Symbolic Debugging of Globally Optimized Code: Data Value Problems and Their Solutions

Ali-Reza Adl-Tabatabai   Thomas Gross

January 1994

CMU-CS-94-105

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152 and by Motorola's Semiconductor Products Sector RISC Division.

Keywords: Symbolic debugging, endangered variables, suspect variables, noncurrent variables, compiler optimization, data flow analysis

Abstract

Symbolic debuggers are program development tools that allow a user to interact with an executing process at the source level. In response to a user query, the debugger must be able to retrieve and display the value of a source variable in a manner consistent with what the user expects with respect to the source statement where execution has halted. However, when a program has been compiled with optimizations, values of variables may either be inaccessible in the run-time state or inconsistent with what the user expects. Such problems that pertain to the retrieval of source values are called *data value* problems. In this paper we address the data value problems caused by global scalar optimizations. We describe in detail how global optimizations cause data value problems and the information a symbolic debugger can provide a user when data value problems occur. We provide a data flow algorithm that detects the impact of two global transformations: *code hoisting* and *dead code elimination*. These two transformations capture the effects of global optimizations that cause data value problems. Common optimizations such as dead store elimination, common subexpression elimination, loop invariant code motion, partial redundancy elimination, or non-speculative global instruction scheduling can be expressed in terms of these transformations.

1 Introduction

Designing a symbolic debugger for globally optimized code is considered difficult, and implementors have generally avoided supporting the debugging of optimized code in the past. A symbolic debugger must solve two types of problems: First, the debugger must be able to both map a source statement to an instruction in the object code to set a breakpoint and map an instruction to the source code to report a fault or user interrupt. Problems relating to these control flow mappings are called *code location* problems [15]. Second, the debugger must be able to retrieve and display the values of source variables in a manner consistent with what the user expects with respect to the source statement where execution has halted. Problems relating to retrieving source values are called *data value* problems [15]. When a program has been compiled with optimizations, mappings between breakpoints in the source and object code become complicated, and values of variables may either be inaccessible in the run-time state or inconsistent with what the user expects.

Optimizations create data value problems by reordering or eliminating assignments to source variables. Since the debugger interacts with the user, only values of *source program* variables are of interest; compiler-internals (temporaries) are never visible to the user. Therefore, data value problems can be handled by restricting attention to transformations that affect assignments to source level variables. Many scalar optimizations, such as strength reduction, constant folding and constant propagation [3] do not affect assignments to source variables. Other optimizations like loop induction variable strength reduction and elimination, allow the debugger to infer source values from compiler temporaries that replace eliminated variables; such an approach was used in both the DOC [9] and CXdb [6] debuggers. Control flow transformations, such as loop unrolling and code replication, change the control flow graph by duplicating basic blocks. These transformations, however, do not reorder the execution of source-level assignments and thus do not cause data value problems.

Of the large number of transformations commonly implemented in a state-of-the-art compiler, only two reorder or eliminate source-level assignments and therefore can cause data value problems for debugging: code *hoisting* and dead code *elimination*. These two transformations capture the effects of global optimizations that cause data value problems, i.e., optimizations such as dead store elimination, common subexpression elimination, loop invariant code motion [3], partial redundancy elimination [14, 7], or global instruction scheduling algorithms that perform *non-speculative* hoisting of instructions [5], can be expressed in terms of these transformations[1].

When code hoisting and dead code elimination have reordered or eliminated assignment expressions, the value in the run-time location of a variable V may no longer correspond to the value of V that the user expects based on the context of the source statement where execution has halted. Such a variable V is called an *endangered* variable [11, 8, 1]. There are two ways in which a variable V can be endangered: the value in the run-time location of V may be either a value of V that has been assigned prematurely by a hoisted assignment expression, or a value that is stale because a dead assignment expression assigns the expected value of V, i.e., the value of V is not up to date. If the run-time value of V is the expected source value of V, then V is said to be *current* (i.e., V is not endangered).

It is important that the debugger detect endangered variables and provide additional information to the user in response to a query of an endangered variable; otherwise the user may be misled by the value displayed by the debugger. A debugger that interacts with the user in such a way is said to behave *truthfully*

[1] The same techniques as those described in this paper have been developed for code sinking and speculative code hoisting transformations. These transformations can also affect assignments to source variables, but they are not as prevalent as non-speculative code hoisting and dead code elimination. Therefore, for the sake of conciseness, we do not discuss them in this paper.

[15]. The techniques described in this paper allow the debugger to detect endangered variables caused by code hoisting and dead code elimination, and convey to the user, in source terms, how optimizations have affected the value of a variable.

The effects of code hoisting and dead code elimination can be described as properties that can be analyzed and propagated by data flow analysis. Data flow analysis is a well-understood technique, so there are limited obstacles to overcome for an implementation of these techniques in a production compiler/debugger. However, the data flow analysis is to be performed on the object code; it is not sufficient to use the results of the data flow analysis done by the compiler. Analysis at the object code level allows the debugger to determine exactly when a variable becomes endangered or current.

Our model of symbolic debugging assumes that the debugger is *non-invasive* [2, 1]; that is, the code generated by the compiler and debugged by the user, is the default code generated with optimizations enabled. The compiler is not allowed to insert additional instructions into the object code to enable or simplify symbolic debugging; we are interested in debugging the *optimized* version of a program.

2 Prior work

Hennessy [11] defined some of the basic terms (e.g., endangered, non-current) and presented measurements of the effects of some local optimizations. Zellweger's thesis [15] concentrated on code location problems.

Our work concentrates on developing a framework for the debugging of (locally and globally) optimized code. As part of this framework, we developed an approach to handle the problems caused by global register allocation and local code scheduling [2, 1]. We do not discuss these aspects any further in this paper. The solution described in this paper is an integral part of this framework, resulting in a debugger that handles the range of global and local scalar optimizations that are included in current state-of-the-art compilers.

DOC [9] and CXdb [6] are examples of two real debuggers for optimized code. These debuggers, however, do not deal with data value problems caused by global optimizations. Their other shortcomings and constraints are discussed in detail in [2, 1].

Other researchers that have investigated the problem of detecting endangered variables caused by global optimizations are Copperman [8] and Wismueller [4]. Both [8] and [4] concentrate on detecting when a variable is current, using data flow analysis of intermediate representations of the program. They do not determine the exact cause of endangerment (i.e., where in the source an endangered variable has been modified), and therefore they cannot provide such information to the user. Furthermore, they do not distinguish between nonresident, suspect and noncurrent variables. [8] does not deal with faults and user interrupts.

Both of these prior works assume that the compiler can mangle the source code arbitrarily, resulting in an arbitrarily difficult problem and in a solution that is difficult to both understand and implement. Our work takes advantage of the fact that optimizations do not transform code arbitrarily. There are a number of invariants that are preserved when a correct compiler transforms a program, and a debugger can exploit these invariants. Both [8] and [4] assume arbitrary code movement and elimination, and fail to recognize that movement and elimination are not unconstrained. If code is hoisted to a different basic block, the basic block is post-dominated by the original block; this limits the range of locations where a variable may be endangered. Or if an expression is eliminated due to redundancy, the value must be available somewhere, and the debugger can provide this value to the user.

By exploiting the invariance maintained by the compiler, our debugger finds a tractable solution to the data value problems that is practical.

3 Debugger model

The debugger is invoked as a result of a *break* that halts the execution of the program either *within* a statement, i.e., during the execution of an operation within a statement, or *at* a statement boundary, i.e., at a point between two statements in the source. A *synchronous* break occurs when control reaches a *control breakpoint* that was set at a source statement by the user. When a synchronous break invokes the debugger, the debugger reports that execution has stopped *at* the statement S where the user has set a breakpoint. S is referred to as the *control reference statement*. A break may also be *asynchronous*, for instance as a result of a user interrupt (i.e., the user types Ĉ or hits a `break` key), an exception (e.g., division by zero, overflow for fixed-sized arithmetic, segmentation fault, etc ...), or a *data breakpoint* (i.e., the next instruction to be executed references a memory location at which the user has set a data breakpoint). When an asynchronous break occurs, the debugger maps the instruction I, at which execution is halted, to the operation O in the source, for which I was generated. O is referred to as the *stopping operation* in the source, and corresponds to the source operation where the fault, data break or interrupt occurred. When an asynchronous break occurs, the debugger reports that execution has stopped *within* the control reference statement S containing O.

When the debugger is invoked, there exists a well-defined *stopping instruction* that caused the break. We assume a target machine with *precise interrupts* [12], so that at a stopping instruction I all prior instructions have executed (i.e, their effects are recorded), none of I's effects are visible (so I has not been executed), and none of the instructions subsequent to I have executed.

If global register allocation has been performed, then the register assigned to a variable V may be shared with other variables as well as temporaries. Thus at a break, V will be *nonresident* [2] if the value in the register assigned to V may be the value of some variable other than V. The debugger reports that the value of a nonresident variable V is unavailable since the value in the register assigned to V does not correspond to a meaningful source-level value of V [2]. If at a break a variable V is resident, then the value in the run-time location of V corresponds to some source-level value of V. This value is referred to as the *actual value* of V, while the value that the user expects V to have based on the context of the source reference statement, is the *expected value* of V. Since the actual value of a variable V is a source-level value of V, it is meaningful to display this value to the user. However, because of optimizations such as instruction scheduling or dead code elimination, the actual value of a variable V may not correspond to the expected value of V, in which case V is *endangered* and additional information should be provided to the user. There are two mutually exclusive classes of endangered variables: *noncurrent* variables and *suspect* variables. Sometimes the debugger can tell that the actual value of V definitely does not correspond to the expected value of V, in which case the actual value of V is displayed to the user with a warning that V is *noncurrent* [11, 8, 1]. However, there are situations when a debugger cannot tell whether V's actual value corresponds to V's expected value, in which case the user is warned that V is *suspect* [1]. If a variable V is endangered, then the debugger can provide additional information to the user about V, such as the source assignment expression(s) that (may have) assigned V's actual value, or the source assignment expression(s) that (possibly) should have assigned V's expected value. Of course if the debugger can determine that the actual value of V definitely corresponds to the expected value of V, then V is *current* and V's actual value is displayed without warnings.

4 Terminology

A control flow graph is a directed graph (B, S, E) where B is the set of basic blocks; $S \in B$ is the entry block; E is the set of edges between blocks such that if $(B_i, B_j) \in E$ then control may immediately reach

B_j from B_i. Each basic block B_i contains a sequence of instructions generated by the compiler, as well as a special *pre-amble instruction* that appears before the other instructions in B_i thus dominating them and a *post-amble instruction* that appears after the other instruction in B_i thus post-dominating them. The pre-amble and post-amble instructions are abstractions used by our algorithms; they are not generated by the compiler nor do they appear in the object code. The pre-amble instruction of a block B_i is denoted by $Preamble(B_i)$, while the post-amble instruction of a block B_i is denoted by $Postamble(B_i)$. Given an instruction I, we define the set of predecessor instructions of I, denoted $pred(I)$, as the set of instructions from which control can immediately reach I. A *point* is defined as being either between two adjacent instructions, before the first instruction in a basic block, or after the last instruction in a basic block. The point immediately before an instruction I is denoted $pre(I)$, and the point immediately after I is denoted $post(I)$. The *entry point* of the control flow graph is the point at the beginning of the source basic block S, and is denoted by *start*. The entry point dominates all other points in the control flow graph. A *path* is defined to be a sequence of points $\langle p_1, ..., p_n \rangle$ such that for each adjacent pair p_i, p_{i+1}, either $p_i = pre(I)$ and $p_{i+1} = post(I)$ for some instruction I, or p_i is a point at the end of a block B_j and p_{i+1} is a point at the beginning of a block B_k and $(B_j, B_k) \in E$. An instruction I is part of a path P, denoted $I \in P$, if $pre(I)$ and $post(I)$ both occur in P, and $pre(I)$ occurs before $post(I)$ along P. A basic block B_i is part of a path P, denoted $B_i \in P$, if $Preamble(B_i) \in P$.

5 Example

In this section, we present an example that illustrates how dead code elimination and code hoisting transform a program. This example is used to introduce terminology pertaining to these transformations. In the next sections, this example is used to illustrate the code location and data value problems these transformations cause for symbolic debugging. Figure 1(a) shows the intermediate representation (IR) flow graph of a program fragment, before optimizations and code generation. This program fragment contains exactly five expressions $E_1 ... E_5$ that assign to a source-level variable `x` and exactly two uses of `x`, one after E_4 inside block B7 and one in block B9. There are no assignments to variables `y` and `z` in this fragment. Figure 1(b) shows the flow graph of the object code after code hoisting and dead code elimination transformations and code generation. The destination register of an instruction is the first operand of the instruction. We assume that other IR expressions and object instructions exist in the program. Variable `x` has been assigned a register represented symbolically by `Rx` in Figure 1(b) and no other definitions of the register `Rx` exist other that the ones shown in the figure.

In this example, assignment expressions E_1 and E_4 have been translated to instructions I_1 and I_3 respectively. Since I_1 and I_3 assign a source-level value of `x` to register `Rx`, these instructions are known as *source definitions* of `x` [2].

In Figure 1(a), variable `x` is dead after expressions E_2 and E_3. Dead code elimination has eliminated these assignment expressions, so that no code has been generated for E_2 or E_3 in Figure 1(b). Expressions that are eliminated by dead code elimination are referred to as *dead* expressions.

In Figure 1, partial redundancy elimination [14] has eliminated expression E_5 and has inserted instruction I_2 into block B4. The expression `x=y+z` is partially available [14] at B8 since this expression is available along paths that reach B8 from B7, but not along paths that reach B8 from B6. In Figure 1(b), a copy of the expression `x=y+z` has been inserted into block B4 (instruction I_2). This insertion makes expression `x=y+z` available along all paths that reach B8 and thus E_5 has been eliminated. In effect, this insertion has hoisted E_5 up from B8 to B4.

The Morel and Renvoise partial redundancy algorithm [14] and its variants [7, 10, 13] compute the set of program points where the insertion of an expression E will make another partially redundant instance of E fully redundant. We refer to all such expressions that are inserted by code hoisting transformations as *hoisted* expressions; the expressions that are made redundant by the insertions and thus eliminated from the program, are referred to as *redundant* expressions. If a hoisted expression E_h is inserted to make one other expression E_r redundant, then E_r is referred to as the *redundant copy* of the hoisted expression E_h and E_h is referred to as a *hoisted copy* of the redundant expression E_r. In general, there will be exactly one redundant copy of a hoisted expression E_h whereas there may be zero or more hoisted copies of a redundant expression E_r. The redundant copy of a hoisted expression E_h is represented by $RedCopy(E_h)$.

Referring back to Figure 1(b), since I_2 assigns a source level value of `x` to `Rx`, I_2 is a source definition of `x`. However, this source definition is one that has been generated for a hoisted expression, and thus I_2 is referred to as a *hoisted definition* of `x`. That is, a hoisted definition of a variable V is a source definition of V that was generated for a hoisted assignment expression. We extend *RedCopy* to apply to hoisted definitions: if D is a hoisted definition generated for a hoisted expression E_h, then $RedCopy(D)$ is the redundant copy of E_h (i.e., $RedCopy(D) = RedCopy(E_h)$). Note that a hoisted definition is an instruction, whereas the redundant copy of a hoisted definition is an expression in the IR. Note also, that the value computed by a hoisted definition D corresponds to the source-level value computed by the redundant expression $RedCopy(D)$. In Figure 1(b), $E_5 = RedCopy(I_2)$ and thus the value assigned to `Rx` by the hoisted definition I_2, corresponds to the source-level value assigned by the redundant expression E_5.

In this paper, we assume that the flow graphs of the IR and the object code are isomorphic, and corresponding blocks in the IR and object code are given the same name. The basic block containing an expression E in the IR is referred to as $Block(E)$, while the basic block containing an instruction I in the object code is referred to as $Block(I)$. This assumption does not limit the applicability of our approach.

6 Code location problems

Code hoisting and dead code elimination make it difficult for the debugger to implement functions that allow the user to manipulate program control. For instance, consider a run-time exception at instruction I_2 in Figure 1(b). The debugger conveys this break to the user by reporting that execution halted during the execution of expression E_5 (Figure 1(a)). This expression, however, has been hoisted from block B8 to block B4. Thus the debugger must inform the user that E_5 has been moved to an earlier point in the program and report a control reference statement inside block B4 (in the source); i.e., the debugger reports that an exception occurred inside E_5 but selects as the control reference statement, a source execution point that maps to B4. Similarly, if the user sets a control breakpoint at a statement S that has been eliminated or hoisted, then the debugger maps the breakpoint to a point in the object within the block containing the dead or redundant IR expression corresponding to S. We do not discuss code location problems in this paper.[2]

7 Data value problems

In this section, we use the example of Figure 1 to illustrate how code hoisting and dead code elimination cause endangered variables.

[2] Zellweger [15] contains a discussion of code location problems. The model of breakpoint mapping we have described is what Zellweger and others [8, 4] refer to as *syntactic breakpoints*.

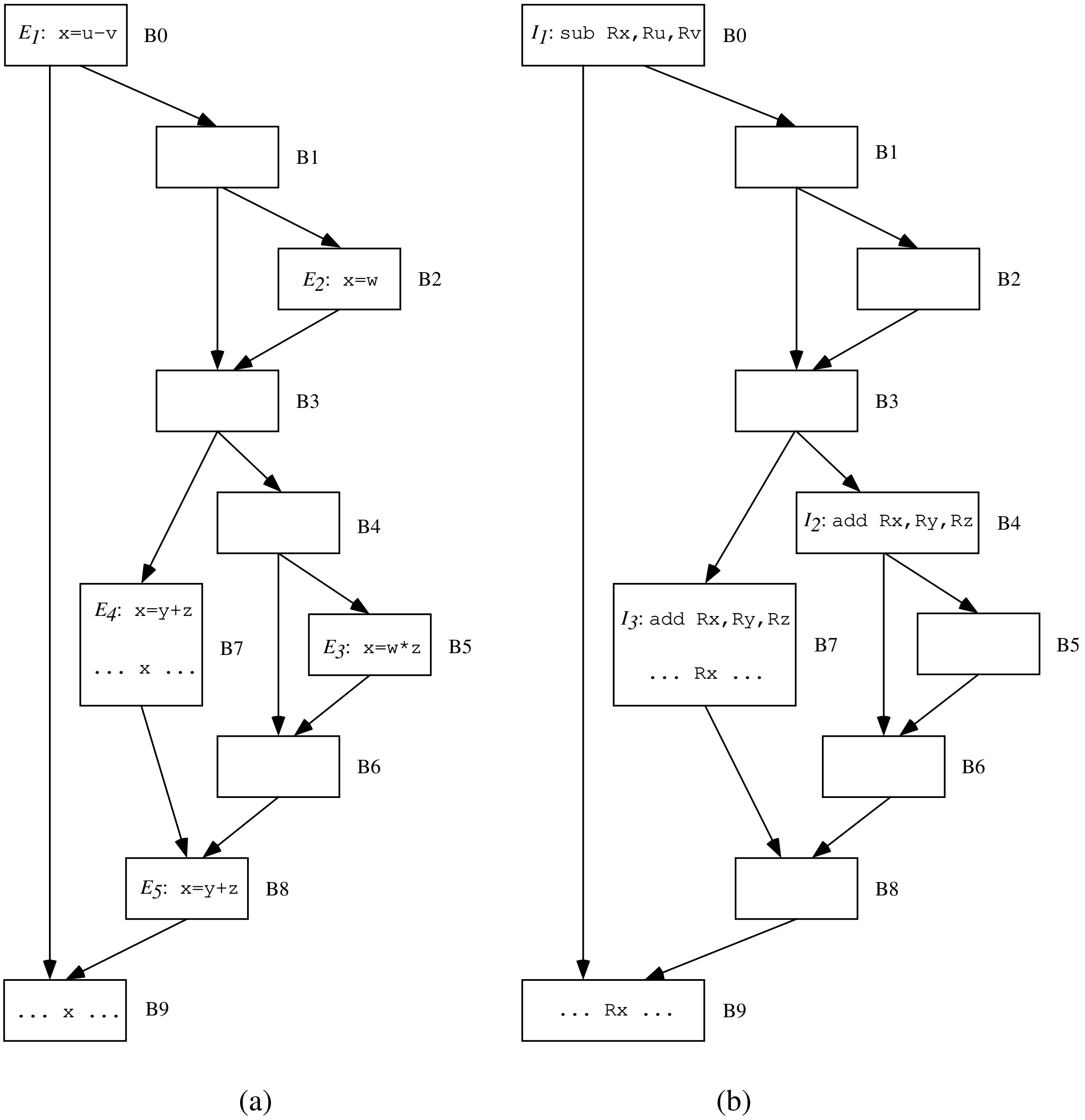

Figure 1: Control flow graphs (a) IR after translation from source (b) Object program after optimizations and code generation

7.1 Endangered variables caused by code hoisting

Code hoisting causes endangered variables by hoisting an expression that assigns to a source-level variable V, thus causing V to be updated prematurely. If the debugger can detect that the value in the run-time location of a variable V is *definitely* from a source-level assignment that has executed prematurely at the break, the debugger reports that V is *noncurrent*. However, control flow introduces ambiguities and the debugger will in general not know which execution path was traversed to reach a break. Thus, the debugger will sometimes detect that the actual value of a variable V is *possibly* from a source-level assignment that has executed prematurely and V is reported as *suspect*. In the case that a variable V is either noncurrent or suspect due to the (possible) premature execution of a source-level assignment, the debugger conveys in source-level terms what the actual value of V (possibly) corresponds to.

7.1.1 Noncurrent variables caused by code hoisting

Consider a stopping instruction I that occurs after I_2 in block B4, B5 or B6 (Figure 1(b)). The actual value of `x` at I will be the value computed by instruction I_2 which corresponds to the source-level value assigned by E_5. The expected value of `x` at the various breaks within these blocks (Figure 1(a)), is the value assigned by E_1, E_2 or E_3, depending on the control reference statement and the execution path leading to the break. Clearly, at all of the above breaks, `x` will be noncurrent, since the actual value of `x` will be different from the expected value of `x` due to the hoisted definition I_2 prematurely assigning to `Rx` the source-level value assigned by E_5. If the user queries the value of `x` at any of these breaks, the debugger can display the actual value of `x` and warn the user that this value is the value assigned by expression E_5, which has executed prematurely.

7.1.2 Suspect variables caused by code hoisting

Now consider a break B inside block B8. Since the expression `x=y+z` is available at B8, the value in `x`'s run-time location at any stopping instruction inside B8, is the value assigned in the source by E_5. Assume that the control reference statement S associated with break B occurs after E_5 inside B8. At break B, `x` is current, since the expected value of `x` is the value assigned by E_5, which corresponds to the actual value of `x`. Now assume that S is before E_5. If execution had reached B8 from B7, then `x`'s expected value would be the value assigned by E_4. The values assigned by E_4 and E_5 are equivalent and therefore `x` would be current at break B. However, if execution had reached B8 from B6 then the expected value of `x` would be the value assigned by any of the expressions E_1, E_2 or E_3 (depending on the path taken) and thus `x` would be noncurrent at break B. In the absence of knowledge regarding execution history, the debugger cannot determine whether execution has reached B8 via B7 or B6, and therefore the debugger cannot determine whether `x` is current or noncurrent and must report `x` as being *suspect* at break B. If the user queries the value of `x` at this break, the debugger can display the actual value of `x` and warn the user that this value *may* be from expression E_5, which *may* have executed prematurely at block B4.[3] The user may be able to determine whether block B4 was executed (e.g., based on the values that determine the outcome of block B3's conditional branch) and thus whether E_5 was indeed executed prematurely. Note that the compiler could have instrumented the object code to collect run-time information (e.g., path determiners [15]), allowing the debugger to determine which path was taken to B8 and thus reporting `x` as either noncurrent or current. This is disallowed, however, in our non-invasive debugging model.

[3] Of course block B4 will be translated to some program point in the source. Also, since several hoisted copies of a redundant operation may exist, the debugger may have to report several such program points to the user. If the user determines that execution reached the break from any of these points, then the user can infer that the suspect variable is noncurrent.

7.2 Endangered variables caused by dead code elimination

Dead code elimination causes endangered variables by eliminating dead assignment expressions, thus eliminating updates to variables. In the case of an endangered variable *V* caused by dead code elimination, the expected value of *V* is the value that would have been assigned by a source-level assignment E_d that was eliminated by dead code elimination, while the actual value of *V* is the value assigned by a source-level assignment other than E_d. When both dead code elimination and code hoisting have been performed, it is possible that the expected value of a variable *V* be from a dead assignment, while the actual value of *V* be from an assignment that has been executed prematurely due to code hoisting; in this case *V* is endangered due to code hoisting. For example in Figure 1, the dead expression E_3 will never directly cause endangerment since I_2 has prematurely updated `x` at all breaks where E_3 assigns the expected value of `x`. Because of control flow ambiguities, dead code elimination can cause both noncurrent and suspect variables. When a variable is endangered due to dead code elimination, the debugger must convey in source-level terms what the actual value of an endangered variable *V* (possibly) corresponds to, as well as the fact that the assignment(s) that would have (possibly) assigned the expected value of *V* have been eliminated.

7.2.1 Noncurrent variables caused by dead code elimination

Consider a break *B* inside block B2. The actual value of `x` at break *B* is the value assigned by I_1, which corresponds to the source-level value assigned by E_1. If the control reference statement *S* associated with break *B* is before E_2, then the expected value of `x` will be the value assigned by E_1 and `x` will be current. On the other hand, if *S* is after E_2, then the expected value of `x` will be the value assigned by E_2, and thus `x` will be noncurrent at break *B*; i.e., the actual value of `x` is a stale value since `x` should have been updated by E_2. If the user queries the value of `x` at break *B*, the debugger can display the actual value of `x` and warn the user that this value does not correspond to the expected value of `x`, which is the value that would have been assigned by the eliminated expression E_2.

7.2.2 Suspect variables caused by dead code elimination

Now consider a break *B* inside block B3. The actual value of `x` is again the source-level value assigned by E_1. Depending on the path that was taken to this break, the expected value of `x` is the value assigned by either E_1 or E_2. If control had reached break *B* without going through B2, then both the expected and actual values of `x` would be the value assigned by E_1, and thus `x` would be current at break *B*. However, had the path through B2 been taken, then `x` would be noncurrent at break *B*. Since the debugger cannot disambiguate which path was actually traversed to reach break *B*, it cannot determine whether `x` is current or noncurrent and must report `x` as suspect at break *B*. If the user queries the value of `x` at break *B*, the debugger can display the actual value of `x` and warn the user that this value *may* not correspond to the expected value of `x`, which *may* be the value that would have been assigned by the eliminated expression E_2. The user may be able to determine whether E_2 should have executed (e.g., based on the values that determine the outcome of block B1's conditional branch). Variable `x` is similarly suspect at a break inside B4, with the stopping instruction before instruction I_2.

In the case where a variable *V* is either noncurrent or suspect due to dead code elimination, the debugger can perform reaching definitions analysis in the object code, to determine which source definition(s) reach a break and thus which source expression(s) (could have) assigned the actual value of *V*. This information can then be presented to the user. For example, in Figure 1, only the source definition I_1 reaches a break at B3, so in addition to telling the user that `x` is suspect, the debugger can tell the user that the actual value of x is the value assigned by E_1.

8 Detecting endangered variables caused by code hoisting

The intuition behind our approach to detecting endangered variables caused by code hoisting is illustrated by the following example. Let D be a hoisted definition of a variable V, let E_r be the redundant copy of D and let P be an execution path in the object, that passes through once both D and $Block(E_r)$. Consider a point O along this path where a break B has occurred. Assume that O occurs after D and that along P, there are no other instructions that define the register assigned to V, so that the actual value of V at B is the source-level value assigned by E_r. We make the following observations:

1. If, along path P, O is *before* the beginning of $Block(E_r)$ then E_r has executed prematurely and V is noncurrent at break B.

2. If O is *after* the end of $Block(E_r)$ along path P, then the expected value of V at break B (assuming no other assignments to V exist in the IR) is the value assigned by E_r and thus V becomes current after $Block(E_r)$. In other words, once execution has passed through $Block(E_r)$, E_r has no longer been executed prematurely by D and thus D no longer causes V to be noncurrent.

3. If O is *inside* $Block(E_r)$ then depending on whether the control reference statement is before or after E_r, V is either noncurrent or current respectively at break B.

Our approach to detecting endangered variables is as follows:

1. **Global analysis.** Data flow analysis is used to detect endangerment caused by D at breaks in blocks other than $Block(E_r)$ (cases 1 and 2 above). This data flow analysis determines whether possible execution paths leading to a break could have executed D without subsequently passing through $Block(E_r)$. Depending on whether this analysis finds all, only some or no paths along which this property holds, V is either noncurrent due to hoisted definitions, suspect due to hoisted definitions or unaffected by hoisted definitions.

2. **Local analysis.** The local analysis of [1] is used to detect whether V is endangered at breaks inside $Block(E_r)$. This analysis detects variables that are endangered due to transformations that reorder code locally (e.g., local instruction scheduling). Since E_r is redundant, it has been eliminated from the program and no instructions have been generated for it. However, at any break inside $Block(E_r)$, the run-time location of V contains the source-level value that would have been assigned by E_r, since the value of E_r is available upon entry to $Block(E_r)$ (assuming no other assignments to V's run-time location). Thus E_r can be modelled as executing at the pre-amble instruction of $Block(E_r)$, i.e., regardless of where the stopping instruction is in $Block(E_r)$, E_r can be considered as having executed since the value in the run-time location of V is the value that would have been assigned by E_r. E.g., in Figure 1, since the expression `x=y+z` is available upon entry to block B8, E_5 can be modelled as executing at the pre-amble instruction of block B8. By modelling the execution of E_5 in this manner, the algorithm of [1] will detect E_5 as having executed prematurely only at control reference statements before E_5.

In the remainder of this section, we describe our data flow analysis solution. Details of the local analysis is left out due to space considerations, and the reader is referred to [1].

The data flow analysis used to detect endangered variables due to code hoisting is similar to the *reaching definitions* data flow analysis [3]. In the following, we concentrate on endangerment caused by code hoisting. We begin with some definitions.

Definition 1 *An instruction I* **reaches along a path** *P in the object code, if $I \in P$ and the destination register of I is not defined by any other instruction in P after the last occurrence of $post(I)$.*

Definition 2 *An instruction I* **reaches a point** *O in the object code, if there exists a path $P = \langle start, ..., O\rangle$ such that I reaches along P.*

After the execution of a hoisted definition D of a variable V, the actual value of V will correspond to the source-level value that would have been assigned by the redundant expression $RedCopy(D)$. Let $P = \langle start, ..., O\rangle$ be an execution path in the object code traversed to a break B. If at break B the actual value of a variable V is the value assigned by a hoisted definition D, then V will be noncurrent at B if the expected value of V does not correspond to the source-level value that would have been assigned by $RedCopy(D)$. This will certainly occur if D reaches along P and after the last occurrence of D, P does not go through the beginning of $Block(RedCopy(D))$. Therefore, given paths along which a hoisted definition D reaches, those paths that do not go through $Preamble(Block(RedCopy(D)))$ are distinguished:

Definition 3 *A redundant assignment expression E_r* **hoist reaches along a path** *$P = \langle start, ..., O\rangle$ in the object code, if there exists a hoisted definition D such that $E_r = RedCopy(D)$ and D reaches along P and $Preamble(Block(RedCopy(D)))$ does not occur after the last occurrence of $post(D)$ along P.*

Note that hoist reaches is defined along paths in the object code, but is a property of redundant IR expressions, not instructions that are hoisted definitions. In Figure 1, expression E_5 hoist reaches along all paths from the start of block B0 to any point within block B6, since I_2 reaches along all such paths ($E_5 = RedCopy(I_2)$) and block B8 ($Block(E_5)$) does not occur along any of these paths. Expression E_5, however, does not hoist reach along any path from the start of block B0 to any point within block B9, since all such paths either go through block B8 or do not execute the hoisted definition I_2.

Lemma 1 *Let E_r be a redundant assignment expression that assigns to a variable V. If E_r hoist reaches along a path $P = \langle start, ..., O\rangle$ in the object code and P is the execution path traversed to a break B, then V is noncurrent at B due to the premature execution of E_r.*

Proof: After execution of P, the actual value of V corresponds to the source-level value assigned by E_r. However, since execution has not yet reached $Block(E_r)$, the expected value of V cannot correspond to the value that would have been assigned by E_r. Hence V is noncurrent at any break at O due to the premature execution of E_r.

Since the debugger does not know which execution path in the object was actually taken to reach a break, all possible paths must be considered. The following lemmas describe the two cases where a redundant assignment expression hoist reaches along all or only some of the paths that lead to a point O, where a break has occurred. Let E_r be a redundant assignment expression that assigns to a variable V:

Lemma 2 *If E_r hoist reaches along all paths leading to a point O in the object code, then V is noncurrent due to the premature execution of E_r, at any break occurring at O.*

Proof: Lemma 1 holds for all possible execution paths leading to O. Thus V is noncurrent due to the premature execution of E_r at any break B occurring at O regardless of the execution path traversed to reach break B.

Lemma 3 *If E_r hoist reaches along at least one but not all paths leading to a point O in the object code, then V is suspect due to the possible premature execution of E_r, at any break occurring at O.*

Proof: Lemma 1 holds for all least one but not all execution paths leading to O. Hence the debugger cannot determine statically whether V is current or noncurrent at any break B occurring at O and thus V is suspect at breaks occurring at O.

Data flow analysis on the object code is used to detect endangered variables caused by code hoisting. Two data flow analysis problems, *AllHoistReach* and *AnyHoistReach*, that together represent the conditions of Lemma 2 and Lemma 3 are defined:

Definition 4 *The predicate AllHoistReach(E_r, O) is true at a point O in the object code if the redundant assignment expression E_r hoist reaches along all paths leading to O.*

Definition 5 *The predicate AnyHoistReach(E_r, O) is true at a point O in the object code if the redundant assignment expression E_r hoist reaches along any path leading to O.*

Note that the data flow analysis problems involve properties of redundant assignment expressions in the intermediate representation, yet they are performed on the object code. If a break B occurs at a point O in the object where *AllHoistReach*(E_r, O) is true, then the variable assigned by E_r is noncurrent at break B due to the premature execution of E_r. Otherwise if *AnyHoistReach*(E_r, O) is true, then the variable assigned by E_r is suspect at break B due to the possible premature execution of E_r.

The in and out sets of the data flow analysis problem, with respect to an instruction I, are as follows:

- *AllHoistReachIn*(I) is the set $\{E_r : \mathit{AllHoistReach}(E_r, \mathit{pre}(I))\}$
- *AllHoistReachOut*(I) is the set $\{E_r : \mathit{AllHoistReach}(E_r, \mathit{post}(I))\}$
- *AnyHoistReachIn*(I) is the set $\{E_r : \mathit{AnyHoistReach}(E_r, \mathit{pre}(I))\}$
- *AnyHoistReachOut*(I) is the set $\{E_r : \mathit{AnyHoistReach}(E_r, \mathit{post}(I))\}$

The only difference between *AllHoistReach* and *AnyHoistReach* is the confluence operator; the kill and gen sets of the two analyses are the same and are represented by *HoistReachKill* and *HoistReachGen*. The confluence operator for *AllHoistReach* is set intersection, while that of *AnyHoistReach* is set union:

$$\mathit{AllHoistReachIn}(I) = \bigcap_{J \in \mathit{pred}(I)} \mathit{AllHoistReachOut}(J)$$

$$\mathit{AnyHoistReachIn}(I) = \bigcup_{J \in \mathit{pred}(I)} \mathit{AnyHoistReachOut}(J)$$

The set *HoistReachGen* is defined as follows. Let D be a hoisted definition, and let E_r be a redundant assignment expression such that $E_r = \mathit{RedCopy}(D)$. E_r hoist reaches the point *post*(D) immediately after D, thus D generates $\mathit{AllHoistReach}(E_r, \mathit{post}(D))$ and $\mathit{AnyHoistReach}(E_r, \mathit{post}(D))$:

- If D is a hoisted definition, then $\mathit{HoistReachGen}(D) = \{\mathit{RedCopy}(D)\}$, otherwise $\mathit{HoistReachGen}(D) = \{\}$.

The set *HoistReachKill* is the smallest set defined as follows. Since a redundant expression E_r is no longer hoist reaching along a path that passes through *Block*(E_r), the pre-amble of *Block*(E_r) kills *AllHoistReach* and *AnyHoistReach* for a redundant expression E_r:

- If E_r is a redundant assignment expression, then $E_r \in \mathit{HoistReachKill}(\mathit{Preamble}(\mathit{Block}(E_r)))$.

Any instruction that defines a register R kills *AllHoistReach* and *AnyHoistReach* for all E_r such that E_r is a redundant assignment expression that assigns to a variable V that has been assigned register R:

- Let I is an instruction that defines a register R. $\forall E_r$ such that

    1. E_r is a redundant assignment expression that assigns to a variable V and
    2. the variable V has been assigned register R

  $E_r \in HoistReachKill(I)$.

The in and out sets of the data flow equations for an instruction I are related by:

$$AllHoistReachOut(I) = (AllHoistReachIn(I) \setminus HoistReachKill(I)) \cup HoistReachGen(I)$$

$$AnyHoistReachOut(I) = (AnyHoistReachIn(I) \setminus HoistReachKill(I)) \cup HoistReachGen(I)$$

As described in Section 7.1.2, if a variable is suspect due to the possible premature execution of a redundant assignment expression E_r, then the debugger can provide additional information to the user by displaying in source-level terms the execution points at which hoisted copies of E_r execute. This information can be gathered at a point O in the object using a reaching definitions data flow analysis in the object code to compute the set $ReachingHoistDefs(E_r, O) = \{D : D \text{ reaches } O \text{ and } E_r = RedCopy(D)\}$. If a break occurs at O, the debugger informs the user of source execution points where hoisted copies of E_r execute, after mapping each instruction $I \in ReachingHoistDefs(E_r, O)$ to an execution point in the source.

9 Detecting endangered variables caused by dead code elimination

The approach to detecting endangered variables caused by dead code elimination uses a data flow analysis similar to that described in Section 8. The data flow analysis for detecting endangered variables at a point O in the object solves for the predicates $AllDeadReach(V,O)$ and $AnyDeadReach(V,O)$ for a variable V. $AllDeadReach(V,O)$ is true if along all paths leading to O the expected value of V is assigned by a dead assignment and the actual value of V is stale. $AnyDeadReach(V,O)$ is true if the conditions holds along only some paths. Thus at a break occurring at O, V is noncurrent due to dead code elimination if $AllDeadReach(V,O)$ is true; otherwise it is suspect if $AnyDeadReach(V,O)$ is true. If neither predicate is true then V is not endangered due to dead code elimination.

Let E_d be a dead assignment expression that assigns to a variable V. Let D be a source definition of V and P be an execution path in the object, such that D and $Block(E_d)$ occur once in P, D occurs after $Block(E_d)$ and $Block(D) \neq Block(E_d)$. Assume that between $Block(E_d)$ and D along P there are no other blocks containing dead assignments to V and no other source definitions of V. Consider a point O along P, where a break B has occurred. We make the following observations:

1. If O is *inside* $Block(E_d)$ then depending on whether the control reference statement is before or after E_d, V is either current or noncurrent respectively at break B.

2. If, along path P, O is *after* $Block(E_d)$ but *before* D then at break B, the expected value of V is the value that would have been assigned by E_d which does not correspond to the actual value of V. V is noncurrent at break B due to the elimination of the dead assignment expression E_d.

3. If O is *after* D along path P, then the actual value of V is the value assigned by D. At break B, this value may either be the current value of V, or the value from a prematurely executed assignment to V. In any case, once execution has passed D, the elimination of E_d no longer causes V to be noncurrent.

Our approach to detecting endangered variables caused by dead code elimination again divides the problem into a global and local component:

1. **Global analysis.** Data flow analysis is used to detect endangerment caused by dead code elimination of E_d at breaks in blocks other than $Block(E_d)$ (cases 2 and 3 above). This data flow analysis determines whether possible execution paths leading to a break could have passed through $Block(E_d)$ for some dead assignment expression E_d that assigns to a variable V, without subsequently executing a source definition D of V. Depending on whether this analysis finds all, only some or no paths along which this property holds, V is either noncurrent due to dead code elimination, suspect due to dead code elimination or unaffected by dead code elimination.

2. **Local analysis.** The local analysis of [1] is used to detect whether V is endangered at breaks inside $Block(E_d)$ for some dead assignment expression E_d that assigns to variable V. Since E_d is dead, it has been eliminated from the program and no instructions have been generated for it. However, at any break inside $Block(E_d)$, the run-time location of V will never contain the source-level value that would have been assigned by E_d. Thus E_d can be modelled as executing at the post-amble instruction of $Block(E_d)$, i.e., regardless of where the stopping instruction is in $Block(E_d)$, the execution of E_d has been delayed until the end of $Block(E_d)$. E.g., in Figure 1, by modelling the execution of E_2 as occurring at the end of block B2, the algorithm of [1] will detect E_2 as having not executed when it should have, only at control reference statements within block B2 that occur after E_2.

Unlike the *HoistReach* data flow algorithm where we solved for whether a *redundant IR expression* is hoist reaching, the data flow algorithm for detecting endangered variables caused by dead code elimination solves for whether a *variable* V is endangered due to the elimination of some dead assignment to V. We again leave out details of the local analysis.

After execution passes through the block containing a dead assignment to a variable V, the actual value of V is stale until a source definition of V is executed:

Definition 6 *A variable V is* **dead reaching along a path** $P = \langle start, ..., O\rangle$ *in the object code, if there exists a dead assignment expression E_d that assigns to V such that $Postamble(Block(E_d)) \in P$ and no source definitions of V occur along P after the last occurrence of $Postamble(Block(E_d))$.*

If a variable V is dead reaching along a path P, and P is the execution path traversed to a break B, then V is clearly noncurrent at B:

Lemma 4 *If a variable V is dead reaching along a path $P = \langle start, ..., O\rangle$ in the object code and P is the execution path traversed to a break B and V is not noncurrent due to the premature execution of a redundant assignment, then V is noncurrent at B because the actual value of V is stale.*

Proof: After execution of P, the expected value of V is from some dead assignment expression E_d. The actual value of V is from an assignment that occurs earlier than E_d in the source. Hence the expected and actual values of V do not correspond and V is noncurrent at break B.

The development of the data flow problem is similar to that of Section 8 (proofs of lemmas are very similar to those of Lemmas 2 and 3, and have been omitted for the sake of conciseness):

Lemma 5 *If a variable V is dead reaching along all paths leading to a point O in the object code, then V is noncurrent at any break occurring at O.*

Lemma 6 *If a variable V is dead reaching along at least one but not all paths leading to a point O in the object code, then V is suspect at any break occurring at O.*

Definition 7 *The predicate AllDeadReach*(V,O) *is true at a point O in the object code if the variable V is dead reaching along all paths leading to O.*

Definition 8 *The predicate AnyDeadReach*(V,O) *is true at a point O in the object code if the variable V is dead reaching along any path leading to O.*

The definitions of the sets *AllDeadReachIn*, *AllDeadReachOut*, *AnyDeadReachIn* and *AnyDeadReachOut*, as well as the confluence operators and flow equations for these sets are similar to the corresponding sets of the data flow analysis in Section 8; we omit the details for conciseness. The set *DeadReachGen* is the smallest set defined by:

- If E_d is a dead assignment expression that assigns to a variable *V*, then $V \in$ *DeadReachGen*(*Postamble*(*Block*(E_d))).

The set *DeadReachKill* is the smallest set defined by:

- If *I* is a source definition of *V*, then $V \in$ *DeadReachKill*(*I*).

If a variable *V* is endangered due to dead code elimination, then it is useful to inform the user which source statements assign the actual and expected values of *V* (as discussed in Section 7.2). At a break the debugger performs a reaching definitions analysis *in the object* to detect the set of reaching instructions that are source definitions. Using this information the debugger can inform the user which source-level expression assigns (or which expressions may assign) the *actual* value of *V*. To inform the user of which expressions assign the *expected* value of *V*, the debugger performs reaching definitions analysis *in the source*.

10 Putting it all together

This section describes how the above data flow problems fit into the rest of our debugger framework, thus providing an integrated approach that handles global register allocation, local instruction scheduling and global transformations. We describe the steps taken by the debugger to detect endangered variables at a break. Consider a break at an instruction *I* in the object. First the debugger uses the results of the *evicted* data flow analysis described in [2] to detect nonresident variables; these variables are reported as unavailable to the user. Then, the debugger uses the results of the *AllHoistReach* and *AnyHoistReach* analysis to detect variables that are endangered due to code hoisting transformations. These variables are ones that have assignment expressions in the sets *AllHoistReachIn*(*I*) and *AnyHoistReachIn*(*I*). Variables that are endangered due to code hoisting are reported to the user in the manner described in Section 7.1. Then local analysis is performed to detect endangered variables caused by local code reordering and by redundant or dead expressions within *Block*(*I*); details of this analysis are described in [1]. Finally, the results of the *AllDeadReach* and *AnyDeadReach* analysis are used to detect and report endangered variables caused by dead code elimination. These are variables that are in *AllDeadReachIn*(*I*) and *AnyDeadReachIn*(*I*). Section 7.2 describes how these endangered variables are conveyed to the user.

11 Conclusion

There are four noteworthy aspects of our approach that allow us to proceed in solving a problem that researchers have struggled with in the past. First, the approach described in this paper concentrates on the two principal global transformations: code hoisting and dead code elimination. Common global optimizations, either can be expressed in terms of these optimizations or do not cause data value problems. Second, the approach takes advantage of invariants maintained by these two transformations. This makes the problem tractable and enables us to provide additional information to the user by conveying the actual value of a variable in source terms. Third, the solutions described are integrated with the implemented solutions to problems caused by local instruction scheduling and global register allocation, described in [1] and [2]. Thus, our debugger is able to address a wide range of common global and local scalar optimizations included in most (if not all) research and production optimizing compilers of the last decade. Finally, the solution described in this paper is cast as a data flow analysis problem which is well understood by compiler developers. To gather the information required for this data flow analysis, the program intermediate representation is annotated during optimizations to mark hoisted, redundant and dead expressions. After code generation, the IR annotations are used to compute data flow sets for each instruction. The data flow analysis can be performed either by the compiler after optimizations and code generation, or by the debugger.

In summary, using the techniques described in this paper it is finally practical to implement a symbolic debugger for globally optimized code.

References

[1] A. Adl-Tabatabai and T. Gross. Detection and recovery of endangered variables caused by instruction scheduling. In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 13–25. ACM, June 1993.

[2] A. Adl-Tabatabai and T. Gross. Evicted variables and the interaction of global register allocation and symbolic debugging. In *Proc. 20th POPL Conf.*, pages 371–383. ACM, January 1993.

[3] A. V. Aho, R. Sethi, and Ullman J. D. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] L. Berger and R. Wismueller. Source-level debugging of optimized programs using data flow analysis. Draft paper, March 1993.

[5] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255. ACM, June 1991.

[6] G. Brooks, G. Hansen, and S. Simmons. A new approach to debugging optimized code. In *Proc. SIGPLAN'92 Conf. on PLDI*, pages 1–11. ACM SIGPLAN, June 1992.

[7] F. Chow. *A Portable, Machine-independent Global Optimizer Design and Measurements*. PhD thesis, Stanford University, 1984.

[8] M. Copperman. Debugging optimized code without being misled. Technical Report UCSC-CRL-93-21, UC Santa Cruz, June 1993.

[9] D. S. Coutant, S. Meloy, and M. Ruscetta. Doc: A practical approach to source-level debugging of globally optimized code. In *Proc. SIGPLAN 1988 Conf. on PLDI*, pages 125–134. ACM, June 1988.

[10] D.M. Dhamdhere. Practical adaptation of the global optimization algorithm of morel and renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1991.

[11] J. L. Hennessy. Symbolic debugging of optimized code. *ACM Trans. on Programming Languages and Systems*, 4(3):323–344, July 1982.

[12] J. L Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufman, 1990.

[13] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 224–234. ACM, June 1992.

[14] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb 1979.

[15] P. Zellweger. *Interactive Source-Level Debugging of Optimized Programs*. PhD thesis, University of California, Berkeley, May 1984. Published as Xerox PARC Technical Report CSL-84-5.