Visualization of Program Dependence Graphs*

Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck

Institute for System Software
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University Linz
Linz, Austria
{wuerthinger,wimmer,moessenboeck}@ssw.jku.at

Abstract. The analysis of a compiler's intermediate data structures helps at debugging complex optimizations. We present a graphical tool for analyzing the program dependence graph of Sun Microsystems' Java HotSpot™ server compiler. The tool saves snapshots of the graph during the compilation. It displays the graphs and provides filtering mechanisms based on customizable JavaScript code and regular expressions. High performance and sophisticated navigation possibilities enable the tool to handle large graphs with thousands of nodes.

1 Introduction

The Java HotSpot™ server compiler [5] of Sun Microsystems uses a program dependence graph [2] as the intermediate data structure when compiling Java bytecodes to machine code. It applies optimizations such as global value numbering, conditional constant propagation, and loop transformations to produce faster code. When debugging the compiler, only a textual output of the graph is currently available.

We present a tool that facilitates analyzing the compiler by providing a graphical representation of the program dependence graph. The tool captures snapshots of the graph during the compilation of a method, so the user can reconstruct the transformations applied to the graph by compiler optimizations. The tool applies filters based on regular expressions to make the appearance of the graph customizable and enables the user to quickly focus on specific parts of the graph, which is especially helpful for the analysis of large graphs.

While the main focus of the tool is currently the visualization of the data structures of the server compiler, it can easily be adapted for other programs that work on directed graphs. A more detailed description of the tool and the program dependence graph can be found in [8].

2 Architecture

The program dependence graph of the Java HotSpot™ server compiler combines control, data, and memory dependencies into a single graph. A node has ordered

* This work was supported by Sun Microsystems, Inc.

L. Hendren (Ed.): CC 2008, LNCS 4959, pp. 193–196, 2008.

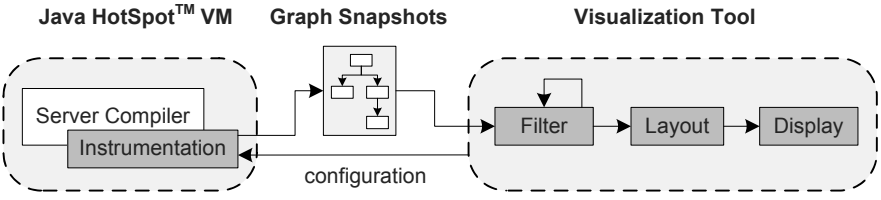

Fig. 1. Interaction between the compiler and the visualization tool

input slots and produces a single output value. Projection nodes are used when a node produces a tuple. Figure 1 shows the interaction between the visualization tool and the server compiler.

We instrument the compiler to take snapshots of the graph. The snapshots are either stored in an intermediate file or directly sent to the visualization tool via a network connection, which additionally allows configuration data to be sent back to the compiler. The user can select a set of filters to be applied to the graph. After the graph is transformed by the filters, the layout algorithm calculates node positions and interpolation points for the edges. Then the graph is displayed on the screen.

Data Model. The graphs are stored in an XML based format. They can be exported to a file to allow subsequent analysis. The snapshot of a graph is serialized as the difference to the previous snapshot. This reduces the storage requirements when snapshots of the graph are taken frequently.

Properties of nodes are stored as textual key-value pairs to improve extensibility. New properties can be introduced without changing the tool. Filters select nodes based on regular expressions on the value of a property with a certain name, and custom filters for arbitrary properties can be defined. This way, the tool can be used to visualize directed graphs whose nodes have other properties than the nodes of the server compiler.

The data transmitted from the server compiler to the visualization tool contains a clustering of the nodes into basic blocks when this information is already available in the compiler. Otherwise the tool calculates an approximate scheduling of the nodes. The basic data model does not contain any display information. Before the filters are applied, the transmitted graph is converted into a graph with display information such as node positions and node colors. A node can have multiple output slots in this model, e.g. if a filter merges several nodes.

Graph Layout. We use a hierarchical layout algorithm based on the approach of Sugiyama [6] and the GraphViz tool *dot* [3]. The main focus of our implementation is performance, because the graph of large methods can have a few thousand nodes and more than ten thousand edges. Long edges are cut so that only the beginnings and endings are drawn. This improves the overview as well as the performance. A special routing for backward edges ensures that they also start at the bottom of their start node and end at the top of their destination node.

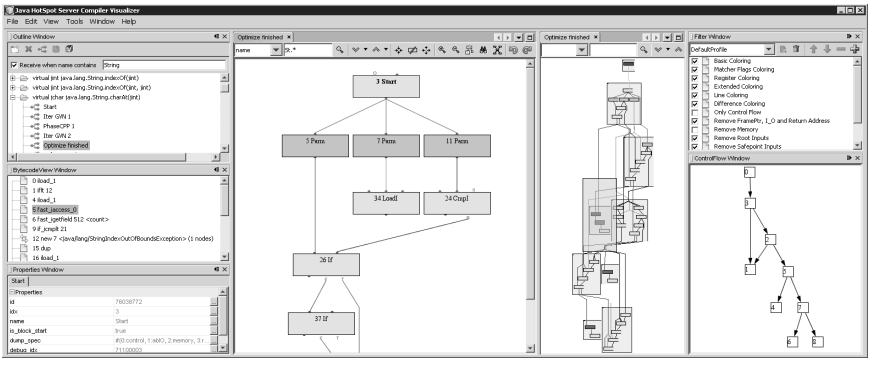

Fig. 2. Screenshot of the visualization tool showing two graphs

3 Usage

Figure 2 shows the tool when displaying an extract of a graph in normal view and a whole graph in satellite view. The methods retrieved from the server compiler are listed in the top left window. Double-clicking on the graph snapshot of a certain method opens the graph in the middle area. The top right window contains the available filters; checkboxes activate them. The list of filters can be edited, and the selected set of filters can be saved as a profile. The bottom right window corresponds to the control flow graph approximation of the active program dependence graph. The bottom left window displays the textual key-value properties of the selected nodes. The middle left window contains the bytecodes of the compiled method.

Filtering. Before layouting and displaying the graph, filters are applied that can remove, add, merge, split, and color nodes and edges. The filters can be specified by predefined JavaScript functions, which use regular expression based rules on the node properties. The following JavaScript statement assigns a red background color to all nodes whose name starts with the letter "I". Semantically, this highlights all integer instructions: `colorize("name", "I.*", red);`

Difference. The tool can display the difference between snapshots graphically. It is also capable of calculating the approximate difference between two arbitrary graphs, e.g. snapshots before and after a compiler change. The difference is made visible using color filters. This helps identifying the effects of modifications in the compiler.

Navigation. Despite standard graph navigation techniques like showing and hiding nodes or going to pre- and successors of a node, the tool provides a way to navigate in the graph by only double clicks on nodes. The tool maintains the set S of fully shown nodes. It draws all nodes that are immediate pre- or successors of a node of the set S as semi-transparent. The user can add one of those semi-transparent nodes to the set S by a double click on it. Performing this action on a node of the set S removes it from the set.

Bytecodes. The bytecodes window shows the input data of the server compiler. If methods are inlined, this is a tree structure where the inlined methods are shown as children of the method call. Navigation between the bytecodes and the graph is available for instructions that may throw an exception. For all other instructions, the server compiler does not track which instructions are created for a certain bytecode.

4 Related Work

Balmas presents a tool that displays the program dependence graph for C source code with focus on creating hierarchical groups of nodes [1]. Krinke developed a similar tool that additionally gives a textual representation of program slices [4]. The main differences to our application are that they do not have built-in filtering mechanisms and they are not designed to visualize a program dependence graph structure that changes during compilation of a method as compiler optimizations are applied.

5 Conclusions

We presented a tool for displaying the program dependence graph of the Java HotSpot^{TM} server compiler at various stages during compilation. It helps at debugging the server compiler and analyze its built-in compiler optimizations. A property-based filter system makes the tool flexible and also usable for the analysis of other directed graph structures. The HotSpot^{TM} compiler team at Sun Microsystems is currently evaluating the tool and integrating the server compiler instrumentation into the upcoming JDK 7 [7]. They are planning to include the visualization tool in the OpenJDK project.

References

1. Balmas, F.: Displaying dependence graphs: A hierarchical approach. In: Proceedings of the Working Conference on Reverse Engineering, pp. 261–270 (2001)
2. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9(3), 319–349 (1987)
3. Gansner, E.R., Koutsofios, E., North, S.C., Vo, K.P.: A technique for drawing directed graphs. IEEE Transactions on Software Engineering 19(3), 214–230 (1993)
4. Krinke, J.: Visualization of program dependence and slices. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 168–177 (2004)
5. Paleczny, M., Vick, C., Click, C.: The Java HotSpot^{TM} server compiler. In: Proceedings of the Java Virtual Machine Res. and Techn. Symposium, pp. 1–12 (2001)
6. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics 11(2), 109–125 (1981)
7. Sun Microsystems, Inc.: JDK 7 Project (2007), `https://jdk7.dev.java.net/`
8. Würthinger, T.: Visualization of program dependence graphs. Master's thesis, Johannes Kepler University Linz (2007)