Recovery of Noncurrent Variables in Source-level Debugging of Optimized Code

Amitabh Srivastava
Symbolic Computing Laboratory
Texas Instruments Incorporated
M.S. 238, P.O. Box 226015
Dallas, Texas 75266
(214)-995-0476

Abstract

Source-level debugging and code optimization are important tools of programming. A debugger has to supply precise information about the values of the variables at any run time error or breakpoint. Code optimization, on the other hand, removes dead stores and reorders code to save time and space, causing intermediate events to happen out of sequence. A thorough implementation of either limits the usefulness of the other. Hennessy, in his paper "Symbolic Debugging of Optimized Code"[2], suggested various techniques to handle this. However, he explicitly chose not to take advantage of the fact that many operations are reversible and many partial results will be available in registers and temporaries. This paper considers these questions in order to extend Hennessy's techniques for recovery of noncurrent variables. The algorithms developed can detect most of the recoverable variables, and run in linear time. We also consider methods to limit our search to only a part of the dag.

---

1This research was done at the Computer Science Department of the Pennsylvania State University.
1 Introduction

Source-level debugging and code optimization are important aspects of programming. A source-level debugger has to supply accurate information about the values of variables at the point when it is invoked. The debugger is generally invoked when the program encounters a stopping point which could be due to a run time error or a user-inserted breakpoint.

The main purpose of code optimizations, on the other hand, is to save time and space by removing dead stores, reordering the code and so on. The only thing code optimization maintains is the overall functional equivalence between the optimized and the unoptimized versions of the program.

The reordering of code and the removal of dead stores creates a problem for the debugger, whose job is to return accurate values of variables at a particular point partway through the execution. At such a stopping point the effect of the optimized and unoptimized programs might temporarily be different. If a variable has a value at the stopping point which is not the same as it would have at the corresponding point in the unoptimized version, then we say the variable is noncurrent.

Thus it is not always possible for the debugger to return accurate source level information when optimizations have been performed on the code. If the user is not appropriately informed, his work will be considerably increased by trying to figure out the values of the noncurrent variables by determining what optimizations were done to his code.

We thus have a conflicting situation in which we want optimized code together with the ability to stop the execution in the middle and examine the program state. A thorough implementation of either limits the usefulness of the other. One very simple solution to this would be to perform the optimizations on the code only after the code is bugfree. There are several arguments against this solution. First, optimizations are a part of the compilation process and it might not be practical to separate the two. One could have two compilers – an optimizing and an unoptimizing one – but it is very difficult to ensure the two compilers implement exactly the same language, and it is better that the program be developed on the same compiler on which it will be used later. Second, unoptimized versions of some programs may not fit on some machines because of space restrictions and so an optimized version would be required from the start. Third, a program is never bugfree and a transient problem in the optimized version might not be reproducible at all in the unoptimized environment. Lastly, source-level debugging is such an important tool that it should not be given up at any stage.

The source-level debugger is invoked when a run time error occurs or a user-inserted breakpoint is encountered. When a run time error occurs in a source statement then the error is reported at just before the statement and so the state of the variable it reports is that of just before the execution of the statement at which the error was encountered. A user is permitted to insert a breakpoint only between statements and not inside a statement. As the two cases in which the debugger is invoked are identical here, for debugging purposes it does not matter whether the stopping point was a run time error or user-inserted breakpoint.

The two main problems of the debugger would be first to determine which variables are non-