Source Level Debugging of Optimized Programs Using Data Flow Analysis*

Roland Wismüller
Department of Computer Science
Munich University of Technology
Arcisstr. 21, D-W-8000 München 2
Tel.: +49-89-2105-8164, Fax: +49-89-2105-8232
E-mail: wismuell@informatik.tu-muenchen.de

1 Introduction

Source level debuggers have been proved to be valuable tools for program development. However, traditional source level debuggers can only be used with code not optimized by the compiler. Since bugs may still appear in prototype versions of a program, where optimization is desirable, there is a need for debugging optimized programs.

An interactive breakpoint based debugger able to deal with optimized code must handle two major problems [Zel84]: The *code location problem* occurs as soon as the user sets a breakpoint on a source line. Then the debugger must decide where to stop the object program. Once the program has been stopped the user usually wants to inspect the program variables. The debugger must be able to compute the values the source variables should have from the values the object variables[1] actually have. This is commonly referred to as the *data value problem.* When trying to inspect a variable v in an optimized program, different situations may arise, e.g:

- The value of v is the same as it would be in the unoptimized program. In this case v is *current.*
- The value of v cannot be determined, since a previous assignment to v has been deleted or moved, so that it has not yet been executed. A similar case arises if a later assignment to v has been moved, so that it has already been executed. In both cases v is *noncurrent.*
- If a loop-invariant assignment of v has been moved in front of a loop, v is noncurrent only in the first iteration of the loop. If there is no information about the current iteration count, v must be regarded as noncurrent.

In the last years several researchers have developed techniques to overcome these problems. We have concentrated on the data value problem following an approach of Hennessy [Hen82], who introduced the term *current* and made some steps towards handling the problem with data flow analysis performed by the debugger. Independent of our work, also Copperman has developed data flow analysis algorithms to detect current variables [CM91].

2 Desired Solution Properties

The algorithm we have developed differs from previously published solutions in being more general. In particular, we desired the following properties to be met:

- Since we are interested in general solutions, the algorithm should be mostly independent from the optimizer. Therefore we should not use data structures specific to certain transformations.
- Most techniques in the literature (e.g. [CMR88, CM91]) require the compiler to generate a unique mapping between source and object variables. This mapping may depend on the program counter, so variables with changing locations (e.g. due to register allocation) can be supported. Since for these variables a compiler usually maintains a mapping only during the variable's life range, using such a mapping can restrict the ability to handle non-live variables. However, the values of non-live variables are often of interest during debugging, so the debugger should determine the proper object variable for some variable v at any point in the program by

*This work is partly funded by the German Science Foundation, Contract: SFB 0342, TP A1

[1] An object variable may be a register or a memory location

itself. This can be done by calculating the set of reaching definitions of v. The variables written by the corresponding stores in the object program are those that correspond to v.

- In contrast to other algorithms that require the program to be stopped at *syntactic breakpoint locations* [Zel84], our algorithm can be used at arbitrary points in a program, given by a pair (b_s, b_o) of locations in the source and object code. This is motivated by the fact that in general syntactic breakpoint locations are neither easy to find nor unique and may even not exist. However, as long as the breakpoint in the object program is reached, if and only if it should be reached according to the source program, its exact location is unimportant. Therefore there is some freedom to choose an optimal location, where as many variables as possible are current. The ability to handle arbitrary breakpoints also allows to use the algorithm when the program stopped due to an exception and enables *recovery* [Hen82].

- To get optimal results, the algorithm should be able to consider information about the paths the program has taken in the past[2]. In order to handle such information, the algorithm must be exact with respect to single paths and definitions. We have shown that data flow analysis must distinguish between instances of definitions in different iterations of loops to meet this requirement. Therefore we cannot use standard algorithms without modifications.

- For usability with real programs, the algorithm must be able to handle assignments to variables via pointers and assignments to array elements.

3 An Algorithm to Determine Current Variables

As an interface between compiler and debugger we decided to use two flow graphs representing source and object code, respectively. Flow graphs have been proved to be suitable representations for data flow analysis and can be easily provided by the compiler by dumping the program's intermediate representation before and after optimization. Only the operations that change the program's state (assignments, calls and jumps) have to be included in these graphs. For each statement additional information needed for data flow analysis is provided. The effects of code transformations are modelled mainly by a relation **code** between corresponding source and object statements and between corresponding flow edges. This relation is built during the optimization phase.

With this representation we test the following condition to check whether a source code variable v_s is current at location (b_s, b_o):

- v_s is current at (b_s, b_o), iff there is an object variable v_o, such that each definition d_s of v_s that reaches b_s is current with respect to v_o.

- d_s is current at (b_s, b_o) with respect to v_o, iff the following holds for each execution (p_s, p_o), each instance d_s^α of d_s in p_s and each instance b_s^β of b_s in p_s:

  If d_s^α writes to v_s and reaches b_s^β along p_s and b_o^δ is the instance of b_o in p_o corresponding to b_s^β then in p_o there is an instance d_o^γ of a store $d_o \in \mathbf{code}(d_s)$ writing to v_o that corresponds to d_s^α and reaches b_o^δ along p_o.

- An execution (p_s, p_o) is defined by the path p_o the optimized program takes with some input and the path p_s the source program would take, if it were executed with the same input.

In other words, for a given execution, we ask whether the statement instance that assigned the current value to v_o in the object program semantically corresponds to the statement instance that should have assigned the current value to v_s according to the source program.

In our algorithm we discriminate between instances of statements (labelled with upper Greek indices in the definition above) by the number of backward-edges on the path to an instance. In order to use standard analysis techniques, the flow graph is unrolled in a special way, such that those statement instances that need to be distinguished are mapped to different nodes in the unrolled graph. For a graph (N, E) we build the new graph (N', E') with $N' \subseteq N \times [-1, m+1], E' \subseteq N' \times N'$ such that $<(n_1, i), (n_2, j)> \in E'$, iff $<n_1, n_2>$ is a forward edge in E and $i = j$ or $<n_1, n_2>$ is a backward edge and $(i = j - 1 \vee i = j = -1 \vee i = j = m+1)$. A node (n, i) thus roughly corresponds to an instance of node n. The subgraphs defined by nodes $(n, -1)$ and $(n, m+1)$ ensure that every path that is possible in original graph can be mapped to the unrolled graph. The number m is bound by the loop connectedness of the graph, so usually $m \leq 2$ holds. Fig. 1 gives an example of an unrolled graph for $m = 1$.

Given a variable v_s and a location (b_s, b_o), the set of reaching definitions of v_s at b_s is computed first.

[2] This information may be gathered by invisible breakpoints or traces

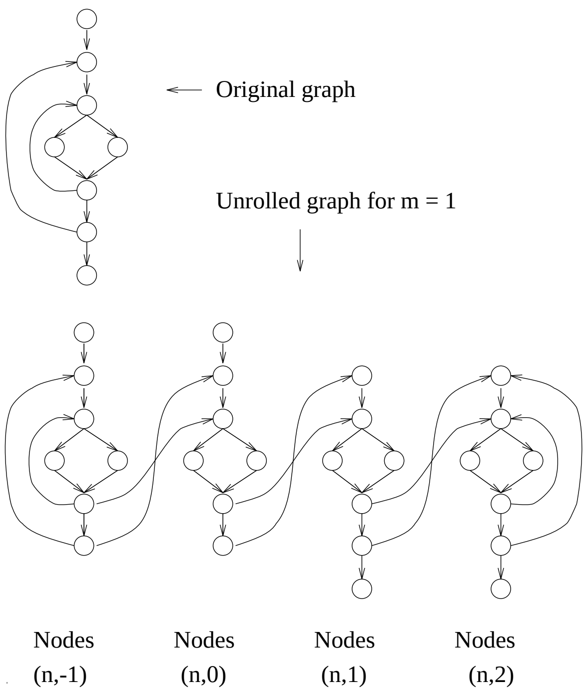

Figure 1: A flow graph and its unrolled graph

From this set the object variable v_o associated to v_s is determined. Then each reaching definition d_s is tested for currentness. For each number of backward edges in the range $0..m$ on a reaching path from d_s to b_s, we test if there is a corresponding instance of a store d_o reaching b_o along a path with the same number of backward edges. To reflect the condition that the path from d_s to b_s and the path between d_o and b_o must belong to the same execution, we restrict the object graph by deleting edges, until only paths that correspond to reaching paths in the source graph are possible. This method of restriction can also be used to exclude paths that are known to be no part of the execution due to run-time information.

If v_s proved to be current, its value can be determined by reading the register or memory location v_o. If v_s is noncurrent, it may be possible to recreate the value by evaluating the right hand side of the last assignment to v_s. This is possible, if there is exactly one reaching definition d_s, and the variables of the right hand side of d_s are current at location (d_s, b_o). Otherwise, the data flow information can be used to explain to the user, why the variable has not an expected value.

With small extensions the algorithm is also able to properly handle indirect assignments via pointers. Furthermore it can use information from the compiler that assures a variable being current at a certain location. Unless the optimizer performs a very detailed analysis of the effects of pointer assignments, each variable that may be accessed by a pointer is current at all assignments using that pointer. With this information, the run-time cost of the analysis can be kept in an acceptable range.

4 State of the Work

We have already finished modifying the SUN C-compiler for the Sparc architecture in order to generate the flow graphs and the mapping between them. A first implementation of the algorithm sketched above is also available. We will integrate this algorithm into the debugger DETOP that is part of the parallel programming environment TOPSYS [BB91] during the next months.

References

[BB91] T. Bemmerl and A. Bode. An Integrated Environment for Programming Distributed Memory Multiprocessors. In A. Bode, editor, *Distributed Memory Computing, 2nd European Conference, EDMCC2*, volume 487 of *Lecture Notes in Computer Science*, pages 130–142, München, FRG, April 1991. Springer-Verlag.

[CM91] M. Copperman and C. E. McDowell. Debugging Optimized Code Without Surprises. In *Proceedings of the Supercomputing Debugging Workshop 1991*, Albuquerque, New Mexico, November 1991.

[CMR88] D. S. Coutant, S. Meloy, and M. Ruscetta. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23(7) of *ACM SIGPLAN Notices*, pages 125–134, June 1988.

[Hen82] J. Hennessy. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[Zel84] P. T. Zellweger. Interactive Source-Level Debugging of Optimized Programs. Technical Report CSL-84-5, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, May 1984.