An Interactive Microarray Call-Graph Visualization

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

***Abstract*—In this paper we present an interactive call-graph visualization tool for viewing large programs. Our space-filling grid-based visualization shows the functions of a programs call-graph. The grid view provides an overview of all of the methods, allowing the user to investigate and view subsets of functions, and finally jump to source code for more details on demand. Our tool assists programmers by reducing large call graphs into smaller subgraphs with function relationships that matter for program comprehension.**

In our benchmarks, we view and explore code relationships in programs with 18,720 functions at interactive frame rates. We provide two use cases with several findings on investigating profile-guided optimizations in C++ and critical sections in concurrent Java programs . Our software visualization tool is Java based and portable across multiple platforms.

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

I. Introduction

Programmers often want to visualize the structure of a program to aid in program understanding. A new programmer joining a large software project can likely understand the architecture of software system more easily with an abstracted visual representation rather than reading hundreds of thousands of lines of source code. Experienced programmers working on legacy software are also aided by being shown which subsystems in a piece of software they should be modifying. In either case, when a programmer wants to visualize a program, extracting a call graph and visualizing it as a node-link diagram is often the first strategy. Node-link diagrams can be presented as a tree showing the hierarchy of functions calls, giving the programmer an understanding of the program structure. However, visualizing the entire program call graph and all the connectivity of functions as a node-link diagram does not scale well past several hundred or a few thousand nodes. Node-link diagrams are also limited in the amount of information that they can present, often limited to only presenting a function name within the node due to screen space constraints.

It is not uncommon for large programs to have tens of thousands of functions. The research question we are asking is how do we visualize call graphs of that magnitude, while still comprehending useful relationships between functions? In order to deal with the problem of scale, we have built a new visualization tool that can show all of the functions of a program in a single grid. Our visualization improves over node-link diagrams by hiding the connectivity information until a user requests to see it by selecting nodes when they want more details on demand. Our visualization then helps programmers find what matters by providing functionality to sort and encode nodes in an exploratory manner. As users find functions of importance our visualization adapts by hiding nodes in the grid that are not relevant. A final subgraph can be extracted as a node-link diagram which can be further analyzed when it is finally appropriate to have all connectivity information.

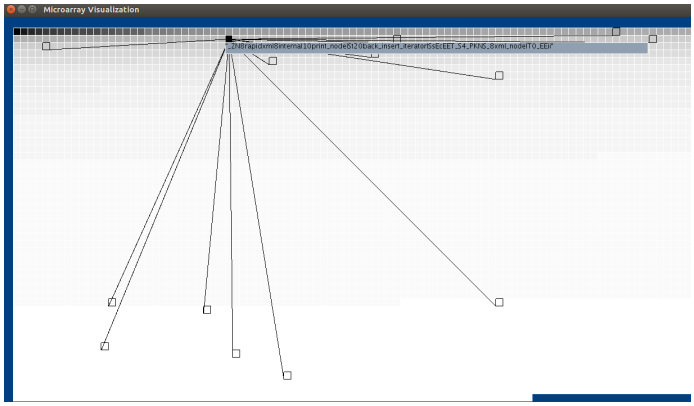

Fig. 1. Users are able to hover over a cell and see function callers and callees. They can then select the function of interest as well as any number of callers or callees in its path along its call tree.

In this paper we present an interactive exploratory call-graph visualization that can scale to viewing tens of thousands of functions while appropriately displaying connectivity information when requested by the user. We have built two call graph generators tested on C/C++ and the Java programming language, and present several use cases on real-world benchmarks. A video demonstration with features of our tool is available at: https://www.cs.tufts.edu/~mshah08/interactivecallgraph.

II. Our Visualization

Our visualization was inspired from a DNA microarray visualization from biology by Zhang et. al where information is laid out in a rectangular grid of pixels [11]. The DNA microarray visualization is a grid of pixels that allows information to be densely packed into a single display.

A. Layout

Our visualization consists of a grid of rectangles (which we will refer to as cells, nodes, methods, or functions interchangeably based on the appropriate context) that are each uniformly sized. The cells will resize to fill the screen depending on

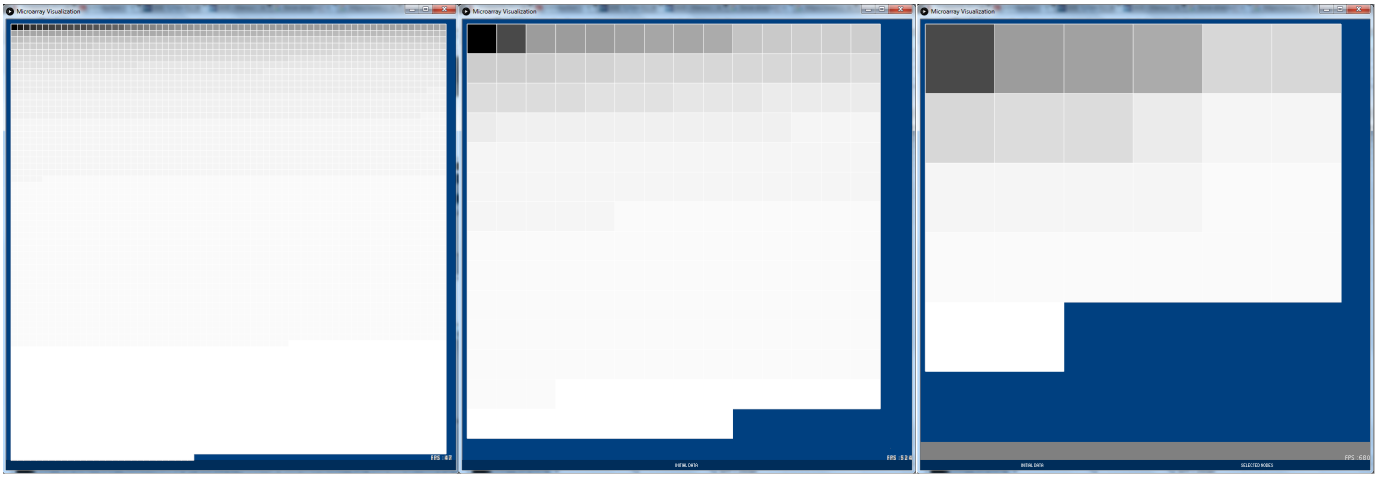

Fig. 2. The far left visualization show the initial program, with cells laid out. The user selected a number of cells, and then generated the middle visualization. The user filtered more cells, and finally had the far right visualization.

how many cells are in the visualization. Each cell in the grid represents a function in the program that we are analyzing. Each cell in the grid is laid out one after the other left-to-right and then top to bottom. This ordering of the cells is based on a user selected criteria of data collected from functions in the program. As users add and remove cells, the ordering of the cells will be preserved in order to assist the user in analysis tasks.

B. Data Structures

Our call graph G of a program is modeled as:

$$G = (n_1, n_2, \dots n_k) \tag{1}$$

Each node n in our visualization has metadata associated with a function. This metadata includes the number of callers, callees, and if the function makes a call to itself. We also generate fields for metadata including: the filename the function is located in, line information, function attributes, opcodes used to make the function, a control flow graph, dynamic data about how often the function is called, and urls to documentation. More fields can be populated, dependent on call graph extracting tools available.

After a user has selected a group of functions they want to further investigate all other nodes are filtered out. In the backend, our program uses a stack data structure to push the selected nodes and generate a new grid visualization containing a subgraph $G\prime$.

$$G\prime = (n_i, n_{i+1}, \dots n_j) \tag{2}$$

The subgraph $G\prime$ consists of a subset of the original nodes in G from some range i to j. The nodes retain all of their original metadata so that it does not need to be recomputed. New metadata will be computed for currently visible nodes regarding the number of callers and callees for that function in the current subgraph. This allows users to then compare how many callees and callers are currently being viewed in $G\prime$ as opposed to G. Figure 2 shows an example of how the visualization changes as we select and push nodes to a new stack and generate a new visualization.

In order to make our visualization run at interactive frame rates, our nodes and their data are stored in hashmaps to retrieve information in O(1) time. Because our visualization lays nodes out in a linear sorted order, we also store them in a list. An additional hashmap is used to retrieve the indexes into this linear sorted list on each level of the stack. This second hashmap is then computed exactly once if we change how the nodes are sorted. Each node in the visualization has an X and Y position in the metadtata associated with it so edges between all of its callers and callees can be drawn. When we push nodes onto a new stack, the layout changes, and we recompute the node positions of all of the callees and callers exactly once. This allows us to draw edges to these nodes while performing our exploration without any additional lag after this one time computation. The linear sorting of the nodes based on user selected criteria helps preserve visual stability however.

C. Interactive Features of Visualization

We interact with the visualization by hovering our mouse over a cell which highlights it in yellow. A separate details pane is coordinated with the visualization showing the metadata associated with that function as we hover over each node. Right-clicking the cell shows us the function name. Left-clicking the mouse highlights the cell in green, and clicking again deselects the cell.

When we mouse over a cell the callees will be highlighted with a black edge between them. An example of this is shown in Figure 1. By holding down a keyboard shortcut callers of the method will be drawn with a red edge between them to distinguish the callees. If edges create clutter the user can alternatively omit the edges and simply highlight callee and caller nodes. Users can quickly select all of the callers and callees from the current node using a shortcut key. A slider widget is available to adjust how many of these edges are traversed and drawn from each method. This gives the user either a partial or entire call tree from each of the methods.

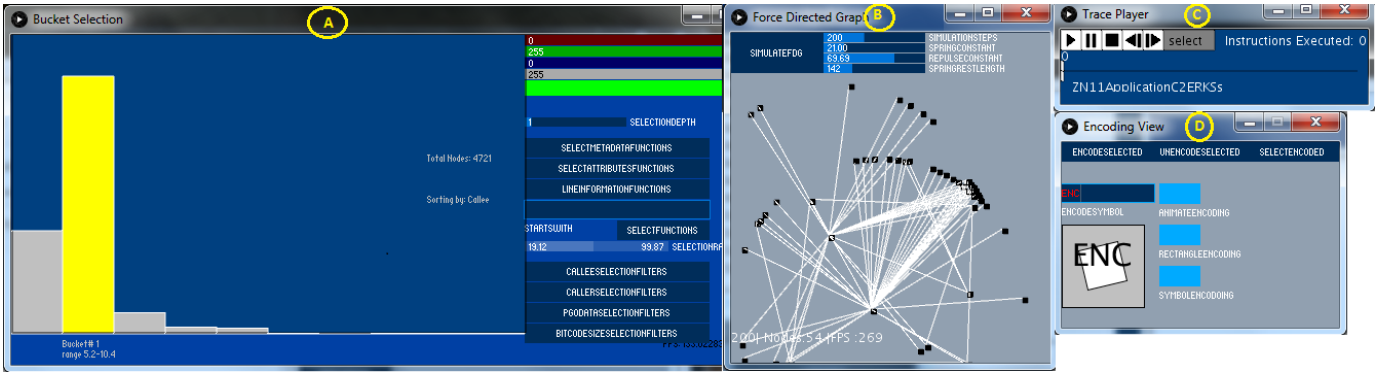

Fig. 3. Four of the five additional windows to support interactions with our microarray visualization are shown above.

When we have selected all of the cells we are interested in, we press the return key to generate a new subgraph. The nodes in the new subgraph are resized to fill the screen, and we may continue exploring the call graph and filtering away cells.

Five separate windows that are coordinated with the microarray are presented to help users query and explore data, four of which are shown in Figure 3 are described below.

- Bucket Selection (Figure 3-A) - The bucket selection view is a histogram that allows highlighting and selecting groups of nodes. Nodes can also be recolored from this view when selected.
- Force-Directed Layout (Figure 3-B) - Users can view all of the selected nodes in a real-time force-directed graph. This view provides a preview of the exported graph, as well as shows clustering of nodes. The user can quickly see if functions are strongly or weakly connected components.
- Animation Pane (Figure 3-C) - A program trace can be loaded, that will highlight and select function names in the order they are called during a program run. The user can estimate the code coverage when running a program trace of an actual execution of the program over a static call graph. The animation pane additionally tallies statistics about the program (number of functions called, total size of all executed functions, etc.).
- Encoding Node (Figure 3-D) - The encoding node view provides options to visually encode nodes with text characters, shapes, and animation that stand out when analyzing the grid. An encoded node will preserve its encoding when pushed to a new subgraph.
- The details view (not pictured in this paper) allows us to jump directly to source if line and file information are available for that method.

Finally, a separate breadcrumbs widget exists that allows users to navigate to different subgraphs that have been generated. The breadcrumbs widget allows us to jump to different subgraphs in our stack analogous to browsing web history with the forward and back buttons in a browser.

D. Call Graph Attributes

Our tool computes attributes about a call graph automatically to save programmers time then doing this by hand manually. The information we provide is the following:

- We return the minimum and maximum depth a method is called along paths from a caller.
- We return the minimum and maximum width at which a method is called at as a callee.
- Users can query if there is a path from method A to B.
- The number of callees and callers the method has in a our full graph G as well as the current subgraph $G\prime$ a user is investigating.
- Users can use the histogram window to quickly query the top ten percent of methods in categories and compute code coverage estimates.

E. Extracting Call Graphs

1) Soot: We built a tool with the Soot framework in order to generate call graphs for programs written in the Java programming language. We extracted additional information about methods for analysis on call graphs with methods annotated as synchronized [15].

2) LLVM: We also built a custom LLVM module pass to generate a call graph as an input file for our visualization [3]. Our LLVM module pass can be used for any programming language that the LLVM frontend supports.

F. Output Formats

When the user has finished their investigation, they can export a subgraph as a DOT file (.dot) or a list of methods. The .dot graph allows users to view a directed-graph of selected functions outside of our system. These .dot files can also be read into our system again, and then used as a starting point for programmers to begin another investigation.

The list of functions can be used as input to additional systems for further code instrumentation. We have created another LLVM module pass as a proof of concept that reads in the list of functions, and annotates those functions to be monitored and have further code transformations applied to them.

III. Case Studies

In order to evaluate the utility of this visualization, we report on two use cases of real-world programs. We analyzed one C++ program and one Java program to test our tools flexibility in different programming languages. Several other programs were analyzed as listed in Table I and the results are summarized in the section Code Exploration.

A. Profile Guided Optimizations

We looked at the open-source C++ Horde3D graphics engine and modified a 3D crowd simulation program provided with the engine. We ran our analysis with the LLVM framework and compiled the program using the Clang-3.9 compiler to generate profile guided data that tells us the execution counts of each method during a program run. This metadata was then loaded into our visualization so we could color our nodes based on execution counts. Our visualization aids in displaying where execution time is spent in the program and how different compiler optimizations increase or decrease that time.

- We use our bucket selection to make a subgraph of the top ten percent of hot functions, and see what functions are calling. We noticed that the crowd simulation was where most of the time was being spent.
- We see which hot methods called into other methods. We then then attempt to apply compiler optimizations, such as inlining methods to see if that could improve overall performance.
- Using our animation pane, we see whether a trace of our Horde3D benchmark compiled with no compiler optimizations (O0) or more compiler optimizations (-O3) made a change in total instruction count.

B. Synchronized Methods

Our next use case is analyzing the concurrency of a large Java program. For this use case, we studied the program Sunflow, which is a multi-threaded ray tracer. Some of the results from our investigation were the following:

- We select an attribute filter to select only the synchronized methods in the program to give us an estimate of the complexity of the programs concurrency. We use our domain knowledge to see if the method names have any intrinsic meaning such as: ”producer”, ”consumer”, ”readers”, ”writers”, and ”barrier” for example.
- We then search to see if any synchronized methods have similar callees and if that gives us insights into refactoring software.
- We search for nested synchronized methods and see if their dependency can be removed to increase performance.
- We also find methods that are not synchronized methods with callees which are synchronized. This can be difficult to find when the method call is several layers down in the call tree, but may reveal where unknown performance penalties could exist.

Program	Program Type	Number of Functions	Average Number of Unique Callees per function
Blender3D	C++ - 3D Animation	18720	3.39
OGRE3D	C++ - Graphics Engine	12423	4.72
Horde3D	C++ - Graphics Engine	4721	2.24
Sunflow	Java - Raytracer	5428	2.54
H2	Java - Banking Application	5139	2.74

TABLE I
Real-world programs tested in our system.

- We also investigate whether the synchronized methods are primarily Java API calls or user written.

C. Code Exploration

We made several observations of code exploration tasks when exploring larger benchmarks like Blender3D to stress our system [2]. During our code exploration task, we focused on looking at the central methods that might be important in the program, as well as outliers.

- We used our tool to gather all of the rendering methods by searching for methods prefixed *gl* to build a call graph of the graphics engine. Previous knowledge of the OpenGL API was required to do this, so this task is applicable to domain experts.
- We selected the function with the most callers, and investigated if the function could be optimized through assembly code which is commonly done in graphics and game applications [7].
- We next visualized attributes of methods across an entire system, which is a strength of our system. These method attributes included: method size, which methods are leafs, and which methods have direct calls to themselves. Methods that have only a direct call to a method with no other callers, are possible candidates to be inlined by the compiler. We exported these methods to a list using our tool to be further investigated.

IV. Discussion

Our tool is only as good as the input data, so if our call graph generator fails to pick up methods this can cause misleading results. In programming languages like Java where code can be loaded dynamically through reflection, users should know our current implementation does not contains this data.

It is also possible to clutter our visualization if too many nodes are selected that are highly connected. Our visualization provides functionality for hiding edges and only highlighting nodes, but this still could be confusing when investigating several methods at a time.

Integrating our tool with an IDE is possible by integrating our call graph generators into the build system and relaunching our standalone program. We believe our tool is valuable to programmers who are brand new to a large programming project. Additionally, associating metadata about performance (e.g. changing compilers, or compiler options) may still be useful with our external tool for analyzing big one time program changes.

V. Related Work

The treemap by Johnson and Schneiderman is a space-filling visualization that visualizes hierarchical data that has been used to lay out call graphs [9] Tools like kcachegrind have made use of the treemap to show how much time a program spends in a given method [10]. An extension of the treemap to three dimensions combined with a city metaphor has been created by Wettel and Lanza [14]. Software cities give a user a sense of locality when being immersed in a large realistic city.

A call matrix is another space-filling visualization that populates an N by N grid (where N is the number of methods in the program) with a program's call graph. The work on the tool AniMatrix demonstrates how to navigate multiple layers of a call graph in a call matrix by Rufiange and Melancon [16]. Additional work on large matrix-based visualizations is presented in a tool called Matrix Zoom by Abello and van Ham [1]. Meister created a tool for analyzing the Mozilla Firefox codebase using a 3D grid that can be explored and displays several code metrics [12]. Work by Henry et al. in their tool NodeTrix takes a hybrid approach of using node-link diagrams with adjacency matrices to analyze large social networks [13].

Node-link diagrams are used to visualize call graphs and can be laid out using tools like Graphvis using several different layouts as documented by Gansner [6]. Reaper is an interactive call graph tool created by LaToza and Myers [19]. Reaper lets users filter and view details of nodes that the user is interested in as they investigate the program. Integrated Development Environments (IDE) like the Eclipse IDE also offer interactive functionality for showing the Call Hierarchy of methods [5]. Text editors such as Vim also have plugins such as CCTree that allow users to browse call trees [20]. A commercial tool called Code Sonar by GrammaTech explores a call graph bottom-up and present different clustering methods in a node-link diagram [4].

Scarle et. al built a visualization tool to represent large programs as a real-time particle system [17]. Their system maps classes, methods, and other entities to a particle, which will then be positioned depending on different user-defined parameters during a simulation.

VI. Future Work and Conclusion

In this paper we have presented a new interactive call graph software visualization tool that scales to programs of tens of thousands of functions supported by two call-graph extracting tools for Java and C++. We believe our work is promising for program comprehension tasks, and will be exploring the following ideas in future work:

- The exploration of additional layout options of grids (possibly 3D) that can pack methods into neighborhoods and minimize edge crossings.
- How best to represent multiple calls from a single caller or callee possibly using different color encodings to indicate hotness along each edge drawn.
- Integration of either a SQL database or Datalog to allow users to write custom selection queries.
- A full user study to evaluate our tool for the program comprehension tasks listed in our use case section. We would like to compare this with other tools and measure the robustness of our tool.
- We believe future user studies could also be performed investigating the visual stability as our visualization evolves, and how that aids or hinders analysis.

Acknowledgment

We would like to thank Sean Silva for his feedback on early versions of this visualization.

References

[1] Abello, J., & van Ham, F. (2004). Matrix zoom: A visual interface to semi-external graphs. In Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on (pp. 183-190). IEEE.

[2] Blender3D. https://www.blender.org/

[3] Chris Lattner, Vikram Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, p.75, March 20-24, 2004, Palo Alto, California

[4] Code Sonar. http://www.grammatech.com/products/code-visualization

[5] Eclipse - The Eclipse Foundation open source community website. https://eclipse.org/

[6] E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. A Technique for Drawing Directed Graphs. IEEE Transactions on Software Engineering, 19(3):214230, March 1993.

[7] Fog, Agner. 2015, December 23. Optimizing Subroutines in Assembly Language.

[8] Horde3D - Next-Generation Graphics Engine. http://www.horde3d.org/

[9] Johnson B., Shneiderman B., Tree-Maps: A Space Filling Approach to the Visualization of Hierarchical Information Space, IEEE Visualization 91, pp. 275-282, 1991.

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

[11] L. Zhang, J. Kuljis, and X. Liu. Information Visualization for DNA Microarray Data Analysis: A critical review. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, 38(1):42 54, 2008.

[12] Meister, Patrick. Interactive Software Visualization, March 2, 2007. https://www.youtube.com/watch?v=bccqxpGouK0

[13] N. Henry, J.-D. Fekete, and M. J. McGuffin. NodeTrix: A Hybrid Visualization of Social Networks. IEEE Transactions on Visualization and Computer Graphics, 13(6):13021309, 2007.

[14] R. Wettel and M. Lanza, Visualizing Software Systems as Cities, Proc. IEEE Intl Workshop Visualizing Software for Understanding and Analysis (VisSoft 07), pp. 92-99, 2007.

[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] Sbastien Rufiange, Guy Melanon, "AniMatrix: A Matrix-Based Visualization of Software Evolution", In IEEE Working Conference on Software Visualization, pp. 1-10, 2014.

[17] Scarle, Simon; Walkinshaw, Neil, "Visualising software as a particle system," in Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on , vol., no., pp.66-75, 27-28 Sept. 2015

[18] Sunflow - Global Illumination Rendering System. http://sunflow.sourceforge.net/

[19] T. D. LaToza and B. A. Myers, "Visualizing Call Graphs," in VL/HCC'2011: IEEE Symposium on Visual Languages and Human-Centric Computing Pittsburgh, PA, 2011, pp. 117-124.

[20] Vim CCTree. https://sites.google.com/site/vimcctree/