Analyzing and Visualizing Whole Program Architectures

Thomas Panas Dan Quinlan Richard Vuduc
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L-550
Livermore, California 94551, USA
{panas2, dquinlan, richie}@llnl.gov

1. Introduction

This paper describes our work to develop new tool support for analyzing and visualizing the architecture of complete large-scale (millions or more lines of code) programs. Our approach consists of (i) creating a compact, accurate representation of a whole C or C++ program, (ii) analyzing the program in this representation, and (iii) visualizing the analysis results with respect to the program's architecture. We have implemented our approach by extending and combining a compiler infrastructure and a program visualization tool, and we believe our work will be of broad interest to those engaged in a variety of program understanding and transformation tasks.

We have added new whole-program analysis support to ROSE [17, 16], a source-to-source C/C++ compiler infrastructure for creating customized analysis and transformation tools. Our whole-program work does not rely on procedure summaries; rather, we preserve all of the information present in the source while keeping our representation compact. In our representation, a million-line application in well less than 1 GB of memory.

Because whole-program analyses can generate large amounts of data, we believe that abstracting and visualizing analysis results at the architecture level are critical to reducing the cognitive burden on the consumer of the analysis results. Therefore, we have extended Vizz3D [21], an interactive program visualization tool, with an appropriate metaphor and layout algorithm for representing a program's architecture. Our implementation provides developers with an intuitive, interactive way to view analysis results, such as those produced by ROSE, in the context of the program's architecture.

The remainder of this paper summarizes our approach to whole-program analysis (Section 2) and provides a concrete example of how we visualize the analysis results (Section 3).

2. Whole-Program Analysis

Our source code analysis and transformation work is part of the ROSE project, sponsored by the U.S. Department of Energy (DOE). Although research in the ROSE project emphasizes performance optimization, ROSE contains many of the components and analyses common to any compiler infrastructure, and thus facilitates the development of a broad range of source-based analysis tools. ROSE routinely compiles million-line applications. This section describes our whole-program analysis implementation in ROSE.

2.1. Representing whole programs

Whole-program analysis is typically implemented either using procedure summaries or by embedding information into the object files to use whole-program context at link-time. In ROSE, we approach the problem differently, namely, by using a space-efficient representation of the complete source.

We use the Edison Design Group C++ front-end (EDG) [9] to parse C and C++ programs. The EDG front-end generates an abstract syntax tree (AST) and fully evaluates all types. We translate the EDG AST into our own object-oriented AST, SAGEIII, based on Sage II and Sage++ [5]. The SAGEIII intermediate representation (IR) has 240 types of IR nodes and can fully represent original structure of the application, including the preservation of comments and preprocessor control structure. From the IR, the original source may be reproduced completely.

The IR is space-efficient by design. In particular, we share parts of the AST (subtrees) that are determined to be identical. This technique is critical for C and C++. For example, a typical million-line applications compiled by ROSE have on the order of 1000 files containing 75K lines contributed from header files and 1K lines of actual source code in the source file. In this scenario, the effective 76K lines of code generate an AST with about 500K IR nodes.

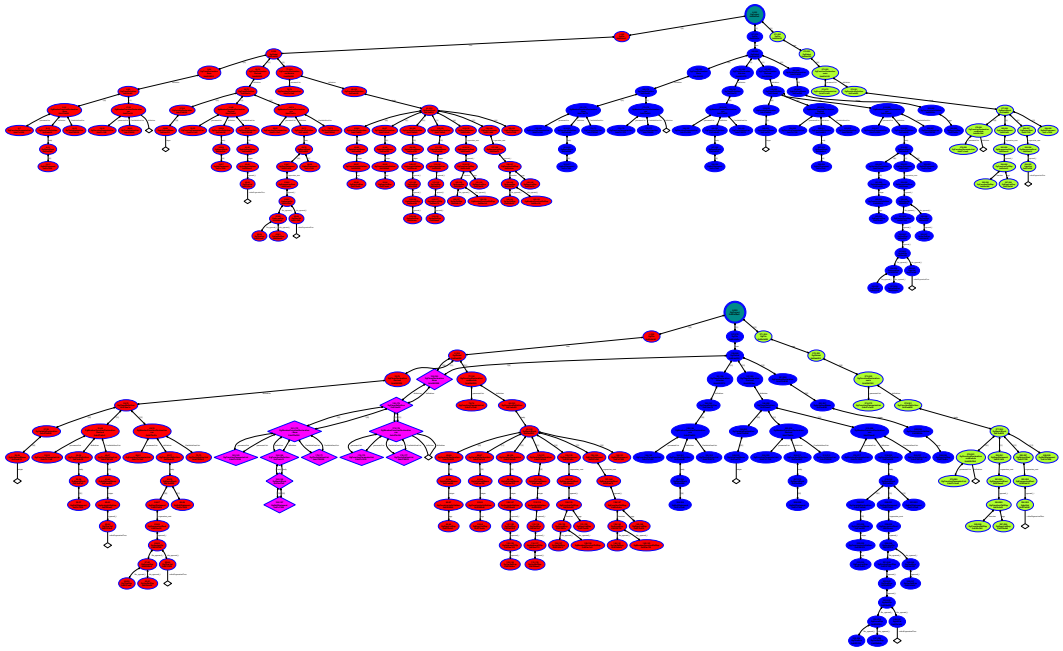

Figure 1. (*Top*) The AST before merging. File 1 = green nodes, File 2 = blue nodes, File 3 = red nodes. (*Bottom*) The AST after merging. The magenta subtree shows common structure that has been merged.

To support whole-program analysis, ROSE merges multiple ASTs from the compilation of different source files into a single AST, without losing project, file, and directory structure. Merging 75K lines over each of the 1000 files saves 75 million lines of code from being represented redundantly in the AST.

Using a 250 KLOC benchmark, we have estimated that a million-line application will fit into approximately 400 MB of memory after merging header files. The AST holding the million-line application can also be saved to and loaded from disk using a custom ROSE-specific binary file format; on current single-processor desktop machines, one of these binary files can be written in roughly 30 sec and read in under a minute. Simple traversals of the whole AST (in memory) are expected to take only a few seconds.

Figure 1 (top) shows the AST for three example source files, with AST subtrees colored by file. The ASTs from the files are not shared. Figure 1 (bottom) shows the AST after the merge process, where the diamond shaped IR nodes of the AST indicate that those IR nodes are shared. To be shared, the declaration at the root of the subtrees had to generate the same internal name (in C++, this includes standard name mangling plus a number of other language specific details) *and* the subtrees had to pass the One-time Definition Rule (ODR) test of equivalence. For a more detailed description of our merge algorithm, see our recent paper [16].

2.2. Analysis

ROSE internally implements a number of forms of procedural and interprocedural analysis, with much of this work in current development. ROSE currently includes support for dependence, call graph, and control flow analysis. In collaboration with academic groups, we are extending the analysis infrastructure to interface with general analysis tools, including PAG [2] OpenAnalysis [19], as well as analysis tools specifically for automated debugging and security, MOPS for finite state machine-based temporal specification checking [6], and coverage analysis tools [8].

ROSE contains a variety of graphs, metrics, and analyses to support program developers in understanding their software quality, software security, performance bottlenecks, and software structure. The following list provides examples of these analyses:

Strongly Connected Components (SCC). We detect cyclic dependencies between functions, classes or files. In general, nodes in a cyclic dependency may be merged to reduce call dependencies, and hence to reduce the structural complexity of the system.

Unsafe Function Calls. Certain aspects of C++ (*e.g.*, unchecked array access, raw pointers), can lead to low-level buffer overflows, page faults, and segmentation faults. In this analysis, we detect calls to "unsafe" functions, such as

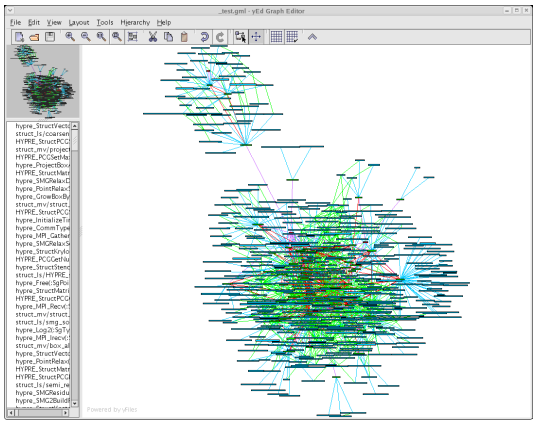

Figure 2. A conventional architecture visualization.

`sprintf`, `scanf`, `strcpy`.

Global Variables. We traverse the program's abstract syntax tree (AST) to check for public declared variables (within the scope of classes) and global variables (outside the scope of classes). Global variables are a bad programming style and should not appear, especially in object-oriented code.

Arithmetic Complexity. For each function, this analysis counts the number of arithmetic operations on float, int, float pointer, and int pointer types. Thus, functions and classes with large arithmetic operation counts can be detected. This property is particularly important in scientific computing codes, since such functions should be the most robust and reliable pieces of the software.

3. Architecture Visualization

The aim of architecture-level visualization is to rapidly summarize and communicate the architecture and design decisions of the overall software system. Architectural visualization is naturally more abstract than low-level visualizations (from low-level analyses) [15], and therefore better suited to visualizations in the large. Abstract visualizations of software architectures combined with metrics can help software developers to answer many questions about a software system.

Common examples of architectural visualizations are function call graphs, hierarchy graphs, and directory structures. There are many ways to present these graphs, such as UML diagrams, graph browsers, and component/connector graph drawings. For example, Figure 2 shows the visualization of various analysis results. The results are represented within one image to reduce the cognitive burden a viewer has in order to associate same entities from multiple views [18]. This image carries a tremendous amount information, and techniques for information reduction become necessary.

To help a viewer better navigate and understand analysis results of large-scale applications, we have implemented a 3D city metaphor, a predictable layout, and abstraction and navigation mechanisms within our visualization tool, Vizz3D. We show an example of visualizing SMG2000 [1], a semicoarsing multigrid code, in Figure 3. This image shows exactly the same information as Figure 2, but in our view, in a more cognitively accessible way.

More specifically, Figure 3 is a snapshot taken of Vizz3D while running on SMG2000. The directories of this application appear as "islands," individual files as "cities" within the island, and individual function definitions as "buildings." In addition, aggregate shaded edges between cities indicates that some function in one file (red end) calls some function in another file (dark end). Other user-selected metrics and analyses (whether static or dynamic) may be rendered as textures, colors, and icons in this view. We believe that the right choice of metaphor and layout are crucial, and we will investigate this claim in future studies.

4. Related Work

Whole-program analysis has traditionally been applied in performance optimization contexts [4, 20], but has recently also been used to find bugs and detect security flaws using global dataflow analyses [12, 13, 10]. Our techniques complement earlier work by providing the basic infrastructure for accurately representing the source of an entire program but for purposes of program understanding. Among other open C or C++ infrastructures [11, 14, 3, 7] and C++ static analysis infrastructures [22], our basic mechanisms for building whole-program representations are unique.

References

[1] The SMG2000 Benchmark, 2001. llnl.gov/asci/platforms/purple/rfp/benchmarks/limited/smg.

[2] AbsInt, Inc. PAG: The Program Analysis Generator, 2006. absint.com/pag.

[3] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF Compiler for Scalable Parallel Machines. In *Proc. SIAM Conference on Parallel Processing for Scientific Computing*, Feb 1995.

[4] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proc. International Conference on Software Engineering*, Berlin, Germany, March 1996.

[5] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

[6] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Proc. Network and Distributed*

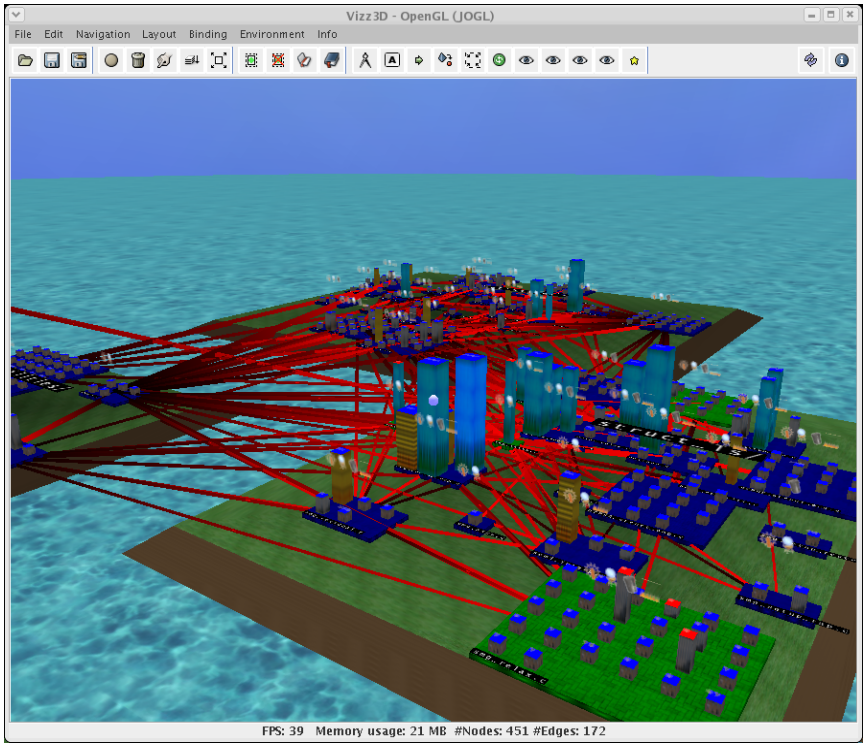

Figure 3. Our architecture visualization.

System Security Symposium, San Diego, CA, USA, February 2004.

[7] S. Chiba. Macro processing in object-oriented languages. In *TOOLS Pacific '98, Technology of Object-Oriented Languages and Systems*, 1998.

[8] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Testing multithreaded Java programs. *IBM Systems Journal: Special Issue on Software Testing*, February 2002.

[9] Edison Design Group. EDG front-end. `edg.com`.

[10] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proc.International Conference on Verification, Model Checking, and Abstract Interpretation*, Venice, Italy, 2004.

[11] F. S. Foundation. GNU Compiler Collection, 2005. gcc.gnu.org.

[12] S. Z. Guyer, E. D. Berger, and C. Lin. Detecting errors with configurable whole-program dataflow analysis. In *Proc. Conference on Programming Language Design and Implementation*, Berlin, Germany, 2002.

[13] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proc. Conference on Programming Language Design and Implementation*, pages 168–181, June 2003.

[14] G. Keating. Inter-module analysis in GCC. In *Proc. GCC Developers' Summit*, Ottowa, Canada, June 2005.

[15] T. Panas. *Towards a Generic Framework for Reverse Engineering*. Licentiate thesis, Växjö University, Sweden, November 2003.

[16] D. Quinlan, R. Vuduc, T. Panas, J. Härdtlein, and A. Sæbjørnsen. Support for whole-program analysis and verification of the One-Definition Rule in C++. In *Proc. Static Analysis Summit*, Gaithersburg, MD, USA, June 2006. National Institute of Standards and Technology Special Publication.

[17] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proc. Joint Modular Languages Conference*, 2003.

[18] M.-A. D. Storey, F. D. Fracchia, and H. A. Mueller. Cognitive design elements to support the construction of a mental model during software visualization. In *Proc. of the 5th Int. Workshop on Program Comprehension (WPC '97)*, Washington, DC, USA, 1997. IEEE Computer Society.

[19] M. M. Strout, J. Mellor-Crummey, and P. D. Hovland. Representation-independent program analysis. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, September 2005.

[20] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *Proc. Conference on Programming Language Design and Implementation*, Ottowa, Canada, June 2006.

[21] Vizz3D. Available at: `http://vizz3d.sourceforge.net`, July 2006.

[22] D. Wilkerson. OINK: A collection of composable C++ static analysis tools, 2005. freshmeat.net/projects/oink.