An Optimizer for Ada – Design, Experiences and Results

Birgit Schwarz, Walter Kirchgässner, Rudolf Landwehr

GMD Forschungsstelle an der Universität Karlsruhe
Haid-und-Neu-Str. 7, D-7500 Karlsruhe 1, Germany

Abstract

In this paper we describe the design of a global machine independent low level optimizer for the Karlsruhe Ada Compiler. We give a short overview on the optimizations and data structures used in the optimizer as well as some experiences with the optimizer. Detailed measurements are provided for a collection of benchmarks. The average improvement of code speed is 40%.

1. Introduction

This paper describes our experiences with the development of a global optimizer that was embedded in the Karlsruhe Ada Compiler [Pers83]. The optimizer is designed to be machine independent (and therefore portable). It is running on the tree structured low level intermediate language of the compiler and allows free and interactive combination of the implemented optimizations. This permits experiments with the choice and the ordering of the optimizations.

In the following we present the data structures and the data flow information used by the optimizer and the implemented optimization techniques. These are local and global common subexpression elimination, strength reduction, induction variable elimination, dead assignment elimination, transformations of expressions and of the flow graph, mapping of local variables to registers, propagation of constants, variables and expressions, and elimination of checks and dead paths. Our algorithms used for global common subexpression elimination and strength reduction are closely related to those described in [MoRe79] and [Chow83]. If a time-space-tradeoff exists, time optimizations are considered to be more important.

Then we describe how the optimizer is parameterized with target dependencies and which requirements the code generator has to fulfill.

In the last chapter we give some results concerning the effectiveness and the order of the applied optimizations. The advantages of the choice of a tree structured intermediate language are discussed.

The appendix contains a detailed example which demonstrates the effects of the optimization techniques described.

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

2. Structure of the Optimizer

The optimizer is a separate pass in the compiler. It is running on the low level intermediate language AIM [JaLa82], i.e. it transforms AIM to AIM. First the control flow graph (with basic blocks as nodes) is constructed. After that data flow analysis (expression numbering and object management, computation of local and global information) is done. The optimizations are independent of each other (in some cases data flow analysis has to be repeated). They can be interactively combined. At the end an AIM program is constructed from the control flow graph and passed to the code generator.

The optimizer is written in Ada. It consists of approximately 13_000 lines of code (without comments, test output and blank lines).

3. Data Structures

In this chapter we briefly describe the data structures used by the optimizer.

3.1 The Intermediate Language

AIM [JaLa82] is a low level but target machine independent intermediate language. An AIM program is a sequence of routines. Each routine is a sequence of basic blocks which are sequences of statements. A statement is represented by an attributed expression tree. The top level operators of each tree represent the only operations with side effects on the values of objects. Control flow is expressed with branches and labels.

The operations available in AIM are similar to operations commonly found on target machines, e.g. integer and float arithmetic. Address paths are visible, i.e. all calculations for selection and indexing are described by expression trees. Checks are integrated in the expressions that access a checked value. This means that there exists a check operator with one operand which is the value to be checked, an attribute which indicates the relation and a second operand which is the bound for the check. The check operator returns the first operand as result or raises an exception.

3.2 The Control Flow Graph

The control flow graph is a partition of an AIM program into basic blocks. Basic blocks contain only one label (at the start) and one transfer statement (at the end). As local and global data flow information is computed blockwise the results of these computations are stored in the control flow graph.

3.3 Management of Expressions and Objects

Aliasing is one of the main problems in optimizing compilers. In AIM we have the concept of areas. An area is a part of the storage. Two different areas are always disjoint. An area can be a scalar variable, the heap for one pointer type or a composite object. All AIM operations that access the storage have an attribute indicating the areas that are concerned.

The area attributes are computed by the transformation phase (transforming the source oriented high level intermediate language DIANA [GoWu83] to AIM) of the compiler. This phase also computes an attribute that indicates whether the whole area is destroyed by an operation.

This is correct as long as no aliasing introduced by parameters passed by reference occurs. Although not exactly defined in [Ada83], the intended interpretation of the Ada reference manual seems to be that alias analysis need not be done because Ada programs are erroneous if the values of variables are accessed after they have been altered by using an alias name of this variable.

Expressions are numbered, i.e. each AIM tree is given a number to be able to detect multiple occurrences of the same expression and to handle sets of expressions. Two AIM trees are given the same number if their evaluation leads to the same result in case no area on which the expressions depend is changed between the two evaluations. Currently we assert this by giving the same number to AIM trees that are syntactically identical or that are syntactically identical after exchanging the operands of a commutative operator. Two trees are syntactically identical if their sons have the same number, if the root operators are identical and if all attributes that are relevant for the value of the tree have the same value. The expression number is stored as an attribute of the numbered tree.

There is one module that does the numbering and the management of expressions and areas. It computes all the information concerning the dependencies between expressions and areas, e.g. which expressions depend on an area, which expressions are subexpressions of an expression, which areas are used by an expression, etc. It also determines whether an exception can be raised by an expression.

4. Data Flow Analysis

In this chapter we describe the information computed by data flow analysis. Data flow analysis is done in three levels: local data flow analysis, global data flow analysis and interprocedural data flow analysis. Problems introduced by tasking and by the exception handling concept of Ada are also handled on the level of data flow analysis.

4.1 Local Data Flow Analysis

Local data flow analysis computes the summary information for one basic block. This is afterwards used to compute the global data flow information.

For areas we compute the KILL and the PALIVE information. KILL indicates whether there is an assignment in the basic block that destroys the whole area. An area is PALIVE (potentially alive) if its current value may be accessed by a content operation in the basic block or in an exception handler which might be reached by an exception raised in the block.

For expressions we compute TRANSP, LAVAIL, LANTICIP and LASSERT. TRANSP(b,e) is true (b is transparent for e) if in the basic block b there is no operation that alters the value of the expression e. LAVAIL(b,e) is true (e is locally available in b) if e is computed in b and if after the last computation of e in b there is no operation that alters

the value of e. LANTICIP(b,e) is true (e can be locally anticipated in b) if e is computed in b and if before the first computation of e in b there is no operation that alters the value of e. LASSERT(b,e) is true (e can be locally asserted in b) if the predicate represented by the expression e can be evaluated to true in b and if until the end of b there is no operation which invalidates e. These predicates are checks and equivalences introduced by assignments (e.g. v=a for v:=a where a does not contain v). In addition LASSERT contains predicates which can be easily derived from an expression (e.g. i>=0 from i:=1) and which an earlier pass has determined to be of interest.

4.2 Global Data Flow Analysis

With the help of the local data flow information we can compute the global information AVAIL, ANTICIP, PAVAIL, ASSERT, and PALIVE. AVAIL, ANTICIP and PAVAIL are used for global common subexpression elimination, code motion and strength reduction, ASSERT is used for propagation of constants, variables and expressions, and for elimination of checks and dead paths, and PALIVE (and KILL) are used for dead assignment elimination and induction variable elimination.

AVAIL(p,e) says that at point p in the program the expression e is available, i.e. e is computed on each path from the entry to p and between the last computations and p the value of e does not change.

ANTICIP(p,e) says that at point p in the program the expression e can be anticipated, i.e. e is computed on each path from p to an exit and between the first computations and p the value of e does not change.

PAVAIL(p,e) says that at point p in the program the expression e is partially available, i.e. e is computed on at least one path from the entry to p and between the last computation and p the value of e does not change.

ASSERT(p,e) says that at point p in the program the predicate represented by the expression e can be asserted on each path from the entry to p.

PALIVE(p,a) says that at point p in the program the area a is potentially alive, i.e. there exists a path from p to an operation using the contents of a and between p and this operation a is not killed. The PALIVE information can also be used to inform the programmer of variables that might be used before initialization. Only the potentially alive local areas at the entry of the procedure need to be determined.

These predicates are only computed for the entry and the exit points of basic blocks by solving a system of recursive equations.

4.3 Interprocedural Data Flow Analysis

The optimizer allows the choice of three different ways of handling the procedure calls:

The first is to make worst case assumptions, i.e. to assume that the call of a procedure alters all expressions, uses all areas and does not kill an area.

The second approach is to use visibility information. A called procedure can only modify memory locations that are visible to it or that may be visible for other routines it can call. An important consequence of the visibility rules is that the call of a global procedure cannot modify local variables of the calling procedure except by parameter passing.

The third possibility (currently the default) is to compute a summary information that indicates which areas are accessed by a content operation and to which areas a new value might be assigned. This is done in a prepass by computing the direct information of the procedures of the actual module. For procedures in other modules or written in a different language the visibility information is used. Then with the help of a call graph for the procedures of the actual module the transitive closure is computed. This information could still be improved if the summary information for procedures of other compilation units were kept in a data base (not implemented).

4.4 Problems with Exception Handling and Tasking

As Ada comprises exception handling we have to take care of expressions raising an exception. One way to handle this problem would be to insert an edge from the point where the exception can be raised to the corresponding exception handlers (when an exception is raised, control is transferred to the corresponding exception handler which is determined by going back along the dynamic calling chain and searching the first Ada block or procedure which has an exception handler). This insertion of edges would introduce many additional basic blocks and make data flow analysis very expensive. Therefore we decided to use another solution.

The first problem was that we had to assert that an assignment to an area which is read in an exception handler could not be eliminated. This is done by computing summary information which objects are alive in the exception handlers and by adding these areas to the alive set of all the expressions which might raise an exception.

The second problem was to prevent that expressions which can raise an exception are moved out of (or into) an Ada block with an exception handler (this could happen during code motion and propagation). This has to be reflected by the data flow predicates: At the begin of an Ada block with

an exception handler a new basic block starts and at the end of an Ada block with an exception handler this basic block ends. Then the ANTICIP information has to be modified in such a way that an expression cannot be anticipated at the exit of a basic block if one of the following basic blocks belongs to another Ada block. AVAIL and ASSERT are modified in such a way that an expression is not available (cannot be asserted) at the entry of a basic block if one of the predecessor blocks belongs to another Ada block. The same mechanism is used to prevent movements beyond synchronization points of tasking constructs.

Our last problem was that the call of a procedure might raise a user-defined exception. In this case expressions which raise a predefined exception must not be moved beyond this call. This problem can be solved by assuming that a call destroys all expressions which might raise a predefined exception. Better results can be achieved if the called procedures are analyzed during a prepass to determine whether they (or a transitively called procedure) raise a user-defined exception. Only calls of such procedures have to be considered as destroying expressions that might raise predefined exceptions.

5. Optimizations

In this chapter we give a short description of the techniques used for optimization. All optimizations rely on the concept that for each expression there exists a corresponding temporary and that temporaries are mapped to registers by the code generator. Whether the computation of an expression is expensive enough to be kept in a register is decided by cost functions (modelling the target dependencies).

5.1 Propagation

Propagation is based on the ASSERT information. ASSERT is very closely related to the AVAIL information. It comprises the AVAIL information for assignments and checks. Additionally some predicates are derived (e.g. p/=NULL at the point of an assignment p:=NEW...). For each successor of a transfer statement ASSERT contains the condition that is valid for this path.

The ASSERT information is used to propagate constants, variables and expressions (if an assignment v:=expr or an equality predicate v=expr are available at a point where v is used, v can be replaced by expr). Checks that can be asserted are eliminated. Dead paths are recognized and eliminated if the negation of the path condition can be asserted. For more details see [Kirc87].

5.2 Dead Assignment Elimination

Assignments are dead if the variable on the left hand side is not used until the end of the program or until the next operation that assigns a new value to the whole variable, i.e. that kills the area. The PALIVE information contains the information whether a path starting at the end of a basic block contains a use of an area before it is killed. Such by initializing a local PALIVE set with this information and by always updating it when an area is used or killed, we can recognize and eliminate dead assignments during a backward traversal of the basic block.

5.3 Global Common Subexpression Elimination and Code Motion

Moving invariant code out of loops and eliminating global common subexpressions are considered as elimination of partial redundancies. A computation of an expression is partially redundant if it is partially available at this point (this is true for loop invariant expressions and for the second computation in the case of common subexpressions). Partially redundant computations can be removed if the expression is made available by introducing computations of the expression on paths leading to the partially redundant computation and not yet containing a computation. This is done by an improved version of the algorithm of Morel and Renvoise [MoRe79] (for more details, especially for a systematic derivation of the data flow equations, see [Schr88]).

They introduced a system of data flow equations which computes a set of insertion points and a set of deletion points. The algorithm guarantees that on each path over an expression that is inserted there is an expression that will be deleted, that between an insertion and a deletion point there is no further insertion and that all deletions are safe, i.e. the expression is computed before the deletion point. So this optimization is correct and does not introduce additional computations at runtime.

In the context of a tree structured language, insertion of an expression means inserting an assignment of the expression to a temporary, and deletion of an expression means replacing the expression by the use of the corresponding temporary.

The main advantage of this algorithm is that code motion and global common subexpression elimination are handled in a uniform way without the necessity of computing the loop structure of the program. This advantage is also inherited by the algorithm for strength reduction (see 5.4).

5.4 *Induction Variable Optimizations*

Strength reduction is considered as a special case of code motion and therefore handled with an algorithm based on the method introduced in chapter 5.3 (a similar approach can be found in [Chow83]). To be able to move candidates for strength reduction (for example i*4, a multiplication of an induction variable with a constant) before a loop, the TRANSP information is modified in such a way that for these candidates induction assignments (e.g. i:=i+1) are ignored. By this trick the candidates become loop invariant and are replaced by uses of the corresponding temporaries, and an assignment of the candidate to the corresponding temporary is introduced before the loop. After the algorithm of Morel and Renvoise the "errors" introduced by ignoring the induction progressions have to be corrected, i.e. at each point where the induction variable is altered the corresponding progression of the temporary has to be inserted. At the points where the expression has been replaced by the temporary the checks which were part of the expression (e.g. 1<=i<=10) have to be reintroduced on the statement level (but without using the result).

After strength reduction it is often possible to get rid of the induction variable if this is used only for initialization, in checks on statement level or in conditions for the loop exit. This is done by first replacing tests for the value of the induction variable by the corresponding tests for the value of the temporary (e.g. i<=10 by t<=40) and by propagating the initial value of the induction variable into the initialization of the temporary. Then dead assignment elimination is done with a modified data flow information for PALIVE (reading of induction variables in induction assignments is ignored). So if the only use of the induction variable was the induction assignment, this assignment is eliminated.

As a summary, strength reduction and induction variable optimizations are handled as special cases of redundancy elimination and dead assignment elimination. This is done by modifying local properties of basic blocks, so no information about the loop structure is required.

5.5 *Local Optimizations*

Local optimizations are optimizations that can be done by considering one basic block at a time.

Local common subexpression elimination is done by two passes over the program. The first pass is a backward pass which for each AIM tree computes an attribute USECOUNT. USECOUNT indicates how often the corresponding expression is still used until the end of the basic block or until the value of one of the operands changes. In parallel this pass can do local dead assignment elimination. The second pass is a forward pass where an expression that is used more than once before the value of one of its operands changes (or before the end of the basic block) is replaced by the use of a temporary. Before the first occurrences of the temporaries an assignment of the expression to the temporary is inserted.

5.6 *Expression Transformations*

Arithmetic and boolean laws are used to do constant folding and to simplify the expressions. Conditional gotos are transformed to unconditional gotos if there exists only one successor or if the value of the condition is known at compile time. Another effect of expression transformations is the normalization of expressions. Tree pattern matching is used to find expressions that can be transformed.

5.7 *Graph Transformations*

The optimizer simplifies the flow graph whenever possible. For example, basic blocks with only one predecessor are fused with this predecessor if it has only one successor. Empty basic blocks (containing only a single goto statement) are eliminated by connecting all the predecessors with the successor. If conditional gotos are transformed to unconditional gotos by the expression transformation, some basic blocks may become unreachable and can be eliminated. The advantage of these optimizations is that the number of basic blocks and jumps is reduced.

5.8 *Mapping of Local Variables to Registers*

Local scalar variables are mapped to registers by replacing all assignments to the variable by assignments to a temporary and all content operations on the variable by uses of this temporary (the code generator will then map the temporaries to registers). The local variables that are mapped to temporaries are chosen by a cost function.

6. Handling of Target Dependencies

Although being performed on a target independent level the effect of the optimizations is target dependent. The availability of resources or address modes can influence the success of an optimization to a high degree. Therefore the optimizing algorithms are parameterized by cost functions and the code generator has to fulfill some requirements.

6.1 *Cost Functions*

As the number of available registers and the costs of computing an expression are highly machine dependent, the optimizer makes use of two cost functions. The first determines whether the computation of an expression is expensive enough to put the expression into a register. This is done by inspecting the expression tree (for example constants and simple address computations are not kept in registers). The second cost function chooses the local variables which are mapped to registers. There is an upper limit on the number of local variables that can be put into registers. Induction variables have the highest priority, because it is assumed that they occur in loops and are therefore very often used. The other variables are chosen arbitrarily if they fulfill the constraint that they need not be saved too often during procedure calls.

6.2 *Requirements on the Code Generator*

The code generator has to map temporaries to registers since this is assumed by the optimizer and by the cost functions. Because the optimizer is doing global optimizations, the code generator has to do global register allocation. Otherwise spill code would be introduced. In our compiler we use a code generator that was generated by a code generator generator [Jans85]. It does global register allocation by graph colouring.

The success of the optimizations depends to a high degree on the ability of the code generator to avoid spill code (and indirectly on the cost functions). Otherwise the program can be slower after optimization because spill code has been introduced.

7. Experiences and Results

In this chapter we want to present our experiences with a tree structured intermediate language, some results on the order of the optimizations and some measurements.

7.1 *Experiences with the Tree Structure*

The tree structure has shown to be very useful because some optimizations, for instance common subexpression elimination and code motion can be done for arbitrarily complex expressions. The same applies to expression transformations because one tree contains the whole pattern to be considered.

The integration of checks in the expression was advantageous for code motion because it guaranteed that the check is moved together with any expression accessing the checked value. On the other hand, strength reduction is more complicated with integrated checks.

7.2 *Order of Optimizations*

There are some constraints on the order of the optimizations:

Expression propagation should only be done before common subexpression elimination because it makes the program slower (variables are replaced by more expensive expressions). Nevertheless, expression propagation should be done because it has a normalizing effect on the program, i.e. after expression propagation there are more opportunities for common subexpression elimination than before.

After propagation, it is useful to do structural transformations (in fact expression transformations are done together with propagation in one pass over the program) because propagation provides more opportunities for constant folding. There are also more possibilities for jump elimination because the information on the values of conditions is better after propagation. For new trees expression transformations should be done because they have a normalizing effect and make the trees smaller.

Another constraint is that it is dangerous to do code motion after strength reduction because then some checks may be separate from the use of the checked expression (see 5.4), and therefore there is no more guarantee that the use of the checked expression will not be moved before the check.

A problem is the ordering of check elimination and common subexpression elimination (together with strength reduction and induction variable elimination):

If check elimination is done first, opportunities for common subexpression elimination may be destroyed. This happens if checks are parts of common subexpressions but only the check of the second computation is eliminated. In this case the second computation will have an expression number different from the number of the first computation and is therefore not recognized as redundant.

On the other hand, the replacement of an induction variable in checks by the temporary of a candidate (after strength reduction) introduces difficulties for the elimination of checks. There are more opportunities for check elimination in loops if the checked induction variable is altered by an increment of 1 instead of an increment of a constant c, $c>1$ (c.f. [Kirc87]).

So check elimination is currently done after strength reduction but before the replacement of the induction variable in checks and conditions.

7.3 Measurements

The figures we give here are for a set of benchmarks originally collected by John Hennessy and then used to test the optimizers of Chow [Chow83] and Powell [Powe84]. It contains a program that computes permutations with recursion (perm), a recursive program solving the Towers of Hanoi problem (towers), a program recursively solving the Eight Queens problem (queens), a program computing the product of two integer matrices (intmm) and a program to compute the product of two real number matrices (mm), a program that solves a puzzle about packing blocks into a cube (partially recursive) (puzzle), a program that performs Quick Sort (quick), a program for Bubble Sort (bubble), a program that performs the recursive Insertion Sort in a binary tree and checks the correctness of the insertions (tree), and a program performing Fast Fourier Transformation (fft).

The execution times for the whole benchmark (Total) and for the different subprograms have been measured with several options (on a SIEMENS S7550). Checks are always generated (but the compiler uses type information to avoid the generation of checks).

The table below consists of two parts: results for single optimizations and results for combinations of all optimizations. For procedure calls, we use worst case assumptions or visibility or summary information as indicated. The execution time for the non-optimized code has been chosen as the reference point (1.0).

In the first part, the results for the optimized code for some selected optimizations are given. These are expression transformation, graph transformation, constant propagation together with check elimination and expression transformation (const prop), global dead assignment elimination (gdae), local dead assignment elimination (ldae), local common subexpression elimination (lcse), mapping of local variables to registers (lova), global common subexpression elimination and code motion (gcse,cm), strength reduction together with global common subexpression elimination and code motion (gcse,cm,sr), strength reduction together with global common subexpression, elimination code motion and induction variable elimination (gcse,cm,sr,ive), and finally local optimization and mapping of local variables to registers ("cheap").

In the second part, the results for three different arrangements of all optimizations are given (case A, B, and C). For case C, which usually gives the best results, figures are given for different levels of information for procedure calls.

Execution Times (normalized)												
		perm	towers	queens	intmm	mm	puzzle	quick	bubble	tree	fft	Total	Aver
not opt		1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
expr transformation		.96	1.0	.86	.99	1.0	.78	.83	.93	1.0	.99	.90	.94
graph transformation		1.0	1.0	1.0	.99	1.0	1.01	1.00	1.01	1.01	1.0	1.0	1.0
const prop	summary	.92	.98	.80	.85	.88	.63	.82	.66	.90	.77	.77	.82
gdae	summary	.96	.94	.99	1.0	.99	.99	.98	1.0	.98	1.0	.99	.98
lcse	visibility	.92	.78	.95	.89	.91	.98	.89	.83	.97	.59	.89	.87
ldae	visibility	1.0	1.0	.98	.99	1.01	1.02	1.0	1.0	.99	.99	1.0	1.0
lova	summary	.93	.98	.92	.94	.95	.89	.86	.94	1.0	.93	.93	.93
gcse,cm	summary	.89	.93	1.03	.85	.89	.74	.92	.69	.86	.87	.82	.87
gcse,cm,sr	summary	.86	.93	.84	.49	.57	.69	.83	.63	.89	.57	.71	.73
gcse,cm,sr,ive	summary	.81	.87	.83	.50	.59	.66	.89	.59	.83	.58	.69	.72
cheap opt	visibility	.90	.77	.77	.90	.91	.68	.74	.70	.91	.56	.75	.78
full opt,A	summary	.68	.64	.74	.38	.47	.53	.60	.44	.79	.59	.56	.58
full opt,B	summary	.73	.64	.73	.39	.52	.52	.59	.46	.75	.59	.57	.59
full opt,C	worst	.92	.69	.79	.59	.64	.66	.67	.49	.81	.62	.67	.69
full opt,C	visibility	.85	.66	.76	.41	.50	.53	.60	.40	.79	.59	.58	.61
full opt,C	summary	.72	.63	.73	.38	.47	.50	.57	.41	.74	.59	.55	.57

In case C we are doing (in this order) expression transformation, mapping of local variables to temporaries, expression propagation (with dead path elimination and expression transformation), graph transformation, local common subexpression elimination, strength reduction (with global common subexpression elimination and code motion), variable propagation (with check elimination, dead path elimination and expression transformation), graph transformation, induction variable elimination, local dead assignment elimination and local common subexpression elimination, and finally global dead assignment elimination. Case A is identical to case C, except that only constants are propagated instead of expressions. In case B check elimination is done together with expression propagation at the beginning rather than after strength reduction (otherwise it is identical to case C).

For full optimization, the table shows a reduction of more than 40% of execution time using interprocedural summary information. Almost the same result can be achieved by considering only visibility information for procedure calls. So the question arises whether it really is worthwhile to invest much effort in the analysis of called procedures. Full optimization with worst case assumptions for procedure calls still yields about 30%.

Local optimizations and mapping of local variables to registers which are very cheap to implement and to perform already account for an improvement of 22% (table entry "cheap").

At the beginning of our experiments, the effect of global optimization was often spoiled because the code generator could not map all temporaries to registers. After some tuning of the cost functions, most of the spill code could be avoided. For the benchmark programs, in the case of full optimization spill code is generated only for the Fast Fourier Transformation (fft), and there mostly for floating point subexpressions, where we considered recomputing to be more expensive than spilling. In the case of single optimization spill code is generated for the Eight Queens problem (for global common subexpression elimination with code motion), for Puzzle (for global common subexpression elimination, for strength reduction and for strength reduction with induction variable elimination) and again for Fast Fourier Transformation (for local common subexpression elimination, for global common subexpression elimination, for strength reduction, for strength reduction with induction variable elimination and for local optimizations with mapping of local variables to registers).

Our optimizer does currently not perform inline expansion of subprograms. This technique provides more opportunities for most optimizations (but also for spill code) and avoids the overhead of procedure calls. Since especially this benchmark contains many calls of small procedures within loops, we did some inline expansion by hand on the Ada source. The result was about 7% improvement in the case of full optimization as well as for the not optimized code. So for this benchmark the expected effect that inline expansion provides more opportunities for optimizations did not hold.

Low level machine properties like caches or instruction pipelines are not considered by our optimizer, but they can influence the execution time more than most of the single optimizations. Properties of particular machine models also strongly influence the effect of optimizations. When we executed the benchmark programs on a SIEMENS 7590, the fastest of this series of machines, we got an average improvement of 38%, and the improvements were differently distributed over the single programs.

Although time optimization had precedence of space optimization the code size has been significantly reduced. The code size after full optimization (case C, with summary information) is 66% of the code size without optimization.

References

[Ada83] The Programming Language Ada, Reference Manual, ANSI/MIL-STD-1815A-1983, LNCS 155, Springer-Verlag Berlin 1983

[Chow83] F.C. Chow. A Portable Machine-Independent Global Optimizer - Design and Measurements, PhD Thesis, Dep. of Electrical Engineering and Computer Science, Stanford University, CA, Technical Report No. 83-254

[GoWu83] G. Goos, W. Wulf, A. Evans, K. Butler. DIANA - An Intermediate Language for Ada, LNCS 161, Springer-Verlag Berlin 1983

[Jans85] H.-St. Jansohn. Automated Generation of Optimized Code, PhD Thesis, GMD-Bericht Nr. 154, R. Oldenbourg Verlag München 1985

[JaLa82] H.-St. Jansohn, R. Landwehr. A Code Generator for Ada, Universität Karlsruhe, Fakultät für Informatik, Bericht 33/82

[Kirc87] W. Kirchgässner. Assertion Propagation - A New Approach to Program Optimization, internal paper

[MoRe79] E. Morel, C. Renvoise. Global Optimization by Suppression of Partial Redundancies, CACM 22(2) (February 1979), 96-103

[Pers83] G. Persch, J. Uhl, H.-St. Jansohn, W. Kirchgässner, R. Landwehr, M. Dausmann, S. Drossopoulou, G. Goos. Ada Compiler Karlsruhe - Overview, Proc. Ada Europe/AdaTEC Joint Conference, Brussels 1983

[Powe84] M.L. Powell. A Portable Optimizing Compiler for Modula-2, Proceedings SIGPLAN Conference on Compiler Construction, SIGPLAN Notices 19(6) (June 1984), 310-318

[Schr88] F.W. Schröer. Districts : A Foundation for the Suppression of Partial Redundancies, Arbeitspapiere der GMD, Nr. 304

Appendix: Example

The following example shows how the optimizer is step by step transforming the AIM program for the scalar product. For brevity AIM is represented in a condensed notation. ↑v denotes the contents of the variable v, and ⊕ stands for address addition. Checks are written as !, e.g. ↑v!<=100 means that the contents of v is checked against 100; if the relation is valid the result is ↑v, otherwise constraint_error is raised.

The Ada Program:

```
TYPE vector IS ARRAY (1..100) OF integer;
PROCEDURE scalar (a, b : IN vector; r : OUT integer) IS
   h : integer := 0;
BEGIN
   FOR i IN 1 .. 100 LOOP
      h := h + a(i) * b(i);
   END LOOP;
   r := h;
END;
```

Step 1: The original AIM program

```
<< 6 >> h:=0
        IF 1>100 THEN 2 ELSE 3
<< 3 >> i:=1
        GOTO 5
<< 5 >> h:=↑h +
          ↑(↑a⊕(((↑i!<=100)!>=1)-1)*4) *
          ↑(↑b⊕(((↑i!<=100)!>=1)-1)*4)
        IF ↑i>=100 THEN 2 ELSE 4
<< 4 >> i:=↑i+1
        GOTO 5
<< 2 >> r:=↑h
        GOTO 1
<< 1 >> RETURN
```

Step 2: Expression transformation, mapping of local variables to temporaries and graph transformation

```
<< 6 >> T26:=0
        T27:=1
        GOTO 5
<< 5 >> T26:=↑T26 +
          ↑(↑a⊕(((↑T27!<=100)!>=1)-1)*4) *
          ↑(↑b⊕(((↑T27!<=100)!>=1)-1)*4)
        IF ↑T27>=100 THEN 2 ELSE 4
<< 4 >> T27:=↑T27+1
        GOTO 5
<< 2 >> r:=↑T26
        RETURN
```

Step 3: Local common subexpression elimination

```
<< 6 >> T26:=0
        T27:=1
        GOTO 5
<< 5 >> T28:=(((↑T27!<=100)!>=1)-1)*4
        T26:=↑T26 + ↑(↑a⊕↑T28) * ↑(↑b⊕↑T28)
        IF ↑T27>=100 THEN 2 ELSE 4
<< 4 >> T27:=↑T27+1
        GOTO 5
<< 2 >> r:=↑T26
        RETURN
```

Step 4: Strength reduction (with global common subexpression elimination and code motion)

```
<< 6 >> T26:=0
        T27:=1
        T28:=(↑T27-1)*4
        T33:=↑b⊕↑T28
        T32:=↑a⊕↑T28
        GOTO 5
<< 5 >> void((↑T27!<=100)!>=1)
        void((↑T27!<=100)!>=1)
        T26:=↑T26 + ↑↑T32 * ↑↑T33
        IF ↑T27>=100 THEN 2 ELSE 4
<< 4 >> T28:=↑T28+4
        T32:=↑T32⊕4
        T33:=↑T33⊕4
        T27:=↑T27+1
        GOTO 5
<< 2 >> r:=↑T26
        RETURN
```

Step 5: Variable propagation (with check elimination, dead path elimination and expression transformation)

```
<< 6 >> T26:=0
        T27:=1
        T28:=0
        T33:=↑b
        T32:=↑a
        GOTO 5
<< 5 >> T26:=↑T26 + ↑↑T32 * ↑↑T33
        IF ↑T27>=100 THEN 2 ELSE 4
<< 4 >> T28:=↑T28+4
        T32:=↑T32⊕4
        T33:=↑T33⊕4
        T27:=↑T27+1
        GOTO 5
<< 2 >> r:=↑T26
        RETURN
```

Step 6: Induction variable elimination (with dead assignment elimination)

```
<< 6 >> T26:=0
        T28:=0
        T33:=↑b
        T32:=↑a
        GOTO 5
<< 5 >> T26:=↑T26 + ↑↑T32 * ↑↑T33
        IF ↑T28>=396 THEN 2 ELSE 4
<< 4 >> T28:=↑T28+4
        T32:=↑T32⊕4
        T33:=↑T33⊕4
        GOTO 5
<< 2 >> r:=↑T26
        RETURN
```