DEBUGGING OF OPTIMIZED ADA CODE

Peter Dencker

Alsys GmbH & Co. KG
Am Rüppurrer Schloß 7
D-7500 Karlsruhe 51
Germany +49 721 883025

Introduction. Modern RISC architectures call for highly optimizing compilers which fully exploit the specific features of RISC architectures - a great number of registers and deeply nested instruction pipelines.

It sounds like an anachronism, but all programming environments still recommend turning off the optimization in the compiler in order to debug a source program in source level terms. This is comparable to the recommendation to throw away the life belt when really going to sea after training on land with the belt.

Optimizations make life more difficult for the debugger in two respects. On one hand they modify the program's control flow and on the other hand they vary the addressing path to objects of the program. Thus one object may exist in different registers throughout its lifetime.

Ada is the first widespread high level programming language [1] which precisely defines the effect optimization may have or may not have on the compilation process. This removes the odor of hacking from the use of optimization ("You can turn it off if something goes wrong"). L. Weber, for example, still describes optimization in this sense [2]. Nowadays, the default for some powerful Ada Compilers is such that even during the development phase programs are compiled with the optimization turned on. This increases the demand for debuggers of optimized Ada code.

Up to now there are few publications on "Debugging of optimized code". Polle Zellweger [3] describes the problems encountered with inline procedure expansion and merging of identical tails of code paths that join for debugging, and the solutions found in a programming environment for the programming language Cedar. The solutions described there, however, do not consider the separate compilation of parts of the program which is an important feature of Ada.

John Hennessy [4] examines the problem of correctness of object values. It is true that optimizations are to sustain the functional equivalence between optimized and non-optimized code; they may, however, alter the structure and the interim results of a program. The consequence may be that in some parts of the optimized program code objects have wrong values with respect to the original logic of the program. For selected local and global code reordering, optimization solutions to the incorrect value problem are given [4]. However, there is no information on an implementation of these solutions.

In this paper we first present the ideal debugger features necessary for our reflections. Then we describe the effects of selected optimization procedures upon these ideal features and the resulting problems. They are accompanied by examples showing parts of sessions with the Alsys Ada Debugger for MIPS [5, 10]. Finally we discuss the solutions which we have implemented in the Alsys Ada System for MIPS.

Ideal Debugger Model. Ideally a source-level Ada Debugger simulates an Ada Machine on which a program can be executed step by step, declaration by declaration[1] and statement by statement. The Debugger simulates the Ada Machine by controlled execution of the object code of the program on a real machine. Therefore it is essential for the simulation that the Debugger can map the object code of a program to its source code. Code optimizations can render the mapping more difficult or even impossible. In the following, those essential properties of the ideal model are enumerated which are related to code optimization techniques. The next chapter discusses how code optimization techniques affect these properties and which different properties are desirable in view of the code optimization techniques.

[1] In Ada, declarations are executed like statements.

```
debug ("beispiel_1"); -- start the program beispiel_1
                      -- under control of the debugger
-->PROGRAM .beispiel_1 STARTED AT
-->>* #MAIN@1       beispiel_1'BODY.(5,4)
-->    1: PROCEDURE beispiel_1 IS
-->    2:    TYPE ar IS ARRAY (1 .. 10, 1 .. 10) OF float;
-->    3:    a : ar;
-->    4: BEGIN
-->    5>    FOR i IN a'RANGE(1) LOOP
-->----------^
-->    6:       FOR j IN 1 .. 11 LOOP
-->    7:          a (i, j) := 3.14;
-->    8:       END LOOP;
-->    9:    END LOOP;
-->   10: END beispiel_1;
go; -- continues the execution of the program.
-->>>> Program abandoned due to
-->>>> unhandled exception .constraint_error - raised at:
-->>* #MAIN@1       beispiel_1'BODY.(5,4).(6,7).(7,16)
-->    7>          a (i, j) := 3.14;
-->----------------------^
```

Example 1: The above log of a Debugger session demonstrates the accuracy of source locations with raised exceptions. Here it is the loop variable j which violates the index constraints of the array a.
The lines starting with "-->" contain the Debugger output. All other lines contain the input commands to the Debugger. The caret "^" in a "...---^" line points to the exact source location in the line above, where the program was stopped. Number tupels like (5,4) or (6,7) designate source positions in the form (line,column) with respect to a compilation unit. A source location is uniquely identified by a source position prefixed with a list of designators for all enclosing declarative regions. The head of the list designates the specification or the body of a library unit. The elements of the list are separated by points. Anonymous declarative regions, like for-loops, are designated by their respective start position.

1. The Debugger may stop the program before any declaration and any statement. These source locations are called *breakpoints*. The Debugger stops the program at a breakpoint only if the breakpoint has been set previously.
2. The Debugger may stop the program at any source location where an exception may be raised (e.g. division by zero). These source locations are called *triggerpoints*.
3. At each breakpoint or triggerpoint the complete, actual program state may be observed with respect to its source level logic. I.e., the correct values of all actual, valid objects are observable.
4. At each breakpoint or triggerpoint the values of all actual, valid, variable objects may independently be modified.
5. At each breakpoint or triggerpoint the modification of a value of an actual, valid, variable object has the expected effect with respect to source level logic.

The properties (2) and (3) require that the Debugger may not only map the object code onto source statements and declarations but also onto source expressions (see example 1). This useful property is still only found in a few Debuggers for high level programming languages.

Effect of selected code optimization techniques on Debugger properties. The selected code optimization techniques discussed in the following are described in the standard compiler construction literature, e.g. [6]. Their specific implementation for the Alsys Ada System has been published in [7]. The selection has been driven by the code optimization techniques used in the Alsys Ada System.

Elimination of inaccessible code. The optimization technique for the elimination of inaccessible code affects property (1) of the ideal Debugger model described in the preceding chapter. No code is generated for statements or declarations which are program logically inaccessible, so the Debugger cannot stop the program at such places. It may be surprising for a user not to be able to place a breakpoint into an unused subprogram whose object code has been eliminated completely. A desired reaction of the Debugger is to inform the user of this optimization (see example 2).

```
debug("beispiel_2");
-->PROGRAM .beispiel_2 STARTED AT
-->>* #MAIN@1       beispiel_2'BODY.(13,4)
-->     1: PROCEDURE beispiel_2 IS
-->     2:    i : integer;
-->     3:    PROCEDURE unbenutzt IS
-->     4:    BEGIN
-->     5:       i := i + 1;
-->     6:    END;
-->     7:    PROCEDURE benutzt IS
-->     8:    BEGIN
-->     9:       i := i + 2;
-->    10:    EXCEPTION WHEN OTHERS => i := 0; benutzt;
-->    11:    END;
-->    12: BEGIN
-->    13>    i := 902;
-->----------^
-->    14:    benutzt;
-->    15: END beispiel_2;
set_break("9"); -- Places breakpoint on statement in line 9.
-->BREAK  6 defined
set_break("2"); -- No object code exists for the declaration in line 2.
-->>>> "2" does not point at a break location. Next possible breakpoint
-->>>> location is: ".beispiel_2'BODY.(13,4)".
set_break("5"); -- The object code for line 5 is inaccessible.
-->>>> Error using path "5": The given source location is not
-->>>> accessible in the current program.
```

Example 2: Shows the different reactions of the Debugger when setting breakpoints.

Elimination of superfluous code. The optimization technique for the elimination of superfluous code affects properties (1) and (3). On property (1) it has the same effect as the technique for the elimination of inaccessible code. However, because the user knows that the control flow passes through the source location whose object code is superfluous, it is desirable for the Debugger to indicate a source location "nearby" where the program may be stopped (see example 2). The reaction of many existing Debuggers is unsatisfactory in that the breakpoint is moved implicitly to the next possible source location where the program may be stopped, without notifying the user. When single stepping, it is desirable that the Debugger stops the program only at declarations or statements for which object code has been generated. This assumes that the user has a general understanding of the effective program optimizations.

The evaluation code for unused variable values is superfluous too. If such code is eliminated then property (3) is lost. A desirable reaction of the Debugger in such a situation is to tell the user that the variable currently has no value (see example 3).

Treatment of common subexpressions. The code optimization technique for common subexpressions [8] affects properties (2) and (5). This technique causes a common subexpression to be calculated only once and causes the code for this calculation to be moved to a suitable location. This optimization destroys the uniqueness of the mapping of object code to source code: All source locations representing the common subexpression refer to the same object code. Thus, property (2) is lost. If an exception is raised during the calculation of this common subexpression, the Debugger can at best display the list of all source locations which represent the common subexpression. Additionally, the code of the common subexpression may be moved by the optimization to a place which is not related to the common subexpression at all. Thus, the user may get the impression that the exception is raised "too early" (see example 4). To prevent this situation, the Debugger should refer to this kind of optimization and additionally should display the breakpoints between which the code of the common subexpression has been moved, if it reports the exception.

If the code of a common subexpression gets moved across a breakpoint H towards the program start then

```
debug("beispiel_3");
-->PROGRAM .beispiel_3 STARTED AT
-->>* #MAIN@1       beispiel_3'BODY.(2,4)
-->    1: PROCEDURE beispiel_3 IS
-->    2>    v1 : integer := 33;
-->----------^
-->    3:    PROCEDURE pp IS
-->    4:       x : integer := v1 - 111;
-->    5:       y : integer := 4;
-->    6:    BEGIN
-->    7:       y := 3;
-->    8:       x := 1;
-->    9:       IF v1 > 0 THEN
-->   10:          x := 2;
-->   11:       ELSE
-->   12:          x := 4;
-->   13:       END IF;
-->   14:       v1 := x;
-->   15:    END;
-->   16: BEGIN
-->   17:    pp;
-->   18: END beispiel_3;
set_break("9");
-->BREAK  6 defined
go;
-->BREAK           6 >* #MAIN@1       pp'3'BODY.(9,7)
-->    9>       IF v1 > 0 THEN
-->-------------^
image("x"); -- Reads the value of x at the current execution location.
-->>>> Error using path "x": The object is dead here
-->>>> and has currently no value.
```

Example 8: Access to a variable x which has no correct value at the current execution location (line 9).

property (5) is lost. The modification of the value of a variable from a common subexpression has no effect on the optimized calculation of the common subexpression, even if it is done at the breakpoint H which precedes the calculation of the common subexpression in source level terms. Property (5) is in direct contradiction to code optimization technique for common subexpressions. No weaker property is known which guarantees the effect logically expected at source level when modifying the value of a variable with the Debugger.

Either we dispense with the optimization of common subexpressions or we accept the uncertainty that the modification of a value of a variable with the Debugger may not show the expected effect. Finding solutions to this problem is still an open research topic.

Register allocation. The code optimization technique of register allocation for objects [9] affects properties (3) and (4). With this technique many program objects may be kept in registers during their life time. The association of program objects to registers is not unique: An object may live in different registers at different times. The association becomes unique only with respect to a certain program location. Conversely, a register may be associated to different objects at the same time (see example 5). This contradicts property (4). If the value of such an object is modified then the values of all other objects sharing the same register get modified at the same time. A desirable reaction of the Debugger is to refuse the modification of one of these objects with a corresponding comment, but to allow an explicit modification of the involved register (see example 6).

Inlined subprograms. Inlining of subprograms affects properties (1) and (3). Property (1) is affected insofar as inlining destroys the unique mapping of a breakpoint to a single object code address. In order to keep property (1) the Debugger must be able to associate a breakpoint to a corresponding object code address in each incarnation (at a call site) of the inlined subprogram.

```
... Program cse1 was started and then stopped at line 5:
-->    1: PROCEDURE cse1 IS
-->    2:    a, c, x : integer := 0;
-->    3:    PROCEDURE p (a,c : IN OUT integer; b : boolean) IS
-->    4:    BEGIN
-->    5>       a := a * a;
-->-------------^
-->    6:       WHILE b LOOP
-->    7:          IF a = c THEN
-->    8:             x := x + 2;
-->    9:             c := 1/a + a*a;
-->   10:             c := c - 1;
-->   11:          ELSE
-->   12:             x := x - 2;
-->   13:             c := 1/a + a*a;
-->   14:             c := c + 1;
-->   15:          END IF;
-->   16:       END LOOP;
-->   17:       c := a * c;
-->   18:    END p;
-->   19: BEGIN
-->   20:    p (a,c,true);
-->   21: END cse1;
step; -- Executes one statement and stops again.
-->BREAK           8 STEPPED THROUGH to
-->    6>       WHILE b LOOP
-->-------------^
step;
-->>>> Program abandoned due to
-->>>> unhandled exception .constraint_error - raised at:
-->>>* #MAIN@1       p'3'BODY.(6,7).(9,19)
-->>>> Location is part of a common subexpression:
-->>>> previous break location: .cse1'BODY.p'3'BODY.(6,7)
-->>>> next break location: .cse1'BODY.p'3'BODY.(6,7).(7,10)
-->    9>             c := 1/a + a*a;
-->------------------------^
```

Example 4: Shows the reaction of the Debugger if an exception is raised "too early". Instead of reaching the if-statement after the last `step` the exception `constraint_error` is raised. The expression in line 9 is shown as the reason for the exception. Additionally the user is informed that this expression is a common subexpression which is calculated between the breakpoints in line 6 and 7. However, the other source of the common subexpression in line 13 is not shown.

Property (3) is affected insofar as inlining may cause different access paths to the parameters and local objects at each incarnation of an inlined subprogram. An incarnation of an inlined subprogram does not need its own activation record. Thus, in order to reconstruct the dynamic calling sequence of subprograms the code of each incarnation of an inlined subprogram must be mappable to the source location of the subprogram call. In order to access the parameters and local objects of an incarnation of an inlined subprogram, the Debugger needs a mapping of these objects to their access paths.

Particularly in view of the automatic inline optimization of compilers, it is highly desirable that the Debugger user interface treats inlined subprogram calls like normal calls. The user should not need to bother whether a given subprogram is inlined or not.

Debugger tables for optimized Ada code. In the Alsys Ada System for MIPS five kinds of Debugger tables exist:

1. Breakpoint table
   It contains a mapping of breakpoints onto relative object module addresses.
2. Source location table
   It contains a mapping of relative object module addresses onto source locations.

3. Register table
   It contains a mapping of Ada objects and source locations onto registers.
4. Inline symbol table
   For each incarnation of an inlined subprogram it contains access path descriptions for the parameters and local objects of the subprogram.
5. Linker table
   It contains a mapping of object modules onto absolute addresses.

The four first tables contain information associated to compilation units. Consequently they are kept in the Ada program library [1,§10.4] together with all other compilation unit specific data. In contrast the linker table refers to the Ada program as a whole. It is stored together with the executable object code of the program.

To address Ada objects not kept in registers, the Alsys Ada Debugger uses the same information from the Ada program library as the compiler. No additional Debugger tables are necessary for this information, so unnecessary information duplication is avoided. However, the inline symbol table information needs to be stored permanently only on behalf of the Debugger. The Compiler needs this table only temporarily for the currently compiled unit.

The information kept in the breakpoint table and the linker table corresponds to what is offered by most of the programming environments for high level programming languages. If an exception occurs, the Debuggers of these environments assume that the object code between two breakpoints is contiguous. Thus, using interval analysis, they may associate the object code address where the exception was raised with a breakpoint. With this method, neither is the accuracy of the shown source location of a raised exception as in example 1 achievable, nor is support provided for the user if an exception is raised within a common subexpression. In the latter case, taking example 4, the Debugger would show the line of the `while`-loop, where no `constraint_error` could be raised according to source level logic!

The accuracy of the source location shown and the support for the user if an exception is raised within a common subexpression is achievable with the newly developed source location table. This table makes no assumption on the order of the object code between breakpoints. Together with the linker table, it returns a source location accurate up to the Ada expression level for each object code address. Additionally, for each instruction a note is made showing, whether it is part of a common subexpression. As source location of an instruction which is part of a common subexpression, one of the source locations which represent the common subexpression is chosen arbitrarily. As shown by example 4, only this one source location is reported to the user. The costs of saving all of the associated source locations to be able to show them to the user are unjustifiable and stands in no relation to the benefit for the user.

The register table is a newly developed Debugger table as well. Besides the mapping of Ada objects and source locations onto registers this table contains an entry showing for each object, whether the object ever lives outside a register.

Measurements with the Alsys Ada System for MIPS have shown that about half of all objects in the register table never live outside of registers. Thus, these objects have a precise life-time description. These objects either have a current value in a known register or they are dead at the given source location (see example 3). For the other half of these objects the validity of the object values is certain only as long as the values are in a register. If the user requests the value of such an object at an execution location where the value is not in a register, then (s)he gets the value in the memory location of the object together with a warning that the value may be invalid due to code optimization (see example 7).

The uncertainty originates in the fact that at some source locations only the dynamic control flow can tell the validity of an object's value. In principle a further refinement of the life-time information for objects is possible; however, so far it was found to be too costly to implement.

Besides the display and modification of values of objects held in registers, the register table supports the disassembler of the Debugger in the association of register operands with Ada objects in instructions. Additionally, the disassembler tries to associate address operands with source locations. Where it succeeds, the operands are displayed twice separated by a "=" character (see example 5). To the left the operands are shown in Ada form, to the right in assembler form. The information in the register table is precise enough for the disassembler to distinguish the association of a register with two different objects within a single instruction (see example 5, instruction at address 100018C).

This form of machine code listing helps the user to map optimized object code to its source code. It supports the method of code-verification often applied to safety-critical applications.

Treatment of inlined subprograms. The treatment of inlined subprograms by the Debugger is a cost

```
debug("tool905");
-->PROGRAM .tool905 STARTED AT
-->>* #MAIN@1       pa'SPEC.(3,7)
-->    1: PROCEDURE tool905 IS
-->    2:    PACKAGE pa IS
-->    3>       v1 : integer := 3;
-->-------------^
-->    4:       v2, v3 : integer;
-->    5:    END pa;
-->    6:    USE pa;
-->    7:    v4 : integer := 4;
-->    8:    PROCEDURE put (i : integer) IS BEGIN pa.v1 := i; END;
-->    9:    PROCEDURE pp IS
-->   10:       v : integer := pa.v1 - 111;
-->   11:    BEGIN
-->   12:       v := v + 1;
-->   13:       put (v);
-->   14:    END;
-->   15: BEGIN
-->   16:    pp;
-->   17:    v1 := v4 - v1;
-->   18:    v2 := v4 - v1;
-->   19:    v3 := v4 - v1;
-->   20:    v4 := v1 + v2 + v3 + (v4 - v1);
-->   21:    pp;
-->   22: END tool905;
show_machine_code(address("17")); -- disassembles from line 17
-->.tool905'BODY.(17,4):
-->10000184         0 sub         v1,v4,v1 = $14,$15,$14
-->10000188         4 sw          v1,-64($30) = $14,-64($30)
-->(18,4):
-->1000018C         0 sub         v2,v4,v1 = $15,$15,$14
-->10000190         4 sw          v2,-68($30) = $15,-68($30)
-->(19,4):
-->10000194         0 sw          {v2,v3},-72($30) = $15,-72($30)
-->(20,4):
-->10000198         0 add         $14,v1,{v2,v3} = $14,$14,$15
-->1000019C         4 add         $13,$14,{v2,v3} = $13,$14,$15
-->100001A0         8 add         v4,$13,{v2,v3} = $15,$13,$15
-->100001A4        12 sw          v4,-76($30) = $15,-76($30)
-->(21,4):
-->100001A8         0 jal         ENTRY .tool905'BODY.pp'9 = 100001C4
```

Example 5: The Ada objects v2 and v3 are assigned to the same register 15 in the code for source lines 19 and 20. In the disassembled code this is shown by {v2,v3}. The disassembler of the Debugger tries to associate the operands of each instruction with Ada objects respectively source locations. Where it succeeds, the operands are displayed twice separated by a "=" character. To the left the operands are shown in Ada form, to the right in assembler form.

```
set_break("tool905.20"); -- Sets breakpoint in the same program
                         -- as in example 5.
-->BREAK  6 defined
go;
-->BREAK           6 >* #MAIN@1       tool905'BODY.(20,4)
-->   20>    v4 := v1 + v2 + v3 + (v4 - v1);
-->----------^
assign("pa.v2",905);
-->>>> Assignment to object denoted by "pa.v2" rejected. The object is
-->>>> currently held in a register shared by the following objects:
-->>>> .tool905'BODY.pa'SPEC.v2
-->>>> .tool905'BODY.pa'SPEC.v3
put_line(string(register_of("pa.v2"))); -- Shows the register of v2.
-->R15
image("pa.v2");
-->       -107
assign(r15,"905"); -- Assignment to register is allowed.
image("pa.v2");
-->        905
image("pa.v3");
-->        905
image(r15,typ=>"integer"); -- Interprets register contents as of
                           -- the given type.
                           -- By default they are displayed as
                           -- address values with base 16.
-->        905
```

Example 6: Modification of an Ada object v2 from the program of example 5, which is associated to the multiple shared register 15.

problem which is far from been solved by most Debuggers. It causes a considerable additional effort within the compiler generated Debugger tables and within the Debugger implementation. Because of lack of space this topic is only briefly discussed here.

In order to be able to access the parameters and local objects of an inlined subprogram, an additional symbol table for each incarnation of the inlined subprogram is kept in the Debugger tables because these objects may be addressed differently in each incarnation depending on optimization.

In the Debugger tables, the different incarnations of an inlined subprogram must be distinguishable with respect to source locations in order to be able to reconstruct the dynamic calling sequence of subprograms. If the user wishes to set a breakpoint into an inlined subprogram then the Debugger implementation must take care that a breakpoint is placed at all incarnations of the inlined subprogram. This is only possible if all calls of a subprogram are enumerable by the Debugger. However, the implementation of this enumeration within the Debugger tables and the Debugger was given lower priority and postponed. In view of the automatic inlining optimization it will be reconsidered.

So far the distinguishability of different incarnations of inlined subprograms has been implemented in the Alsys Ada System [10]. Consequently, the user must explicitly set a breakpoint into each incarnation of an inlined subprogram in which (s)he wants to stop the program. The distinguishability is implemented in such a way that each source location in the breakpoint table and the source location table is represented by a list of source positions whose prefix contains the source positions of all enclosing inlined subprogram calls, and the last element of the list contains a source position within the innermost inlined subprogram.

Cost of Debugger tables. By applying proper compression techniques, the space for the three Debugger tables breakpoint table, source location table, and register table was reduced to about 6-8% of the total space required for a single compilation unit in the Ada program library. The inline symbol table for a compilation unit make up for another 3-4%. The share of the linker table with about 1% of the object code size of a program is almost negligible.

The compilation of the PIWG benchmarks [11] into an empty Ada program library showed the following percentage results with respect to the total Ada program library size:

```
show; -- Shows the source at the current execution location.
-->   15: BEGIN
-->   16:    pp;
-->   17:    v1 := v4 - v1;
-->   18:    v2 := v4 - v1;
-->   19:    v3 := v4 - v1;
-->   20>    v4 := v1 + v2 + v3 + (v4 - v1);
-->----------^
-->   21:    pp;
-->   22: END tool905;
put_line(string(access_to("pa.v1"))); -- Returns the accessibility
     -- of the object pa.v1 at the current execution location.
-->IN_REGISTER
image("pa.v1"); -- Returns (the image of) the value of pa.v1.
-->        111
step; -- Executes one statement and stops again.
-->BREAK          14 STEPPED THROUGH to
-->   21>    pp;
-->----------^
put_line(string(access_to("pa.v1")));
-->IN_MEMORY_OR_NO_VALUE
image("pa.v1");
-->>>> The value "        111" of object "pa.v1"
-->>>> may be invalid due to code optimization.
```

Example 7: Shows the reaction of the Debugger on a user request for the value of object pa.v1 at a source location (Line 20) where the value of the object lies in a register and at a source location (Line 21) where the value of the object does not lie in a register.

Breakpoint tables	2.7 %
Source location tables	3.5 %
Register tables	1.5 %
Inline symbol tables	3.4 %
Sum	11.1 %

The self-compilation of the Alsys Ada compiler into an empty Ada program library showed the following percentage results with respect to the total Ada program library size:

Breakpoint tables	3.2 %
Source location tables	1.8 %
Register tables	1.4 %
Inline symbol tables	-
Partial sum	6.4 %

The cost of the inline symbol tables for the compiler was not available but is expected to be in the same range of 3-4% as with the PIWG results.

Conclusion. Debugging of optimized Ada code is possible. This has been demonstrated with the development of the Alsys Ada System for MIPS. However, the effects of optimization may not and should not be hidden from the user. Trying to keep up the ideal Debugger model would mean hiding the effects of optimization.

The goal of the debugging of optimized Ada code is to make transparent the effects of optimization. The described qualities of the Alsys Ada System Debugger contribute to achieve this goal.

The memory requirement for the Debugger tables is low. For a program the size of the Alsys Ada System compiler this is in the range of 11% of the total information that is stored in an Ada program library.

The problem of keeping up property (5) of the ideal Debugger model while at the same time handling common subexpressions in the best possible way still remains unsolved.

Acknowledgements. Jakob Schauer contributed significantly to the design and implementation of the register table. He made the register allocator generate the contents of the register table. Peter Kleiner contributed much to the design and implementation of the Debugger tables and their exploitation within the Debugger. I personally want to thank our development team for their good collaboration on the Debugger issues.

Short Biography. Peter Dencker received his diploma (masters) in Computer Science from the University of Karlsruhe in 1978. Diploma thesis: A portable LALR(1) Parser Generating System (PGS). Research assistent of Computer Science at University of Karlsruhe: Language design and compiler construction projects for the progamming languages LIS and Modula-2. Joined SYSTEAM KG in 1984: Responsible for the design and implementation of a retargetable debugger for the SYSTEAM Ada System. PhD thesis (February 1985) on "Generative attributierte Grammatiken" an extension of attributed grammars to allow for the generation of structure trees in the course of attribution (better known as "Higher Order Attribute Grammars"). Between 1986 and 1989 SYSTEAM's team leader of two European Community funded projects DESCARTES and Ada-IDAS. As of September 1990 Product and Marketing Manager of Alsys GmbH & Co. KG (former SYSTEAM KG).

References.

[1] The Programming Language Ada Reference Manual,
American National Standards Institute, Inc.
ANSI/MIL-STD-1815A-1983,
Springer Lecture Notes in Computer Science 155, 1983

[2] Larry Weber
Conversion factors, Systems International, June 1990, pp. 48-50

[3] Polle T. Zellweger
An Interactive High-Level Debugger for Control-Flow Optimized Programs, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, Sigplan Notices Vol. 18, No. 8, August 1983

[4] John Hennessy
Symbolic Debugging of Optimized Code, Transactions on Programming Languages and Systems, Vol.4, No.3, July 1982, pp. 323-344

[5] Gerry Kane
MIPS R2000 RISC Architecture, Prentice Hall, Englewood Cliffs, 1987

[6] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullmann
Compilers - Principles, Techniques, and Tools, Addison-Wesley Publishing Company, 1986

[7] Rolf Holzapfel, Jakob Schauer, Manfred Dausmann
Alsys Ada System, AIM Optimizer Overview, Alsys Document No. 24/88, 1988

[8] E. Morel, C. Renvoise
Global Optimization by Suppressing of Partial Redundancies, Communications of the ACM, Vol. 22, No. 2, February 1979

[9] James R. Larus, Paul N. Hilfinger
Register Allocation in the SPUR Lisp Compiler, Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, Sigplan Notices Vol. 21, No. 7, July 1986

[10] Alsys Ada System, User Manual for MIPS R3000/RISC/os,
Alsys Document No. 5/84, 1990

[11] Ada Performance Issues, Ada LETTERS, Vol. 10, No. 3, Winter 1990