To appear in the
Proceedings of PLDI'96, ACM SIGPLAN'96 Conf. on Programming Language Design and Implementation, May 1996, Philadelphia, PA

Source-Level Debugging of Scalar Optimized Code

Ali-Reza Adl-Tabatabai[1] and Thomas Gross[1,2]

[1]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[2]Institut für Computer Systeme
ETH Zürich
CH 8092 Zürich

Abstract

Although compiler optimizations play a crucial role in the performance of modern computer systems, debugger technology has lagged behind in its support of optimizations. Yet debugging the unoptimized translation is often impossible or futile, so handling of code optimizations in the debugger is necessary. But compiler optimizations make it difficult to provide source-level debugger functionality: Global optimizations can cause the runtime value of a variable to be inconsistent with the source-level value expected at a breakpoint; such variables are called *endangered* variables. A debugger must detect and warn the user of endangered variables otherwise the user may draw incorrect conclusions about the program. This paper presents a new algorithm for detecting variables that are endangered due to global scalar optimizations. Our approach provides more precise classifications of variables and is still simpler than past approaches. We have implemented and evaluated our techniques in the context of the `cmcc` optimizing C compiler. We describe the compiler extensions necessary to perform the required bookkeeping of compiler optimizations. We present measurements of the effect of optimizations on a debugger's ability to present the expected values of variables to the user.

1 Introduction

Designing a source-level debugger for globally optimized code is still an open problem and users must choose between debugging or optimization. Compiler implementors have generally avoided supporting the debugging of optimized code in the past, although some systems provide debugger mechanics without warranties for the "debugging" of optimized code. To cite from the description of the options in the C manual of a major Unix vendor:

> **-g3** Have the compiler produce additional symbol table information for full symbolic debugging for fully optimized code. This option makes the debugger inaccurate.

For non-Unix systems, the situation is no better (to give an example from the PC/Macintosh world):

> **Enable Debugging** turns off all optimizations. When the compiler optimizes it sometimes rearranges object code so that it does not correspond exactly to the source code. This rearrangement may confuse the debugger's source code view.

A source-level debugger must solve two types of problems: First, the debugger must 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 (*code location* problems [26]). Second, the debugger must 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 (*data-value* problems [26]). When a program has been compiled with optimizations, mappings between breakpoints in the source and object code become complicated, and values of variables can be inaccessible in the runtime state or inconsistent with what the user expects.

This paper presents a solution to deal with the the data-value problems caused by optimizing compiler transformations. Our focus are the global and local *scalar* optimizations that are included in current state-of-the-art compilers. There are two novel aspects. First, we develop a unified model of the data-value problem caused by global scalar optimizations; Figure 1 sketches the questions that a debugger must address. Then we present an approach based on data-flow analysis to analyze and propagate the effects of optimizing

This research was sponsored in part by the Advanced Research Projects Agency/ITO monitored by SPAWAR under contract N00039-93-C-0152 and in part by donations from Intel Corp. and Motorola Inc.

transformations. These algorithms have been implemented in an optimizing C compiler: the data-flow analysis required to support the debugger is similar to the data-flow analysis performed for global optimization and in our compiler uses the same modules. Compiler extensions are only necessary to generate the information required to analyze the effect of optimizing transformations on the data-value problems. Moreover, our algorithms work on a single representation of a program; this is in contrast to other approaches (e.g., [24]) that keep around a copy of the original source program representation. Our paper concludes with an empirical evaluation; we report on measurements for the eight C programs of the SPEC92 suite.

1.1 The data-value problem

We summarize here the terminology; for examples and motivation we defer to the references.

When global register allocation is performed, the register assigned to a variable V may be shared with other variables as well as temporaries. Thus at a breakpoint there may be no value of V available; V is *nonresident* [3] 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 [3]. If at a breakpoint a variable V is resident, then the value in the runtime 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 breakpoint 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, if optimizations have moved or eliminated assignment expressions, the actual value of a variable V may not correspond to the expected value of V, in which case V is *endangered* [19, 13, 2] and additional information must 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* [19, 13, 2]. However, there are situations when a debugger cannot tell whether the actual value of V corresponds to the expected value of V, in which case the user is warned that V is *suspect* [2]. Suspect variables are caused by ambiguities due to either multiple paths reaching a breakpoint [1], or pointer assignment that are executed out of order [2]. If a variable V is endangered, the debugger can provide additional information about V to the user, such as the source assignment expression(s) that (may have) assigned the actual value of V, or the source assignment expression(s) that (possibly) should have assigned the expected value of V.

If the debugger can positively determine that the actual value of V corresponds to the expected value of V, then V is *current* and the actual value of V is displayed without warnings. The techniques described in this paper allow the debugger to detect whether a variable is noncurrent, suspect or current, and convey to the user in source terms how optimizations have affected the value of a variable.

1.2 Debugger model

Our model of debugging assumes that the debugger is *noninvasive* [3, 2]; 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 source-level debugging; we are interested in debugging the *optimized* version of a program.

Figure 1 orders the states of a variable with respect to debugging. The debugger can determine whether a variable should contain a source-level value by performing simple reaching analysis. This analysis helps, since a register promoted variable that is uninitialized is likely to be nonresident, and the debugger can provide a better classification by informing the user that the variable is uninitialized. If a variable is initialized, the debugger detects whether the variable is resident. If the variable is resident, the analysis described in this paper determines whether the variable is noncurrent, suspect, or current, and its value is shown to the user, possibly with a warning. This approach never misleads the user; an endangered values is always accompanied by a warning (that the value is noncurrent or suspect).

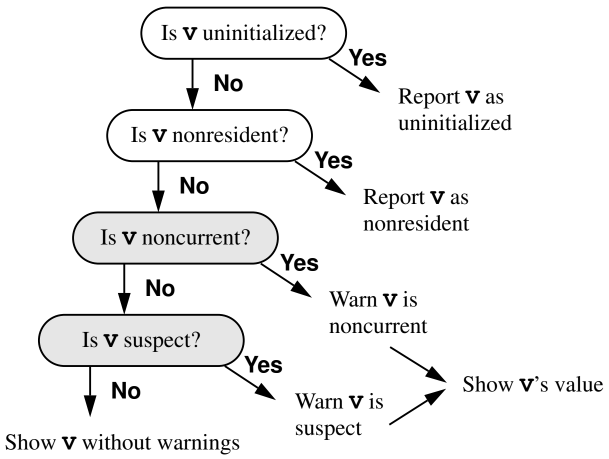

Figure 1: Outline of our algorithm; shaded part corresponds to the novel aspects discussed in this paper.

The above approach presents a baseline that can be improved, for example, by using runtime values to reconstruct a variable's expected value (*recovery* [19, 1]). Runtime values can also be used to differentiate between suspect and current values (e.g., by determining which path was taken to reach a specific breakpoint; an example is provided in Section 2.2). However, all of these refinements are improvements to this overall model, which therefore plays a central role in the organization of a debugger.

1.3 Prior work

There exists a small body of literature on debugging optimized code, starting with Hennessy's paper [19], which defined some of the basic terms (e.g., endangered, noncurrent). DOC [14] and CXdb [8] are examples of two real debuggers for optimized code. These debuggers detect whether a variable is nonresident (using a conservative approach based on a variable's live range [3]). CXdb only detects whether a variable is resident; it is up to the user to determine the correspondence between a variable's actual value and source code values. DOC can detect variables that are endangered due to instruction scheduling, but cannot deal with data-value problems caused by global optimizations. The shortcomings and constraints of these debuggers are discussed in detail in [3, 2, 1].

Our own earlier work [3, 2] concentrates on the effects of global register allocation and local code scheduling. We do not discuss these aspects any further in this paper, as underlined by Figure 1. This paper concentrates on a framework for the source-level debugging of *locally* and *globally* optimized code and discusses how to deal with endangered variables caused by these optimizations. Except for the italicized entries, we discuss the optimizations of Table 1 here for the first time. Register optimizations are included in our compiler (otherwise, we would not claim to have an optimizing compiler) and therefore included in our evaluation.

Other researchers that have investigated the problem of detecting endangered variables caused by global optimizations are Copperman [13] and Wismueller [24]. Both of these works provide formal frameworks but do not specify how their solutions can be extended to the problems faced by a real compiler. Copperman's approach [13] is based on data-flow analysis of intermediate representations of the program. This representation captures the effects of transformations (global optimizations), but does not cover all aspects of the translation (e.g., register allocation) and does not deal with faults and user interrupts. Without an implementation, it is difficult to evaluate the practicality of Copperman's approach. Wismueller [24, 25] concentrates only on detecting whether the expected value of a variable can be displayed to the user; his algorithms do not distinguish between nonresident, suspect, and noncurrent variables.

Both [13] and [24] 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 compilers transform programs. For example, if code is hoisted to a different basic block, the basic block is post-dominated by the original block; this limits the range of breakpoints where a variable is 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. [13] and [24] fail to take these constraints into account.

Another major difference between the earlier work by Copperman and Wismueller is that they attempt to capture a "summary effect" of all optimizations. Then they attempt to relate the optimized code back to the source code. In contrast, our approach models each optimization step (of Table 1); the compiler propagates the information about the effect of these optimization steps through all all optimization phases. This issue is discussed in more detail in Section 3.

program	`gcc -O2`	`cc -O2`
`li`	0.98	1.05
`eqntott`	1.13	1.09
`espresso`	1.06	1.05
`gcc`	1.02	0.89
`alvinn`	1.06	0.94
`compress`	0.84	0.95
`ear`	1.07	0.95
`sc`	1.09	1.03

Table 3: Performance of optimized code generated by `cmcc`, relative to optimized code generated by `gcc` (version 2.3.2) and MIPS `cc` on a DECstation 5000/200.

1.4 Experimental framework

The algorithms described in this paper are implemented in the `cmcc` compiler, CMU's optimizing C compiler. `cmcc` uses the `lcc` ANSI C front end [18]. Table 1 lists the optimizations performed by `cmcc`. These optimizations are based mostly on the standard bit-vector algorithms described in [12]. Our implementations of partial redundancy elimination, strength reduction and partial dead code elimination are based on the algorithms described in [21], [20] and [22]. The global register allocator is a Chaitin-style register allocator [9] with the improvements described in [7]. So far, we have retargeted `cmcc` to the MIPS, SPARC, DLX, and iWarp architectures. We used the MIPS code generator to gather the results that we report in this paper.

In this paper, we base our empirical evaluation on the set of eight C programs from SPEC92. Table 2 shows the sizes of these programs and statistics relevant to source-level debugging. The third and and fourth column of this table show the total number of source-level breakpoints in each program and the average number of breakpoints per function.

Loop unrolling and peeling	Linear function test replacement
Induction variable expansion	Induction variable simplification
Constant propagation and folding	Induction variable elimination
Assignment propagation	Partial dead code elimination
Dead assignment elimination	Partial redundancy elimination
Strength reduction	Branch optimizations
Global register allocation (using graph coloring)	*Register coalescing*
Instruction scheduling	

Table 1: Optimizations performed by `cmcc`.

Program	Lines of code	Total source breakpoints	Breakpoints per function	Variables per breakpoint
`li`	7741	2594	7.4	5.2
`eqntott`	3483	1267	21.6	5.1
`espresso`	14842	7424	21.5	9.4
`gcc`	102389	28433	20.7	9.3
`alvinn`	322	140	8.3	6.3
`compress`	1503	429	26.9	5.8
`ear`	4466	1108	11.8	6.9
`sc`	8491	3400	23.1	7.1

Table 2: Programs used in this study.

The last column shows the average number of local variables that were in scope at each source-level breakpoint. Table 3 shows that the optimized code generated by `cmcc` is roughly of the same quality as the optimized code produced by `gcc` and the native MIPS `cc` compiler. (This table presents the performance relative to these compilers; a number of less than one means that `cmcc` produces better code.)

2 Our approach

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 [4] do not directly affect assignments to source variables[1]. Other optimizations, like loop induction variable strength reduction and elimination, allow the debugger to infer source values from compiler temporaries that replace eliminated variables. There are other situations where the overall effect of a series of transformations is the replacement of a source-level variable with a compiler temporary, again allowing the debugger to infer values from compiler temporaries. 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.

Compiler transformations cause endangered variables by either *eliminating* or *moving* assignments to source variables. An assignment A may be eliminated because it is either *dead* (i.e., the value computed by A is never used), or *available* (i.e., the value computed by A is already computed on all paths leading to A). Code motion algorithms move an expression either upwards against the direction of control flow (hoisting), or downwards towards the direction of control flow (sinking). Code motion is constrained by *safety* considerations: a computation cannot be introduced into a path where it did not exist before. Therefore, code hoisting optimizations copy an expression E from a block B to one or more blocks that are *post-dominated* by B, while code sinking optimizations move an expression E from a block B to one or more blocks that are *dominated* by B. By taking advantage of these code motion invariants, our algorithms are greatly simplified. In fact, it is these invariants that have allowed us to produce a solution to the problem that is significantly simpler than the approaches described in [25] and [13]. Examples of code hoisting optimizations include partial redundancy elimination [23, 12, 15, 21] and global instruction scheduling algorithms that perform non-speculative hoisting of instructions [5][2]. Examples of code sinking optimiza-

[1] Their effects may be indirect, for instance, constant propagation or copy propagation may eliminate all uses of an assignment's left hand side, thus subjecting the assignment to elimination by dead code elimination.

[2] The algorithms described in this paper have been extended to deal with

tions include partial dead code elimination [22], unspeculation [17], global instruction scheduling algorithms that sink code past conditional branches [10], superblock dead code elimination [11] and forward propagation [6].

2.1 Core optimizations

Code motion and elimination are related, since some code motion algorithms operate by computing the set of program points where insertions of expressions render other expressions either dead or available [22, 23]. If the debugger can detect endangered variables caused by code hoisting and dead code elimination, then we have the foundation to debug optimized code, since these two transformations capture the effects of the elimination and movement transformations discussed above (including code sinking).

Code hoisting causes endangered variables by hoisting an expression that assigns to a source-level variable V, thus causing V to be updated prematurely. Dead code elimination causes endangered variables by 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 stems from a dead assignment, while the actual value of V stems from an assignment that has been executed prematurely due to code hoisting; in this case V is endangered due to code hoisting. Because of control flow ambiguities, both transformations may cause variables to be suspect; that is, the debugger may sometimes only be able to tell that a variable is *possibly* noncurrent due to code hoisting or dead code elimination. We now provide an example to clarify these concepts.

2.2 Illustration of the effect of code hoisting

Figure 2 shows the intermediate representation (IR) flow graph of a program fragment. Code hoisting has inserted expression E_3 inside block B2, rendering expression E_2 in block B3 redundant. We refer to expressions that are inserted by code hoisting (e.g., E_3) as *hoisted* expressions, while expressions that are made redundant by insertions and thus eliminated from the program (e.g., E_2) 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* $RedCopy(E_h)$ of the hoisted expression E_h. In Figure 2, $E_2 = RedCopy(E_3)$.

Figure 2 shows three possible breakpoints that may occur: Bkpt1, Bkpt2, and Bkpt3. At Bkpt1, x is definitely noncurrent, since the actual value of x is different from the expected value of x due to E_3 prematurely assigning to x the source-level value assigned by E_2. If the user queries the value of x at this breakpoint, the debugger can display the actual value of x and warn the user that this value is the value assigned by the source-level assignment E_2, which has executed prematurely. At breakpoint Bkpt2, x is current if execution has reached block B3 from block B1 and noncurrent if execution has reached from block B2. In the absence of knowledge regarding execution history, the debugger cannot determine whether execution has reached B3 via B1 or B2, and therefore the debugger cannot determine whether x is current or noncurrent and must report x as being *suspect* at breakpoint Bkpt2. If the user queries the value of x at this breakpoint, the debugger can display the actual value of x and warn the user that this value *may* be from expression E_2, which *may* have executed prematurely at block B2. The user may be able to determine whether block B2 has executed (e.g., based on the values that determine the outcome of block B0's conditional branch) and thus whether E_2 has indeed executed prematurely. Note that the compiler can instrument the object code to collect runtime information, allowing the debugger to determine which path was taken to B3 and thus reporting x as either noncurrent or current. This is disallowed, however, in our non-invasive debugger model.

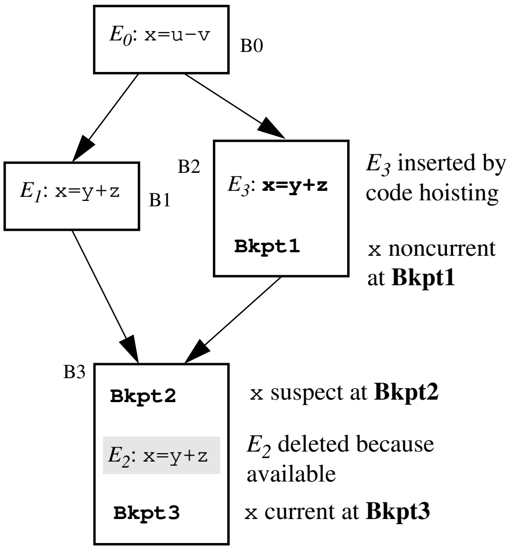

Figure 2: Example of code hoisting.

Now consider breakpoint Bkpt3. At this breakpoint, the expected value of x is the value assigned by E_2. The actual value of x is the value assigned by either E_1 or E_3, depending on the path traversed to this breakpoint. The values of either E_1 or E_3 are the same as the value that would have been assigned by E_2 (otherwise E_2 would not have been eliminated due to redundancy), and thus x is now current.

speculative code hoisting transformations [1].

2.3 Detecting endangered variables caused by code hoisting

The key idea is to determine if there exists a path to a breakpoint that includes a hoisted expression that is not followed (on the same path) by a redundant copy. Once execution has progressed to the point that all paths include a redundant copy, the variable is current, since the actual value of the variable is the expected value. Path problems are easily solved by an appropriate data-flow framework.

The data-flow analysis used to detect endangered variables due to code hoisting is similar to the *reaching definitions* analysis [4]. After the execution of a hoisted expression E_h that assigns to a variable V, the *actual* value of V corresponds to the source-level value that would have been assigned by the redundant expression $RedCopy(E_h)$; for example, in Figure 2, the actual value of x at breakpoint Bkpt1 is the value assigned by the hoisted assignment E_3, which is the source-level value that would have been assigned by E_2.

Let $P = \langle start, ..., O\rangle$ be an execution path traversed to a breakpoint B. If at B the actual value of a variable V is the value assigned by a hoisted expression E_h, then V is noncurrent at B if the *expected* value of V does not correspond to the source-level value that would have been assigned by $RedCopy(E_h)$. Therefore, if E_h reaches along P and after the last occurrence of E_h, P does not include $RedCopy(E_h)$, then V is noncurrent. Or, expressed positively, V is current whenever a path P includes $RedCopy(E_h)$, and E_h does not occur on P after $RedCopy(E_h)$. For instance, in Figure 2, E_3 reaches Bkpt1 and Bkpt3. x is noncurrent at Bkpt1, since any path taken to Bkpt1 does not go through E_2, and current at Bkpt3, since all paths taken to Bkpt3 go through E_2.

Therefore, given paths along which a hoisted expression E_h reaches, those paths that do not go through $RedCopy(E_h)$ are distinguished:

Definition 1 *A redundant expression E_r* **hoist reaches along a path** $P = \langle start, ..., O\rangle$, *if there exists a hoisted expression E_h such that $E_r = RedCopy(E_h)$, and E_h reaches along P, and E_r does not occur after the last occurrence of E_h along P.*

Note that hoist reach is a property of redundant expressions only (i.e., expressions that are eliminated by partial redundancy elimination). In Figure 2, the redundant expression E_2 hoist reaches Bkpt2 on the path from block B2. E_2 does not hoist reach on paths that reach via block B1.

Lemma 1 *Let E_r be a redundant assignment expression that assigns to a variable V. If E_r hoist reaches along a path $P = \langle start, ..., O\rangle$ and P is the execution path traversed to a breakpoint B, then V is noncurrent at B due to the premature execution of E_r*[3].

[3] We omit the proofs of lemmas as they are straight forward.

Since the debugger does not know which execution path was actually taken to reach a breakpoint, it must consider all possible paths. 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 breakpoint 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, then at any breakpoint occurring at O, V is noncurrent due to the premature execution of E_r.*

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

In Figure 2, x is noncurrent at Bkpt1, since the redundant assignment expression E_2 hoist reaches on all paths to Bkpt1. At Bkpt2, x is suspect since E_2 hoist reaches on only some paths. At Bkpt3, x is current since no expressions that assign to x hoist reach.

Detecting whether a redundant expression hoist reaches along all or only some paths can easily be done using data-flow analysis. This data-flow analysis is performed on the final instruction-level intermediate representation of a program, that includes annotations describing the effects of optimizations. In Section 3, we describe how this representation can be built and maintained by the compiler. The hoist reach data-flow attribute of an assignment expression E is generated by any code inserted by code hoisting that computes E. The hoist reach of E is killed by any eliminated redundant code that also computes E. Two flow analyses are done to determine whether an assignment expression hoist reaches on some or all paths to a breakpoint. Typically, a variable is endangered over a small region of the program, and only a few variables are endangered. Therefore, an efficient implementation of our analyses can be based on slotwise analysis [16]. (For more implementation details, see [1].) Note that the data-flow analysis does not need to determine which instance of an expression hoist reaches, but rather that *some* expression hoist reaches; that is, the compiler need not determine that $E_2 = RedCopy(E_3)$, but rather that E_3 is a hoisted instance of E, and that E_2 is redundant. Note also that given a redundant expression E_r that is the redundant copy of a hoisted expression E_h, E_r post-dominates E_h, and thus the hoist reach of E_r is eventually killed on any path leading from E_h. Therefore the region of endangerment caused by code hoisting is limited.

2.4 Detecting endangered variables caused by dead code elimination

The program fragment of Figure 3 is used to demonstrate the effects of dead code elimination on debugging. In this figure,

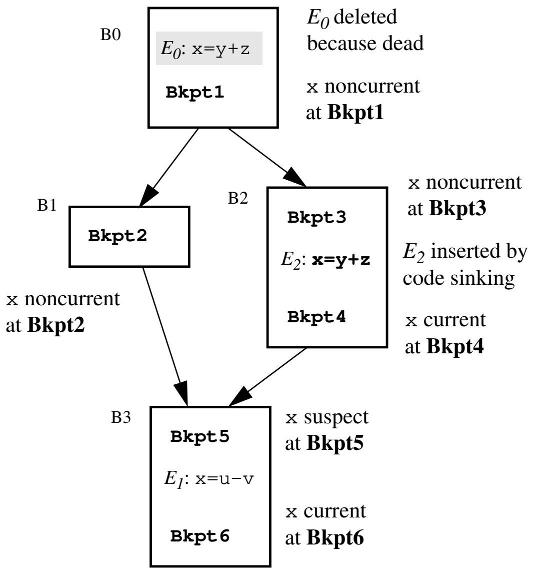

Figure 3: Example of dead code elimination.

assignment sinking has inserted E_2 and deleted expression E_0. At breakpoint Bkpt1, x's expected value is the value that would have been assigned by E_0, while x's actual value is the value assigned by the last assignment that was executed prior to this program fragment. Therefore, the actual value of x is stale, and x is noncurrent. x is similarly noncurrent at Bkpt2 and Bkpt3. At Bkpt4, x is current since expression E_2 assigns the expected value of x (i.e., the value that would have been assigned by E_0). At Bkpt5, x is noncurrent if execution has reached this breakpoint from block B1 and current if execution has reached from block B2. Therefore, x is suspect at Bkpt5. Finally, at Bkpt6, both the expected and actual values of x are from E_1, and thus x is current.

Unlike the hoist reaching 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. After execution passes through a dead assignment to a variable V, the actual value of V becomes stale until another assignment to V is executed. For example, in Figure 3, x becomes noncurrent after E_0, until after the assignments E_1 and E_2. Therefore, we distinguish those paths where a variable's value is stale due to a dead assignment:

Definition 2 *A variable V is* **dead reaching along a path** $P = \langle start, ..., O\rangle$, *if there exists a dead assignment expression E_d that assigns to V such that E_d occurs in P and no assignments to V occur along P after the last occurrence of E_d.*

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

Lemma 4 *If a variable V is dead reaching along a path $P = \langle start, ..., O\rangle$, and P is the execution path traversed to a breakpoint 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.*

Lemma 5 *If a variable V is dead reaching along all paths leading to a point O, then V is noncurrent at any breakpoint 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, then V is suspect at any breakpoint occurring at O.*

In Figure 3, x is dead reaching along all paths leading to Bkpt1, Bkpt2 and Bkpt3, and thus x is noncurrent at these breakpoints. At Bkpt5, x is dead reaching only along those paths that pass through B1 and thus x is suspect. At Bkpt6, x is not dead reaching and thus x is current.

The data-flow analyses to detect whether a variable is dead reaching on only some or all paths can be derived in a straight forward manner from the above definitions and lemmas. The dead reach of a variable V is generated by a dead assignment to V and killed by any other kind of assignment to V[1].

2.5 Recovery

If dead code elimination eliminates an assignment to a variable V, it may be possible to recover the expected value of V from the values of compiler temporaries. Consider the example in Figure 4(a). The right hand side of the expression x=y+z at statement S_1 is propagated to the two uses of x at statements S_2 and S_3. After this assignment propagation, no uses of x remain and S_1 is eliminated (Figure 4(b)). Common subexpression elimination detects the common subexpression y+z, replacing the two computations of y+z with fetches from the temporary tmp (Figure 4(c)).

S_1:	x=y+z				tmp=y+z
S_2:	..x..	S_2:	..y+z..	S_2:	..tmp..
S_3:	..x..	S_3:	..y+z..	S_3:	..tmp..
	(a)		(b)		(c)

Figure 4: Recovery example (a) Original source program (b) After copy propagation and dead code elimination (c) After common subexpression elimination.

In cmcc, assignment propagation is performed to improve partial redundancy elimination [12, 6] and the situation described above occurs quite often. The final effect of this series of transformations is that the source-level variable x is replaced with tmp. If the user queries the value of x at a breakpoint that occurs after statement S_1, the debugger can display the value of tmp, since these two variables are aliased. This is one form of *recovery*, where the debugger reconstructs the expected value of a variable from other runtime values.

Recovery is performed by checking each expression E inserted by code replacement transformations (Section 3 describes how we keep track of such transformations). If E replaced a fetch from a source-level variable V in the original program, then the value computed by E aliases V, and V can be recovered from the storage location holding the value of E. E may be a constant, a fetch from a temporary, or some more general computation such as addition. In the case that E is a fetch from a temporary T, we generate the residence [3] of V in the storage location assigned to T. If E is a constant, we generate a special constant residence for V, indicating that the value of V is a constant. If E is neither a constant nor a fetch, then we generate the residence of V in the storage location assigned to the result register of the instruction that computes E's value. In all cases, the dead reach of V is killed by E. A similar approach is used to recover the value of a source-level induction variable that is replaced by a strength-reduced expression. Linear function test replacement [12] replaces a loop test involving a source-level variable with a compiler synthesized temporary. The source-level induction variable V can then be eliminated if all other uses of V have been eliminated (most likely by strength reduction).

3 Tracking compiler transformations

To allow the debugger analyses described in Section 2, the compiler must perform bookkeeping to record the effects of optimizations in the program representation. In the `cmcc` compiler, this bookkeeping is performed by annotating the nodes of `cmcc`'s IR. These annotations record whether an operation was inserted by optimizations (and if so by which). Bookkeeping also inserts special *IR marker nodes* to mark points of interest to the debugger. These annotations and markers are ignored by optimizations and optimizations are not constrained in any way. This is in contrast to the approach described in [24] where a representation of the original source program is kept as a copy, and links are maintained between the intermediate representation used for optimizations and the original representation (e.g., an abstract syntax tree).

The different ways in which global optimizations may transform a program and the manner in which bookkeeping is performed for these transformations, are as follows:

Code insertion Code motion and common subexpression elimination transformations insert new code into the program representation. Expressions that are inserted by code hoisting or code sinking are marked as hoisted or sunk expressions. Assignment expressions that are marked as hoisted will generate the hoist reach of variables.

Code replacement Copy propagation and redundancy elimination replace one expression with another. Copy propagation replaces a reference to a variable with a propagated expression, while redundancy elimination replaces an available expression with a fetch from a compiler temporary. When an expression E replaces a fetch from a variable V, a reference to V is kept in E. This information is needed only for recovery (as described in Section 2.5) and can otherwise be omitted.

Code deletion Dead code elimination and partial redundancy elimination delete assignment expressions that are dead or available. When an assignment to a variable V is eliminated, it is replaced with a special IR *marker* node, unless this assignment has been previously marked as sunk or hoisted. A marker node contains a reference to the variable V and an indication why the assignment to V was eliminated (i.e., whether the assignment was dead or available). Markers are ignored by optimization phases and are used only for the debugger analysis algorithms. Markers that indicate an available assignment to a variable V will kill the hoist reach of the assignment, while markers indicating a dead variable V will generate the dead reach of V.

Code duplication Control flow optimizations such as loop peeling duplicate code. Code duplication, however, does not create data-value problems since no movement or elimination of assignments occur. Therefore, the effects of this transformation need not be recorded. However, marker nodes inside a block B must also be duplicated when B is duplicated. Moreover, if an IR node containing debugging annotations is duplicated, the annotations must be duplicated along with the node.

Basic block deletion A block of code can be eliminated if the optimizer determines that this code is unreachable. This transformation usually occurs after a conditional branch is folded. Since the code that is eliminated would not have executed in the original program, this transformation does not cause data-value problems and its effects need not be recorded.

Basic blocks can also be deleted because they become empty after other optimizations, or because they contain only unconditional branches (and are deleted by branch chaining). If a deleted basic block contains any information relevant to debugging (i.e., markers), then such information must be retained and is transferred to the deleted block's successor.

Basic block insertion Edge splitting and preheader insertion insert new (empty) basic blocks into the program representation. These transformations do not cause data-value problems[4].

[4] Code duplication, basic block deletion and basic block insertion create

Only after the final object code is produced are all optimizations exposed, and thus the analyses for detecting endangered variables are performed on an instruction-level representation of the program [3, 2]. Like most compilers, `cmcc` has a two-level intermediate representation consisting of a machine-independent IR used for global optimizations (e.g., partial redundancy elimination), and an instruction-level representation used for machine-dependent optimizations (e.g., register allocation and instruction scheduling). Most of the bookkeeping is performed on the machine-independent IR (since most optimizations operate on this IR), and the annotations and markers are passed along to the instruction-level representation as the program representation is lowered. This is similar to passing high-level information such as aliasing information along to a compiler back end for use by an instruction scheduler. During code selection, annotations are transferred from nodes in the machine-independent IR, to the selected instructions. IR marker nodes are lowered to special *marker instructions*, that convey essentially the same information as the IR marker nodes. Instructions are also annotated with information indicating which instructions correspond to source-level assignments. Additional information is passed along for detecting endangered variables caused by instruction scheduling, as described in [2, 1].

4 Experimental results

To better understand the effect of global optimizations on source-level debugging, we instrumented our algorithms to count the number of variables that are endangered at each breakpoint. The charts in this section show the average number of variables that are uninitialized, current, endangered, and nonresident at a breakpoint [5]. These numbers were collected by counting the number of variables in each category, for each possible breakpoint in the source program, and averaging the results by the number of breakpoints. (These static numbers assume that all breakpoints are equally likely. We note that a long-term user study that records actual usage patterns is still outstanding.)

Our measurements showed that although there are a large number of *global* variables that can be queried at each breakpoint, very few global variables are endangered on the average. Therefore, our figures depict only the results for *local* variables.

Code hoisting did not affect source-level debugging for these programs, and the measurements in this section show endangerment caused by elimination and sinking of assignments. The `cmcc` optimizer hoisted mainly address computations. The few source-level assignments that were hoisted were also partially dead, and so the subsequent partial dead code elimination phase sunk the hoisted assignments down past their original locations, to points where they were less frequently executed. Aggressive global scheduling may increase the number of source-level assignments that are hoisted.

Program	% Suspect
`li`	3.5%
`eqntott`	15.6%
`espresso`	9.3%
`gcc`	4.9%
`alvinn`	5.0%
`compress`	1.5%
`ear`	12.3%
`sc`	9.6%

Table 4: Percentage of endangered variables that are suspect in Figure 5(a).

Figure 5(a) shows the results when the programs are compiled *with* global optimization but *without* global register allocation. Since register allocation is not performed, nonresident variables cannot occur. On average, only about 10-30% of the variables are endangered at each breakpoint. Table 4 shows the percentage of endangered variables that are suspect. This table shows that the majority of endangered variables are noncurrent.

Figure 5(b) shows the results when the programs are compiled with global optimizations *and* with register allocation. About half the variables are current or uninitialized; these are the "good" cases, since the debugger can provide accurate and meaningful information to the user. It is interesting to note that almost all the variables that cause problems for the debugger are nonresident.

It is worthwhile to compare Figures 5(a) and 5(b): adding global register allocation decreases the number of current variables (the number of uninitialized ones is obviously unaffected), and there are only a few endangered variables; nonresident variables complicate the life of the debugger. This result is not surprising, since we expect the register allocator to reuse registers assigned to dead variables. (Note that on a machine like the MIPS R3000, there are only 26 integer and 16 floating point registers available for register allocation.) This result suggests that if register allocation is performed with dead code elimination, the effects of dead code elimination are manifested in the form of nonresident variables, rather than endangered variables.

Note that these results are no indication of how often a debugger will be able to respond with a variable's expected value during a debugging session since we report an average result for all possible breakpoints. Such measurements would require a user study. The numbers presented above, however, do give an indication of how different optimizations may

code location problems, since they affect setting and reporting of breakpoints. Code location problems are discussed in [26] and [1].

[5] We use a variant of the nonresidency algorithm described in [3]. This algorithm was modified to handle live range splitting [1].

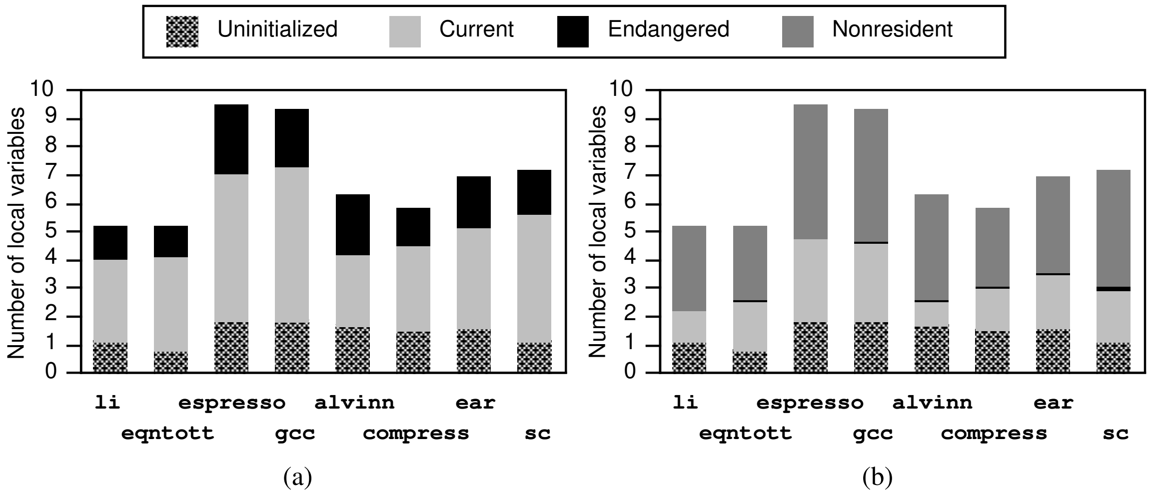

Figure 5: Average number of local variables at a breakpoint (a) Global optimizations only (b) Global optimizations and register allocation.

affect a debugger's ability to recover a variable's expected value. We conclude from our results that a debugger's ability to retrieve a source-level value will more likely be affected by nonresident variables than endangered variables, if optimizations are performed in conjunction with global register allocation. Moreover, code hoisting does not cause much of a problem in practice, since this transformation very rarely results in a hoisted source-level assignment.

5 Conclusions

In this paper we presented a concise model for the data-value problem and described a simple approach to detecting endangered variables caused by global scalar optimizations. The solution described in this paper is cast as a data-flow analysis problem. Since data-flow analysis is a well-understood technique, there are limited obstacles to overcome for an implementation of these techniques in a compiler/debugger. Moreover, the analyses are very similar to other analyses that are done by the compiler and can thus take advantage of an infrastructure that is already present. This is in contrast to other approaches that require specialized data-flow analyses and program representations [13, 24]. To gather the information required for our data-flow analysis, the program intermediate representation is annotated during optimizations to mark hoisted and sunk assignments, and additional markers are inserted to indicate points from which source-level assignments are eliminated. The data-flow analysis can be performed either by the compiler after optimizations and code generation, or by the debugger. Neither the execution time of the analysis phase nor the storage requirements are significant.

We have used our implementation to measure the effects of optimizations on a source-level debugger's ability to retrieve variable values. Measurements show that a debugger is more likely to be affected by register allocation than by other global optimizations. Furthermore, hoisting of assignments almost never occurs. Therefore, a debugger can take a conservative approach to detecting endangered variables caused by code hoisting (e.g., a hoisted assignment can cause a variable to become nonresident). Hence, a combination of residence detection and our simple data-flow algorithm for detecting endangered variables caused by dead code elimination is good enough for most practical situations. Moreover, since assignments are almost never hoisted, the code location issue of syntactic versus semantic breakpoints [26, 13, 24] is not important; the simple syntactic breakpoint model is good enough for a useful debugger.

There are three noteworthy aspects of our approach that allow us to proceed in solving a problem that researchers have struggled with in the past. First, our approach concentrates on two principal global scalar optimizations: code hoisting and dead code elimination. Other global optimizations either can be expressed in terms of these optimizations, or do not cause data-value problems. Second, our 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, our approach is integrated with other implemented solutions to problems caused by local instruction scheduling and global register allocation, described in [2] and [3]. Thus, we are able to address a wide range of common global and local scalar optimizations included in most research and production optimizing compilers of the last decade.

Acknowledgments

We thank Ken Lueh for his many contributions to the `cmcc` compiler. We appreciate comments, suggestions, and encouragement by John Ruttenberg and Fred Chow, Silicon Graphics, Geoff Lowney, DEC, and Bruce Olson, HP.

References

[1] A. Adl-Tabatabai. *Source-Level Debugging of Globally Optimized Code*. PhD thesis, Carnegie Mellon University, 1996.

[2] A. Adl-Tabatabai and T. Gross. Detection and recovery of endangered variables caused by instruction scheduling. In *Proc. ACM SIGPLAN'93 Conf. on Prog. Language Design and Implementation*, pages 13–25. ACM, June 1993.

[3] A. Adl-Tabatabai and T. Gross. Evicted variables and the interaction of global register allocation and symbolic debugging. In *Conf. Record of the 20th Annual ACM Symp. on Principles of Prog. Lang.*, pages 371–383. ACM, January 1993.

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

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

[6] P. Briggs and K. Cooper. Effective partial redundancy elimination. In *Proc. ACM SIGPLAN'94 Conf. on Prog. Language Design and Implementation*, pages 159–170. ACM, June 1994.

[7] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. ACM SIGPLAN'89 Conf. on Prog. Language Design and Implementation*, pages 275–284. ACM, July 1989.

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

[9] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proc. ACM SIGPLAN 1982 Symp. on Compiler Construction*, pages 98–105, June 1982. In SIGPLAN Notices, v. 17, n. 6.

[10] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *Proc. 18th Intl. Symp. on Computer Architecture*, pages 266–275. ACM/IEEE, May 1991.

[11] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, Dec 1991.

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

[13] M. Copperman. Debugging optimized code without being misled. *ACM Trans. on Prog. Lang. Syst.*, 16(3):387–427, May 1994.

[14] D. S. Coutant, S. Meloy, and M. Ruscetta. Doc: A practical approach to source-level debugging of globally optimized code. In *Proc. ACM SIGPLAN'88 Conf. on Prog. Language Design and Implementation*, pages 125–134. ACM, June 1988.

[15] 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, April 1991.

[16] D.M. Dhamdhere, B.K. Rosen, and F.K. Zadeck. How to analyze large programs efficiently and informatively. In *Proc. ACM SIGPLAN'92 Conf. on Prog. Language Design and Implementation*, pages 212–223. ACM, June 1992.

[17] D. Ebcioglu, R. Groves, K. Kim, G. Silberman, and I. Ziv. Vliw compilation techniques in a superscalar environment. In *Proc. ACM SIGPLAN'94 Conf. on Prog. Language Design and Implementation*, pages 36–48. ACM, June 1994.

[18] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[19] J. L. Hennessy. Symbolic debugging of optimized code. *ACM Trans. on Prog. Lang. Syst.*, 4(3):323–344, July 1982.

[20] J. Knoop, O. Ruthing, and B. Steffen. Lazy strength reduction. *Journal of Prog. Languages*, 1(1):71–91, 1993.

[21] J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Trans. on Prog. Lang. Syst.*, 16(4):1117–1155, July 1994.

[22] J. Knoop, O. Ruthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN'94 Conf. on Prog. Language Design and Implementation*, pages 147–158. ACM, June 1994.

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

[24] R. Wismueller. Debugging of globally optimized programs using data flow analysis. In *Proc. ACM SIGPLAN'94 Conf. on Prog. Language Design and Implementation*, pages 278–289. ACM, June 1994.

[25] R. Wismueller. *Quellsprachorientiertes Debugging von optimierten Programmen*. PhD thesis, Technische Universitaet Muenchen, Munich, Germany, Dec. 1994. (in German). Published (1995) by Shaker Verlag, Aachen (Germany), ISBN 3-8265-0841-6.

[26] 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.