# A Further Note on Hennessy's "Symbolic Debugging of Optimized Code"

Max Copperman
Charles E. McDowell

UCSC-CRL-92-24
Supersedes UCSC-CRL-91-04
April 1992

Board of Studies in Computer and Information Sciences
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

When attempting to debug optimized programs, most debuggers may give misleading information about the value of variables at breakpoints. Hennessy proposed a set of algorithms for generating optimized code and determining when, in the generated code, the reported values would be misleading, and under certain circumstances actually recovering the "expected" value of the variable (i.e., one that would not be misleading). We point out where the assumptions made by Hennessy need to be revised due to advances in compiler and debugger technology, and give references for current work on this revised problem.

# 1 Introduction

Hennessy's paper "Symbolic Debugging of Optimized Code" [Hen82] discusses one of the problems of source-level debugging of optimized code: how to provide information about the value of a variable that has been affected by optimization without confusing or misleading the user. When the debugger gains control due to a program trap or breakpoint, it should provide values that are consistent with the order of evaluation and assignment in the source program, which corresponds to the the order of evaluation and assignment in an unoptimized version of the program. Unfortunately, this is not always possible, because various kinds of optimizations may cause a variable to have a value that has been assigned earlier or later than in the source (or unoptimized) code. Such a variable is termed *noncurrent* at points in optimized code where its value differs from its value at corresponding points in the source or unoptimized code and *current* at points where its value is the same.

The bulk of Hennessy's paper deals with problems that arise from local optimization. He describes information to be used to determine which variables are noncurrent in the presence of local optimizations and algorithms that use that information to make the determination and, if possible, recover the correct values of such variables. Wall et al [WST85] describes errors in Hennessy's algorithms and gives corrected algorithms.

The code generation algorithm used in Hennessy's work (a standard algorithm from Aho and Ullman [AU77], [AU73], [AU72]) generates code that contains only one assignment to any variable within a basic block. In the next section, we show how Hennessy's algorithms (and the modified algorithms of [WST85]) are dependent on this characteristic.

Due to a decade's progress in computer architecture and compiler technology, highly optimized code may contain more than one assignment to a given variable within a basic block. At one time registers were expensive and variables were assumed to reside in memory; if a variable could be kept in a register for its entire lifetime, it was considered to have been 'optimized away'. Many architectures today provide enough registers that many variables never need to be stored in memory. Symbol table formats recognize this fact by allowing a variable to be mapped to a register; debuggers will retrieve the value in the register when queried about the variable. In fact, it is becoming more and more common to allow a variable to be mapped to a number of locations over its lifetime [CMR88], [Str91], [Wan91], [Sil92].

Assume that in some source code, a variable $V$ is assigned into more than once in a basic block. Assuming that $V$ is used after each such assignment, the code generation technique used by Hennessy would use temporaries (registers) for all but the last such assignment. Under the old assumption that $V$ is stored in memory and has a single location known to the debugger, this is an optimization that eliminates memory references and thereby speeds up the program, at the cost of making the value of $V$ noncurrent between the first and last assignments into $V$ in the block. With current compiler and debugger technology, each temporary is an alias for $V$ (over the appropriate range of code), and $V$ is current throughout the block.

The machine code that is produced by the code generation algorithm used by Hennessy contains more than one assignment into a variable $V$ within a single block $B$ if

```
x = a*b+c;
d = e*x;
f = sqrt(x);
g = (h*x)/i;
x = j*k;
```

Figure 1.1: In the above code for a basic block, the first assignment to `x` must take place because `x` is used within the block. The second assignment to `x` must take place because `x` must have the correct value on block exit.

- the source code contains more than one assignment into $V$ within $B$,
- the result of some such assignment other than the last one is used within $B$, and
- each temporary into which such a result is computed is considered to be an alias for $V$ over the appropriate range of code addresses (and the debugger is so informed).[1]

Figure 1 comprises a block in which two assignments are made to variable `x`.

## 2 The Problem

The algorithms given by Hennessy for determining whether a variable $V$ is current at a breakpoint do not give correct results when there are multiple assignments into $V$ in a block, that is, when what were once considered unavailable compiler temporaries are considered to be aliases for $V$. Hennessy's algorithms will report that `x` is noncurrent at a breakpoint reached between the two assignments to `x` within the block shown in Figure 1.

Hennessy's debugger algorithms take as input an augmented expression dag. A normal expression dag has a variable name (or several) optionally labeling each node, indicating that the value of that node should be assigned to that variable when the value is computed. If a variable $V$ is assigned at more than one node, as the dag is constructed the label is moved to the latest node at which $V$ is assigned, so that subsequent uses of $V$ within the dag refer to the node containing the correct value of $V$. Thus there is only one instance of any given variable label in the dag. Hennessy augments the dag in a number of ways, only one of which is germane here: rather than moving labels, a field is added to distinguish between *old* and *current* labels. In the dag construction algorithm, the node representing the last value assigned to a variable within a basic block is given a "current" label for that variable. Nodes corresponding to previous values assigned to the same variable within the basic block are given an "old" label for that variable. By construction, there is only one current label for each variable assigned to within the dag.

In Hennessy's paper, an old label represents an assignment in the unoptimized version that is not present in the optimized version, and a current label represents an assignment that appears in both unoptimized and optimized versions.

[1] Coutant et al [CMR88], Streepy [Str91], and the DWARF standardization effort [Sil92] discuss different formats whereby a debugger can be informed of such aliases.

When what were once considered compiler temporaries are considered valid locations for variables, this construction produces assignments with old labels that appear in both unoptimized and optimized versions, contrary to the intended meaning of the old label.[2]

Due to this construction, Hennessy's algorithms for determining whether a variable is current (and the modified versions of Wall et al [WST85]) will provide all current variables in programs that either:

- contain no old labels, or
- contain old labels, but due to optimization do not perform the assignments represented by those labels (even into temporaries),

that is, programs that do not perform more than one assignment to any variable within a basic block. These algorithms will fail to provide some available current values in programs that perform more than one assignment to any variable within a basic block.

These algorithms cannot provide all current variables for highly optimized code. It clearly follows that the algorithms cannot provide all current variables for partially optimized code, which in addition may contain dead stores to variables subsequently assigned within a block. A general solution must work not only on highly optimized code, but on partially optimized and unoptimized code as well.[3]

## 3 Examples

Figure 3.1 is a reproduction of Figure 7 from Hennessy's paper.[4] In this example, if execution is suspended at instruction 2′ then **F** is noncurrent because the code that should have assigned into **F** has been eliminated and **D** is noncurrent because the code that assigns into **D** has been executed earlier than in the unoptimized code.

Figure 3.2 is a modified version of the same example. A use of **F** has been inserted between the first and second assignments into **F**, thus code must be generated for the first assignment into **F**: **F** is assigned into twice in the same basic block. If execution is suspended at the corresponding point in this example (now instruction 3′ rather than 2′ due to the presence of the additional assignment), **F** is current. **F** has the same value at that point in the optimized and unoptimized code. However, the augmented dag for this modified example contains an old label for **F** at node 1, just as in the case when no code was generated for the node. Algorithm 1 from Wall et al [WST85], shown in Figure 3.4, therefore reports **F** as being noncurrent (to relate this example to Algorithm 1, let L be the label on node 1: nodepointer of L $< 4$ (the node number of the breakpoint location), L is an old label, and **F** is not in *Fixed*).[5]

[2] It should be noted that this will occur independently of the compiler's choice for the location of the temporary – it will happen even if the compiler chooses the storage location of the variable as the 'temporary' location. It was once safe to assume that this would not occur, because variables were located in memory, but today variables may be located in registers, and the compiler may well choose the same location for the 'temporary' and 'permanent' variable locations.

[3] Compilers routinely offer partial optimization in the form of optimization levels. These typically allow a trade-off between speed of compilation and completeness of optimization.

[4] A typographical error in the original figure has been corrected: the roll forward variable (RFV) is **F**, as is clear from the code and the text, but is shown in the original figure as **A**.

[5] The problem addressed by Wall et al in [WST85] is orthogonal to the problem introduced by our new interpretation of the

| Unoptimized Code | Optimized Code | Noncurrent Variables | |
|---|---|---|---|
| | | RBV | RFV |
| 1. F := D | | | |
| 2. A := B + C | 1′. D := B | D | F |
| 3. D := B | 2′. A := B + C | | |
| 4. F := E | 3′. F := E | | |

| Root Order | Code Order |
|---|---|
| $1 < 4 < 5$ | code start (2) < code start (4) < code start (5) |

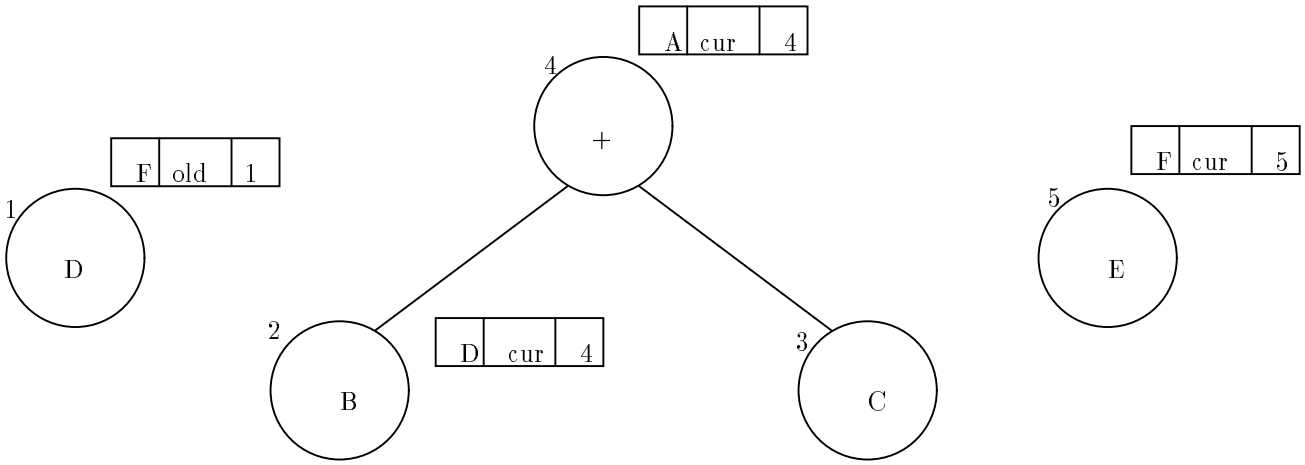


Figure 3.1: An Example (Figure 7) from Hennessy's Paper

The problem results from the fact that a current variable corresponds to a node labelled old. It should be possible to modify the expression dag construction algorithm to allow multiple current labels, so that every assignment into a variable within a block is labelled current, and a node labelled old truly represents an eliminated assignment. However, Hennessy's algorithms assume that there is only one current label for a variable in a block. This can be seen by examining Figure 3.3, which is a further modified version of the example. The label on node 1 is current because the assignment does take place. A dead store into **F** has been added (statement 5, node 6) and eliminated by optimization. If execution is suspended at statement 5′ (corresponding to statement 6 in the unoptimized code), **F** is noncurrent because of the eliminated assignment. However, Algorithm 1 reports **F** as current because the current label of **F** on node

machine code – Hennessy's original Algorithm 1 also reports **F** as being noncurrent.

| Unoptimized Code | Optimized Code | Variables Reported as Noncurrent | |
|---|---|---|---|
| | | RBV | RFV |
| 1. F := D | 1′. F := D | | |
| 2. A := B + C | 2′. D := B | D | F |
| 3. D := B | 3′. A := B + C | | |
| 4. G := F + A | 4′. G := F + A | | |
| 5. F := E | 5′. F := E | | |

| Root Order | Code Order |
|---|---|
| $1 < 4 < 5 < 6$ | code start (1) < code start (2) < code start (4) < code start (5) < code start (6) |

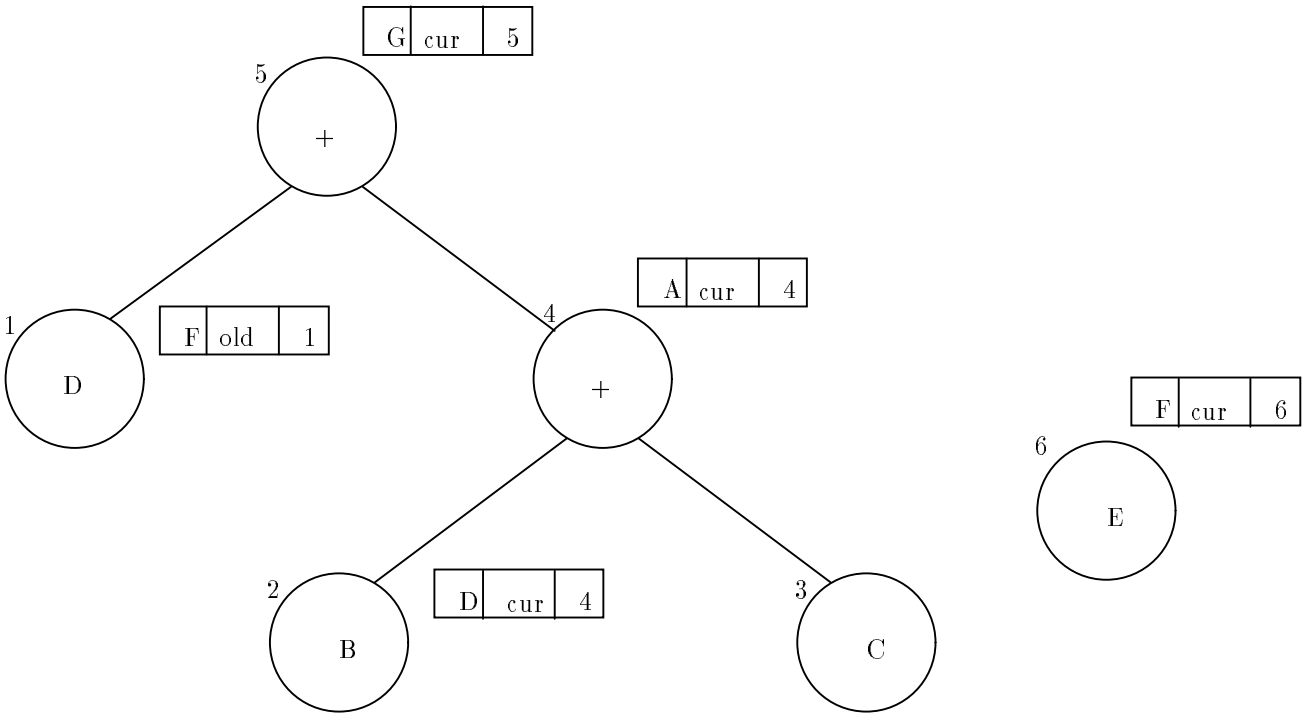


Figure 3.2: A Modified Example on which Hennessy's Algorithms Are Conservative

| Unoptimized Code | Optimized Code | Variables Reported as Noncurrent | |
|---|---|---|---|
| | | RBV | RFV |
| 1. F := D | 1′. F := D | | |
| 2. A := B + C | 2′. D := B | D | |
| 3. D := B | 3′. A := B + C | | |
| 4. G := F + A | 4′. G := F + A | | |
| 5. F := H | 5′. F := E | | |
| 6. F := E | | | |

| Root Order | Code Order |
|---|---|
| $1 < 4 < 5 < 7$ | code start (1) < code start (2) < code start (4) < code start (5) < code start (7) |

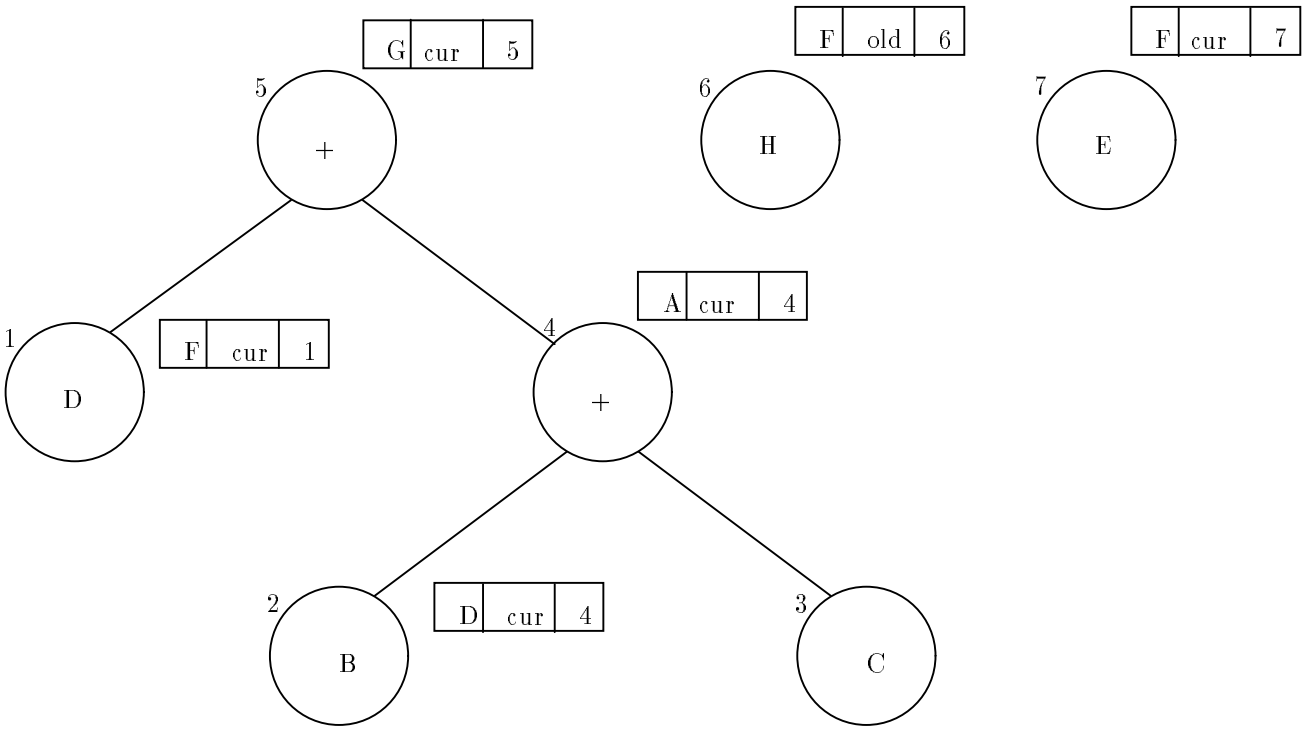


Figure 3.3: A Further Modified Example: if multiple current labels are allowed for a single variable in a block (here **F**), Hennessy's algorithms will give incorrect results.

Input: A dag $D$ and a node $d$ that is the error or breakpoint location.
Output: The sets RFV and RBV.

Method:
RFV := $\emptyset$; RBV := $\emptyset$;
{ First pass: find the identifiers that were stored into }
Fixed := $\{v | v$ is a current label on some node $n$ with codestart($n$) < codestart($d$)$\}$
for each label $L$ in $D$ do begin
  $n$ := the node labeled by $L$;
  $v$ := the variable in the label $L$;
  if (nodepointer of $L < d$)
      { i.e., the assignment should have been done }
    and ((codestart($n$) > codestart($d$))
      { but was not because of code motion }
      or ($L$ is an old label) and ($v$ is not in Fixed))
      { or because of a deleted store whose absence is not made irrelevant by a
      later store that is actually executed }
  then RFV := RFV $\cup\{v\}$;
  { compute the RBV set }
  if ($L$ is current) and (codestart($n$) < codestart($d$))
      { i.e., the assignment was executed }
    and (nodepointer of $L \geq d$)
      { but should not have been executed yet }
  then RBV := RBV $\cup\{v\}$;
end

Figure 3.4: Algorithm 1 from Wall et al

1 causes **F** to be placed in the set *Fixed*; when subsequently the old label on node 6 is examined, **F** is rejected from membership in the RFV (Roll Forward Variable) set because **F** is in *Fixed*. Note also that in the computation of the RBV (Roll Backward Variable) set, if an assignment to a variable $V$ and the corresponding uses of $V$ are moved up (occur earlier in the optimized code than in the unoptimized code) so that they cross both the breakpoint and another (current) assignment to $V$, $V$ will be incorrectly placed in RBV. Algorithm 2 also relies upon the assumption that there is a single current label for a variable in the dag, particularly in the function *Available*.

## 4 Related Work

Since Hennessy's paper, several groups have worked on this or related problems. Coutant et al [CMR88] describes a compiler/debugger pair modified to allow source-level debugging of optimized code. The part of their work relevant to this discussion is that the debugger reports when variables have been made noncurrent due to instruction scheduling within a basic block. The compiler tracks stores to user variables that have been moved across statement boundaries. As of the 1990 implementation, a variable was reported to be noncurrent after its last use, although it may still be available in a register or in memory [Mel90]. Streepy

[Str91] describes a rich compiler/debugger interface designed to allow source-level debugging of optimized code. The interface defines source units beyond subroutines and statements (it includes loops, blocks, and expressions) and includes a *source range table* mapping between source units and sequences of object code generated from them. This provides sufficient information to determine whether a variable is current at a breakpoint when local optimizations have been performed: within a basic block, there is a total order on source statements and a total order on object code addresses, and the source range table gives a mapping between them, allowing differences in order to be determined.[6] Across basic blocks, the orderings on source statements and object code addresses are partial orders, thus the source range table does not provide sufficient information to determine whether a variable is current at a breakpoint: when global optimizations have been performed, data flow information is needed.

Data flow techniques have been successfully applied to problem of currentness determination in the presence of both local and global optimization by Copperman and McDowell [Cop92], [CM91] and independently by Bemmerl and Wismueller [BW92]. Reaching definitions are computed before optimization is performed. In Copperman and McDowell's work, a second reaching definitions computation is done after optimization is performed, and a comparison of the definitions of a variable that reach a breakpoint before and after optimization is used to determine if the variable is current at the breakpoint. In Bemmerl and Wismueller's work, each definition of a variable found to reach a breakpoint before optimization is tested to determine whether it is killed along any path along which it is 'supposed' to reach the breakpoint.

## 5 Summary

Many debuggers occasionally give misleading information about the value of a variable at a breakpoint in an optimized program. Hennessy's algorithms for determining when a reported value would be misleading due to local optimizations assume that optimized code contains only one assignment to any variable within a basic block. This assumption was valid when he made it, but is now overly restrictive: compiler and debugger technology has progressed so that what were once considered compiler temporaries are now considered aliases for variables, thus there may be more than one assignment to a variable within a block. The problem of currentness determination is more general than the problem that Hennessy solved. Several groups have worked on the more general problem with some success.

## References

[AU77] Aho, A.V. and Ullman, J.D. *Principles of Compiler Design,* Addison Wesley, Menlo Park, CA, 1977.

[AU73] Aho, A.V. and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling,* Prentice Hall, Englewood Cliffs, N.J., 1973.

[6] The information is sufficient provided that assignment is considered to be an expression, and thus a mapping between assignment operators and the object code that sets the variable (the store into memory, computation into a register, or register copy operation) is provided. It is not clear from Streepy's paper that this is the case, but if not, it is clearly possible to extend the interface in this direction.

[AU72] Aho, A.V. and Ullman, J.D. "Optimization of Straightline Code," Siam J. Comput. 1, 1(Jan. 1972), 1-19.

[Cop92] M. Copperman, "Debugging Optimized Code Without Being Misled," UCSC Technical Report UCSC-CRL-92-01, January 1992. Submitted for publication to *ACM Transactions on Programming Languages and Systems.*

[CM91] Copperman, M., McDowell, C. "Debugging Optimized Code Without Surprises (Extended Abstract)," *Proceedings of the Supercomputer Debugging Workshop*, Albuquerque NM, November 1991

[CMR88] D. Coutant, S. Meloy, M. Ruscetta "DOC: a Practical Approach to Source-Level Debugging of Globally Optimized Code," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125-134, 1988.

[Sil92] Personal Communication, J. Silverstein, ed., "DWARF Debugging Information Format," Proposed Standard, UNIX International Programming Languages Special Interest Group, April 1992.

[Hen82] Hennessy, J. "Symbolic Debugging of Optimized Code," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 323-344, 1982

[Mel90] Personal Communication, Meloy, S., Hewlett-Packard, 3345 Mount Pleasant Rd., Lincoln, CA

[Hen90] Personal Communication, Hennessy, J., Center for Integrated Systems, Stanford University, Stanford, CA

[Str91] L. Streepy, "CXdb A New View On Optimization," *Proceedings of the Supercomputer Debugging Workshop* , Albuquerque, November 1991.

[Wan91] Personal Communication regarding Microtec Research's Xray Debugger, Wang, F., Microtec Research, Inc., Santa Clara, CA, January 1992

[Wis92] Personal Communication, Wismueller, R., Institut fur Informatik, Technische Universitat Munchen, Munich, Germany, March 1992

[BW92] T. Bemmerl, R. Wismueller, "Quellcode-Debugging von global optimierten Programmen", Proceedings of the GI-ITG Workshop "Parallelrechner und Programmiersprachen", Schloss Dagstuhl, Germany, Feb. 1992

[WST85] D. Wall, A. Srivastava, R. Templin, "A note on Hennessy's *Symbolic Debugging of Optimized Code*," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 176-181, Jan. 1985.