DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code

Deborah S. Coutant
Sue Meloy
Michelle Ruscetta

Hewlett-Packard
19447 Pruneridge Ave.
Cupertino, CA 95014

Abstract

As optimizing compilers become more sophisticated, the problem of debugging the source code of an application becomes more difficult. In order to investigate this problem, we implemented DOC, a prototype solution for Debugging Optimized Code. DOC is a modification of the existing C compiler and source-level symbolic debugger for the HP9000 Series 800. This paper describes our experiences in this effort. We show in an actual implementation that source-level debugging of globally optimized code is viable.

1. Problem

Code transformations performed by global optimization introduce problems which affect the ability to provide a source-level debugging environment. Our optimizer was designed for a RISC-based architecture providing a large number of registers and an exposed instruction pipeline. [1]. In order to take advantage of the architecture, the compiler must make efficient use of the registers and must schedule instructions effectively.

An optimizing compiler faces two main problems which affect the ability to provide source-level debugging [2, 3]. 1) The *code location* problem concerns maintaining a mapping between source lines and machine instructions. 2) The *data value* problem concerns finding the current location of a variable's value, and displaying it consistent with the user's perception of the source.

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

2. Related work

2.1. Collection of compile-time information

The first area of research related to this work concerns the idea that information about a program can be collected at compile-time. This information can be very useful to the programmer in trouble shooting and understanding his program [4, 5, 6, 7].

DOC is an extension of this idea in that the information collected is added to the usual debug information and is used by the debugger during an interactive debug session.

2.2. General solutions

The problem of symbolically debugging globally optimized code is presented by Zellweger [8, 2] and Seidner [3]. Both describe the general data value and code location problems. These works, as well as Teeple's [9] work, discuss inlining and cross-jumping. These two types of transformations exemplify the traditional control-flow problem. DOC deals almost exclusively with the data value problem.

Conway [10] describes a PL/I compiler that does some local code optimizations, and interfaces with a symbolic debugger. DOC is based on a global optimizer.

Hennessy [11] provides algorithms for recovering the values of non-current variables from

labeled DAGs. His work addresses the data value problems in detail, and therefore is the most closely related to our work. Some differences are:

- His work is at a higher level in the compiler model, where the nodes in the DAG correspond to symbol table entries for variables. DOC is at a lower level, since our optimizer transforms compiler-generated assembly code.
- He does not address the problem of tracking the locations of a variable's values from memory through registers. Providing a solution to this problem was a major goal of DOC.

2.3. Providing partial information

Warren [12] suggests a system in which the user receives partial information concerning the values of variables. Our solution is similar.

Warren also defined data structures to hold the information the debugger would require. He estimated the memory requirements, and concluded that the cost would not be prohibitive. While our data structures are quite different, we show in an actual implementation that source-level debugging of globally optimized code is viable.

The TLD Systems Ada debugger [7] supports source-level debugging of globally optimized code. It warns the user if a variable's memory location might not have an up-to-date value at all points in its lifetime. DOC attempts to be more specific, by tracking the locations of variables through registers and memory.

3. The DOC prototype

The DOC prototype was implemented as an extension to a family of compilers, each of which use the same debugger and optimizer.

3.1. Overview of the compiler

The compiler used for the prototype solution consists of a language-dependent front-end, and a language-independent global optimizer. The compiler front-end provides aliasing information for the optimizer, and provides information about user variables and source lines for the debugger.

The aliasing information is communicated to the optimizer through a table containing entries for every memory location and register used in the program [13].

The global optimizer is a multiple pass optimizer [14] that uses global data flow information contained in intervals to perform various local and global optimizations. These optimizations include register promotion, common subexpression elimination, loop optimizations, coloring register allocation, and instruction scheduling [15].

Most optimizations are performed by separate passes of the optimizer [16].

3.2. Overview of the debugger

The debug information generated by the compiler is contained in the object file, in an area that is not loaded during normal execution of the program.

The debugger is a source-level symbolic debugger [17], which displays the source in one window while commands are entered in another.

4. Goals

The overall design constraint placed on the DOC team was to implement a *practical* solution to the problem. The purpose of DOC was to demonstrate the feasibility of providing this capability in a product. At the same time, we wanted to provide useful functionality to our users.

In addition to the constraints of practicality and usability, other goals of DOC included:

- There would be no change in the optimized code when compiling with debug enabled.
- The user would never be mislead by receiving incorrect information from the debugger.
- The implementation would have a minimal impact on performance and resource consumption.

In order to meet the above objectives we decided to implement a solution that would give the debug user partial information. In the cases where the user wants data that we cannot supply, we try to provide enough background information to allow him to recreate the data himself.

We also decided to disallow data and control flow modification from within the debugger. This simplified the complexity of the task, since no optimization assumptions would be invalidated at runtime.

5. Optimizations addressed by DOC

In order to prove the feasibility of our solution, we wanted to address the optimizations causing the most trouble for a source-level symbolic debugger. DOC concentrates on this subset of optimizations performed by our compiler, includ-

ing:

- register promotion and assignment
- loop variable elimination (induction variable elaboration)
- instruction scheduling

6. DOC session example

The following simple program fragment demonstrates some of the features of DOC. There is a mistake in the **for** statement of the second loop, where **i** is compared to 0 rather than assigned 0. The first **if** statement causes the optimizer to believe that **i** might be initialized. Without this **if** statement, DOC would display the following message at compile time:

WARNING: use of undefined variable i in function main at line 15

```
12:  k = 15;
13:  if (!init)
14:          i = 0;
15:  for (i==0; i!=9; i++) {
16:          for (j=i+1; j!=10; j++) {
17:                  if (a[i].n > a[j].n) {
18:                          tmp = a[i];
19:                          a[i] = a[j];
20:                          a[j] = tmp;
21:                  }
22:          }
23:  }
```

Optimization causes the following debugging problems in this program:

- Line 12 is eliminated.
- **k** is a constant; it does not exist in memory or in a register.
- The assignment into **j** at line 16 is moved across a statement boundary.
- **j** is eliminated and replaced by an induction variable within the second **for** loop.

The debug session is begun with a breakpoint on line 12.

```
>b 12
Added:
1: count: 1 Active  main: 13: if (!init) i = 0;
```

Line 12 has been eliminated, so the breakpoint is set on line 13. The user now starts up the program.

```
>r
Starting process 25796: "a.out"
breakpoint at 0x00001200
```

The user asks to print the value of **k**. DOC is able to recreate the value.

```
>p k
k = 5
```

The user then asks to print the value of **j**, which does not yet have a value.

```
>p j
The value of variable j is not available at this line
```

The user now single steps, which takes him to line 15 (since the **if** test fails), and requests the value of **i**. At this point **i** has been placed in a register by the optimizer, and contains a spurious value.

```
>s
>p i
i = 5
```

The user now steps to line 16, and requests the value of **j**. The assignment to **j** has already occurred, because it was moved into the code for line 15.

```
>s
>p j
Variable j was set early
New value was set during execution of line 15
j = 6
```

After stepping to line 17, the user requests the value of **j** again. At this point, **j** has been replaced by an induction variable, so DOC must recreate the value.

```
>s
>p j
j = 6
```

7. General solution

7.1. Code location solution

In our early investigation, we discovered that most of the problems turned out to be data value problems. Even when code is moved or eliminated, the main effect is that data values may not be as the user expects at every point during execution of the program.

Statement boundaries are tracked by attaching a label to the first instruction of the statement. When the first instruction is moved or deleted, the label is moved to the next instruction. One instruction can have multiple labels, if the code for an entire statement has been moved or deleted.

7.1.1. Breakpoints on eliminated statements

When the user requests a breakpoint on an eliminated statement, the breakpoint is set at the instruction associated with the (moved) statement label. When the user is concerned with a relative control flow position, such as breaking at the first statement in a procedure or a loop, this is the desired interpretation.

When this is not what he wants, then either:

1. He wants to see the value of a variable after modification by a previous statement, or
2. He wants to see the value of a variable before it is modified by a subsequent statement.

In case 1, moving the breakpoint to a later statement will not cause debugging problems. The variable has already been modified, and remains modified at the breakpoint.

In case 2, moving the breakpoint to a later statement will only cause debugging problems if the variable that the user is interested in was modified by the eliminated statement. If the eliminated statement did not modify the variable, its value will remain unmodified at the breakpoint. If the eliminated statement did modify the variable, its value will not be what the user expects at the breakpoint.

For example, see Figure 1.

In either case, what really matters is eliminating variable updates, or moving them across statement boundaries. These transformations can cause debugging problems even when we *can* set a breakpoint on the desired line.

```
1           a = 5;
2           x = a;  this stmt is eliminated
3           y = x;

break at 2  actually breaks at 3
print a     no problem
print y     no problem
print x     problem
```

Figure 1.

7.1.2. Moving instructions

Only those instructions that modify user data are troublesome, and only when they are moved across a statement boundary. Thus, many code motion problems can be solved by the same techniques as for data value problems.

7.2. Data value solution

Our solution for data value problems is to communicate to the debugger *ranges* of information about variables affected by optimization. These ranges can contain location information, or information about updates of user variables that have been moved. All the problems with debugging optimized code that we have investigated can be solved by variations on one basic range data structure.

7.2.1. The range data structure

Our first task was to design an efficient way to represent ranges in the debug information. The information needed for most uses of ranges is minimal. For example, Figure 2 shows a code fragment, and three ranges that could be used to describe the locations of the variable x.

```
0x1FFC    inst          x=500
0x2000    inst
          . . .
0x2020    def 28        x=expr1
0x2024    inst
          . . .
0x2080    last use 28
          . . .
0x2100    store x       x=expr2
          . . .
0x2200    last load x
```

Low Addr	High Addr	Location	Type
0x2000	0x2020	500	constant
0x2024	0x2080	28	register
0x2100	0x2200	-88	memory

Figure 2.

In this example, x lives in three different locations. By comparing the program counter against the low and high addresses of each range, the debugger can determine which range, if any, is currently active.

The type field tells the debugger how to interpret the location. If the type of range is **register**, the value of x can be found in the specified register. If the type is **constant**, the location field is

the actual constant value of x. If the type is **memory**, the location field contains the memory location (in this case, a stack offset).

If the program counter is not contained in any of the ranges, the value of x is unavailable at that point, and the debugger prints a message to this effect.

7.2.2. Range implementation details

In addition to minimal space consumption, another consideration for the range representation was to minimize the time the debugger needed to search for the appropriate range. We decided to sort the range list in some order. This allows the debugger to tell when it has searched far enough, without always looking through the entire list. We chose to sort by the low address, although a sort on the high address would also have worked.

The ranges are emitted to the output file sequentially. The ranges for each variable are delimited by a flag set on the last range for that variable. The debug entry for each variable described by one or more ranges contains an index to the first range for that variable.

Since a global may have ranges in more than one procedure, these ranges must be linked together. This is done by allocating a *continuation* range as the first range for that global in each procedure. This range contains the index of the procedure for which the following list of ranges applies. All continuation ranges for a particular variable are linked together (see Figure 3).

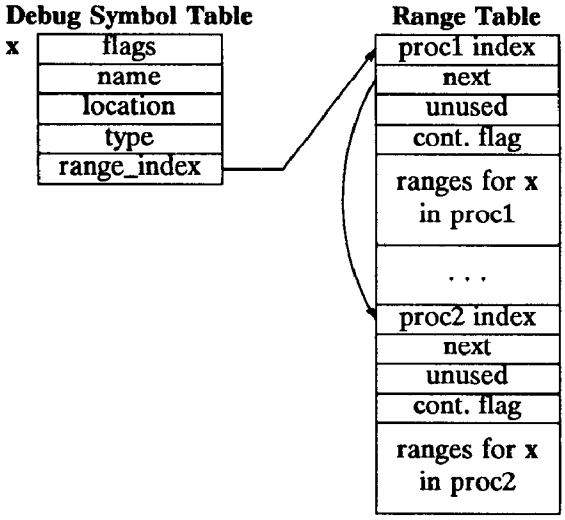

Figure 3.

When the debugger needs to find the value of a global, it traverses the continuation chain until it finds the range list corresponding to the current function, and then searches the range list to find the range(s) enclosing the current code location.

8. Compiler implementation details

8.1. Communication of variable information to the optimizer

Since the optimizer operates at the assembly-level, it does not explicitly deal with user variables. Instead, it tracks memory locations and registers through *resource IDs*. Whenever the front-end wishes to reference a new memory location or register, it allocates a resource ID which is then attached to the instruction. Register resources are allocated by the code generator based on an infinite register set; actual registers will be assigned by the optimizer during register allocation.

In order to support DOC, we now need to associate debug information with the resource IDs corresponding to user variables.

8.2. Register promotion

Information about the definitions and uses of resources is kept in a flow-based data structure similar to def-use chains [18]. One resource could have several chains associated with it. When register promotion occurs, it is actually a def-use chain that is promoted. Thus, for a single user variable, there may be some chains that stay in memory and some chains that are promoted to registers.

When register promotion involves a user variable, we now need to associate the new register resource ID with the original memory resource ID. We also need to mark the instruction defining the new register resource as a *pseudo-store*. This tells later optimizer components to keep track of any movement or deletion of the instruction, since it "stores" into a user variable.

8.3. Register allocation

The register allocator attempts to assign the registers in the most efficient way possible. Thus, a single variable can reside in different locations in different parts of the code. There may also be regions where it does not exist at all. Conversely, a single register can be used to hold multiple variables.

8.3.1. Range calculation

Ranges of sequential instructions must be calculated using data flow information. For example, see Figure 5.

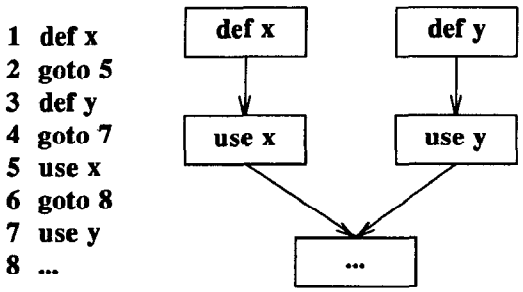

Figure 5.

The live ranges for x and y are disjoint, so the same register could be assigned to both. A simple sequential pass through the code without regard for data flow would result in the range for x including instructions 3 and 4.

The register allocator performs register coloring based on interferences [19]. It constructs an initial interference graph using the def-use chains and interval data flow information. If the graph cannot be successfully colored with the available actual registers, registers are spilled to memory and the graph is adjusted.

The interference graph is built by traversing each basic block interval in reverse order. The initial interference set at the end of the basic block has been previously calculated by interval analysis [20, 21]. As each instruction in the basic block is examined for definitions and new uses of registers, interferences are deleted from and added to the interference set.

Ranges can be calculated as the initial interference graph is constructed. When a new interference is added to the graph, it marks one endpoint of a range. When an interference is removed, it marks the other endpoint. The beginning and end of a basic block may also mark endpoints of ranges. (See Figure 6.)

In this example, there are three basic blocks (instructions 1-2, 3-4, and 5-8). The ranges for x are calculated as follows:

Instruction 8 is the end of a basic block. The initial interference set, i_set, is empty. Instruction 7 has a use of x. Since x is not already in i_set, it marks a high bound. Instruction 6 is another use of x, but x is already in i_set. Instruction 5 is the top of a basic block, so it marks a low bound for any resources in i_set (in this case, x).

1	def x	i_set=	*low bound* x
2	goto 5	i_set=x	*high bound* x
3	inst	i_set=	
4	goto 8	i_set=	
5	inst	i_set=x	*low bound* x
6	use x	i_set=x	
7	use x	i_set=x	*high bound* x
8	inst	i_set=	

Figure 6.

Instruction 4 is the end of a basic block, and the initial i_set is empty. Instruction 3 is the top of a basic block, and i_set is still empty.

Instruction 2 is the end of a basic block, and the initial i_set contains x, so it marks a high bound. Instruction 1 is a definition of x, so it marks a low bound.

In addition to tracking register ranges, we need to track memory ranges for user variables that have had any of their def-use chains promoted to registers. (Normal memory variables do not need to be tracked, since their locations will not change.) This tracking can be done in a similar fashion, by keeping track of definitions (stores) and uses (loads).

We cannot generate the register ranges as they are calculated, since register assignment will not occur until later. Register spill may also alter the location of the variable. We save the list of ranges calculated for each def-use chain until we know exactly what information to generate.

8.3.2. Copy elimination

Once the initial interference graph has been built, the register allocator performs copy elimination. It does this by merging def-use chains that are joined by a single copy. The resource ID for one of the chains is changed to that of the other chain.

This merging could effectively remove a user variable. The ranges for both chains must be saved so that they can all be generated when the merged chain is assigned a register. We need to associate each list of ranges with the appropriate user variable (see Figure 7).

When an actual register is finally assigned to resource r1, the ranges for x and y are generated. The location for both variables will be the same register.

Before	After
10 def r1	10 def r1
20 inst	20 inst
30 copy r1,r2	
40 inst	40 inst
50 use r2	50 use r1
r1: var x lines 20-30	r1: var x lines 20-30,
r2: var y lines 40-50	var y lines 40-50

Figure 7.

8.3.3. Register spill

Register spill occurs when the register allocator decides that keeping a value in a register is more costly than reloading it, or when the interference graph cannot be colored with the available actual registers. When a user variable is spilled, the ranges already calculated for it can still be used; the location is all that changes. If the register is spilled to memory, the location is the stack offset of the spill location. (This spill location is not the same as the original memory location.) The range can be built at this point, since we have all the necessary information.

In the case of "spilling" a constant, the constant is created before each use, rather than actually saving the value and loading it from memory. If the register containing the constant is a user variable, this can effectively eliminate the variable. In this situation, we generate constant ranges, again using the previously calculated range bounds. The location field in each range holds the value of the constant.

8.4. Recreating loop variables

The values of loop variables eliminated due to strength reduction and induction variable elaboration can be recreated by the debugger.

Induction variable temporaries are introduced when a multiplication is strength-reduced into successive additions. The location of the temporary, and the value or location of the multiplication factor are communicated to the debugger with an induction variable range. The debugger can recreate the value of the loop variable by dividing the value of the temporary by the multiplication factor.

DOC currently supports strength reduction of multiplications only, although the technique can be extended to other replacement functions.

8.5. Instruction scheduling

The scheduler component of the optimizer is responsible for rearranging instructions in order to minimize memory interlocks, and to take advantage of branch delay slots. Instruction scheduling can move instructions across statement boundaries, corrupting the mapping between source and object code. This can cause both code location and data value problems.

8.5.1. The scheduler code location solution

In our architecture, the instruction following a branch is executed before the branch completes. The instruction scheduler is responsible for filling these branch delay slots with useful instructions. The target instruction of the branch can often be moved or copied into the delay slot. The branch is then adjusted to go to the instruction following the original target.

Copying the target into the delay slot may introduce a code location problem. If the original target is the first instruction of a source statement, a breakpoint set there will not always be hit.

The solution is to move the statement label to the new target instruction. Thus, the original target is treated as if it had been moved, even though it actually still exists in the same relative position (see Figure 8).

Before sched.	After sched.	After DOC
stmt2		
tgt store x	stmt2 store x	store x
		stmt2
inst1	tgt inst1	tgt inst1
...	...	...
goto tgt	goto tgt	goto tgt
nop	store x	store x

Figure 8.

8.5.2. The scheduler data value solution

User variables are the only data resources that a user has access to in a source-level debugging environment. Therefore, the solution for the data value problem only needs to address the modification of user variables.

The scheduler may move instructions that update user variables. If such a move crosses a statement boundary, a debugging problem can occur. The value of the affected variable may not be what the user expects, if he asks to see it between the original position of the instruction and

the new position. In order to avoid giving misleading information, we must keep track of the ranges of source code affected by all such moves.

There are two types of updates that need to be tracked: ordinary store instructions, and pseudo-stores.

The first instruction generated for a source statement will be referred to as a *sentry* (statement entry). Sentry instructions can be recognized by a flag which is set by the code generator when debug is enabled.

Before scheduling is done, instructions are numbered in ascending order. The instruction numbers are used as the basis for detecting instruction movement caused by scheduling.

A *sentry table* is built during the pass over the instructions prior to scheduling. Each table entry contains a sentry instruction number and its corresponding line number.

After the scheduling pass is run, another pass over the instructions is made to check for early and late updates of user variables.

An *early update* occurs when a store or pseudo-store instruction is moved before the sentry of the statement which originally contained the instruction. A *late update* occurs when a store or pseudo-store instruction is placed after the sentry of the statement following that which originally contained the instruction.

When an early or late update is detected, a range record is built for that variable. The range record contains information about the location of the variable, and the range of source lines over which the variable will not have its expected value.

Within a debugging session, when a user asks for the value of a variable having a scheduler range, the debugger informs the user of the early or late variable update.

8.5.3. Aggregate resource updates

The scheduler moves instructions without regard for resource type, and thus could make transformations on aggregate (aliased memory) resources. Aggregates include array element references and pointer indirection. Debug entries are not created for source expressions that reference these types of resources.

In order to provide information to the user about these types of resources, a flag in the resource table entry indicates whether the resource is an aggregate. The debug information for the aggregate resource is associated with the base for the aggregate (the pointer or array). When an early or late update involves an aggregate resource, the type field of the range indicates this. If the debugger encounters such a range, the user is informed with one of the following messages:

A pointer dereference using *ref_var* was set early, or

A pointer dereference using *ref_var* was set late

One problem with this solution is that this message will be displayed for user requests of any element of the aggregate within the update range. In most cases, however, the user should be able to determine which element of the aggregate is affected by the early update due to the source context.

9. Range update

Since the optimizer is multi-pass, later passes may invalidate ranges created by earlier passes. For example, the register allocator may build a range indicating that a variable resides in a register for some range of instructions. The instruction scheduler may then decide to move the instruction that sets the register into the code for another statement. The register range would no longer be correct.

In order to solve this problem, we have prioritized the types of ranges. The priority of a range type is generally based on the order of execution of the optimizer pass that generates it. When a lower priority range overlaps with a higher priority range, the lower priority range is shortened to eliminate the overlap. In Figure 9, the instruction scheduler range has higher priority (since it is generated later), so the register range is shortened.

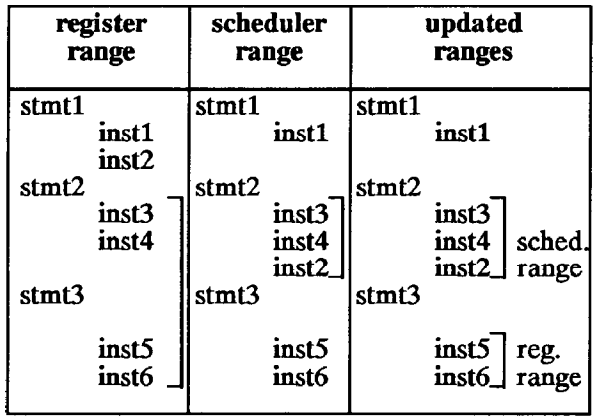

Figure 9.

10. Debugger modifications

The modifications to the debugger were minimal. When looking for the location of a variable, it is now necessary to check a flag to see if any ranges are present. If the flag is set, the list of ranges for that variable is traversed to see if the program counter lies within any of them.

If a low bound is seen that is greater than the program counter, or if the last range is encountered, the variable is assumed to be dead. If a valid range is found, the information is used to find the correct value or print a message.

11. Results

The range information generated by DOC averages 3.5% of the relocatable object file size. Optimization decreases file size more than the range information increases file size, so DOC object files average 2.6% smaller than unoptimized debug object files. Based on an analysis of all components of our optimizer, we expect comparable results when range information is generated for the full set of optimizations performed by our compiler.

In order to calculate the effect of DOC on compiler performance and memory usage, we calculated the effects for a normal compile, for debug alone and for optimization alone. The increases for debug and for optimization were added to the base effects of a normal compile, and this total used to compare to a DOC compile.

Compile time for DOC increased an average of 3%. This represents the additional time to calculate and process the range information.

Compiler memory usage for DOC increased an average of 0.4%.

Scanning the range information does not add significant overhead to debugger search time (the time it takes to find a variable and display its value). Measurements show that 91% of the variables with ranges have only 1 to 5 ranges. The debugger search time averaged less than 0.25 seconds real time to find and print the information contained in the 30th range of a local variable.

The extra overhead for processing globals does not have a significant impact since only 12% of variables with ranges are globals, and these average only two continuation ranges each. Debugger search time for a global with 10 continuation ranges averages less than 0.10 seconds real time.

12. Further work

Further work could be done to determine if there is some way to map machine instructions back to the source in a space efficient manner. This would provide a means to report the location of an exception consistent with the user's perception of the source.

DOC does not currently allow the user to modify the values of variables or the program control flow within the debugger. To do so would introduce a level of complexity similar to the problem of incremental optimization [22]. However, not all data and control flow modifications invalidate optimizer ranges. Those that do not could be allowed during a debug session if the debugger had access to additional information.

Improvements could be made in the ability to recreate the values of variables at debug time, in situations where we currently tell the user that the value is unavailable.

Assembly-level debuggers are used when debugging low-level system code. There might be an opportunity to apply DOC concepts to assembly-level debugging, especially in the area of register/memory tracking.

The process of ascertaining whether an object code patch is safe in globally optimized code is at best ad-hoc. A tool that extended some of our prototype's techniques might be of benefit.

13. Conclusions

- One additional debugger data structure will handle all the data value and code location problems our optimizer introduces.
- The values of eliminated variables can often be reconstructed.
- Only instructions that modify user data need to be tracked, and only when they are moved across statement boundaries.
- Only a small amount of extra information is necessary to provide adequate usability.

14. Acknowledgements

This work was made possible through the support of the management and staff of the Hewlett-Packard Computer Languages Laboratory. The following people provided valuable assistance in gaining an understanding of the compiler and debugger internals, and in discussing solutions for problems: Paul Chan, Richard Holman, Suneel Jain, Steve Lilker, Daryl Odnert, Karl Pettis, Vatsa

Santhanam, Carol Thompson, and Shankar Unni.

15. References

[1] Birnbaum, J., and Worley, W., "Beyond RISC: High Precision Architecture", *Hewlett-Packard Journal*, 36(8):4-10, August 1985.

[2] Zellweger, P., *Interactive Source-Level Debugging of Optimized Programs*, Xerox PARC Report CSL-84-5, [P84-00047], May 1984.

[3] Seidner, R., and Tindall, N., "Interactive Debug Requirements", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging*, March 1983, pp. 9-22.

[4] Lew, A., "On Interconnections Between Optimization and Debugging", *Proceedings of the Seventh Hawaii International Conference on Systems Sciences*, January 1974, pp. 164-166.

[5] Tischler, R., Schaufler, R., and Payne, C., "Static Analysis of Programs as an Aid to Debugging", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 1983.

[6] Ottenstein, K. and Ottenstein, L., "High-level Debugging Assistance Via Optimizing Compiler Technology", *ACM SIGPLAN Notices*, 18(8):152-154, August 1983.

[7] Personal conversation with Clark Oliphant, TLD Systems Ltd., Torrance, CA.

[8] Zellweger, P., "An Interactive High-Level Debugger for Control-Flow Optimized Programs", *ACM SIGPLAN Notices*, 18(8):159-171, Aug. 1983.

[9] Teeple, D., and Anderson, J., *The Debugging of Optimized Code*, Unpublished draft from MacDonald, Dettwiler and Associates Ltd., Richmond B.C., Canada, March 1980.

[10] Conway, R., and Wilcox, T., "Design and Implementation of a Diagnostic Compiler for PL/I", *Communications of the ACM*, 16(3):169-179, March, 1973.

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

[12] Warren, H., and Schlaeppi, H., "Design of the FDS Interactive Debugging System", *IBM Research Report RC7214*, (IBM Yorktown Heights), July 1978.

[13] Coutant, D., "Retargetable High-Level Alias Analysis", *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, January 1986.

[14] Coutant, D., Hammond, C., and Kelley, J., "Compilers for the New Generation of Hewlett-Packard Computers", *Hewlett-Packard Journal*, 37(1):4-18, January 1986.

[15] Gibbons, P., and Muchnick, S., "Efficient Instruction Scheduling for a Pipelined Architecture", *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, June 1986, pp. 11-16.

[16] Johnson, M., and Miller, T., "Effectiveness of a Machine-Level, Global Optimizer", *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, June 1986, pp. 99-108.

[17] *HP Symbolic Debugger User's Guide*, Hewlett-Packard technical manual, June 1987.

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

[19] Chaitin, G., "Register Allocation and Spilling via Graph Coloring", *Proceedings of the SIGPLAN Symposium on Compiler Construction*, June 1982, pp. 98-105.

[20] Sharir, M., "Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers", *Computer Languages*, 5, Pergamon Press Ltd., 1980.

[21] Jain, S., and Thompson, C., "An Efficient Approach to Data Flow Analysis in a Multiple Pass Global Optimizer", *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, June, 1988.

[22] Pollock, L., and Soffa, M., "INCROMINT - An Incremental Optimizer for Machine Independent Transformations", *Softfair II - A Second Conference on Software Development Tools, Techniques, and Alternatives*, IEEE Computer Society Press, Washington, D.C., 1985, pp. 162-171.