Debugging of Globally Optimized Programs Using Data Flow Analysis*

Roland Wismüller
Institut für Informatik (Department of Computer Science)
Technische Universität München (Munich University of Technology)
D-80290 München, Germany
Tel.: +49-89-2105-8164
wismuell@informatik.tu-muenchen.de

Abstract

Advanced processor and machine architectures need optimizing compilers to be efficiently programmed in high level languages. Therefore the need for source level debuggers that can handle optimized programs is rising. One difficulty in debugging optimized code arises from the problem to determine the values of source code variables. To ensure correct debugger behaviour with optimized programs, the debugger not only has to determine the variable's storage location or associated register. It must also verify that the variable is *current*, i.e. the value determined from that location is really the value the variable would have in unoptimized code. We will deduce requirements on algorithms for currentness determination and present an algorithm meeting this requirements that is more general than previous work. We will also give first experiences with an implementation. To our knowledge this is the first implementation of a currentness determination algorithm for globally optimized code.

1 Introduction

Source level debuggers have been proved to be valuable tools for program development. Current implementations, however, rely heavily on the code not being optimized by the compiler. Although earlier phases of program development do not have a strong need for optimizers, in later stages, e.g. prototype versions, optimization is desirable. There is a couple of reasons why debuggers should be usable with optimized code. Even if the optimizer is correct, there may be bugs in a program that only have effects in the optimized version [Cop93]. Furthermore, some bugs may not appear until the code is already in production use, at a point when it should be optimized.

Figure 1 shows a fragment of a C-program together with its corresponding Sparc assembly code[1]. Using this example, we will demonstrate some of the problems a symbolic debugger for optimized code must deal with.

The *code location problem* [Zel84] arises as soon as the programmer sets a breakpoint. If he wants to break at line 4, where should the debugger halt the optimized program? The appropriate mapping depends on the programmer's intention in setting the breakpoint. If he wants to examine the old value of f the program should be stopped at instruction 2 (*semantic mapping*). However, if he wants to determine which $a[i]$ are positive, the breakpoint should be mapped to instruction 10 or 11 (*syntactic mapping*). If he is interested in the value of y, it may be the best to stop at instruction 17, if register $i5$ is positive.

Suppose the program has been stopped at instruction 11. When the user tries to inspect the value of f, the debugger must first determine the register assigned to f. In our example this is register $l6$. In addition, the debugger must ensure that f is *current* in $l6$, i.e. the value contained in $l6$ is the value f should have according to the source code. This is commonly referred to as the *data value problem*.

In Fig. 1, f is noncurrent in the first loop iteration, since f should still contain the value assigned prior to the loop, but instruction 2 has already assigned a new value to register $l6$. Since the value of f is loop invariant, f is current in all later iterations. However, without information about the loop's iteration count, the debugger must conservatively regard f as noncurrent.

Several techniques have been developed to avoid these problems by disabling code transformations that hinder transparent debugging. They either require specification of debugging requests prior to compilation [Gup90, PS88] or restrict the program points where a debugger can be invoked [HCU92, ZJ91].

*This work is partly funded by the German Science Foundation, Contract: SFB 342, TP A1

[1] All variables except x are assumed to be live after the loop.

```
for (i=0;i<10;i++) {
  if (a[i] > 0) {
    y = y + a[i];
    f = g+h;
    w = a[i] * 2;
    z = z+w;
    x = a[i] + 1;
  }
  else {
    w = a[i] * 2;
    x = -a[i] + 1;
    z = z-w;
    y = y + a[i];
    f = g+h;
  }
}
```

```
     mov  0,%i3         ! i = 0
     add  %o0,%l5,%l6   ! f = g+h
     add  %fp,-48,%i2   ! i2 = &a[i]
                        ! loop:
 L1: ld   [%i2],%i5     !   i5 = a[i]
     tst  %i5
     ble  L2
     sll  %i5,1,%i1     !   w = a[i] * 2
                        !   if (a[i] > 0)
     b    L3
     add  %i4,%i1,%i4   !     z = z+w
                        !   else
 L2: sub  %i4,%i1,%i4   !     z = z-w
 L3: inc  %i3           !   i++
     cmp  %i3,10
     add  %l7,%i5,%l7   !   y = y + a[i]
     bl   L1
     inc  4,%i2         !   i2 = &a[i]
                        !   if (i<10) goto loop
```

Figure 1: A program fragment before and after optimization

Although these methods allow for some optimization during program development, they are of limited use for debugging production code or if bugs only show up in fully optimized code. In these cases optimization must be handled by the debugger. Additional symbol tables can inform the debugger of the effects of the transformations applied [Zel84, CMR88, BHS92]. Different transformations often require different tables that must be updated in each optimization step, so the debugger depends on the optimizer's implementation and usually handles only few optimizations. Furthermore, tables cannot easily support different breakpoint mappings or the use of run-time information.

The most promising approach to handle a large number of optimizations in a general, flexible way is to use data flow analysis to detect their effects on debugging. Until now, all algorithms based on this concept have severe drawbacks. [Hen82] and [ATG93a] are usable exclusively for local optimization, [ATG93b] only solves the restricted problem of finding the register assigned to a variable. The most general algorithm so far has been published in [Cop93]. However, it has never been implemented, cannot use run-time information and supports only a fixed syntactic breakpoint mapping.

In this paper we present an algorithm solving the data value problem that is more general than previous work and has been fully implemented. In Section 2 we will specify the properties met by this algorithm. Section 3 gives the formal definition of currency, that is the basis of the algorithm outlined in Section 4. Finally, Section 5 presents first results obtained with an implementation.

2 Requirements on Solutions of the Data Value Problem

Independence of Optimization Details. Since the algorithm should be independent of the optimizer's internals and optimization specific data structures, the interface between compiler and debugger must be general enough to represent at least a large subset of common global optimizations. A suitable interface consists of flow graphs of both source and object program and an explicit mapping between corresponding entities. Representing source and object code in a single graph as in [Cop93] is problematic, if the basic block structure is heavily changed by the optimizer.

Arbitrary Breakpoint Mapping. In contrast to previous work, that is based on a fixed syntactic breakpoint mapping, our algorithm can be used with any mapping. As shown in Section 1, syntactic breakpoints are neither unique, nor necessarily the best points to stop the program. If currency determination allows for arbitrary breakpoint mappings, there is some freedom to choose an optimal mapping, where most variables that are of interest for the user are current.

In addition, an algorithm with this property can be used when the program is halted due to an exception and allows recovery of noncurrent variables [Hen82]. In both cases no syntactic mapping can be assumed.

Usage of Run-Time Information. As shown in Section 1, run-time information can lead to more precise results when analyzing currency. In order to use information about the paths the program has taken in the past[2], the algorithm

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

must distinguish between different instances of the same statement.

Assume that the program in Fig. 1 is halted at statement 4 and instruction 11 in the second loop iteration. If $a[0] > 0$, the value of y has been computed by statement 3, while the value of the associated register $l7$ has been stored by instruction 16. Although instruction 16 has been generated from statement 3, y is noncurrent, since its value should have been computed in the current loop iteration but the store to $l7$ has been performed in the previous one.

Handling Non-Live Variables. Most current techniques, e.g. [CMR88, Cop93], rely on a unique mapping between source and object variables, that may depend on the program counter to support variables with changing locations (e.g. due to register allocation). The optimizer maintains this mapping by storing the register or memory location used for each segment of a variable's live range. So the values of non-live variables cannot be determined by the debugger, since no information on the proper location is available.

The algorithm in [ATG93b] computes the proper location of non-live variables, but requires that the same register is used for all segments of a variable's live range, which is not the case for a variety of compilers. The algorithm presented in Section 4 does not rely on this assumption.

Handling of Indirect Assignments. To be usable for real programs, an algorithm determining currency must be able to handle indirect assignments through pointers correctly. Section 4.3 shows how this is accomplished in our algorithm.

Assurance of Currency by the Compiler. Many compilers regard assignments through pointers as an implicit use of all variables the pointer may access. So these variables must be current at the location given by the assignment and its associated store instruction. Using such information may significantly speed up analysis and lead to more precise results.

3 A General Definition of the Term "Current"

Intuitively, at a breakpoint a variable v_s is current in a register or memory location v_o, iff the value of v_o is always the same as the value v_s should have according to the source program. In this context "always" means "in each possible execution of the program". Since we are concerned about the the original source program and the optimized object program, we need a definition of the term execution that characterizes the behaviour of both program versions:

Definition 1: A *path* is a sequence $(x_1 \dots x_n)$ of statements, where x_{i+1} is a possible successor statement of x_i. An *execution* of a program is a pair (p_s, p_o), where p_o is the path the object program takes with some input, while p_s is the path the source program would take, if it were executed with the same input.

Since the value of a variable is determined by its last assignment, v_s is current at a breakpoint in v_o, iff for each execution of the program the statement instance[3] d_s^α that assigned the last value to v_s according to the source code has a semantically corresponding statement instance d_o^γ in the object code that stored the same value into v_o and reached the breakpoint[4]. In the most general case, the breakpoint position is identified by a pair of statement instances (b_s^β, b_o^δ).

We will first define currency of single definitions, that is important for the algorithm presented in Section 4.

Definition 2: A *definition* d_s is *current* at a breakpoint (b_s^β, b_o^δ) with respect to a source variable v_s and an object variable v_o, iff the following condition holds for each execution (p_s, p_o) and each instance d_s^α of d_s on p_s:

If d_s^α writes to v_s and reaches b_s^β along p_s then on p_o there is an instance d_o^γ of a store that semantically corresponds to d_s^α, writes to v_o and reaches b_o^δ along p_o.

Note that the terms "writes" and "reaches" in this definition refer to an actual program execution. Since in the presence of indirect (pointer) assignments static analysis can only decide whether or not a definition *may* write to a variable and *may* reach a breakpoint, Definition 2 is not directly usable, if the paths to be tested contain such assignments. In Section 4.3 we will present an efficient static analysis method for this case.

Now, a variable is current, if each definition that may have assigned the variable's last value is current.

Definition 3: Let $\mathrm{RD}(b, v)$ be the set of definitions of variable v that reach location b. Then a *variable* v_s is *current* at (b_s^β, b_o^δ) in object variable v_o, iff each $d_s \in \mathrm{RD}(b_s, v_s)$ is current at (b_s^β, b_o^δ) with respect to v_s and v_o.

The information about semantic correspondence between statement instances summarizes all program transformations performed by the optimizer. If a vector of iteration indices is used to distinguish between instances (as in [PW86]), the above definitions also hold in the presence of loop transformations, e.g. *interchanging* or *skewing*. However, using index vectors requires the program's loop structure to be known and well defined.

In the rest of this paper we assume that the program's loop structure is not changed by optimization. Then the number of back edges on the path to a statement can be used to distinguish between different instances[5].

[3] A statement instance is an occurrence of a statement in a path. We will use upper Greek indices to distinguish between different instances of the same statement.

[4] I.e. no instruction between d_o^γ and the breakpoint stored into v_o.

[5] In well structured programs, back edges correspond to loops. See [Hec77] for an exact definition.

4 An Algorithm to Determine Current Variables

4.1 Program Representation

A program is represented by a pair of flow graphs for each procedure: The *source graph* $\mathcal{S}$ represents the original source code, while the *object graph* $\mathcal{O}$ represents the optimized code. The nodes of these graphs are basic blocks that contain a sequence of statements. Since the analysis only has to consider operations that either change variables or control flow, only these operations are included. They are characterized by the following information:

- Statement type: we distinguish between assignments, indirect assignments, procedure calls and jumps.
- For assignments there is a field $\mathrm{assigned\_var}$ containing a description of the destination variable that is either a variable name, an address or a register number. Offset and length fields are used to handle assignments to structure or array components.
- Indirect assignments and calls are associated with a set MAYKILL of variables that may be modified.

An additional relation CODE mapping source statements to semantically equivalent instructions summarizes all code transformations. So an assignment in the source code is mapped to the instruction that stores the value of the assignment's right hand side. Relation CODE also holds between corresponding edges of branch or join nodes in order to model control flow transformations. In general, CODE is neither a total nor a unique mapping.

Figure 2 shows the representation of the program in Fig. 1[6]. Instruction 8 corresponds to both statement 5 and 10, since it always stores the same value as these (equivalent) statements. $\langle 7, 13\rangle \in \textsc{code}(\langle 2, 10\rangle)$ holds, since in each execution, where 10 is the successor of statement 2, instruction 7 branches to instruction 13.

In order to simplify analysis, flow graphs have single entry nodes with zero in-degree, that contain initial pseudo assignments to each variable used in the graph. In this way there is at least one reaching definition for a variable along each path.

Although the flow graph nodes are basic blocks, for ease of explanation we subsequently assume a node being a single statement. If a node consists of more than one statement, additional local analysis must be performed.

4.2 Algorithm

Disregarding indirect assignments for the moment, we can use Definitions 2 and 3 to develop an algorithm computing currency of a given variable v_s. The idea is to look at each instance of a definition of v_s that reaches the breakpoint in the source program. Each such definition reaches the breakpoint along a set of paths in the source program. This set defines a set of corresponding paths in the object program. Likewise, the tested source definition leads to a set of semantically corresponding definition instances in the object code. If along each of the above paths one of these definitions reaches the breakpoint in the object code, and all definitions write to the same object variable v_o, then v_s is current in v_o.

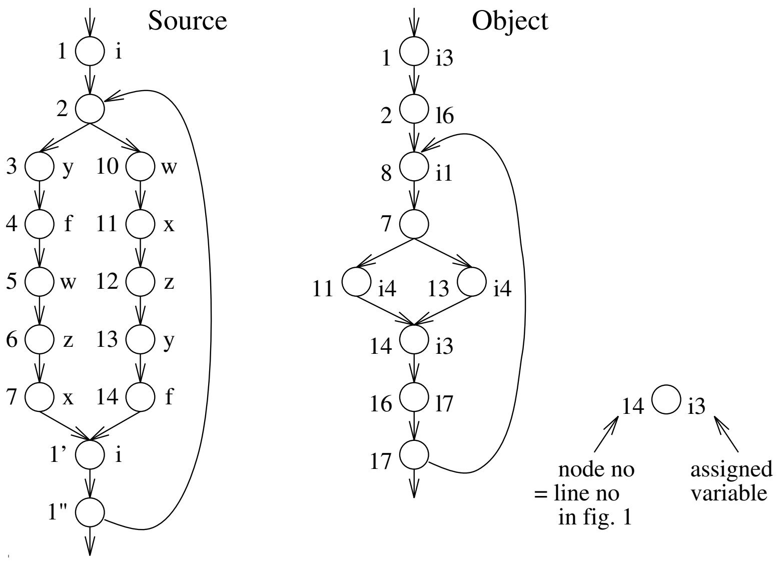
Unfortunately this idea cannot be converted to an algorithm immediately, since an infinity of possible instances and an infinity of program executions is tested. Furthermore, since the instance of the breakpoint is usually unknown, all possible breakpoint instances have to be considered.

However, if no loop transformations have been applied, there is usually no reason to stop the program at a breakpoint (b_s^β, b_o^δ) where b_s^β and b_o^δ are located in different iterations of the same loop. For syntactic breakpoints, b_s and b_o are always located in the same loop (or outside any loops) and have to be reached in the same loop iterations to achieve transparent debugger behaviour. So in this case it is justified to require that $\beta = \delta$. To support semantic breakpoints, we must also consider cases where b_o is ahead of the loop that contains b_s, e.g. when a semantic breakpoint is set on a statement that has been moved out of a loop. In this case, b_o^δ should be the last instance of b_o ahead of the loop iteration containing b_s^β. The reverse situation occurs, if b_o is located after the loop containing b_s. Other situations do not occur regularly. A compiler may be able to move some load instructions to a previous loop iteration, but the statement's key effect, i.e. the computation of its value that determines the position of the semantic breakpoint, is either left in the same iteration or moved out of the loop.

So instances of corresponding breakpoint locations are no longer independent, but coupled by above conditions. The same conditions can also be used to extend the CODE relation to statement instances: d_o^γ semantically corresponds to d_s^α, if $d_o \in \textsc{code}(d_s)$ and the above relation holds between d_s^α and d_o^γ.

To determine currentness of a definition, all paths ending in an arbitrary instance of a breakpoint must be examined. Standard data flow analysis algorithms can be used to compute combined information, e.g. reaching definitions, from all such paths. However, as shown in Section 2, we must be able to distinguish between different instances of a definition reaching along some path. This can be accomplished by unrolling the flow graphs in a way, such that all instances that need to be distinguished are represented by different nodes in the unrolled graph. So standard data flow analysis in the transformed path provides information on different statement instances. A parameter m determines the number of instances prior to the breakpoint that must be distinguished. The unrolled graph then represents all possible paths that contain at least m back edges. Figure 3 shows an example of such a graph for $m = 1$, where the original graph

[6] Nodes 7 and 8 in the object graph are interchanged, with respect to the assembler listing since instruction 8 always executes before the delayed branch 7 is executed.



CODE mapping	
Source	Object
1	1
$1'$	14
3,13	16
4,14	2
5,10	8
6	11
12	13
$\langle 2,3\rangle$	$\langle 7,11\rangle$
$\langle 2,10\rangle$	$\langle 7,13\rangle$
$\langle 7,1'\rangle$	$\langle 11,14\rangle$
$\langle 14,1'\rangle$	$\langle 13,14\rangle$
$\langle 1,2\rangle$	$\langle 2,8\rangle$
$\langle 1'',2\rangle$	$\langle 17,8\rangle$

Figure 2: Representation of the program in Fig. 1

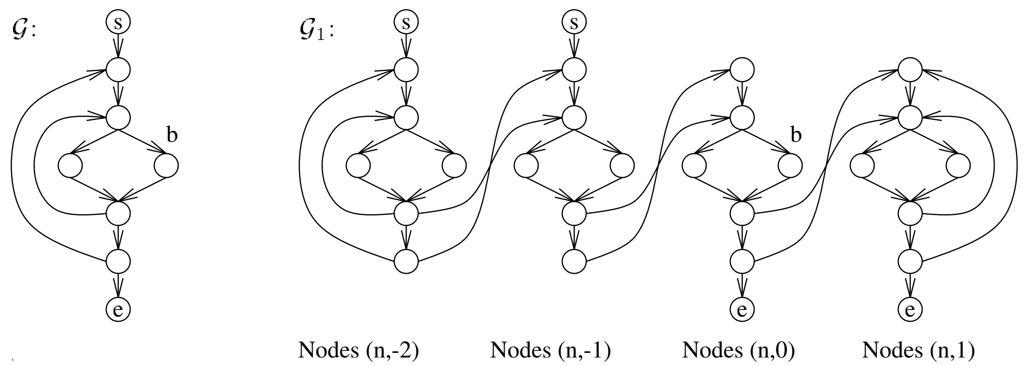


Figure 3: A flow graph $\mathcal{G}$ with two nested loops and its unrolled graph $\mathcal{G}_1$

contains two nested loops. Roughly nodes $(n,0)$ represent the "current" loop iteration, while nodes $(n,-1)$ represent the previous iteration of either the inner or outer loop. The subgraphs $(n,-2)$ and $(n,1)$ summarize all instances that need not be distinguished.

More formally, the unrolled graph $\mathcal{G}_m = (N_m, E_m)$ of a graph $\mathcal{G} = (N, E)$ is defined as follows:

1. $N_m \subseteq N \times [-m-1, 1]$.
2. If $\langle x, y\rangle \in E$ is a forward edge, then $\langle (x,i),(y,i)\rangle \in E_m$ for all $i \in [-m-1, 1]$.
   If $\langle x, y\rangle \in E$ is a back edge, then $\langle (x,-m-1),(y,-m-1)\rangle \in E_m$, $\langle (x,1),(y,1)\rangle \in E_m$ and $\langle (x,i),(y,i+1)\rangle \in E_m$ for all $i \in [-m-1, 0]$.
3. If $s \in N$ is the start node of $\mathcal{G}$ and $e \in N$ the exit node, then $(n,i) \in N_m$, only if there is a path from a node (s,j) with $j \le -m$ to a node (e,k) with $k \ge 0$ that contains (n,i), i.e. all nodes are removed that are not part of a path through $\mathcal{G}_m$.

Using unrolled graphs to model the execution of the source and object program, currency of a definition d_s can be tested in the following way, according to Definition 2: In $\mathcal{S}_m$, the considered instance of the source breakpoint (i.e. b_s^β) is represented by node $(b_s, 0)$. If some instance d_s^α of a definition reaches this breakpoint along a path containing m back edges, node $(d_s,-m)$ reaches $(b_s,0)$ in $\mathcal{S}_m$. Now we have to test, whether in all executions where this instance reaches the breakpoint in the source program, a semantically corresponding definition instance reaches the breakpoint in the object program. In order to test all these execution simultaneously, we restrict the unrolled graph $\mathcal{O}_m$ in a way, that only those paths remain that belong to these executions. This is done using information computed in $\mathcal{S}_m$. In the restricted object graph, for each path there has to be a definition corresponding to d_s^α that reaches the breakpoint. This data flow problem is very similar to the *available expressions* problem and can be solved by a standard analysis algorithm.

In order to test all possible executions, these steps must be repeated for $m = 0, 1, \ldots$. However, it is not necessary to consider paths of arbitrary length. In standard (scalar) data flow problems, all possible executions are examined by testing paths containing at most l back edges, where l is the flow graph's *loop connectedness* [Hec77]. For the analysis above we have proved that $3 \cdot l + 2$ is a legal upper bound for m. The proof is based on the fact that a path containing more than l back edges must have a cycle. If there is a program execution, where some definition reaching the source breakpoint along a path with more than $3 \cdot l + 2$ back edges is noncurrent, then there is also a shorter execution with that property. The factor of three shows up, since the path must be divided into three sub-paths where cycles may be removed, in order not to remove the object breakpoint or some instruction killing the definition's value from the execution. Finally the back edges immediately after the definition and immediately preceding the breakpoint in the source program must not be removed, increasing the bound by two. There is some evidence that the least upper bound is reasonably smaller, but there is no proof yet. In practical cases, l is usually not greater than 2, for most well structured programs our target compiler creates flow graphs with $l = 1$.

There are still three problems that have to be solved in the algorithm:

1. How can $\mathcal{O}_m$ be restricted in the required way?
2. Which nodes in $\mathcal{O}_m$ represent possible instances of the object breakpoint and which represent an instance corresponding to d_s^α?
3. How can we find the object variable associated with v_s?

4.2.1 Restricting the Unrolled Object Graph

In the unrolled graph $\mathcal{S}_m$ we examine all paths where node $(d_s, -m)$ reaches node $(b_s, 0)$. An edge $\langle (x_s, i), (y_s, j) \rangle$ cannot be a part of such a path, if $(d_s, -m)$ reaches (y_s, j) but does not reach the point immediately after (x_s, i)[7]. In this case (y_s, j) is a join node and must be reached along another incoming edge. Therefore the corresponding edge in the object graph (i.e. $\langle (x_o, i), (y_o, j) \rangle$ with $\langle x_o, y_o \rangle \in$ $\textsc{code}(\langle x_s, y_s \rangle)$) cannot be part of a path through $\mathcal{O}_m$ for these program executions and is deleted from $\mathcal{O}_m$. Likewise, we can remove an edge from $\mathcal{O}_m$, if its corresponding edge in the unrolled source graph is not part of any path through $\mathcal{S}_m$ containing the breakpoint location $(b_s, 0)$. Finally, all nodes that are not part of a path through $\mathcal{O}_m$ are deleted, since they are not relevant in the considered program executions.

By properly restricting both $\mathcal{S}_m$ and $\mathcal{O}_m$, run-time information can be taken into account in a very similar way.

[7] I.e. either $(d_s, -m)$ doesn't reach (x_s, i) or is killed by (x_s, i).

4.2.2 Finding Possible Breakpoints and Definitions in the Unrolled Object Graph

In Section 4.2 we have defined which instance b_o^δ of a breakpoint in the object program correspond to an arbitrary instance b_s^β of the source breakpoint: If b_o and b_s are located in the same loops, then $\delta = \beta$ must hold. If b_o has been moved in front of a loop surrounding b_s, then b_o^δ is the last instance of b_o prior to the iteration containing b_s^β. Likewise, b_o^δ is the first instance after this iteration, if b_o has been moved behind the loop.

In the unrolled flow graphs, b_s^β is mapped to $(b_s, 0)$ in $\mathcal{S}_m$. Which nodes in $\mathcal{O}_m$ represent the corresponding instance b_o^δ? In the first case, b_o^δ is mapped to $(b_o, 0)$, since $\delta = \beta$ holds. In the second case, b_o^δ is represented by node $(b_o, 0)$, if we consider the first loop iteration, or by the last node (b_o, i) with $i \in [-m-1, -1]$ prior to the current iteration. In this situation, the current loop iteration is reached via a back edge, that corresponds to some edge $\langle (x_o, -1), (y_o, 0) \rangle$ in the unrolled object graph. b_o is located ahead of this loop, if $b_o < y_o$ in the *reverse post order*[8] [Hec77] in $\mathcal{O}$, i.e. if there is a path from b_o to y_o that doesn't contain any back edge. Similar conditions arise, if b_o is located after the loop containing b_s.

So the set of nodes in $\mathcal{O}_m$ that may represent a possible instance of breakpoint b_o can be summarized as:

$\{\ (b_o, i) \mid i \in [-m-1, -1] \wedge$ there is a path in $\mathcal{O}_m$ starting at (b_o, i) that ends in a back edge $\langle (x_o, -1), (y_o, 0) \rangle$ with $b_o < y_o \wedge$ there is no (b_o, j) between (b_o, i) and $(y_o, 0)\ \}$
$\cup\ \{\ (b_o, 0)\ \}$
$\cup\ \{\ (b_o, 1) \mid$ there is a path to $(b_o, 1)$ starting with a back edge $\langle (x_o, 0), (y_o, 1) \rangle$ with $b_o > x_o \wedge$ there is no (b_o, j) between $(x_o, 0)$ and $(b_o, 1)\ \}$

As an example, consider a breakpoint at line 4 in the program given in Fig. 1. Figure 5 shows the unrolled object graphs $\mathcal{O}_0$ and $\mathcal{O}_1$ of this program[9]. If the breakpoint is mapped to instruction 2, node $(2, 0)$ represents the only instance of this breakpoint in $\mathcal{O}_0$. In $\mathcal{O}_1$, both nodes $(2, -2)$ and $(2, -1)$ are possible breakpoints, since there is a path from either of these nodes ending in back edge $\langle (17, -1), (8, 0) \rangle$ and node 2 precedes node 8 in $\mathcal{O}$, i.e. instruction 2 is located prior to this loop. Since there are no other nodes $(2, i)$ on these paths, both $(2, -2)$ and $(2, -1)$ represent the last instance of instruction 2 ahead of the current loop iteration. Although not considered explicitly in the explanation above, the set is also correct for nested loops.

Since the relation $\textsc{code}$ defining semantic correspondence between statements has been extended to statement instances using the same mapping as above, the set of nodes corresponding to d_s^α is very similar. However, there is a shift in the indices, since d_s^α is mapped to $(d_s, -m)$. Furthermore, semantic correspondence must hold for all possible paths.

[8] Also called *depth first order*.

[9] Nodes are only partially numbered in order to keep the picture clear.

Input: Flow graphs $\mathcal{S}$ and $\mathcal{O}$, a source variable v_s and a breakpoint location b_s.

Output: The set $\text{CURRENT}(b_o)$ of object variables, such that v_s is current at (b_s, b_o) in each $v_o \in \text{CURRENT}(b_o)$. If v_s is noncurrent, $\text{CURRENT}(b_o) = \emptyset$.

Algorithm:

Determine the set OBJECT_VARS of object code variables corresponding to v_s as explained in **4.2.3**

For all nodes n in $\mathcal{O}$:

$\text{CURRENT}(n) = \text{OBJECT\_VARS}$

For $m = 0$ to $3 \cdot l + 2$:

For each definition $(d_s, -m)$ of v_s that reaches $(b_s, 0)$ in $\mathcal{S}_m$:

Restrict $\mathcal{O}_m$ such that only paths are possible, that correspond to reaching paths from $(d_s, -m)$ to $(b_s, 0)$ in $\mathcal{S}_m$ (see **4.2.1**).

Determine the set of nodes representing possible breakpoints and definitions as shown in **4.2.2**

Solve the data flow equation for $\text{AVAIL}(n, i)$ in $\mathcal{O}_m$:

$$\text{AVAIL}(n,i) = \bigcap_{(p,j) \in \text{PRED}(n,i)} \left[\left[\text{AVAIL}(p,j) - \text{MAYKILL}(p) \right] \cup \text{MUSTGEN}(p,j) \right]$$

where

$$\text{MUSTGEN}(p,j) = \begin{cases} \{\text{assigned\_var}(p)\} \cap \text{OBJECT\_VARS} & \text{if } (p,j) \text{ represents a definition} \\ \emptyset & \text{else} \end{cases}$$

For each (n, i) representing a possible breakpoint:

$\text{CURRENT}(n) = \text{CURRENT}(n) \cap \text{AVAIL}(n, i)$

Figure 4: The complete algorithm for currency determination

So the following set results:

$\{\ (d_o, -m-1) \mid d_o \in \text{CODE}(d_s) \wedge$ for all paths from $(d_o, -m-1)$ ending in some back edge $\langle (x_o, -m-1), (y_o, -m) \rangle :\ d_o < y_o \wedge$ there is no (d_o, j) between $(d_o, -m-1)$ and $(y_o, -m)$ $\}$

$\cup\ \{\ (d_o, -m) \mid d_o \in \text{CODE}(d_s)\ \}$

$\cup\ \{\ (d_o, i) \mid d_o \in \text{CODE}(d_s) \wedge i \in [-m+1, 1] \wedge$ for all paths to (d_o, i) that start with a back edge $\langle (x_o, -m), (y_o, -m+1) \rangle :\ d_o > x_o \wedge$ there is no (d_o, j) between $(x_o, -m)$ and $(d_o, i)\ \}$

4.2.3 Finding Appropriate Object Variables

In principle, the algorithm in Fig. 4 doesn't require computation of object variables corresponding to v_s prior to currency determination, since it computes the set of object variables where v_s is current in. However, there are a lot of set operations that are more efficient for small sets, so it is beneficial to start with a small set of variables that might correspond to v_s rather than the set of all variables.

When a single definition of v_s is executed, the value of v_s is stored in the destination variable of the corresponding definition in the object code. If there is more than one corresponding definition, the value is stored in more than one object variable. So after a given definition d_s has been executed, its value is stored in all variables contained in:

$$\bigcup_{d_o \in \text{CODE}(d_s)} \text{assigned\_var}(d_o)$$

At a given breakpoint b_s in the source code, the value of v_s may have been computed by any definition in $\text{RD}(b_s, v_s)$. Since it is unknown, which of these definitions actually assigned the current value of v_s, an object variable is eligible only, if it is contained in the above set for all $d_s \in \text{RD}(b_s, v_s)$. So the set of variables that may contain the value of v_s can be computed by:

$$\bigcap_{d_s \in \text{RD}(b_s, v_s)} \left(\bigcup_{d_o \in \text{CODE}(d_s)} \text{assigned\_var}(d_o) \right)$$

4.2.4 The Complete Algorithm

The resulting algorithm is shown in Fig. 4. There is no need to explicitly build the unrolled graph $\mathcal{S}_m$, since the definitions $(d_s, -m)$ reaching $(b_s, 0)$ in $\mathcal{S}_m$ can be computed by a slightly modified iterative algorithm for reaching definitions in $\mathcal{S}$. An iterative algorithm [Hec77, MJ81] can also be used to solve the equation for $\text{AVAIL}(n, i)$.

4.2.5 An Example

Again we consider variable f at breakpoint (4,11) in the program shown in Fig. 1 and Fig. 2. Assuming that statement 0

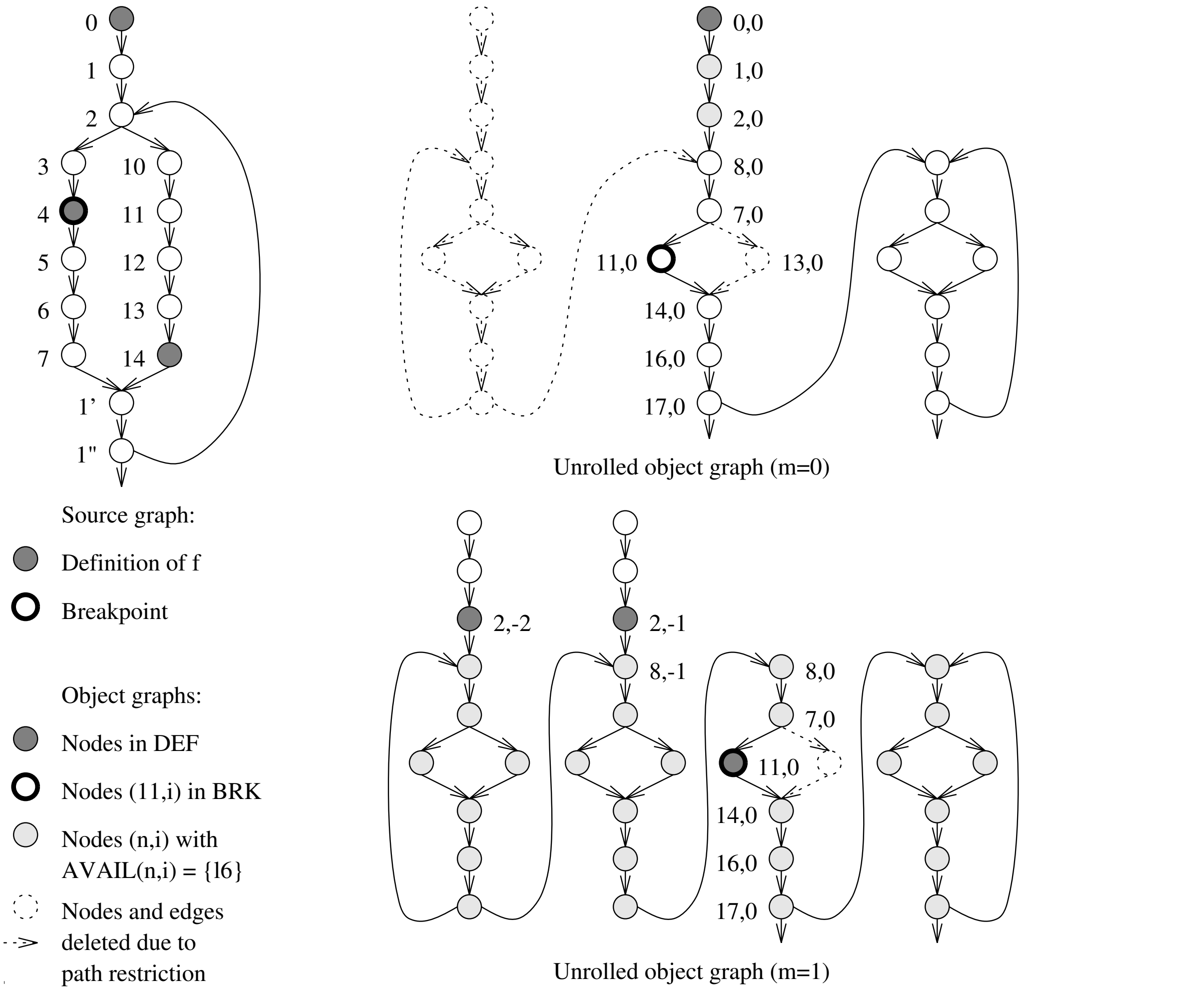


Figure 5: An example for currency determination

is the definition of f prior to the loop, $\text{RD}(4, f) = \{0, 4, 14\}$. If statement 0 has a corresponding instruction 0 storing to $l6$, the set of object variables that may contain the value of f evaluates to $\text{assigned\_var}(0) \cap \text{assigned\_var}(2) \cap \text{assigned\_var}(2) = \{l6\}$. If instruction 0 had been eliminated or stored to another register, this set would be empty, so the value of f could not be determined, since f would have no associated location. In the rest of this example we assume that f is associated with $l6$.

In the first step ($m = 0$), the algorithm tests all executions, where a definition of f reaches the breakpoint along a path without back edges. There is only one such execution, that is shown in Fig. 6 as a trace of both source and object program. Looking at this execution, it is clear, that f is noncurrent, since the value assigned to f by statement 0 is stored in register $l6$ by instruction 0, but is overwritten by instruction 2. The same result is computed by the algorithm in Fig. 4.

First, analysis of the unrolled source graph shows, that only node $(0, 0)$ reaches node $(4, 0)$, i.e. statement 0 is the only definition to look at. In a next step, edges and nodes, that cannot occur in these executions are removed from the unrolled object graph. The resulting graph $\mathcal{O}_0$ is shown in the upper half of Fig. 5. Edge $\langle(17, -1), (8, 0)\rangle$ has been eliminated, since no reaching path from $(4, 0)$ to $(0, 0)$ in $\mathcal{S}_0$ can contain the corresponding edge $\langle(1'', -1), (2, 0)\rangle$. Likewise $\langle(13, 0), (14, 0)\rangle$ has been deleted, since a path in $\mathcal{S}_0$ containing node $(4, 0)$ can't contain $\langle(14, 0), (1', 0)\rangle$. Elimination of all nodes that are not part of a path through $\mathcal{O}_0$ removes the dotted nodes and edges in Fig. 5. So this graph roughly contains the same information as the object program's trace in Fig. 6.

Next, $(11, 0)$ is determined as the node representing the appropriate instance of the breakpoint instruction 11, while node $(0, 0)$ is computed as the only representative of a definition that semantically corresponds to statement 0. Finally, the algorithm checks, whether the value assigned by this definition is available at the breakpoint. Since instruction 2 assigns to register $l6$, node $(2, 0)$ kills $l6$, so the result is $\text{AVAIL}(11, 0) = \emptyset$ and therefore $\text{CURRENT}(11) = \emptyset$. In

```
0 f = ...;
...
1 for (i=0;i<10;i++) {
2   if (a[i] > 0) {
3     y = y + a[i];
```

```
0      mov  ...,%l6      ! definition
...
1      mov  0,%i3
2      add  %o0,%l5,%l6  ! kills %l6
3      add  %fp,-48,%i2
5 L1:  ld   [%i2],%i5
6      tst  %i5
7      ble  L2           ! not taken
8      sll  %i5,1,%i1
```

Figure 6: A program trace of fig.1 until the first occurrence of breakpoint $(4, 11)$

```
S1  a = x;
S2  *p = y;
S3  b = *p;
S4  ...
```

```
I1  sethi %hi(_a),%o0
I2  st    %l1,[%o0+%lo(_a)]  ! a = x
I3  st    %l2,[%i5]          ! *p = y
I4  mov   [%i5],%l3          ! b = *p
I5  ...
```

Figure 7: An example program with an indirect assignment

---

later iterations of the m-loop, this result can't change any more, so f is noncurrent.

When there is some runtime information, telling the debugger that the program has not been stopped in the first loop iteration, a better result can be achieved. To reflect the given runtime information, edge $\langle(1,0),(2,0)\rangle$ can be deleted in the unrolled source graph $\mathcal{S}_0$, since it is only part of a program execution, when the breakpoint occurs in the first loop iteration. Now for $m = 0$ there is no node $(d_s, 0)$ in $\mathcal{S}_0$ that reaches node $(4, 0)$, i.e. there is no longer any definition of f reaching the breakpoint within the same loop iteration. In this case, $\text{CURRENT}(11)$ is still initialized with the set $\{l6\}$, when analysis is repeated with $m = 1$. For $m = 1$, definitions $(4, -1)$ and $(14, -1)$ reach the breakpoint in $\mathcal{S}_1$. For both definitions the same situation occurs, so only the analysis for $(4, -1)$ is explained.

The unrolled graph $\mathcal{O}_1$ is shown in the lower half or Fig. 5. As above, node $(13, 0)$ has been removed during path restriction, node $(11, 0)$ represents the current breakpoint instance. Since $\text{CODE}(4) = 2$ and instruction 2 has been moved in front of the loop, both nodes $(2, -2)$ and $(2, -1)$ represent instances of an instruction that semantically corresponds to the source definition being tested. Node $(2, -2)$ is included, since for all paths from this node to some back edge $\langle(x, -2), (y, -1)\rangle$, the only candidate for y is instruction 8 that is preceded by instruction 2 in reverse post order. In addition there are no other occurrences of instruction 2 on these paths, so $(2, -2)$ represents the last instance of this instruction prior to the loop iteration containing the breakpoint.

In this case, $\text{AVAIL}(11, 0) = \{l6\}$, since on every path in $\mathcal{O}_1$ a possible definition reaches node $(11, 0)$ in $l6$. Therefore $\text{CURRENT}(11) = \{l6\}$ after iteration $m = 1$. No subsequent iteration changes this result, so f is current in $l6$, if the breakpoint is not reached in the first loop iteration.

4.3 Currency Determination in the Presence of Indirect Assignments

In the presence of indirect assignments, e.g. $*p = x$, the algorithm presented in Section 4.2 is too conservative. Consider the program fragment in Fig. 7, where p may point to a. If variable a is examined at breakpoint (S4,I5), two reaching definitions are tested. While definition S2 is current with respect to a and $\_a$, the algorithm will classify S1 as noncurrent, since instruction I3 may kill $\_a$. The problem is that in assuming S1 to be a's last definition, it is assumed that S2 does not access a. However, the algorithm doesn't take into account that in this case I3 cannot write to $\_a$, if it semantically corresponds to S2. The following theorem allows a slightly modified recursive version of our algorithm to be used in the presence of pointer assignments.

Theorem 1: Let $\text{ID}(b, v)$ be the set of definitions d of v, such that there is a path from d to b that must not kill v. So $\text{ID}(b, v)$ is the minimal subset of $\text{RD}(b, v)$ such that for each path to b, the last reaching definition of v is in $\text{ID}(b, v)$.

Then v_s is current at (b_s, b_o) in v_o, if each $d_s \in \text{ID}(b_s, v_s)$ is current at (b_s, b_o) with respect to v_s and v_o and, if d_s is an indirect assignment, v_s is current in v_o at (d_s, d_o) for each $d_o \in \text{CODE}(d_s)$.

We will justify this theorem using the example above. S2 $\in \text{ID}(\text{S4}, a)$, so there are no statements between S2 and S4 that may write to a. Since S2 is current at (S4,I5), there are also no instructions between I3 and I5 that may change $\_a$. So in an arbitrary execution, if S2 writes to a, a is current in $\_a$. Otherwise, a is current in $\_a$ at (S4,I5), if and only if

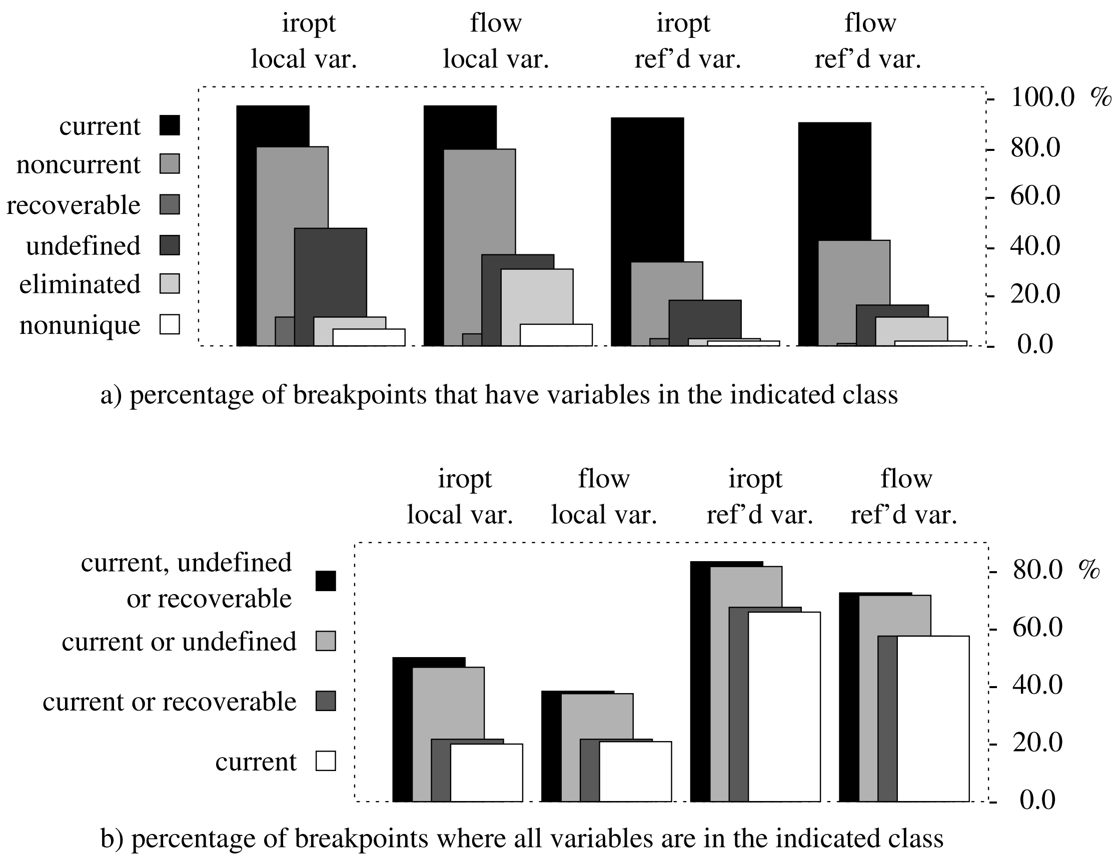


Figure 8: Effects of optimization on debugging

it is current at (S2,I3), since between these points neither a nor $\_a$ is modified.

Theorem 1 also allows us to reduce the algorithm's complexity by using compiler information that assures currency of a variable. In the example, if a is guaranteed to be current in $\_a$ at (S2,I3), no recursive analysis is needed.

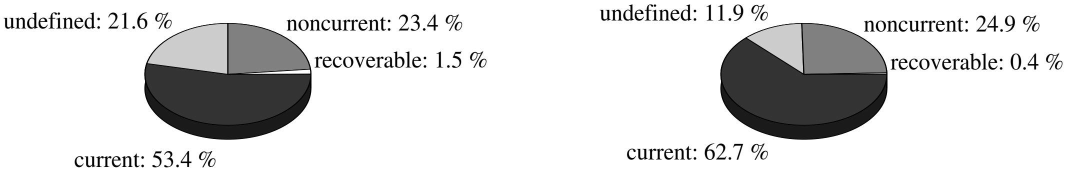
5 Results and Future Work

The algorithm developed in Sections 4.2 and 4.3 has been implemented based on the SUN C-compiler for Sparc workstations [Muc90]. Optimizations performed by this compiler include global common subexpression elimination, register allocation, global copy and constant propagation, global dead store elimination, loop invariant code motion and composite breaking. We have modified the optimizer and code generator to generate the proposed program representation in addition to traditional debugging information. Of about 94,000 lines of code, only 2,200 lines had to be changed or added for this purpose, where most of the additional code consists of output routines.

The overhead in compile time and object file size introduced by this debugging information has been measured by compiling the optimizer itself, that consists of 69 files. Creating traditional debugging information for unoptimized code increases compile time by 19%, while file size increases by a factor of 5.5. In comparison, generating the extended information for optimized code results in a compile time overhead of 29% and output files that increase by a factor of 20 in size. However, the output format is very redundant at the moment, so we expect that the size of the program representation could be reduced by a factor of 2, resulting in a comparable reduction in compile time and file size overhead.

Currency analysis is performed by a separate program that takes as input a breakpoint location, given by line number and object code address, and a variable, given by its name, an offset and its length. Although worst case complexity of this analysis is quadratic in the number of basic blocks, in practice response time of the algorithm is only a few tenths of a second on a SPARCstation 10, so the algorithm is usable for interactive debugging.

To gain empirical results on the effects of global optimization on debugging, we have analyzed several programs using input scripts that request currency state of each variable at each possible breakpoint. The usual syntactic breakpoint mapping, that is generated by the compiler in the same



a) iropt   b) flow

Figure 9: Average number of current and noncurrent variables

way as for unoptimized programs, has been used for this analysis. Figure 8 shows some results for two different programs: `iropt` is the optimizer of our target compiler, `flow` is the currency algorithm itself. The figure comprises about 16,800 breakpoints and 170,000 currency tests. Variables include locals and parameters of all types, where structures and arrays are broken into their components. In order to keep the size of the input scripts manageable, global variables have not been considered in this picture.

In the columns labelled *local var.* all variables have been examined at all breakpoints. Fig. 8a shows that there are virtually no breakpoints where no variable is current, however at a large number of breakpoints (80%) also noncurrent variables occur. Some noncurrent variables can be recovered at 4 to 10% of all analyzed locations, while variables that have not yet been assigned occur at about half of all breakpoints (37-48%). Although in Fig. 8 these variables are labelled as undefined, they also include procedure parameters that are not changed prior to the breakpoint. For these parameters, the variable's value can be correctly determined by the debugger, if no interprocedural optimization has been performed, as it is the case with our compiler.

Figure 9 shows another view of these results, namely what percentage of all variables existing at a breakpoint is current, noncurrent, undefined or recoverable. Only one forth of all variables are noncurrent, so on an average there is a 75% chance to determine the expected value of a variable. Recovery is moderately successful for global optimizations: about 2 to 5% of all noncurrent variables can be recovered.

To some extent, also the reasons, why variables are noncurrent are shown in Fig. 8a: At up to 31% of all breakpoints there is no associated object variable due to code elimination, in about 7% of all cases, the object variable is not unique, since different registers are used for different sections of a variable's live range. It is not easily possible to further analyze which optimization caused a variable to be noncurrent, since the program representation only models the accumulative effect of all optimizations.

Finally, Fig. 8b shows the relative number of breakpoints, where full debugging is possible, i.e. all considered variables are current, undefined or recoverable. If only variables are taken into account that are referenced at the breakpoint line or the preceding line in the source code, the results change significantly, as the columns labelled *ref'd var.* in Fig. 8 show. This restriction models a situation, where a user sets a breakpoint either to see the value of a variable used at this point or defined immediately ahead of it. In this situation, the number of breakpoints with noncurrent or undefined variables is greatly reduced, while the number of breakpoints with full debugging capabilities increases.

Although it is possible to obtain statistical data, it is not yet known how the presence of noncurrent variables affects debugging in practice, i.e. how often a programmer is interested in the value of such a variable in a real debugging session. To answer this question, we will integrate our algorithm into the source level debugger DETOP, that is part of the programming environment TOPSYS [BB91]. We will also investigate how the usage of run-time information affects the results presented in this section. Further work will be done in algorithms to determine syntactic breakpoint mappings and optimal mappings with respect to the number of current variables.

References

[ATG93a] A. Adl-Tabatabai and T. Gross. Detection and Recovery of Endangered Variables Caused by Instruction Scheduling. In *Proceedings of the ACM/SIGPLAN Conference on Programming Language Design and Implementation PLDI'93*, volume 28(6) of *ACM SIGPLAN Notices*, pages 13–25, Albuquerque, New Mexico, June 1993.

[ATG93b] A. Adl-Tabatabai and T. Gross. Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging. In *Proceedings of the 20th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 371–383, Charleston, South-Carolina, January 1993. ACM Press.

[BB91] T. Bemmerl and A. Bode. An Integrated Environment for Programming Distributed Memory Multiprocessors. In A. Bode, editor, *Distributed*

Memory Computing, 2nd European Conference, EDMCC2, volume 487 of *Lecture Notes in Computer Science*, pages 130–142, München, FRG, April 1991. Springer-Verlag.

[BHS92] G. Brooks, G. J. Hansen, and S. Simmons. A New Approach to Debugging Optimized Code. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27(7) of *ACM SIGPLAN Notices*, pages 1–11, San Francisco, California, June 1992. ACM Press.

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

[Cop93] M. Copperman. *Debugging Optimized Code Without Being Misled*. PhD thesis, Univ. of California, Santa Cruz, May 1993. Technical Report UCSC-CRL-93-21.

[Gup90] R. Gupta. Debugging Code Reorganized by a Trace Scheduling Compiler. *Structured Programming*, 11(3):141–150, 1990.

[HCU92] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27(7) of *ACM SIGPLAN Notices*, pages 32–43, San Francisco, California, June 1992. ACM Press.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs*. The Computer Science Library. Elsevier North-Holland, New York, 1977.

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

[MJ81] S. S. Muchnick and N. D. Jones. *Program Flow Analysis, Theory and Applications*. Prentice Hall, Englewood Cliffs, 1981.

[Muc90] S. S. Muchnick. Compiling for RISC-Based Systems. In *RISC Compilers Tutorial, SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990.

[PS88] L. L. Pollock and M. L. Soffa. High-Level Debugging with the Aid of an Incremental Optimizer. In *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, volume II, Software Track, pages 524–532, 1988.

[PW86] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

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

[ZJ91] L. W. Zurawski and R. E. Johnson. Debugging Optimized Code with Expexted Behaviour. unpublished manuscript, April 1991.