FULLDOC: A Full Reporting Debugger for Optimized Code *

Clara Jaramillo[1], Rajiv Gupta[2], and Mary Lou Soffa[1]

[1] Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, U.S.A.
{cij, soffa}@cs.pitt.edu
[2] Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A.
gupta@cs.arizona.edu

Abstract. As compilers increasingly rely on optimizations to achieve high performance, the effectiveness of source level debuggers for optimized code continues to falter. Even if values of source variables are computed in the execution of the optimized code, source level debuggers of optimized code are unable to always report the expected values of source variables at breakpoints.
In this paper, we present FULLDOC, a debugger that can report all of the expected values of source variables that are computed in the optimized code. FULLDOC uses statically computed information to guide the gathering of dynamic information that enables full reporting. FULLDOC can report expected values at breakpoints when reportability is affected because values have been overwritten early, due to code hoisting or register reuse, or written late, due to code sinking. Our debugger can also report values that are path sensitive in that a value may be computed only along one path or the location of the value may be different along different paths. We implemented FULLDOC for C programs, and experimentally evaluated the effectiveness of reporting expected values. Our experimental results indicate that FULLDOC can report 31% more values than are reportable using only statically computed information. We also show improvements of at least 26% over existing schemes that use limited dynamic information.

1 Introduction

Ever since optimizations were introduced into compilers more than 30 years ago, the difficulty of debugging optimized code has been recognized. This difficulty has grown with the development of increasingly more complex code optimizations, such as path sensitive optimizations, code speculation, and aggressive register allocation. The importance of debugging optimized code has also increased over the years as almost all production compilers apply optimizations.

* Supported in part by NSF grants CCR-940226, CCR-9808590 and EIA-9806525, and a grant from Hewlett Packard Labs to the University of Pittsburgh and NSF grants CCR-9996362 and CCR-0096122 to the University of Arizona.

Two problems surface when trying to debug optimized code from the viewpoint of the source code. The *code location* problem relates to determining the position of a breakpoint in the optimized code that corresponds to the breakpoint in the source code. The *data value* problem is the problem of reporting the values of the source variables that a user *expects* to see at a breakpoint in the source code, even though the optimizer may have reordered or deleted the statements computing the values, or overwritten the values by register allocation.

Techniques have been developed that tackle both the code location and data value problems with the goal of reporting expected values when they can be determined from the optimized code but also reporting when an expected value cannot be determined. Progress has been made in the development of debuggers that report more and more expected values. The early techniques focused on determining expected values using information computed statically [8,4,3,15,1]. Recent techniques have proposed using information collected during execution, along with the static information, to improve the reportability of values [5,16]. Dhamdhere et al. [5] time stamp basic blocks to obtain part of the execution path of the optimized code, which is used to dynamically determine currency (whether the actual values of source variables during the optimized code execution are the expected values) at breakpoints. Wu et al. [16] selectively take control of the optimized program execution and then emulate instructions in the optimized code in the order that mimics the execution of the unoptimized program. This execution reordering enables the reporting of some of the expected values of source variables where they occur in the source. Despite all the progress, none of the techniques are able to report all possible expected values of variables at all breakpoints in the source program.

In this paper, we present FULLDOC, a **FULL** reporting **D**ebugger of **O**ptimized **C**ode that reports all expected values that are computed in the optimized program. We call this level of reporting "full reporting." That is, the only values we cannot report are those that are deleted; however in these cases, we report the value has been deleted. It should be noted that techniques exist for recovering some of these values in certain circumstances [8]. For example, if a statement is deleted due to copy propagation, it is sometimes possible to report the value if the copy is available. Since these recovery techniques can be incorporated into all debuggers, regardless of what else they do, we choose not to include these techniques, knowing that they can improve the results of all debuggers of optimized code, including FULLDOC. As illustrated in Figure 1, FULLDOC can report more expected values that are computed in the optimized code than Wu et al. [16] and Dhamdhere et al. [5]. Our technique is non-invasive in that the code that executes is the code that the optimizer generated. Also, unlike the emulation technique [16], we do not execute instructions in a different order and thus avoid the problem of masking user and optimizer errors. FULLDOC works on programs written in C, syntactically mapping breakpoints in the source code to the corresponding positions in the optimized code.

FULLDOC extends the class of reportable expected values by judiciously using both static and dynamic information. The overall strategy of our technique

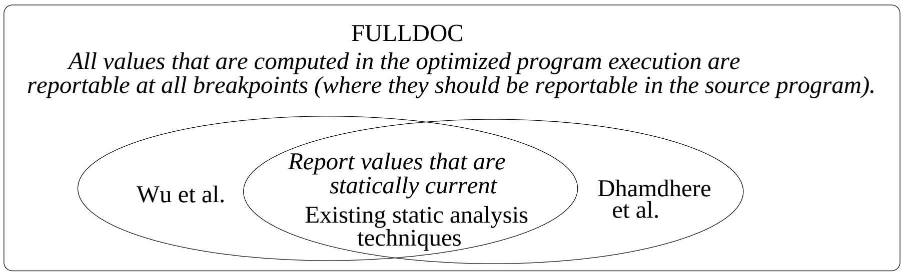

Fig. 1. Reportability of debugger strategies

is to determine by static program analysis those values that the optimizer has placed in a precarious position in that their values may not be reportable. The reportability of these values may depend on run time and debugging information, including the placement of the breakpoints and the paths taken in a program's execution. Thus, during execution, our strategy employs *invisible breakpoints* [17] to gather dynamic information that aids in the reporting of precariously placed values. We employ three schemes, all transparent to the user during a debugging session, to enable full reporting. To report values that are overwritten early with respect to a breakpoint either because of code motion or register reuse, FULLDOC saves the values before they are overwritten and deletes them as soon as they are no longer needed for reporting. FULLDOC only saves the values if they are indeed the expected values at the breakpoint. To report values that are written late with respect to a breakpoint because of code sinking, FULLDOC prematurely executes the optimized program until it can report the value, saving the values overwritten by the roll ahead execution so that they can be reported at subsequent breakpoints. When reportability of a variable at a breakpoint is dependent on the execution path of the optimized code, FULLDOC dynamically records information to indicate the impact of the path on the reportability of a value, and thus is able to report values that are path sensitive either because the computation of the value or the location is dependent on the path.

We implemented our technique and demonstrate its effectiveness and practicality through experimentation. We also show that the technique is practical in terms of the run time overhead.

The capabilities of FULLDOC are as follows.

- Every value of a source level variable that is computed in the optimized program execution is reportable at all breakpoints in the source code where the value of the variable should be reportable. Therefore, we can report more expected values that are computed in the optimized program execution than any existing technique. Values that are not computed in the optimized program execution are the only values that we do not report. However, FULLDOC can incorporate existing techniques that recover some of these values.
- Run time overhead is minimized by performing all analysis during compilation. FULLDOC utilizes debugging information generated during compilation to determine the impact of reportability of values at user breakpoints

and to determine the invisible breakpoints that must be inserted to report affected values.
- Our techniques are transparent to the user. If a user inserts a breakpoint where the reportability of values is affected at the breakpoint or a potential future breakpoint, FULLDOC automatically inserts invisible breakpoints to gather dynamic information to report the expected values.
- Errors in the optimized code are not masked.
- User breakpoints can be placed between any two source level statements, regardless of the optimizations applied.
- The optimized program is not modified except for setting breakpoints.
- Statement level optimizations that hoist and sink code are supported, including speculative code motion, path sensitive optimizations (e.g., partial redundancy elimination), and register allocation.

This paper is organized by Section 2 describing the challenges of reporting expected values using examples. Section 3 describes our approach and implementation. Section 4 presents experimental results. Related work is discussed in Section 5, and concluding remarks are given in Section 6.

2 Challenges of Reporting Expected Values

The reportability of a variable's value involved in an optimization is affected by 1) register reuse, code reordering, and code deletion, 2) the execution path, including loop iterations, and 3) the placement of breakpoints. In this section, we consider the effect of optimizations that can cause a value of a variable to be overwritten early, written late, or deleted. Within each of these cases, we consider the impact of the path and the placement of breakpoints. We demonstrate how our approach handles these cases. In the figures, the paths highlighted are the regions in which reportability is affected; reportability is not affected in the other regions.

2.1 Overwritten Early in the Optimized Program

A value *val* of a variable v is *overwritten early* in the optimized program if val' prematurely overwrites v's value. The application of a code hoisting optimization and register reuse can cause values to be overwritten early. For example, consider the unoptimized program and its optimized version in Figure 2(a), where X^n refers to the n^{th} definition of X. X^2 has been speculatively hoisted, and as a result, the reportability of X is affected. Regardless of the execution path of the optimized code, a debugger cannot report the expected value of X at a breakpoint b along region 1 by simply displaying the *actual* contents of X. The *expected* value of X at b is the value of X^1, but since X^2 is computed early, causing the previous value (i.e., X^1) to be overwritten early, the actual value of X at b is X^2.

The path can also affect reportability. Assume now that a breakpoint b is placed in region 2. The expected value of X at b is either X^2, if the true

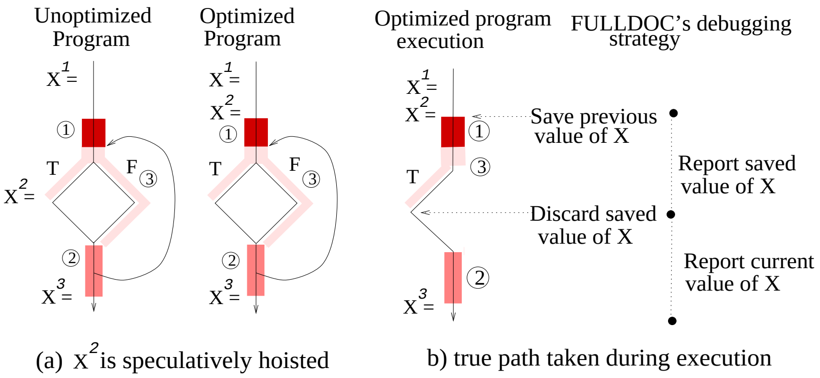

Fig. 2. Overwritten early example

path is taken, or X^1, if only the false path is taken within each loop iteration. However, since X^2 is computed before the branch, the actual value of X at b in the optimized code is X^2. Thus, when execution follows the true path, the expected value of X at b can be reported, but when only the false path is taken, its value cannot be reported.

The number of loop iterations can also affect reportability. The expected value of X at a breakpoint b along region 3 depends not only on whether the true path was taken but also on the current loop iteration. During the first loop iteration, the expected value is X^1. On subsequent loop iterations, the expected value is either X^2 (if the true path is taken) or X^1 (if only the false path is taken on prior loop iterations). However, since X^2 is computed before the loop, the actual value of X at b in the optimized code is X^2. When execution follows the true path, the debugger can report the expected value of X at b on subsequent loop iterations; otherwise, the debugger cannot report the expected value of X.

Using only dynamic currency determination [5], the expected value of X at breakpoints along region 1 cannot be reported because the value has been overwritten. The emulation technique [16] can report the expected value of X along region 1 and along the true path of region 3, but since the technique is not path sensitive, the expected value cannot be reported along region 2 and along the false path of region 3 due to iterations.

FULLDOC can report all of these expected values. During the execution of the optimized code, if a value is overwritten early **with respect to a breakpoint**, FULLDOC saves the value in a *value pool*. FULLDOC only saves what is necessary and discards values when they are no longer needed for reporting. Figure 2(b) illustrates FULLDOC's strategy when the optimized program in Figure 2(a) executes along the true path, assuming the loop executes one time. FULLDOC saves X^1 before the assignment to X^2 and reports the saved value X^1 at breakpoints along regions 1 and 3. FULLDOC discards the saved value when execution reaches the original position of X^2. At breakpoints along the non-highlighted path and region 2, FULLDOC reports the current value of X. Notice that values are saved only as long as they could be reportable in the source program, and thus, our save/discard mechanism automatically disambiguates which value to report at breakpoints along region 2. If X^1 is currently saved at the breakpoint, then only the false path was executed and the saved value is reported. Otherwise if X^1 is not currently saved, then the true path was executed and the current value of X is reported. Notice that this saving strategy, as well as the other strategies, are performed with respect to user breakpoints. In other words, if a user does not insert breakpoints along the regions where the reportability of X is affected, then FULLDOC does not save the value of X.

2.2 Written Late in the Optimized Program

A value *val* of a variable v is *written late* in the optimized program if the computation of *val* is delayed due to, for example, code sinking and partial dead code elimination. In Figure 3(a), suppose X^2 is partially dead along the false path and moved to the true branch. As a result, the expected value of X at a breakpoint b along regions 1 and 2 is not reportable in the optimized code.

Consider a breakpoint b placed in region 3. The expected value of X at b is X^2. However, the actual value of X at b in the optimized code is either X^2 (if the true path is taken) or X^1 (if the false path is taken). Thus, only when execution follows the true path, can the expected value of X at b be reported. Reportability can also be affected by loop iterations, which has the same effect as for the overwritten early case.

Using only dynamic currency determination [5], the expected value of X at breakpoints along region 3 can be reported provided the true path is taken but not along regions 1 and 2. Since the emulation technique [16] is not path sensitive, the expected value of X along region 3 cannot be reported. We can report values in 1 and 3 provided the true path is taken. Note that values in regions 1, 2, and 3 could possibly be reported by all schemes if recovery techniques are employed.

If a requested value is written late **with respect to a breakpoint**, FULLDOC prematurely executes the optimized code, saving previously computed values before they are overwritten (so that they can be reported at subsequent

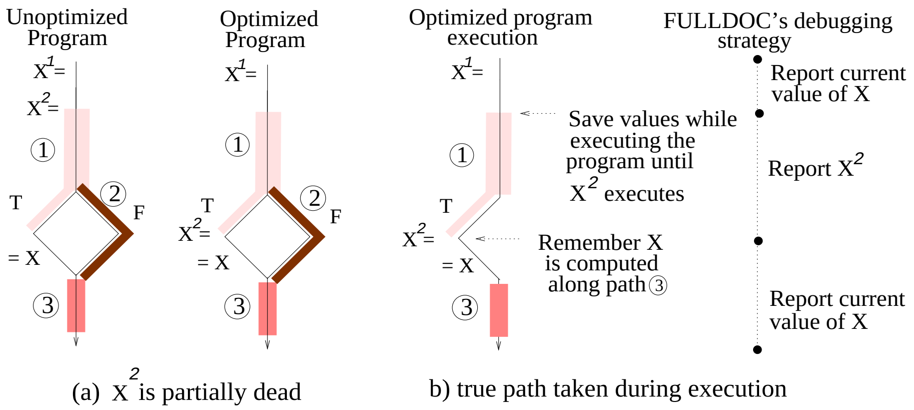

Fig. 3. Written late example

breakpoints). Figure 3(b) illustrates FULLDOC's strategy when the optimized program in Figure 3(a) executes along the true path. At breakpoints along region (1), FULLDOC reports the expected value of X by further executing the optimized code, saving previously computed values before they are overwritten. The roll ahead execution stops once X^2 executes. At breakpoints along the non-highlighted path and region (3), FULLDOC reports X^2.

2.3 Computed in the Unoptimized Program but not in the Optimized Program

Finally, we consider the case where a statement is deleted and thus its value is not computed in the optimized code. For example, in Figure 4(a), suppose Y^2 is dead in the unoptimized program and deleted. The expected value of Y at a breakpoint b along region (1) is Y^2, which cannot be reported in the optimized code.

Now consider placing a breakpoint at region (2). The expected value of Y at b along region (2) is either Y^1 (if the true path is taken) or Y^2 (if the false path is taken). However, since Y^2 was deleted, the actual value of Y at b in the optimized code is Y^1. Thus, along the true path, the actual value is the expected value and can be reported, but along the false path, the expected value cannot be reported.

The emulation technique [16] cannot report the expected value of Y along region (2) because it is not path sensitive. Dynamic currency determination [5] as well as our technique can report the expected value of Y at breakpoints along region (2) if the true path is taken.

Figure 4(b) illustrates FULLDOC's strategy when the optimized program in Figure 4(a) executes along the false path. At a breakpoint along the non-highlighted paths, FULLDOC reports the current value of Y. When execution reaches the original position of Y^2, FULLDOC knows Y is not reportable along regions (1) and (2), and reports the expected value of Y is not computed. When execution reaches Y^3, FULLDOC disregards the non-reportability information of Y.

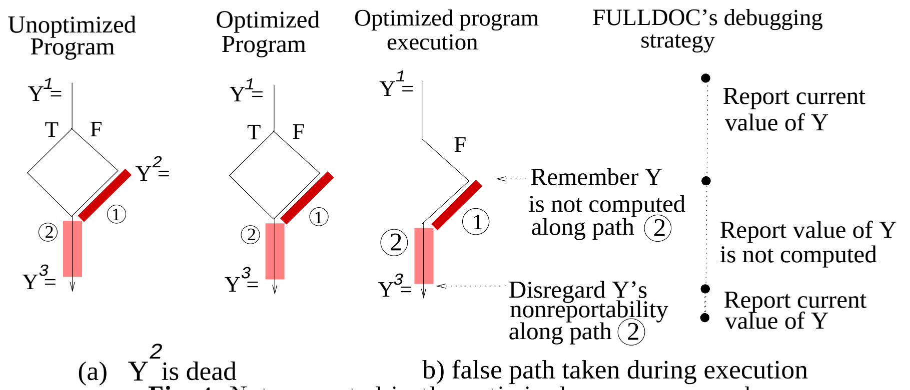

Fig. 4. Not computed in the optimized program example

3 FULLDOC's Approach and Implementation

FULLDOC uses three sources of *debug information* for its debugging capabilities. First, as optimizations are applied, a *code location mapping* is generated between the source and optimized code. Second, after code is optimized and generated by the compiler, static analysis is applied to gather information about the reportability of expected values. This *reportability debug information* is used when user breakpoints are inserted, special program points are reached in the program execution, and when a user breakpoint is reached. Third, during execution, *dynamic debug information* indicating that these special points have been reached is used as well as the position of the user breakpoints to enable full reporting.

Figure 5 illustrates FULLDOC's strategy with respect to a user inserting breakpoints. When the user inserts breakpoints either before the program executes or during program execution, FULLDOC uses the code location mapping to determine the corresponding breakpoints in the optimized code. FULLDOC uses the reportability debug information to determine the impact on reportability at the breakpoints and potential future breakpoints:

• If a value is *overwritten early* with respect to a breakpoint, FULLDOC inserts *invisible breakpoints* [17] to *save* the value during execution as long as the value should be reportable and *discard* the value when it is no longer needed.

• If the reportability of a variable with respect to a breakpoint is path sensitive, FULLDOC inserts invisible breakpoints to update the dynamic debug information regarding the reportability of the value.

Figure 6 illustrates FULLDOC's strategy when a breakpoint is reached. If a user breakpoint is reached, FULLDOC informs the user. FULLDOC responds to user queries by using both static and dynamic information. For invisible breakpoints, FULLDOC performs the following actions. For a value that is *overwritten early*, FULLDOC *saves* the value in a *value pool* and *discards* the value when it is no longer needed for reporting. For a value that is path sensitive, FULLDOC updates the *path sensitive info* regarding the reportability of the value depending on the execution path taken.

When execution reaches a user breakpoint and the user requests the value of a variable, FULLDOC uses the reportability debug information and dynamic debug information to determine the reportability of the value. If the value is available at the location (in memory or register) of the variable or in the value pool, FULLDOC reports the value. If the requested value is *written late* with re-

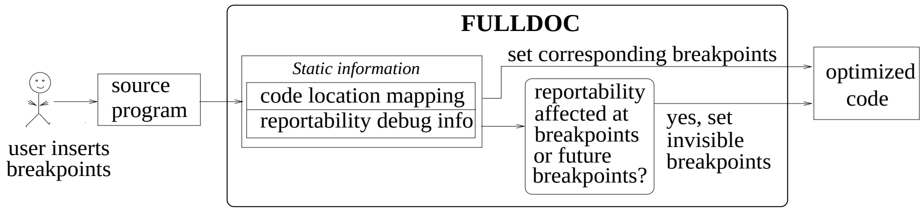

Fig. 5. FULLDOC's strategy with respect to user inserting breakpoints

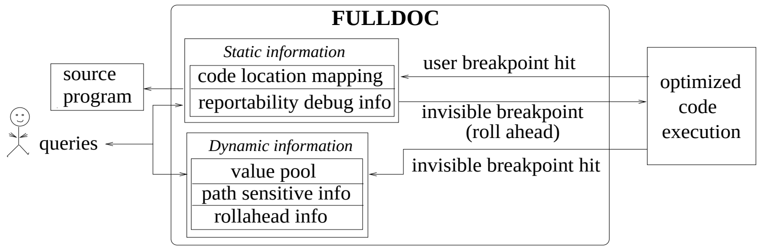

Fig. 6. FULLDOC's strategy with respect to breakpoints hit

spect to the breakpoint, FULLDOC uses the reportability debug information to *roll ahead* with the execution of the optimized code, saving previously computed values before they are overwritten. It *stops* execution once the value is computed and reports the value to the user if it is computed. If the value is not computed in the execution, FULLDOC informs the user that the value is not reportable.

3.1 Code Location Mapping

The code location mapping captures the correspondence between the optimized code and the source code. This code location mapping is used by FULLDOC to map between user breakpoints in the source code and corresponding breakpoints in the optimized code. This mapping is also used to compute the reportability debug information, described in the next section. For each statement in the source code, the code location mapping associates the statement with (1) its original position in the optimized code, that is, the position in the control flow graph G_{opt} prior to the application of optimizations and (2) its corresponding statement(s) in the optimized code. Initially the optimized code starts as an identical copy of the source program with mappings between original positions and corresponding statements in the two programs. As optimizations are applied, the code location mapping is maintained between the source and optimized code.

3.2 Reportability Debug Information

We now describe the reportability debug information computed through static analysis of the optimized code that is provided to FULLDOC as well as how FULLDOC employs this information at run time and collects dynamic debug information in response to the user setting breakpoints and requesting values of variables at these breakpoints.

Simply Reportable

$\boxed{\texttt{AvailAtBkpts[b,v] = }\{\texttt{l}\}\texttt{ or }\{\texttt{(d1,l1), (d2,l2), ...}\}}$

If the value of variable `v` is always reportable at breakpoint `b`, then `AvailAt-Bkpts[b,v]` provides the location (memory location or register name) where the value of `v` can be found. In case the value can always be found at the same location, no matter what execution path is taken, `l` provides the location.

However, it is possible that the location of `v` depends on the path taken during execution because `b` is reachable by multiple definitions of `v`, each of which stores the value of `v` in a different location (e.g., a different register). In this case, the execution path taken determines the latest definition of `v` that is encountered and hence the location where the value of `v` can be found. Each of the potential definition-location pairs (`(di,li)`) are provided by `AvailAtBkpts[b,v]` in this case. When a breakpoint is set at `b`, the debugger *activates* the recording of the definition of `v` that is encountered from among (`d1, d2, ...`) by inserting invisible breakpoints at each of these points. When an invisible breakpoint is hit during execution, the debugger records the latest definition encountered by overwriting the previously recorded definition.

Overwritten Early

```
EarlyAtBkpts[b] = {es: es overwrites early w.r.t. breakpoint b}
SaveDiscardPoints[es] = (save, {discard1, discard2, ...})
```

If the user sets a breakpoint at `b`, then for each statement `es` that overwrites early in `EarlyAtBkpts[b]`, we activate the save and discard points in `SaveDiscardPoints[es]` by inserting invisible breakpoints. This ensures that the values of variables overwritten early with respect to breakpoint `b` will be saved and available for reporting at `b` from the value pool in case they are requested by the user. Note that the save and discard points must be activated immediately when a breakpoint is set by the user so that all values that may be requested by the user, when the breakpoint is hit, are saved. If a discard point is reached along a path and nothing is currently saved because a save point was not reached along the same path, the debugger simply ignores the discard point. The example in Figure 2, where X is overwritten early, is handled by this case.

Written Late

```
LateAtBkpts[b] = {ls: ls writes late w.r.t. breakpoint b}
StopPoints[ls] = {stop1, stop2, ...}
```

Assume the user sets a breakpoint at `b`. Then for each statement `ls` $\in$ `LateAtBkpts[b]`, we must first determine if `ls` is written late with respect to the next instance of the breakpoint `b`. If the original position of `ls` is reached during execution but the current position of `ls` is not reached (before the breakpoint `b` is hit), then `ls` is written late. We determine this information as follows. For each statement `ls` that is written late, we insert invisible breakpoints at the original and current positions of `ls` and record if the original position of `ls` is encountered during execution. When the current position of `ls` is reached during execution, the recorded information is discarded. Now, suppose execution reaches `b`, and the user requests the value of a variable `v` such that `v` is written late by a statement `ls` in `LateAtBkpts[b]`. If the original position of `ls` is currently recorded, then `v` is late at the current instance of the breakpoint `b` and the execution of the program rolls ahead until one of the stop points in `StopPoints[ls]` is encountered. At a stop point, either the value of `v` has just been computed or it is known that it will definitely not be computed (recall that sinking of partially dead code can cause such situations to arise). Unlike the overwritten early case where the save and discard points were activated when a breakpoint was set, here the stop points are activated when the breakpoint is hit and a request for a value that is written late is made. The example in Figure 3, where the reportability of X along region 1 is affected, is handled by this case.

Never Reportable because Deleted Along a Path

```
NotRepDelAtBkpts[b] = {v: v is never reportable at b (deleted)}
NotRepLateAtBkpts[b] = {v: v is never reportable at b (late)}
```

When (partial) dead code removal is performed, the value of a variable defined by the deleted statement becomes unreportable. For each breakpoint `b`, the variables whose values are never reportable at `b`, no matter what execution path is taken, are recorded in `NotRepDelAtBkpts[b]` and `NotRepLateAtBkpts[b]`, for statements removed from paths by dead code elimination and partial dead code elimination, respectively. When the user requests the value of a variable `v` at breakpoint `b`, if `v` is in `NotRepDelAtBkpts[b]` or `NotRepLateAtBkpts[b]`, we report to the user that the value is not reportable because the statement that computes it has been deleted along the execution path. The example in Figure 4, where the reportability of Y is affected along region 1, is handled by this case. Also, the example in Figure 3, where the reportability of X is affected along region 2 is handled by this case.

Path Sensitive Nonreportability/Reportability When Deleted

```
MaybeDelAtBkpts[b] = {ds: ds may be deleted w.r.t. breakpoint b}
EndDelPoints[ds] = {EndDel1, EndDel2, ...}
PotFutBkptsDel[b] = {ds: ds may be deleted at later breakpoints}
```

A value may be deleted on one path (in which case it is not reportable) and not deleted on another path (in which case it is reportable). In this path sensitive case, the reportability information must be updated during execution, based on the paths that are actually executed (i.e., program points reached).

If a user sets a breakpoint at `b`, invisible breakpoints are set at each of the original positions of any deleted statement `ds` in `MaybeDelAtBkpts[b]` to record if one of these positions is encountered during execution. Invisible breakpoints are also set at the end of the definition range of `ds`, stored in `EndDelPoints[ds]`. When `EndDeli` in `EndDelPoints[ds]` is reached during execution, the recorded information is discarded. Now consider the case when breakpoint `b` is reached, and the user requests the value of variable `v` defined by some statement `ds` in `MaybeDelAtBkpts[b]`. If the dynamically recorded information shows that the original position of `ds` was encountered, the debugger reports that the value of `v` was not computed as `ds` was deleted. Otherwise the debugger reports the current value of `v`. The example in Figure 4, where the reportability of Y along region 2 is path sensitive, is handled by this case.

We use the same strategy for each deleted statement in `PotFutBkptsDel[b]`, which prevents FULLDOC from setting invisible breakpoints too late. `PotFutBkptsDel[b]` holds the deleted statements where reportability could be affected at potential future breakpoints even though reportability is not necessarily affected at `b`, and invisible breakpoints must now be set so that during the execution to breakpoint `b`, FULLDOC gathers the appropriate dynamic information for the potential future breakpoints.

Path Sensitive Nonreportability/Reportability When Written Late

```
MaybeLateAtBkpts[b] = {ls: ls may be late w.r.t. breakpoint b}
EndLatePoints[ls] = {EndLate1, EndLate2, ...}
PotFutBkptsLate[b] = {ls: ls may be late at later breakpoints}
```

Sinking code can also involve path sensitive reporting, because a statement may be sunk on one path and not another. This case is opposite to the previous one in that if a late statement is encountered, it is reportable. If the user sets a breakpoint at `b`, the debugger initiates the recording of the late statements in `MaybeLateAtBkpts[b]` by setting invisible breakpoints at the new positions of the late statements. The debugger will discard the recorded information of a late statement `ls` when a `EndLatei` in `EndLatePoints[ls]` is encountered (`EndLatePoints[ls]` holds the end of the definition range of `ls`). Now consider the case when breakpoint `b` is reached, and the user requests the value of variable `v` defined by some statement `ls` in `MaybeLateAtBkpts[b]`. If the dynamically recorded information shows that the late statement `ls` was encountered, the debugger reports the current value of `v`. Otherwise the debugger reports that the value of `v` is not reportable. The example in Figure 3, where the reportability of X along region 3 is path sensitive, is handled by this case.

The same strategy applies for each late statement `ds` in `PotFutBkptsLate[b]`, which prevents FULLDOC from setting invisible breakpoints too late.

3.3 Computing the Reportability Debug Information

The code location mapping is used to compute the reportability debug information. The algorithm in Figure 7 gives an overview of how this debug information is computed. Lines $2-6$ determine what values are overwritten early and compute the `SaveDiscardPoints[] and EarlyAtBkpts[]` information. Lines $7-10$ determine what values are written late and compute the `StopPoints[] and LateAtBkpts[]`. Lines 11-13 determine the rest of the debug information by using data flow analysis. More details about the particular steps follow.

Determining Statements that Overwrite Early or Write Late. We determine where values are overwritten early due to register reuse. Suppose D_x is a definition of a variable x and the location of x is in register r in the optimized code. If D_x reaches an assignment to r in which r is reassigned to another variable or temporary, then x is overwritten early at the reassignment.

For each source definition D_v
    If D_v overwrites x early then
        Let `discard1, discard2, ...` $=$ the corresponding positions of original definitions of x that are reachable from $ARHead(D_v)$ in the optimized code
        `SaveDiscardPoints` $[D_v]$ = $(ARHead(D_v)$, {`discard1, discard2,...`})
        For each breakpoint B along a path from D_v to `discard1, discard2,...`,
            `EarlyAtBkpts[B] = EarlyAtBkpts[B]` $\cup \{ D_v \}$
    Else If D_v writes late in the optimized code then
        `StopPoints` $[D_v]$ = $\{ARHead(D_v)\} \cup \{p : p$ is an earliest possible program point along paths from $ORHead(D_v)$ where D_v will not execute$\}$
        For each breakpoint B along paths $ORHead(D_v)$ to $p \in$ `StopPoints` $[D_v]$,
            `LateAtBkpts[B] = LateAtBkpts[B]` $\cup \{ D_v \}$
Compute `AvailAtBkpts[,]`, `NotRepDelAtBkpts[]`, and `NotRepLateAtBkpts[]` by comparing ranges using $ORHead(D_v)$ and $ARHead(D_v)$
Compute `MaybeDelAtBkpts[]` and `MaybeLateAtBkpts[]` by determining when deleted and late statements occur on one path and not another
Compute `EndDelPoints[]`, `EndLatePoints[]`, `PotFutBkptsDel[]`, and `PotFutBkptsLate[]` by using reachability

Fig. 7. Algorithm to compute the reportability debug information

To determine where values of variables are overwritten early due to code hoisting optimizations, we compare, using G_{opt}, the original positions of the definitions and their actual positions in the optimized program. Let $ARHead(D_v)$ denote the actual position of a definition D_v and let $ORHead(D_v)$ denote the corresponding original position of D_v. We determine the existence of a path P from $ARHead(D_v)$ to $ORHead(D_v)$ such that P does not include backedges of loops enclosing both $ARHead(D_v)$ and $ORHead(D_v)$. The backedge restriction on P ensures that we only consider the positions of the same instance of D_v before and after optimization. This restricted notion of a path is captured by the *SimplePath* predicate.

Definition. The predicate $SimplePath(x, y)$ is true if $\exists$ path P from program point x to program point y in G_{opt} and P does not include backedges of loops enclosing both x and y.

If $SimplePath(ARHead(D_v), ORHead(D_v))$ is true and the location of v at the program point before $ARHead(D_v)$ is the same location that is used to hold the value of D_v, then v is overwritten early at D_v in the optimized code. For example, in Figure 2, $SimplePath(ARHead(X^2), ORHead(X^2))$ is true, and thus, X is overwritten early at X^2.

To determine where values of variables are written late in the optimized program, we similarly compare, using G_{opt}, the original positions of the definitions and their actual positions in the optimized program. That is, for a definition D_v, we determine the existence of a path P from $ORHead(D_v)$ to $ARHead(D_v)$ such that P does not include backedges enclosing both points. Thus, if $SimplePath(ORHead(D_v), ARHead(D_v))$ is true, then definition D_v is written late in the optimized code. For example, in Figure 3, X is written late at X^2 because $SimplePath(ORHead(X^2), ARHead(X^2))$ is true.

Computing `SaveDiscardPoints[] and EarlyAtBkpts[]`. If a value of x is overwritten early at D_v in the optimized code, then a save point is associated at the position of D_v in the optimized code, and discard points are associated at the corresponding positions of original definitions of x that are reachable from D_v in the optimized code. Data flow analysis is used to determine reachable original definitions, which is similar to the reachable definitions problem. After the save and discard points of D_v are computed, we determine the breakpoints where reportability is affected by D_v. $D_v \in$ `EarlyAtBkpts[b]` if b lies along paths from save to corresponding discard points of D_v. `EarlyAtBkpts[]` is easily computed by solving the following data flow equation on G_{opt}:

$$EarlyAt(B) = \bigcup_{N \in pred(B)} Gen_{ea}(N) \cup (EarlyAt(N) - Kill_{ea}(N))$$

where

$Gen_{ea}(B) = \{D_v : D_v \text{ overwrites early and a save point of } D_v \text{ is at } B\}$ and
$Kill_{ea}(B) = \{D_v : D_v \text{ overwrites early and a discard point of } D_v \text{ is at } B\}$.

Then $D_v \in$ `EarlyAtBkpts[B]` if $D_v \in EarlyAt(B)$. For example, in Figure 2, `SaveDiscardPoints[`X^2`]` $= (ARHead(X^2), \{ORHead(X^2), ORHead(X^3)\})$. For a breakpoint b along regions 1, 2, and 3, `EarlyAtBkpts[`b`]` $= \{X^2\}$.

Computing `StopPoints[] and LateAtBkpts[]`. For a definition D_v that is written late, `StopPoints` $[D_v]$ are the earliest points at which execution can stop because either (1) the late value is computed or (2) a point is reached such that it is known the value will not be computed in the execution. A stop point of D_v is associated at the $ARHead(D_v)$. Stop points are also associated with the earliest points along paths from $ORHead(D_v)$ where the appropriate instance of D_v does not execute. That is, $p \in StopPoint(D_v)$ if

$$p = ARHead(D_v) \vee \tag{1}$$

$$(D_v \notin ReachableLate(p) \wedge \tag{2}$$

$$\nexists\, p'(SimplePath(p', p) \wedge p' \in StopPoint(D_v))). \tag{3}$$

Condition 1 ensures a stop point is placed at D_v. Condition 2 ensures the rest of the stop points are not placed at program points where the appropriate instance of the late statement would execute. Condition 3 ensures stop points are placed at the earliest points. $ReachableLate(p)$ is the set of statements written late that are reachable at p. $ReachableLate()$ is easily computed by solving the following data flow equation on G_{opt}:

$$ReachableLate(B) = \bigcap_{N \in succ(B)} Gen_{rl}(N) \cup (ReachableLate(N) - Kill_{rl}(N))$$

where

$Gen_{rl}(B) = \{D_v : ARHead(D_v) = B\}$ and
$Kill_{rl}(B) = \{D_v : ORHead(D_v) = B\}$.

Consider the example in Figure 3. `StopPoints` $[X^2] = \{ARHead(X^2)$, program point at the beginning of the false path$\}$. After the stop points of D_v are computed, we determine the breakpoints where reportability is affected by D_v. $D_v \in$ `LateAtBkpts[b]` if b lies along paths from $ORHead(D_v)$ to the stop points of D_v. `LateAtBkpts[b]` is easily computed using data flow analysis.

Computing `AvailAtBkpts[,]`. The code location mapping is used to construct program ranges of a variable's value which correspond to the unoptimized code (*real* range) and the optimized code (*actual* range). By comparing the two ranges, we can identify program ranges in the optimized code corresponding to regions where the value of the variable is always available for reporting. If breakpoint B is in this program range for a variable v then `AvailAtBkpts[B,v]` is computed by performing data flow analysis to propagate the locations (memory and registers) of variables within these program ranges.

Computing `NotRepDelAtBkpts[]` and `NotRepLateAtBkpts[]`. To determine the values of variables that are not reportable along a breakpoint because of the application of dead code elimination, we propagate the deleted statements where reportability is affected (regardless of the execution path taken) through the optimized control flow graph G_{opt} by solving the data flow equation:

$$NonRepDel(B) = \bigcap_{N \in pred(B)} Gen_{nrd}(N) \cup (NonRepDel(N) - Kill_{nrd}(N))$$

where

$Gen_{nrd}(B) = \{D_v : ORHead(D_v) = \{B\} \wedge D_v \text{ is deleted}\}$ and
$Kill_{nrd}(B) = \{D_v : ORHead(D'_v) = \{B\} \wedge D'_v \text{ is a definition of } v\}$.

Then for each breakpoint B, $v \in$ `NotRepDelAtBkpts[B]` if $\exists\, D_v$ such that $D_v \in NonRepDel(B)$. For example, in Figure 4(a), for a breakpoint B along region 1, $Y \in$ `NotRepDelAtBkpts`$[B]$. `NotRepLateAtBkpts[]` is computed similarly.

Computing `MaybeDelAtBkpts[] and MaybeLateAtBkpts[]`. To determine the values of variables that may not be reportable along a path when deleted, we first compute the data flow equation on G_{opt}:

$$MaybeDel(B) = \bigcup_{N \in pred(B)} Gen_{md}(N) \cup (MaybeDel(N) - Kill_{md}(N))$$

where

$Gen_{md}(B) = \{D_v : ORHead(D_v) = \{B\} \wedge D_v \text{ is deleted}\}$ and
$Kill_{md}(B) = \{D_v : ORHead(D'_v) = \{B\} \wedge D'_v \text{ is a definition of } v\}$.

Then $v \in$ `MaybeDelAtBkpts[B]` if $\exists\, D_v$ such that $D_v \in MaybeDel(B) \wedge D_v \notin NonRepDel(B)$. For example, in Figure 4(a), for a breakpoint B along region 2, $Y \in$ `MaybeDelAtBkpts`$[B]$ because $Y^2 \in MaybeDel(B) \wedge Y^2 \notin NonRepDel(B)$. `MaybeLateAtBkpts[]` is computed similarly.

Computing `EndDelPoints[] and EndLatePoints[]`. For a variable v of a deleted statement `ds` $\in$ `MaybeDelAtBkpts[]`, `EndDelPoints[ds]` are the corresponding positions of original definitions of v that are reachable from *ORHead*(`ds`) in G_{opt}. For example, in Figure 4(a), `EndDelPoints[Y]` = the original position of Y^3, which is $ORHead(Y^3)$. Similarly, for a variable v of a late statement `ls` $\in$ `MaybeLateAtBkpts[]`, `EndLatePoints[ls]` are the corresponding positions of original definitions of v that are reachable from $ORHead$(`ls`).

Computing `PotFutBkptsDel[]` and `PotFutBkptsLate[]`. For each deleted statement D_v in `MaybeDelAtBkpts[]`, $D_v \in$ `PotFutBkptsDel[b]` if b lies along paths from the $ORHead(D_v)$ to the corresponding positions of original definitions of v that are reachable from $ORHead(D_v)$ in the optimized code. `PotFutBkptsLate[]` is computed similarly.

4 Experiments

We implemented FULLDOC by first extending LCC [6], a compiler for C programs, with a set of optimizations, including (coloring) register allocation, loop invariant code motion, dead code elimination, partial dead code elimination, partial redundancy elimination, copy propagation, and constant propagation and folding. We also extended LCC to perform the analyses needed to provide the debug information to FULLDOC, given in the previous section. We then implemented FULLDOC, using the debug information generated by LCC, and fast breakpoints [11] for the implementation of invisible breakpoints.

We performed experiments to measure the improvement in the reportability of expected values for a suite of programs, namely YACC and some SPEC95 benchmarks. Rather than randomly generate user breakpoints, we placed a user breakpoint at every source statement and determined the improvement in reportability of FULLDOC over a technique that uses only static information. We also report for each breakpoint, the reasons why reportability is affected, and thus we can compare the improvement of our technique over techniques that cannot report overwritten values or path sensitive values.

Table 1 shows for each benchmark, the percentage of values that could not be reported by (1) using only statically computed information and (2) FULLDOC. The first row gives the percentages of values that were deleted along all paths, and are thus not reportable in FULLDOC (as noted, FULLDOC could recover some of these values, as other debuggers can [8]). The next two rows give the percentages of values whose reportability is affected because they are overwritten early, either because of code hoisting (row 2) or a register being overwritten

Table 1. Percentage of local variables per breakpoint that are not reportable

Problems	yacc		compress		go		m88ksim		ijpeg	
	static info	FULL DOC	static info	FULL DOC	static info	FULL DOC	static info	FULL DOC	static info	FULL DOC
deleted-all paths	0.96	0.96	15.03	15.03	0.75	0.75	1.87	1.87	10.42	10.42
code hoisting	0.19	0.00	0.34	0.00	0.30	0.00	0.14	0.00	4.15	0.00
reg overwrite	42.65	0.00	17.24	0.00	9.44	0.00	1.83	0.00	15.87	0.00
code sinking (rf)	0.19	0.00	0.64	0.09	1.40	0.39	0.57	0.07	1.79	0.09
del on path	0.00	0.00	0.02	0.02	0.10	0.10	0.06	0.06	0.28	0.28
path sens late	0.00	0.00	0.18	0.09	0.51	0.18	0.41	0.37	0.58	0.39
path sens delete	8.27	6.07	0.18	0.00	2.25	0.74	0.00	0.00	2.36	1.20
path sens location	3.95	0.00	0.07	0.00	1.14	0.00	0.32	0.00	1.43	0.00
total	56.21	7.03	33.70	15.23	15.89	2.16	5.20	2.37	36.88	12.38

Table 2. Static statistics

		yacc	compress	go	m88ksim	ijpeg
no. source statements		168	354	10876	5778	8214
% statements affected		85	57	59	52	56
number of table entries	code hoisting	10	77	1502	987	2374
	reg overwrite	517	234	11819	3961	9655
	code sinking (rf)	13	177	5355	1839	3745
	path sens late	0	117	2912	1203	1833
	path sens delete	66	37	1785	397	1452
	path sens location	48	59	1937	301	1447
% increase compile time		12.1	8.8	11.0	9.6	13.1

early (row 3). If a debugger does not include some mechanism for "saving" values overwritten early, it would not be able to report these values. The next three rows give the percentages of values whose reportability is affected because the statements that computed the values were affected by partial dead code elimination. Row 4 indicates the percentages of values that are not reportable along paths before the sunk values. Row 5 indicates the percentages of values that are not reportable along paths where the sunk values are never computed. Row 6 indicates the percentages of values that are not reportable along paths because the reportability of the values sunk is path sensitive. If a debugger does not include some mechanism to "roll ahead" the execution of the optimized program, it would not be able to report these values. The next two rows give the results when reportability is affected by path sensitive information. The seventh row gives the percentages that were not reportable for path sensitive deletes. In this case, the values may have been deleted on paths that were executed. The eighth row gives the results when the location of a value is path sensitive. A technique that does not include path sensitive information would fail to report these values. The last row gives the total percentages that could not be reported. On average, FULLDOC cannot report 8% of the local variables at a source breakpoint while a debugger using only static information cannot report 30%, which means FULLDOC can report 31% more values than techniques using only statically computed information. From these numbers, FULLDOC can report at least 28% more values than the emulation technique [16] since neither path sensitivity nor register overwrites were handled. FULLDOC can report at least 26% more values than dynamic currency determination technique [5] since early overwrites were not preserved and no roll ahead mechanism is employed.

In Table 2, we present statistics from the static analysis for FULLDOC. The first two rows show the number of source statements and the percentage of source statements whose reportability is affected by optimizations. The next 6 rows give the number of entries in each of the tables generated for use at run time. It should be noted that the largest table is for register overwrites. The last row shows that the increase in compilation for computing all the debug information averaged only 10.9%.

Table 3. Runtime statistics

		yacc	compress	go	m88ksim	ijpeg
	% breakpoints where reportability affected	94	95	67	21	92
avg. no. invisible breakpoints per source statement	code hoisting	0.12	0.03	0.04	0.05	0.35
	reg overwrite	1.03	0.13	0.26	0.02	0.35
	code sinking (rf)	0.03	0.03	0.07	0.03	0.12
	path sens late	0.10	0.05	0.13	0.04	0.23
	path sens delete	0.09	0.00	0.03	0.01	0.23
	path sens location	0.07	0.02	0.02	0.00	0.05
	overall	1.44	0.26	0.56	0.18	1.37
	(duplicates removed) overall	0.56	0.14	0.37	0.17	0.43
% source assignments executed for roll forwards		1.33	4.11	17.39	6.01	19.8
maximum roll forward length		5	60	314	4102	1482
average roll forward length		2	4	7	5	4

In Table 3, we show the average number of invisible breakpoints per source code statement that was encountered during execution. These numbers are shown for each of the various types of invisible breakpoints. These numbers indicate that not much overhead is incurred at run time for invisible breakpoints. The last three rows display the overhead imposed by the roll ahead execution of the optimized program. On average, 9.7% of the source assignment statements were executed during the roll aheads. The maximum number of statements executed during a roll forward ranges from 5 to 4102 values, which means at most 5 to 4102 number of values are saved from the roll ahead at any given moment. The average roll ahead of source assignment statements ranges from 2 to 7 statements. The size of the value pool holding values that are overwritten early was small with the maximum size ranging from 8 entries to 77 entries, indicating that optimizations are not moving code very far.

Thus, our experiments show that the table sizes required to hold the debug information and the increase in compile time to compute debug information are both quite modest. The run time cost of our technique, which is a maximum of less than one fast breakpoint per source level statement if all possible values are requested by the user at all possible breakpoints, is also reasonable. The payoff of our technique is substantial since it reports at least 26% more values than the best previously known techniques.

The presence of pointer assignments in a source program can increase our overheads because our strategies rely on determining the ranges in which the reportability of variables are affected. For control equivalent code motion (assignments are not introduced into new paths nor removed from paths), we can statically determine the ranges in which reportability of values are affected even in the presence of pointer assignments. For the case when the reportability of a value of a variable is affected and the end of its reportable range is possibly at a pointer assignment (because of code deletion and non-control equivalent code motion), our strategy has to dynamically track the range in which the reportability of the value of the variable is affected.

5 Related Work

The difficulty of debugging optimized code has long been recognized [8], with most work focusing on the development of source level debuggers of optimized code [8, 17, 13, 4, 7, 9, 2, 12, 3, 15, 1] that use static analysis techniques to determine whether expected values of source level variables are reportable at breakpoints. Recent work on source level debuggers of optimized code utilizes some dynamic information to provide more expected values. By emulating (at certain program points) the optimized code in an order that mimics the execution of the unoptimized program, some values of variables that are otherwise not reportable by other debuggers can be reported in [16]. However, as pointed out in [16], altering the execution of the optimized program masks certain user and optimizer errors. Also, the emulation technique does not track paths and cannot report values whose reportability is path sensitive. The dynamic currency determination technique proposed in [5] can also report some values of variables that are not reportable by other debuggers by time stamping basic blocks to obtain a partial history of the execution path, which is used to precisely determine what variables are reportable at breakpoints; but values that are overwritten early by either code hoisting or register reuses are not always reportable. Recovery techniques [8], which can be incorporated into all debuggers including FULLDOC, are employed in [16] and [5] to recompute some of the nonreportable values in certain circumstances.

Instead of reporting expected values with respect to a source program, the Optdbx debugger [14] reports values with respect to an optimized source program version. Also, Optdbx uses invisible breakpoints to recover evicted variables.

Another approach to debugging optimized code is COP [10], a comparison checker for optimized code, which verifies that given an input, the semantic behaviors of both the unoptimized and optimized code versions are the same. This can be incorporated into a debugger to report all values, including deleted values. However, this technique requires the execution of both the unoptimized and optimized programs.

6 Conclusions

This paper presents FULLDOC, a **FULL** reporting **D**ebugger of **O**ptimized **C**ode that reports all expected values that are computed in the optimized program. That is, every value of a source level variable that is computed in the optimized program execution is reportable at all breakpoints in the source code where the value of the variable should be reportable. Experimental results show that FULLDOC can report 31% more values than techniques relying on static information and at least 26% more over existing techniques that limit the dynamic information used. FULLDOC's improvement over existing techniques is achieved by statically computing information to guide the gathering of dynamic information that enables full reporting. The only values that FULLDOC cannot reported are those that are not computed in the optimized program execution.

References

1. Adl-Tabatabai, A. and Gross, T. Source-Level Debugging of Scalar Optimized Code. In *Proceedings ACM SIGPLAN'96 Conf. on Programming Languages Design and Implementation*, pages 33–43, May 1996.
2. Brooks, G., Hansen, G.J., and Simmons, S. A New Approach to Debugging Optimized Code. In *Proceedings ACM SIGPLAN'92 Conf. on Programming Languages Design and Implementation*, pages 1–11, June 1992.
3. Copperman, M. Debugging Optimized Code Without Being Misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, 1994.
4. Coutant, D.S., Meloy, S., and Ruscetta, M. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. In *Proceedings ACM SIGPLAN'88 Conf. on Programming Languages Design and Implementation*, pages 125–134, June 1988.
5. Dhamdhere, D. M. and Sankaranarayanan, K. V. Dynamic Currency Determination in Optimized Programs. *ACM Transactions on Programming Languages and Systems*, 20(6):1111–1130, November 1998.
6. Fraser, C. and Hanson, D. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1995.
7. Gupta, R. Debugging Code Reorganized by a Trace Scheduling Compiler. *Structured Programming*, 11(3):141–150, 1990.
8. Hennessy, J. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.
9. Holzle, U., Chambers, C., and Ungar, D. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings ACM SIGPLAN'92 Conf. on Programming Languages Design and Implementation*, pages 32–43, June 1992.
10. Jaramillo, C., Gupta, R., and Soffa, M.L. Comparison Checking: An Approach to Avoid Debugging of Optimized Code. In *ACM SIGSOFT Symposium on Foundations of Software Engineering and European Software Engineering Conference*, pages 268–284, September 1999.
11. Kessler, P. Fast Breakpoints: Design and Implementation. In *Proceedings ACM SIGPLAN'90 Conf. on Programming Languages Design and Implementation*, pages 78–84, June 1990.
12. Pineo, P.P. and Soffa, M.L. Debugging Parallelized Code using Code Liberation Techniques. *Proceedings of ACM/ONR SIGPLAN Workshop on Parallel and Distributed Debugging*, 26(4):103–114, May 1991.
13. Pollock, L.L. and Soffa, M.L. High-Level Debugging with the Aid of an Incremental Optimizer. In *21st Annual Hawaii International Conference on System Sciences*, volume 2, pages 524–531, January 1988.
14. Tice, C. *Non-Transparent Debugging of Optimized Code.* PhD dissertation, University of California, Berkeley, 1999. Technical Report UCB-CSD-99-1077.
15. Wismueller, R. Debugging of Globally Optimized Programs Using Data Flow Analysis. In *Proceedings ACM SIGPLAN'94 Conf. on Programming Languages Design and Implementation*, pages 278–289, June 1994.
16. Wu, L., Mirani, R., Patil H., Olsen, B., and Hwu, W.W. A New Framework for Debugging Globally Optimized Code. In *Proceedings ACM SIGPLAN'99 Conf. on Programming Languages Design and Implementation*, pages 181–191, May 1999.
17. Zellweger, P.T. An Interactive High-Level Debugger for Control-Flow Optimized Programs. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 159–171, 1983.