Capturing the Effects of Code Improving Transformations*

Clara Jaramillo, Rajiv Gupta and Mary Lou Soffa
Department of Computer Science
University of Pittsburgh
Pittsburgh,PA 15260

Abstract

Symbolic debugging of transformed code requires information about the impact of applying transformations on statement instances so that the appropriate values can be displayed to a user. We present a technique to automatically identify statement instance correspondences between untransformed and transformed code and generate mappings reflecting these correspondences as code improving transformations are applied. The mappings support classical optimizations as well as loop transformations. Establishing mappings requires analyzing how the position, number, and order of instances of a statement can change in a particular context when transformations are applied. In addition to enabling symbolic debugging of transformed code, these mappings can be used to understand transformed code and to compare values computed in both program versions either manually or automatically.

1. Introduction

Compilers apply code improving transformations to achieve high performance for various types of computing platforms including scalar, superscalar, and parallel machines. Although code improving transformations are extremely beneficial, their application creates challenges in symbolically debugging the transformed code. Transformed code must be debugged because its behavior may differ from that of the untransformed code. First, to maximize performance, the compiler may apply unsafe transformations. For instance, the reordering of operations under some conditions can cause overflow or underflow of values and may even cause a program to crash (e.g., division by zero). Second, the user may override a data dependency to manually apply a code transformation (e.g., parallelizing transformation), which is actually invalid and introduces an error in the transformed program. Third, the compiler itself may contain an error in the implementation of a particular transformation. And lastly, the transformed program may uncover an error that was not detected in the untransformed program (e.g., uninitialized variable). For these reasons, it is important to debug the transformed version of a program.

When the behaviors of the transformed and untransformed program versions differ erroneously, we must locate the cause of the errors through debugging. The existing approach to this problem is to utilize source level debuggers that allow a user to debug from the point of view of the source level program but actually execute the transformed code [1, 2, 3, 4, 5, 6, 7, 10, 11, 13]. To accomplish this task, mappings are used to track the relationship between an untransformed program and its transformed version. The mappings are statically analyzed to determine the placement of breakpoints as well as to determine if values of source level variables are reportable at given breakpoints. However, with these mappings, all values of source level variables are not reportable and code improving transformations, such as loop interchange, are not supported. These limitations are partly due to mapping only statements (and not their instances) from the untransformed program to corresponding statements in the transformed program.

Code improving transformations restructure a program by moving, modifying, deleting, and adding code, and as a result, change the position, number, and order of executing statement instances. Thus, it is necessary to establish a correspondence between statement *instances* of transformed and untransformed versions using *mappings*, which identify the type of correspondence between statement instances. A statement instance in the untransformed program and a statement instance in the transformed program *correspond* if the values computed by the two instances should be the

*Partially supported by National Science Foundation grant CCR-9402226 and a grant from Hewlett-Packard.

same and the latter was derived from the former by the application of some transformations. To be practical, these correspondences must be generated automatically as the code is being transformed. Moreover, we must consider a broad class of transformations including classical optimizations and loop transformations.

In this paper, we present a technique for generating mappings between statement instances in the untransformed and transformed versions of a program when classical optimizations and loop transformations are applied. We demonstrate the usefulness of the mappings in symbolically debugging a transformed program. A technique is developed to automatically generate mappings. The mappings reflect the effects of transformations and are established by analyzing how the position, number, and order of instances of a statement change as transformations are applied. No restrictions are placed on the order or number of transformations. We assume transformations are applied either at the source level or intermediate levels, but this work is also applicable at the target code level. In this paper, due to space limitations, we focus on loop transformations.

The paper is organized as follows. Sections 2 and 3 define and describe how to generate mappings for loop transformations. Section 4 demonstrates the usefulness of the mappings in a symbolic debugger, and conclusions are given in Section 5.

2. Capturing effects through mappings

Loop transformations move, modify, delete, and add statements in a program. As a result, the number of instances of a statement can increase, decrease, or remain the same in the transformed program as compared to the untransformed program. Also, the instances of a statement can be reordered and distributed among several statements. Therefore, during execution of the untransformed and transformed programs, corresponding statements may execute in a different order, and the number of their instances may differ. For example, loop invariant code motion (LICM) moves a loop invariant statement S out and above loops. The number of times statement S executes in the untransformed code is greater than the number of times S executes in the transformed code. Loop interchange interchanges two nested loops. Although statements within the effected loop bodies remain within the same loops in both programs and therefore execute the same number of times, the execution order of their instances differs in both programs. Also, corresponding loop headers of the interchanged loops appear in different loop nest levels, and thus, execute a different number of times in both programs.

To capture the effects of code improving transformations, we developed mappings that associate *statement instances* in the untransformed program and the corresponding *statement instances* in the transformed program as transformations are applied. We represent mappings by labeled edges between corresponding statements in both programs. Labels identify the instances in the untransformed program and the corresponding instances in the transformed program. Let OS(S) denote an ordered sequence of instances of S. A label of a mapping between statement S from the untransformed program and statement S' from the transformed program is of the form:

$$OS(S) \rightarrow OS(S').$$

The ordered sequences in the mappings express the correspondences between instances of two statements. The number of elements in the two sequences may be the same or may differ. For example, if there is an one-to-one correspondence between the instances, then the number of instances would be the same. Corresponding instances may appear in the same or different order (e.g., reverse order). If the number of instances is not the same, a consecutive subsequence of instances in one sequence corresponds to a single instance in the other. It should be noted that corresponding statement instances are computed statically but the mappings are between all potential dynamic instances. All of the instances in both sequences may not execute, but for the instances that do execute, the mappings capture the dynamic correspondences.

To refer to one or more instances of a statement, the loop iterations in which the instances execute are specified. Therefore, each statement in the program is viewed with respect to the looping structure in which it is enclosed. A statement S not enclosed within a loop is referred to as "*one*" instance of S. A statement S within a **single** loop is referred to by the following terms:

- "*one*" denotes *an instance* of S that executes in *each iteration* of the loop.
- "*all*" denotes *all instances* of S that execute in *all of the iterations* of the loop.
- "*last*" denotes *the instance* of S that executes in the *last iteration* of the loop.
- "c" denotes the instance of S that executes in the c^{th} iteration of the loop, where c is a constant.
- "$\{l, u, i\}$" and $i \geq 0$ denotes the instances of S that execute in the increasing sequence of iterations $(l, l+i, l+2*i, \ldots, end)$ where $((end \leq u$ and $(end+i) > u))$ of the loop.[1]

[1] A decreasing sequence can also be denoted similarly.

Let S represent a statement in the untransformed program, S' a statement in the transformed program, and $S_{(i)}$ denote instance i of statement S. Examples of mapping labels generated from applied transformations are:

- Code reordering: $one \rightarrow one$ indicates that for each iteration i of a loop, $S_{(i)}$ corresponds with $S'_{(i)}$.
- Loop invariant code motion: $all \rightarrow one$ indicates that for all iterations i of a loop, $S_{(i)}$ corresponds with $S'_{(1)}$.
- Loop reversal: $\{1, 10, 1\} \rightarrow \{10, 1, -1\}$ indicates that for each iteration i of a loop, $S_{(i)}$ corresponds with $S'_{(11-i)}$.
- Loop unrolling: $\{1, 10, 2\} \rightarrow \{1, 5, 1\}$ indicates that for each iteration i of a loop, $S_{(i*2-1)}$ corresponds with $S'_{(i)}$.

To refer to statements within **nested** loops, the notation is extended by using a vector to represent iterations of the loop nest. Without loss of generality, we assume a program is enclosed within a loop of one iteration, denoted by L_0. A statement S is identified as being nested within a loop nest $L = L_0, L_1, \ldots, L_n$ where L is all of the loops enclosing S, numbered successively from the outermost to the innermost loop, and $n+1$ is the number of loop nest levels. Each iteration of loop nest L uniquely identifies instances of statement S, and instances of statement S are ordered by the order of iterations of loop nest L. An ordered sequence of instances of a statement within loop nest $L = L_0, L_1, \ldots, L_n$ is specified by an $(n+1)$-dimensional vector where each element is subscripted such that an element with subscript i represents an ordered sequence of iterations of loop L_i. The order in which vector elements are specified determines the order of instances in the sequence.

Examples of mapping labels for nested loops are:

- $(one_0, one_1) \rightarrow (one_0, one_1)$ indicates that for each iteration (i, j) of a loop nest, $S_{(i,j)}$ corresponds with $S'_{(i,j)}$. As a shorthand, $\overrightarrow{one} \rightarrow \overrightarrow{one}$ denotes $(one_0, one_1) \rightarrow (one_0, one_1)$.

- Partial dead code elimination: $(one_0, last_1) \rightarrow (one_0)$ indicates that for each iteration of a loop nest, $S_{(i,last)}$ corresponds with $S'_{(i)}$.

- Loop interchange (on the loop body): $(one_0, one_1) \rightarrow (one_1, one_0)$ indicates that an instance of S executed in iteration (i, j) corresponds to the instance of S' executed in iteration (j, i); that is, $S_{(i,j)}$ corresponds with $S'_{(j,i)}$.

- Loop interchange (on the loop header): $(one_0, all_1, one_2) \rightarrow (one_0, one_1)$ indicates that an instance of S executed in iteration (i, j, k) corresponds to the instance of S' executed in iteration (i, k).

3. Generating mappings

The transformed program initially starts as an identical copy of the untransformed program with mappings between corresponding statements in the two programs. Initially, all of the mapping labels are $\overrightarrow{one} \rightarrow \overrightarrow{one}$ instance associations because corresponding statements are enclosed by the same loops and thus execute within the same loop iterations. Mappings change as transformations are applied. That is, the labels of mappings may change and/or new mappings may be established. The mappings for individual transformations are determined by using the semantics of those transformations with respect to the untransformed program. The effects on mappings after a single (initial) transformation is applied are described first, followed by the effects when a series of transformations are applied.

Consider loop invariant code motion, illustrated in Figure 1. LICM moves a statement S from a loop at nesting level n to an outer loop at nesting level $n-1$. For explanatory purposes, we refer to the statement in the transformed code as statement S'. After LICM is applied to statement S, in each iteration of this outer loop, *all* instances of S correspond to *one* instance of S', and the mapping label of S and S' is changed from $one_{0..n} \rightarrow one_{0..n}$ to $one_{0..n-1}, all_n \rightarrow one_{0..n-1}$.

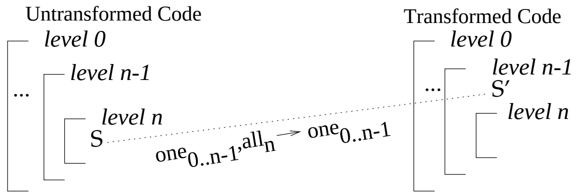

Figure 1. Mapping label after applying LICM

Loop interchange exchanges the positions of two loops in a loop nest. As a result, the order of loop iterations differs in the untransformed and transformed code. If a loop at nesting level i is interchanged with a loop at nesting level $i+1$, as shown in Figure 2, then the instances of the statements of the inner loop body are reordered but execute the same number of times. The instances referred by these statements in the mapping labels are permuted to match the interchanged order, and therefore, the mapping

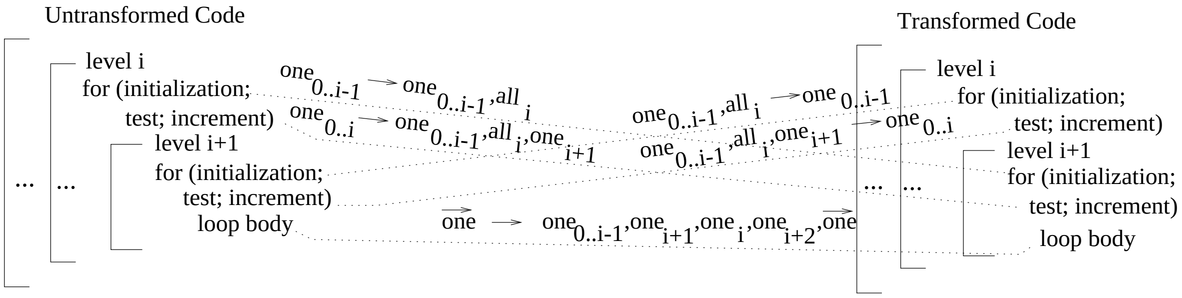

Figure 2. Mapping label after applying loop interchange

labels of these statements change from $\overrightarrow{one} \rightarrow \overrightarrow{one}$ to $\overrightarrow{one} \rightarrow one_{0..i-1}, one_{i+1}, one_i, one_{i+2}, \overrightarrow{one}$. Also, the mapping labels of the loop headers at nesting level i and $i+1$ in the untransformed code are also affected, as shown in Figure 2. The statement instances of the loop initialization, test, and increment of the i^{th} loop in the untransformed code will execute more often in the transformed code because they are in an inner loop in the transformed code. The statement instances of the loop initialization, test, and increment of the $i+1^{th}$ loop in the untransformed code will execute less often in the transformed code because they are in an outer loop in the transformed code.

In practice, transformations are applied repeatedly. Transformations can be applied in any order and as many times as desired. As a subsequent code transformation is applied, the mappings are changed to reflect the composition of the previous mapping (the effects of all previously applied transformations) by the effects of the current transformation. Since only a finite number of program transformations are supported and the instance associations depend on the iterations of the loops in the program, the number of possible ways in which instance associations can be made is restricted.

Table 1 displays the form of the mapping labels generated as a result of applying a particular code transformation to a statement S' in the transformed program, given an existing mapping of S and S' where S is a corresponding statement in the untransformed program. Let the existing mapping label of S and S' be $\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ where $\langle * \rangle_i$ denotes element i. We show how elements in the vector are affected as code transformations are applied. Due to space limitations, only the effects of loop transformations on loop bodies are shown, but the effects on loop headers (i.e., initialization, test, and increment) are similar. We also omit entries when the instances referred by S in the mapping labels do not change after the application of a transformation; this situation occurs when statement S in the untransformed code is not enclosed by the effected loop.

When statements are moved out of a loop at nesting level j for LICM and PDE, element $\langle * \rangle_j$ in the vector for S' is removed. Also, *all* or *last* is applied to element $\langle * \rangle_j$ in the vector for S. Now *all*, *last*, c, and $\{i, l, u\}$ refer to a specified set of instances.

Loop reversal reverses the iterations of a loop at nesting level j. As a result, the instances of the statements of the loop body at nesting level j are reversed. This effect is captured by applying $\{last, 1, -1\}$ to the element j in the vector for S. As a result, element j changes from $\langle * \rangle_j$ to $\{last, 1, -1\}(\langle * \rangle_j)_j$.

Loop peeling removes iterations of a loop by placing copies of the loop body before or after the loop and modifying the loop header. Instances of effected statements in the loop body of the untransformed program are distributed among several statements in the transformed program. Suppose a loop is peeled k times before the loop. Instances 1 through k of the loop body in the untransformed code refer to the peeled copies of the loop body in the transformed code. The first entry for loop peeling specifies the mapping labels for the instances 1 through k of the loop body in the untransformed code. The remaining instances of the loop body in the untransformed code refer to the instances of the loop body in the transformed code.

Loop distribution, jamming, normalization, and skewing do not reorder nor change the number of instances in the loop body and thus, the mapping labels of the loop bodies do not change when these transformations are applied. Strip mining, which creates nested loops, uses indices in the sequences to reflect the dependence of the inner loop limits on the outer loop index "is". We do not explicitly show tiling as it can be derived from strip mining and loop interchange.

4. Applications

Our mappings are beneficial for source level debugging of transformed code. Previous work for source

Transformation	loop body mapping labels
loop invariant code motion (LICM)	$\langle * \rangle_{0..j-1}, all(\langle * \rangle_j)_j, \overrightarrow{\langle * \rangle} \rightarrow \langle * \rangle_{0..j-1}$
partial dead code elimination (PDE)	$\langle * \rangle_{0..j-1}, last(\langle * \rangle_j)_j, \overrightarrow{\langle * \rangle} \rightarrow \langle * \rangle_{0..j-1}$
loop interchange	$\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}, \langle * \rangle_{j-1}, \langle * \rangle_{j+1}, \langle * \rangle_j, \langle * \rangle_{j+2}, \overrightarrow{\langle * \rangle}$
loop reversal	$\overrightarrow{\langle * \rangle}, \{last, 1, -1\}(\langle * \rangle_j)_j, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop distribution, jamming,	$\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop normalization, skewing	$\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop unroll k times (n iterations)	$\overrightarrow{\langle * \rangle}, \{i, n, k+1\}(\langle * \rangle_j)_j, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$, for $1 \leq i \leq k+1$
loop peel k times (before the loop)	peeled copies: $\overrightarrow{\langle * \rangle}, i(\langle * \rangle_j)_j, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$, for $1 \leq i \leq k$ loop: $\overrightarrow{\langle * \rangle}, \{k+1, last, 1\}(\langle * \rangle_j)_j, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop unswitching	conditional : $\overrightarrow{\langle * \rangle}, all(\langle * \rangle_j)_j, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ other statements : $\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
software pipelining (k stages)	startup code : $\overrightarrow{\langle * \rangle}, i(\langle * \rangle_j)_j, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$, for $1 \leq i \leq k-1$ loop: $\overrightarrow{\langle * \rangle}, \{i, last-(k-i), 1\}(\langle * \rangle_j)_j, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$, for $1 \leq i \leq k$ cleanup code: $\overrightarrow{\langle * \rangle}, i(\langle * \rangle_j)_j, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$, for $last-(k-2) \leq i \leq last$
strip mining	$\overrightarrow{\langle * \rangle}, (\{is=1, last, strip\}\{i=is, min(last, is+strip-1, 1\})_j, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$

Table 1. Loop transformation effects on mappings with $\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ labels on loop bodies

level debuggers of transformed code developed mappings of *statements* to track an untransformed program to its transformed version [1, 2, 3, 4, 7, 10, 11, 13, 14]. Mappings are statically analyzed to determine the placement of breakpoints and determine if values of source level variables are reportable at given breakpoints. All values of source level variables are not reportable and code improving transformations, such as loop interchange, are not allowed. One reason is that the mappings are too coarse. Only statements from the untransformed program are mapped to corresponding statements in the transformed program. Consider the example in Figure 3, in which the loops are interchanged, unrolled, and jammed, and as a result, some instances of statements are reordered and deleted. If only mappings of *statements* (i.e., no labels) are used by a source level debugger that executes transformed code, the debugger is ineffectual. For any breakpoint placed within the loops, all variables (except k) inside the loop are considered noncurrent and their expected values cannot be accurately reported. The debugger does not have knowledge of what values can be reported because it does not have information about loop iterations and statement instances. For example, if the user places a breakpoint after statement S in the innermost loop of the untransformed program and requests the value of $a(i, j)$, the debugger cannot report the expected value of $a(i, j)$, regardless of where the breakpoint is placed in the loop nest of the transformed code. The debugger does not know if the expected value has been computed because the instances of S have been reordered and distributed between statements S'_1 and S'_2, and it has no information of how the instances of S correspond with instances of S'_1 and S'_2.

By using our mappings, shown in Figure 3, all source level values computed in the transformed program can be tracked to the source program and accurately reported. Thus, a source level debugger can provide precise information without burdening the user. However, since values may be overwritten, a mechanism is needed to save values that may be overwritten during reportable ranges. Our mappings can be used to identify what values have to be saved and for how long.

Our mappings have been used in the development of a tool [8] that automatically compares source level values produced during execution by the untransformed and transformed versions of a program and reports if all the values are the same; if there is a difference, it reports where in the code the values differ. Mappings are generated automatically as program transformations are applied and thus, the user is not burdened with inserting mappings. Another work that uses map-

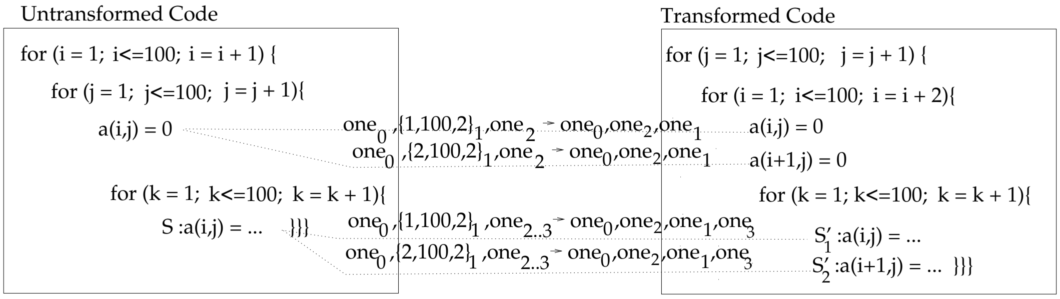

Figure 3. Example of mapping instances of statements

pings and compares two programs is Guard[12], a relative debugger, which compares the execution of one program, the reference program, with the execution of another program, the development version. Guard has also been extended to a parallel relative debugger, but in both cases, the user must insert the mappings.

Finally, our mappings can help users understand transformed code and can further enhance information provided by programming environment tools [9], which help users decide how to restructure programs by analyzing and performing transformations to detect and exploit parallelism.

5. Conclusions

This paper develops mappings that capture the effects of program transformations with respect to a source level program. Statement instances in the untransformed program are mapped to corresponding statement instances in the transformed program. This allows us to track statement instances effected by classical optimizations and loop transformations.

The mappings are automatically generated as code improving transformations are applied, and thus, there is no need for a programmer's interaction. The mappings produced reflect the effects of applying a series of transformations without carrying any history about the particular transformations applied.

References

[1] Adl-Tabatabai, A. *Source-Level Debugging of Globally Optimized Code.* PhD dissertation, Carnegie Mellon University, 1996. Technical Report CMU-CS-96-133.

[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 Trans. Prog. Lang. Syst.*, 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] Fritzson, P. A Systematic Approach to Advanced Debugging through Incremental Compilation. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symp. on High-Level Debugging*, pages 130–139, 1983.

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

[7] Hennessy, J. Symbolic Debugging of Optimized Code. *ACM Trans. Prog. Lang. Syst.*, 4(3):323–344, July 1982.

[8] Jaramillo, C., Gupta, R., and Soffa, M.L. Debugging of Optimized Code using Value Comparisons. Technical Report 96-18, Computer Science Department, Pittsburgh, PA, September 1996.

[9] Kennedy, K., McKinley, K.S., and Tseng, C. Interactive Parallel Programming Using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[10] Pineo, P.P. and Soffa, M.L. A Practical Approach to the Symbolic Debugging of Code. *Proceedings of International Conference on Compiler Construction*, 26(12):357–373, April 1994.

[11] 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.

[12] Sosic, R. and Abramson, D. A. Guard: A Relative Debugger. *Software Practice and Experience*, 27(2):185–206, February 1997.

[13] 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.

[14] Zellweger, P.T. An Interactive High-Level Debugger for Control-Flow Optimized Programs. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symp. on High-Level Debugging*, pages 159–171, 1983.