Front End Based Optimization: Premature or Inevitable?

Toon Moene
The GNU Fortran Team
toon@moene.indiv.nluug.nl

Abstract

The process of optimization in a compiler is finding a shorter and/or faster sequence of instructions that computes the same results as the "obvious" sequence. Traditionally, optimizations are performed after the source program has been fully converted to an intermediate language that is independent of the source language. This paper presents cases where source language knowledge is important for large gains in Fortran optimization. It also discusses further criteria for front-end optimizations.

No specific prior knowledge (outside general knowledge of the GNU Compiler Collection structure) is needed to understand this paper - all relevant Fortran (and Fortran front end) concepts will be explained.

1 Introduction

Classical compiler theory [1], Figure 1.2, assumes that a compiler will first transform a program from its *source language* (e.g., Fortran) to some *intermediate language* before code optimization takes place.

In this paper it is argued that - at least for Fortran, but probably also for other High Level Languages - it is useful to perform optimizations on the internal structures of the *Front End* (i.e., the part of the compiler that "knows" about the programming language the source is written in).

For GNU Fortran, the Front End consists of the files in the *gcc/fortran* directory.

2 The Tasks of the Front End

The Fortran Front End has to perform the following tasks:

1. Parse the characters of the source file into something manageable on the basis of the syntaxis of the language. The result is stored into structures held inside the Front End.
2. Gather the references to all the names used in the various scopes of the scanned source, and make them point to the right objects.
3. Convert array(-section) expressions to loops breaking them down to scalar expressions (a process known as *scalarization*).
4. Convert the resulting structures (front-end trees) into the intermediate language of the *middle end* - the language-independent part of the compiler.

Items 1 and 2 were provided by Andy Vaught, items 3 and 4 were written by Paul Brook with assistance from Steven Bosscher.

Optimizations by the Front End are useful after items 1 and 2 have been processed, because the former are largely of a "convert text to internal representation" type. It follows that Front End optimizations are useful when *scalarization* occurs (because then actual code transformation takes place) - this will be borne out in the next paragraphs.

3 Terminology and Notation

In Fortran

1. An *array* is a set of scalar data whose elements are arranged in a rectangular pattern.
2. The *rank* of an array is the number of dimensions.
3. The *extent* along a dimension is the number of elements along that dimension.
4. The *shape* of an array is an integer rank 1 array with as elements the extents of the dimensions.
5. The *size* of an array is the total number of elements of the array.

In array expressions and assignments all arrays have the same shape.

Furthermore, in assignments it is guaranteed that the right hand side is evaluated fully before the left hand side is affected.

Array assignments and expressions take the same form as scalar ones. E.g., the following assigns the sum of array B and C to A (each element of A will be the sum of the *corresponding* elements of B and C):

$A = B + C$

Parts of arrays are indicated by a "colon" notation (called an *array section*), as follows:

$A(first : last : step)$

indicates the elements of A from $first$ to $last$ taking every $step$'th value (all three of them can be omitted - they then revert to their default values: $first$ = lower bound, $last$ = upper bound of array, and $step$ = 1).

Compilers use *descriptors* to describe rank, shape and other relevant properties of arrays and array sections.

Arrays of rank 1 and 2 are the most common - they are the representation of vectors and matrices that are the subject of the branch of mathematics known as *Linear Algebra* (see for instance the book by the same name [3]).

Of course, the constituent mathematical entities in Linear Algebra are elements of a *field*, which the floating point numbers and their addition and multiplication operations are not.

However, Fortran considers "an approximation to the outcome of operation X" (where operation X is the mathematical "equivalent" of the floating point operation) good enough.

4 Existing Fortran Front End Optimizations

4.1 Reduction Operators on Array Expressions

Several array reduction operators are defined in the Fortran language:

1. SUM - determine the sum of the elements of the array given as argument.

2. PRODUCT - determine the product of the elements of the array given as argument.

3. ANY - determine the inclusive OR of the (logical) elements of the array given as argument.

4. ALL - determine the AND of the (logical) elements of the array given as argument.

5. MAXLOC - determine the location of the maximal element of the array given as argument.

6. MINLOC - determine the location of the minimal element of the array given as argument.

7. MAXVAL - determine the value of the maximal element of the array given as argument.

8. MINVAL - determine the value of the minimal element of the array given as argument.

A *reduction* operator is one that returns an array with rank smaller than the one of its input.

For instance, assuming that array $A(N,M)$ contains floating point values, the expression `SUM(A)` returns the sum of all elements of A, a single number out of $N*M$ numbers.

Interesting optimizations can be obtained by recognizing that a single number (or logical, in case of `ANY` and `ALL`) is obtained, regardless of the number of (logical, floating point) entities that went into the computation.

E.g., the following is regular Fortran idiom:

```
SUBROUTINE AAP(A, B, N, M)
REAL A(N, M), B(N, M)
IF ( ALL(A == B) ) THEN
 PRINT*,'AAP: A equals B!'
ELSE
 ... do some real work ...
ENDIF
END
```

The obvious (and expensive) code to generate would be to first evaluate $A == B$ in a temporary array of extents $N*M$, and then compute the total `AND` of the resulting $N*M$ logical array.

GNU Fortran evaluates the complete `ALL` expression at once, along the following lines (the '&'s as last and first character "glue" together two subsequent source lines):

```
...
RESULT = .TRUE.
DO I = 1, M
 DO J = 1, N
  RESULT = RESULT .AND. &
 &            A(J,I) == B(J,I)
 ENDDO
ENDDO
...
```

The reduction operators `ANY` and `ALL` were already treated this way originally.

4.2 Inlining of `DOT_PRODUCT`

`DOT_PRODUCT` is a reduction operator implementing `SUM(A*B)`, where A and B are rank 1 arrays.

Originally, the GNU Fortran run time library contained an implementation of this routine, but it was determined, after extensive testing, that expanding the computations comprising this intrinsic inline always gave better performance.

4.3 Special treatment of $A^T.B$

The matrix multiplication of the transpose of a matrix A with matrix B is a frequent operation. A naive implementation will need a temporary for A^T.

Matrix multiplication for arrays with rank <= 2 is done by the intrinsic `MATMUL(A,B)`. The transpose of a rank 2 matrix is formed by the intrinsic `TRANSPOSE(A)`.

The obvious way to evaluate the expression `MATMUL(TRANSPOSE(A),B)` indeed is to evaluate `TRANSPOSE(A)` in a temporary and pass that temporary and B as the two arguments to `MATMUL`.

However, given that it is necessary to pass array descriptors to `MATMUL` anyway, it is sufficient to pass an alternative descriptor, that describes A in a way that exchanges its first and second dimension.

This saves the allocation of a temporary of the size of A and the subsequent copying of A^T into it.

The reason to special-case this expression is that it is very common in physical problems, like quantum mechanics, optics and quantum chemistry.

4.4 Necessity of temporary storage in array assignment

Given the Fortran rules of assignment (the right-hand-side is always evaluated fully before the left-hand-side is affected) it is always safe to evaluate an array expression in freshly allocated temporary storage, and then, on assignment, copy the contents of that storage to the left-hand-side's array and deallocate the temporary storage afterwards.

However, very often it is possible to prove that the allocation, copying and deallocation of the temporary storage is not necessary, and the assignment can directly fill the left-hand-side array.

1. If the right hand side doesn't contain the array on the left hand side (e.g., $A = B * C$).
2. If the use of the left-hand-side array on the right hand side exactly overlaps (e.g., $A = A + B$).
3. If the scalarization loop can be set up so that overlap between left and right hand side is handled correctly (e.g., in $A(1 : n-1) = A(2 : n)$, start the loop at 1).

This analysis is now routinely done in the Front End.

4.5 Purification and Constification of intrinsic procedures

Quoting the GCC documentation, on the discussion of function attributes:

> Many functions have no effect except their return value *and* their return value depends only on the parameters and/or global variables. Such a function can be subject to common subexpression elimination (see below) and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute *pure*.

In fact,

> Many functions do not examine any values except their arguments, and have no effects except the return value. Basically this is just a slightly more strict class than the *pure* attribute above, since such a function is not allowed to read global memory. These function should be declared with the attribute *const*.

The *pure* attribute applies clearly to most of Fortran's simple intrinsic functions, like $sin(x)$, $atan2(x,y)$ and $log(x)$. When setting this attribute for these functions, further optimization by the compiler is possible. If the compiler directly encodes calls to the C math library routines, then they can get attribute *const*, as they only depend on the *value(s)* of their argument(s).

It is therefore beneficial to have the Front End set these attributes for those functions.

[Note that a Fortran program doesn't have access to *errno*, so the function result *is* the only effect.]

5 Optimization Assisted by the Fortran Front End

5.1 Use of Preinstalled BLAS libraries

BLAS stands for Basic Linear Algebra Subprograms, and is a library of well-known vector and matrix operations.

The `-fexternal-blas` compiler option will force the compiler to generate code to use the BLAS routines `[sdcz]gemm` in case of matrix multiplication by the intrinsic `MATMUL`. This is beneficial (for not too small matrices) if the installed external BLAS library is highly optimized for the system the program is compiled for.

The extent at which `-fexternal-blas` will be effective is detemined by the command line option `-fblas-matmul-limit=n` with n being 30 by default.

Fortran reference implementations of the BLAS routines can be found at `http://netlib.org/blas/index.html`.

6 Proposed Future Optimizations

6.1 All Fortran intrinsic procedures are *pure*, except `RANDOM_NUMBER`

Having the Front End set the *pure* attribute for them would already result in a number of optimizations. Note that GCC uses a different definition of "pure" than the Fortran (2003) Standard.

GCC's *pure* definition is most easily understood as follows:

Given a Fortran intrinsic function $f(x)$, $2 * f(x)$ can be substituted for $f(x) + f(x)$.

6.2 Inline `MATMUL` for small arrays

For small arrays, it is probably advantageous to expand `MATMUL` inline. For instance, when the inner loop has four iterations, it will be fully expanded by loop unrolling.

Given that it is profitable to inline `DOT_PRODUCT` completely, it can be assumed that for rank <= 1 (vector * vector, or scalar * vector), inlining would always be right.

6.3 MATMUL is called twice in $MATMUL(a,b)/(1+MATMUL(a,b))$

Just as we do not compute $sin(x)$ twice when evaluating $sin(x)/(1+sin(x))$, neither should we do so for $MATMUL(a,b)$.

This is the subject of *Problem Report* 22572.

The optimization that takes care of this unnecessary duplication is called *Common Subexpression Elimination*, also known as CSE.

Unfortunately, the Fortran Front End does evaluate $MATMUL(a,b)$ twice, and allocates temporary storage for the results twice.

Even if the *pure* attribute were set for MATMUL, still two temporary arrays would be allocated, instead of one.

Obviously, this analysis should be extended to all *pure* Fortran intrinsics.

6.4 Further reduction of temporary allocation on assignment

Allocation of a temporary to compute the right-hand-side of an assignment into can be avoided if the scalarization loops are build such that overlap between arrays in the left- and right-hand-side is harmless

1. E.g., in the following assignment: $A(2:n) = A(1:n-1)$ the temporary can be avoided by running the scalarization loop "backwards".

The code to allow this treatment (and others more general) is already present in the part of the compiler that translates array expressions - however, it is not activated and, hence, not debugged.

7 General Reflections on Front End Optimizations

All of the described optimizations can be classified as *local optimizations*. They can be performed without knowledge of the flow of control, because they all act on a single expression or assignment.

To get a good overview of classical optimizations on scalar expressions when using the information acquired by control and data flow analysis, one wants to look at [1], Chapters 12, 13 and 14.

Almost all of this theory carries over if one considers *arrays* to be the basic entity, instead of *scalars*, for these optimizations (except for induction variable strength reduction and elimination, which are typical optimizations *on* array indices and addressing).

Obviously, the optimizations described below have to take place before scalarization.

7.1 Basic Blocks

To quote [1], a *Basic Block* is:

> A sequence of consecutive statements only entered at the beginning and executed without halt or branch until the end [of the basic block].

Computing the basic blocks in a routine is the first step towards deriving the control flow graph.

Basic blocks can be quite large, if one disregards the underlying loops that will ultimately comprise the array operations.

An example will clarify this - this is the Newton-Raphson inverse finder for matrices as discussed in [2], Chapter 2.5, where $B0$ is a first guess of the inverse of A:

```
FUNCTION AINV(A, B0) RESULT (B)
INTEGER I
REAL A(:, :), B0(:, :)
! B has the size of A
! (and therefore of B0)
REAL B(SIZE(A,DIM=1),SIZE(A,DIM=2))
! This converges quite fast
B = B0
DO I = 1, 7
   B = 2 * B - &
  &    MATMUL(B, MATMUL(A, B))
ENDDO
END
```

There are only three basic blocks in this routine - the one before the loop (containing the copy of $B0$ to B, initializing the result variable with the first guess), the one after the loop (empty) and the loop body.

Large basic blocks are beneficial, because it will make control flow analysis in this part of the compiler fast.

7.2 Loop Invariant Compututations

Once basic blocks have been determined and the control flow graph derived, data flow information can be computed (see [1], Chapter 13).

When this information is available, *Loop Invariant Computations* can be determined.

E.g., the following is a naive implementation of Jacobi's method of solving a linear system $A.x = b$ (see [2], Chapter 19.5), which uses the fact that a matrix can always be split into $L + D + U$, with L the lower triangular, U the upper triangular and D the diagonal part (*DINV* is the inverse of D):

```
FUNCTION XR(L, DINV, U, B, X0) RESULT (X)
INTEGER I
REAL L(:,:), DINV(:,:), U(:,:)
REAL B(:), X0(:)
REAL X(SIZE(B))
X = X0      ! Suitable first guess...
DO I = 1, 7 ! Seven is probably too low.
   X = MATMUL( -MATMUL(DINV,L+U), X) + B
ENDDO
END
```

After this analysis, the expression `-MATMUL(DINV,L+U)` will be recognized as a loop invariant, and moved to be computed before the loop.

8 Conclusions

We have shown that several optimizations are available to the Fortran Front End that are impossible to perform later, because these optimizations involve the prevention of allocating temporary storage.

Setting function properties like *pure* or *const* on intrinsic functions is only possible in the Front End, because it knows the rules of the language.

In one case, the underlying knowledge of the description of arrays is used to prevent an unnecessary copy into a temporary.

9 Acknowledgements

As stated, the driving force behind writing the translation from the parser internal structures to the middle end's GENERIC language and thereby implementing the first Front End optimizations was Paul Brook, assisted by Steven Bosscher.

This paper discusses Front End optimizations developed by the following persons (in alphabetical order): Steven Bosscher, Paul Brook, François-Xavier Coudert, Andrew Pinski, Richard Sandiford, Roger Sayle and Paul Thomas.

Those that helped to improve this paper are: Paul Brook, Paul Thomas and Steven Bosscher.

The author is also indebted to the rest of the GNU Fortran Team, for providing Fortran using professionals with an outstanding compiler and run time library.

References

[1] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley Publishing Co., 1978.

[2] William H. Press et al. *Numerical Recipes in Fortran (second edition)*. Cambridge University Press, 1992.

[3] Serge Lang. *Linear Algebra*. Addison-Wesley Publishing Co., 1972.