

D3.js provides the ability to make network diagrams in general using its *Force Directed Layout* diagram feature using its *d3-force component*. It calculates the position of each node by simulating an attractive force between each pair of linked nodes, as well as a repulsive force between the nodes. After that, we can draw the calculated graph with custom node design and links.

Although this is a very powerful D3.js feature, it does not fit our use case well as we intended to show a hierarchical graph instead of a general one.

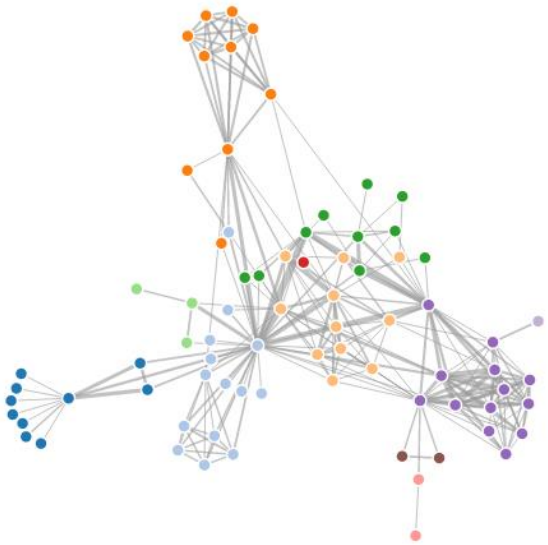

Figure 15 A sample Force Graph showing character co-occurrence in Les Misérables. [4]

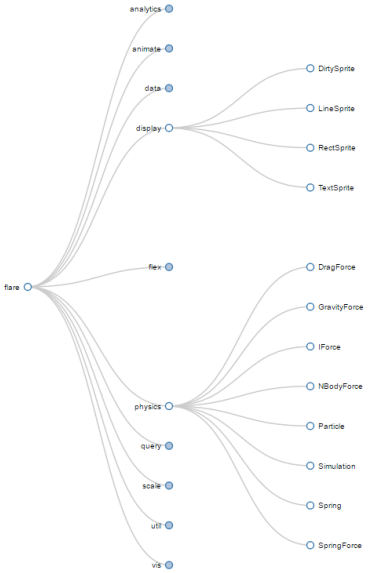

Figure 16 Tree layout made using D3.js. Nodes can be expanded or contracted by clicking. [5]

3.4.2 D3.js Tree Layout

D3.js has a component for visualizing hierarchical data called *d3-hierarchy*. This can be used to create a tree layout using data for nodes and edges. By default, it will calculate layout for a graph to be layout horizontally. We experimented with this but ran into issues such as nodes overlapping as it seems that D3.js does not take the content of the nodes into consideration when calculating the node position. Also, there is no support for multi-parent nodes and cycles in the tree, which are both needed for making call graphs. However, given D3.js's flexibility, a solution could have possibly been made for these problems but at the cost of time.

3.4.3 Vis.js Network

Vis.js is a JavaScript visualization library that aims to be easy to use, provide interactivity with data and be able to handle large amount of data. Their network diagram seemed to be most suited to our work. Vis.js features such as custom colours, node shapes, node content, labels and layouts suited our various requirements. Vis.js also provides an API to add data dynamically to the dataset after it has been rendered. Vis.js network diagram understands the hierarchy of nodes and takes it into consideration when rendering the graph.

Hence, we tried out Vis.js with sample data from a memory profile to see how it performs.

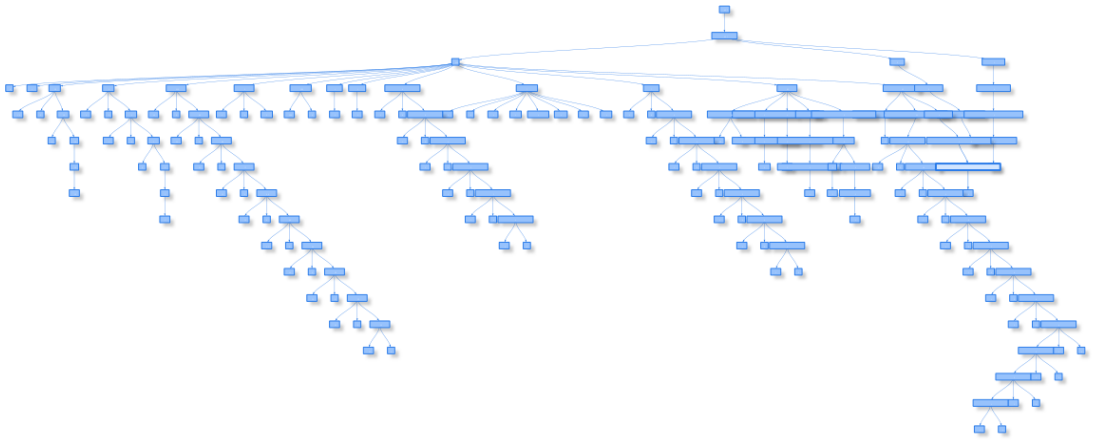

Figure 17 Vis.js network used to create a call graph for a test C++ program.

However, Vis.js does not take content of node into consideration when calculating node positions. This makes some nodes overlap and some nodes to be too far apart.

Overall, Vis.js was the closest solution to our problem that met many of our requirements out of the box. We could zoom into the graph, move it around, colour the nodes and add custom content to nodes.

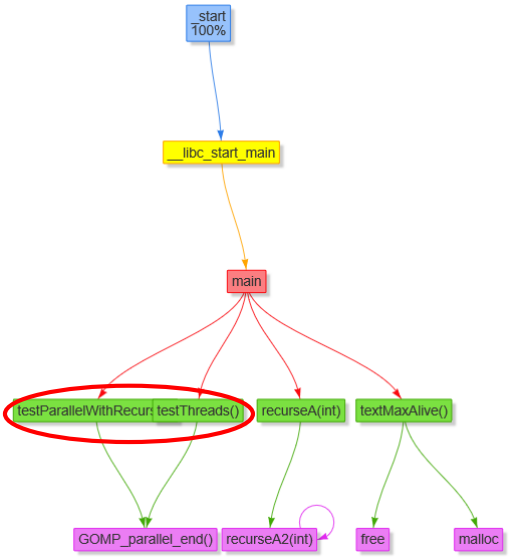

Figure 18 Nodes overlap as Vis.js fails to take node content into consideration for layout.

3.4.4 Viz.js

Graphviz is open source graph visualization software. It has important applications in networking, bioinformatics, software engineering, database and web design, machine learning, and in visual interfaces for other technical domains.

Graphviz comes as a standalone program and is available on most Linux system from the default package repository making it widely available. Graphviz meets many of our requirements as it directly has support for hierarchical diagrams with a lot of features that can be used by using Graphviz's *Dot language*. The Dot language is a *DSL* (Domain Specific Language) that can be used to describe graphs along with content, styling and layout hints. Graphviz takes a Dot file and produces a graph as an image output in any of the common image formats. However, since Graphviz is a C++ program, no version is directly available for the browser so we used a version of Graphviz, known as *Viz.js*, that was compiled from C++ into JavaScript along with slight modifications to run it on the browser. Our code first produced a Dot code description of the call graph which was then passed onto Viz.js that generated an *SVG* (Scalable Vector Graphics) output which we then showed in the page.

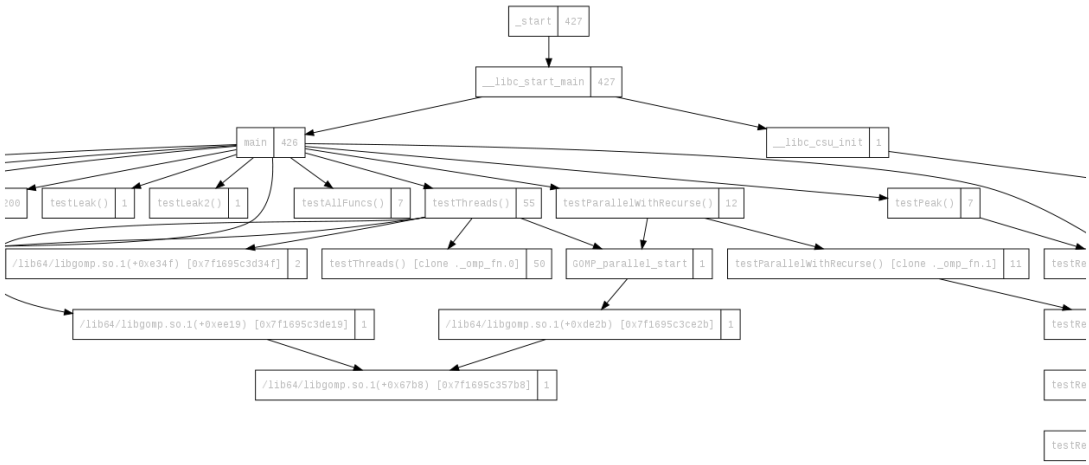

Figure 19 A portion of the call graph generated using Viz.js.

Thanks to Viz.js, we were getting a neat diagram and were quickly able to integrate memory statistics such as memory allocation count as shown in the diagram above.

However, tests with large graphs showed that Viz.js slowed down considerably with the size of data. In some cases, Viz.js took so long that the browser had to close the script to recover the tab. Also, doing all filtering and rendering of data on the client-side made the page very slow to load.

3.4.5 Graphviz

Instead of using Graphviz on browser side, we decided to use the original compiled version on the server side to generate the SVG, and send the SVG to the browser side to render it and to add interactivity to it. This was easy as we already had the Dot code generator from our previous experiment. This time we added the ability to colour nodes as well to our Dot code generator.

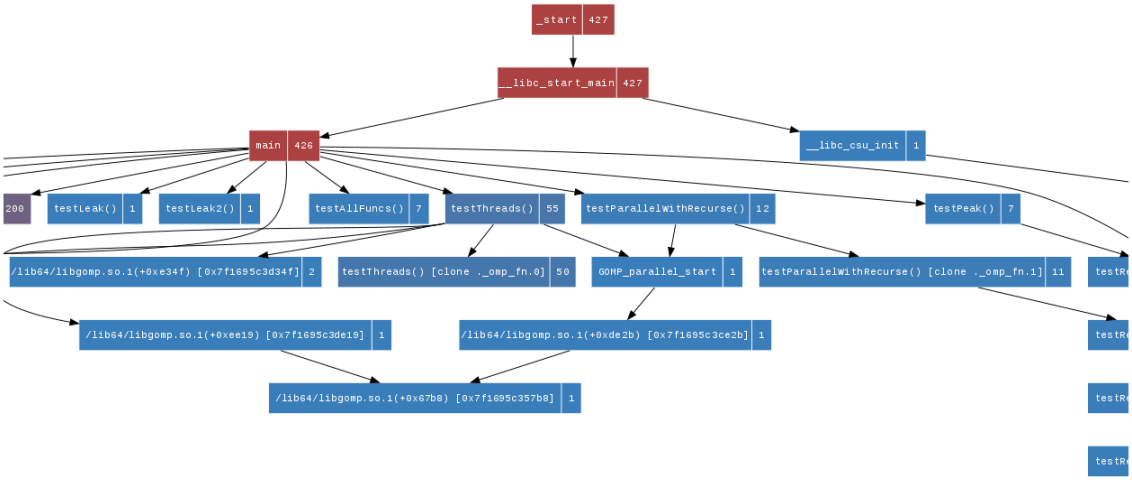

Figure 20 Same graph as before, generated on server-side and displayed in the browser.

Initial tests showed vast improvements in speed which can be understood was Viz.js was an inferior compiled-to-JavaScript version of Graphviz. We also tried rendering large graphs which did render

but took 30-40 seconds which is still understandable given we had ~3000 nodes. Overall, this seemed to be the closest solution to our problem; hence, we decided to use it.

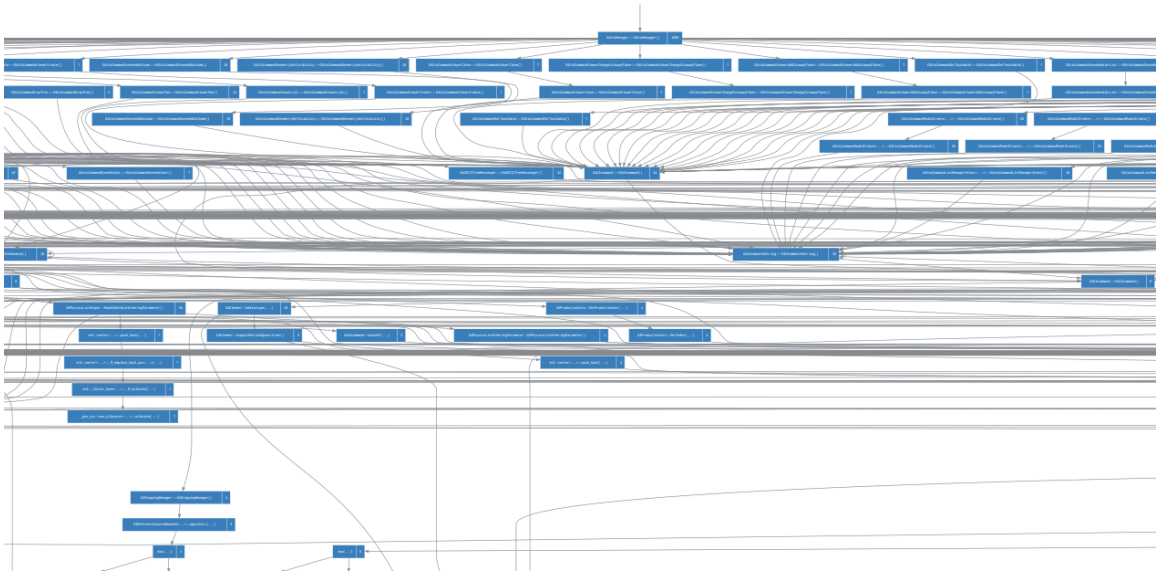

Figure 21 A small portion of a ~3000 node call tree for a Geant4 Simulation memory profile.

3.5 Filtering and Navigating

Since our graphs can grow to be quite large due to the size of some programs, we added features to filter out nodes from graphs to only show those that are making a significant impact on the memory. The user can choose a *Node Cost* which is a percentage to determine what should be the minimum impact for a node to be included in the graph. This removes nodes with zero allocations or too small an allocation to be considered.

Also, the user can choose to limited the depth of the graph by selecting the number of levels that should be visible. Same applies to the height of the graph. To allow navigating through the graph, the user double clicks on a node to *select* it causing the graph to re-render to show all ancestors and descendants of the selected nodes with the given filters applied. This way of filtering and navigating through the graph allows user to go through an arbitrarily large graph without slowing down the application. Note that node can also be selected by searching it in the Symbol List on the left-nad side of the page. A table for the memory statistics of the selected node (function) is also shown at the bottom of the page besides the filter controls.

Any node can be right-clicked to bring up the context menu for the node which can to be used to select the node or to jump to its source. Clicking on the call tree in general brings up a context menu with option to export the call tree diagram as an SVG file or as a Dot file. Having the Dot file means the user can edit the diagram for his own purpose, making MALT Call Tree Viewer a powerful general purpose tool as well.

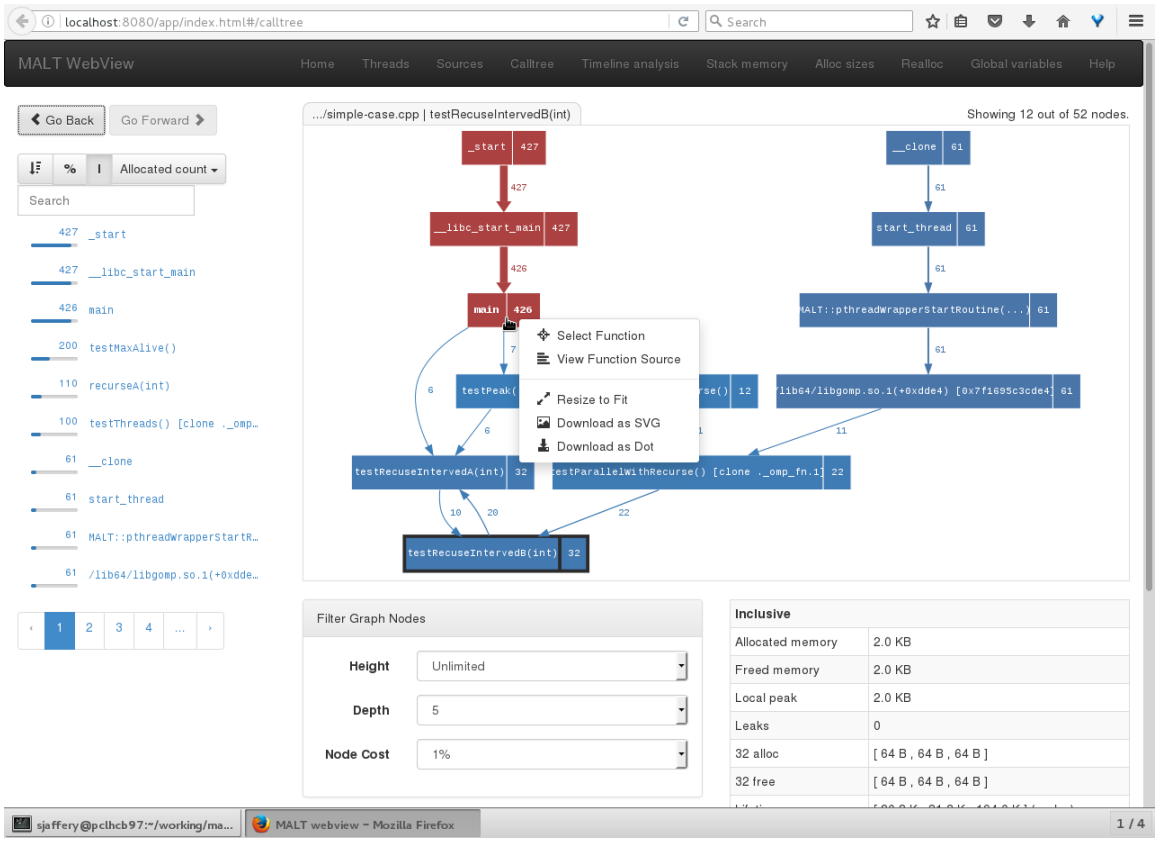

Figure 22 The completed Call Tree page. Notice the filter controls at the bottom and the Symbol List on the left-hand side. The context menu is also open in this particular screenshot.

3.6 Annotating the Call Tree

To make the call tree as useful as possible, we annotate the diagram with memory statistics such as memory allocation count in the sample graph shown below for a test C++ program. The value shown in the node is for cumulative memory allocations made by this function inclusive of its children (calls to other functions) throughout the program lifetime. The label on the edge shows the memory allocations that happened along this path.

The nodes and the edges are colour coded to show the relative weight of the memory allocations. Red means large and blue means small. Also, the thickness of the edges varies with the weight of the allocation happening along the path. All of this helps the user in quickly identifying hotspots or regions of high memory usage.

Note that the user can change the type of memory statistic that is used to annotate the graph from the metric dropdown box on the page.

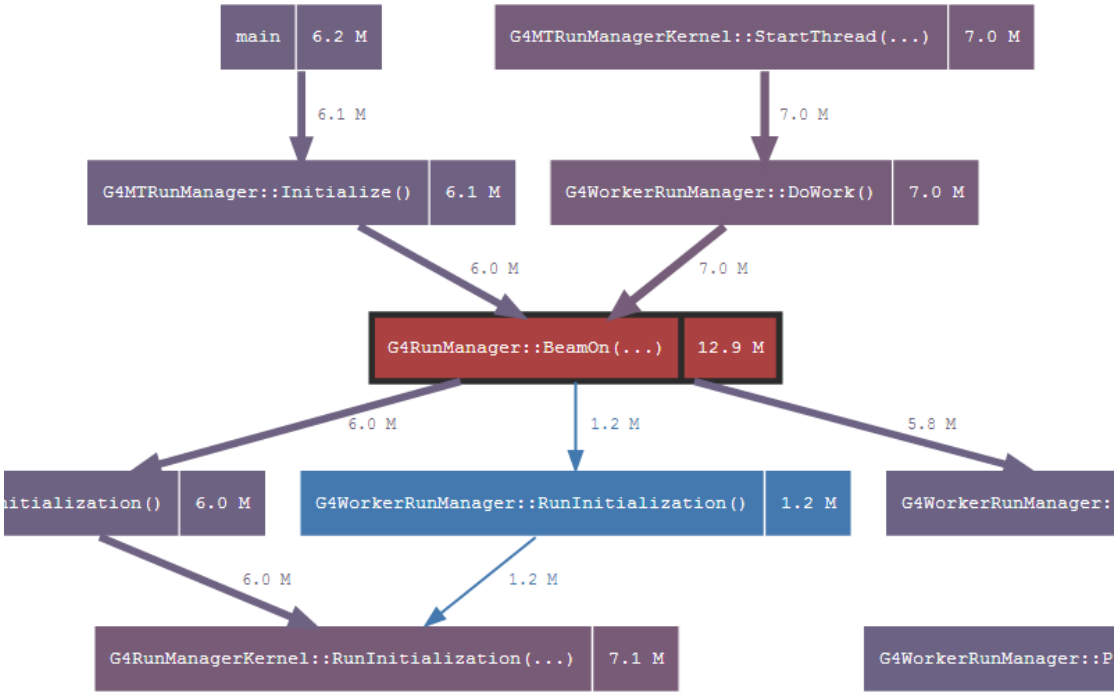

Figure 23 A close-up of a call tree generated using MALT Call Tree Viewer.

3.7 Data Flow

This section describes how original data from the memory profiler is transformed and used to create the call tree diagram.

1. The JSON profile file contains a record of all the stack traces for instances where memory allocations (or deallocations) were done. Each stack trace comes with complete statistics about the quantity or size of the allocations and the exact location of the code that does the allocation.
2. MALT GUI merges the stack traces together to build a hierarchical tree of function calls and allocations for each call. Merging functions across stack traces means their statistics have to be merged too (counts are summed, min and max are recalculated). MALT also sums up the statistics for edges. While merging, we need to take care of not duplicating nodes and to prevent double counting of statistics due to cycles in the graph.
3. Then MALT GUI uses the metric selected by the user to calculate the colour codes for all nodes and edges as well as the thickness of the edges.
4. MALT then converts this internal presentation of the call tree into the Dot code using a Dot Code generator which goes through all the nodes and vertices and generates a Dot code text file while also adding the styles to the code.
5. The Graphviz binary is then executed with the generated Dot code text file as input. The generated SVG file is then passed to the client (browser).
6. The client-side parses the SVG and adds it to the *DOM* (Document Object Model). The client then adds interactivity to the SVG element using JavaScript and DOM events, such as the ability to zoom in, pan the diagram, to select nodes and to select context menu.
7. In case the filter for the graph changes or a different node is selected, the MALT server needs to regenerate the Dot file from the call tree object (cached this time) and call Graphviz to regenerate the SVG file which is then sent to the browser for displaying.

4 Conclusion

In this project, we were able to add two new major features to the MALT GUI. Changes were made both to the client-side and the server-side code for the MALT GUI. The successful integration of the new Source Code Viewer made the page much faster and removed bugs that existed previously. Prism.js was selected as the highlighting library as it was a fast, regex based highlighter and its small size allowed us to make changes to its code base to add markers to the highlighted code to show memory related statistics.

The second feature that we added was the MALT Call Tree Viewer which used the stack traces captured in the profile data to build and render a call tree. Graphviz was used to generate call tree while custom code in Node.js performed the transformation and filtering of the data to fit the needs of rendering a call graph.

The next step will be to improve the performance of the existing MALT GUI, refactoring the code for the GUI, including making sure our code fits the paradigm of Angular.js. Redesigning of the GUI should be prioritized to make the UI easier to use and more accessible.

5 Citations

1. MALT: MALLoc Tracker, a presentation by Sébastien Valat at Concurrency Forum, CERN.
2. CodeViz official site: http://www.csn.ul.ie/~mel/projects/codeviz/
3. KCachegrind Screenshots page: https://kcachegrind.github.io/html/Shot3Large.html
4. Force-Directed Graph demo by Mike Bostock: https://bl.ocks.org/mbostock/4062045
5. Collapsible Tree demo by Mike Bostock: https://bl.ocks.org/mbostock/4339083