A method for optimizing and transforming a compiler program in a computer system. The method comprises the steps of constructing a compiler comprising a program augmentation capability; and, locating this capability in association with phases of a standard compilation process. The program augmentation capability may comprise symbolic automatic differentiation, or generation of Taylor series, or generation of Hessian or Jacobian matrices.
COMPUTER-PROGRAM COMPILERS
COMPRISING A PROGRAM
AUGMENTATION CAPABILITY

This is a continuation of application(s) Ser. No. 08/327,086 filed on Oct. 21, 1994 now abandoned.

FIELD OF THE INVENTION

This invention relates to computer-program compilers comprising a program augmentation capability.

BACKGROUND OF THE INVENTION

Computer-program compilers comprise programs or hardware that can translate programs written in a source language into those written in a target language. For example, the source language may comprise a high level language such as Fortran, and the target language may also comprise a high level language e.g., a transformed Fortran, or alternatively, an assembly code or machine language.

SUMMARY OF THE INVENTION

Our work comprises combining and integrating two disparate concepts, as they relate to computer program compilers.

The first concept centers around compiler techniques comprising code optimization, which seek a transformation of the program with an aim of improving (optimizing) an efficiency or performance of a target program. For example, a goal of an optimizing compiler may be to generate a smaller or a faster set of object code that exactly duplicates a function of the program as it was written.

The second concept comprises directly providing a compiler with a program augmentation capability; e.g., an automatic symbolic differentiation capability (comprising a forward, reverse, or hybrid mode) which can augment the compiled program to include values for derivatives of the program’s function. This second concept also comprises other augmentations to the compiled program e.g., consistency verification under dimensional analysis.

We combine these two concepts in the following way.

First, we recognize that an efficient employment of symbolic derivatives may be enhanced by identifying expressions that have equal values and eliminating redundant calculations along any path in the target program. Hence, no mathematical expression along any path in a target program is evaluated more than once. This is an optimization of the target program and it may be achieved for any operation, or sequence of operations, that are valid in the language of the source program. These include, but are not limited to, feedback loops, conditional branching and GO TO jumps in the control flow, subroutine calls, MAX, MIN or ABS (Absolute) evaluations, and table look-up data. We refer to this optimization as redundant expression elimination. Furthermore, we recognize that not all intermediate derivatives are needed. Therefore, the program augmentation capability preferably does not generate them in the first place. We refer to this optimization as an employment of global dependency information.

Secondly, we observe that extant compilers do not directly comprise a symbolic differentiation capability. Instead, this function can be done by an automatic symbolic differentiation precompiling program.

Thirdly, we observe that extant automatic differentiation programs comprise a structure/sequence wherein automatic symbolic differentiation is done locally on a statement-by-

statement basis. That is to say, the structure/sequencing of an extant automatic differentiation program is such that it cannot avail itself of global dependency information or redundant expression elimination as it references automatic symbolic differentiation. Accordingly, from a vantage point of our invention, it is not possible to obtain an optimal or highly efficient code using extant automatic differentiation program structures.

We have now discovered a novel compiler structure/sequencing apparatus that is predicated on our employment of global dependency information and redundant expression elimination (in contrast to prior art local schema), which enables one to comprehend automatic symbolic differentiation as being inherently enhanced by its incorporation in proximity to, or within, a compiler’s code optimization, thereby generating highly efficient code.

In a first aspect, the present invention comprises a method for optimizing and transforming a program to be compiled in a computer system. The method comprises the steps of:

1) constructing a compiler comprising a program augmentation capability; and
2) locating this capability in association with phases of a standard compilation process.

In a second aspect, the present invention comprises a compiler apparatus for compiling a program to be executed on a general purpose target computer system. The compiler apparatus comprises:

1) a front end (FE) for initially processing an input program;
2) a symbol-information data structure (SIDS) in communication with the front end for recording information about symbols in an input program;
3) an intermediate language generator (ILG) in communication with the front end and the symbol-information data structure for producing intermediate language instruction;
4) an optimizer (OPT) in communication with the symbol-information data structure and the intermediate language generator;
5) a means for locating a program augmentation capability in operative association with the optimizer; and
6) a back end (BE) in communication with the optimizer and/or the intermediate language generator for translating a program into target code.

The present invention as defined can realize several significant advantages.

First of all, as alluded to above, the qualities and attributes of the highly efficient code presently generated, arise in part from the fact that in our employment of e.g., symbolic differentiation done in operative association with the compiler optimizer, we can immediately avoid redundant calculations in the target program. This situation, concomitantly, can advantageously reduce the time needed to perform a required calculation which, in turn, can save money and speed up developmental processes. Other advantages are enumerated below.

BRIEF DESCRIPTION OF THE DRAWING

The invention is illustrated in the accompanying drawing, in which:

FIG. 1 shows a block diagram of a representative extant computer compiler;
FIG. 2 shows a block diagram of a compiler apparatus of the present invention; and
FIGS. 3, 4 and 5 show additional alternative embodiments of the FIG. 2 compiler apparatus.
3

DETAILED DESCRIPTION OF THE INVENTION

We now reference the present invention by first setting forth a conceptual backdrop and insights into various aspects of the prior art. This approach, when set in apposition to a following detailed description of the present invention, can highlight novel aspects of the present invention.

Attention, accordingly, is now directed to FIG. 1 which shows a block diagram of a representative extant computer program compiler 10. In overview, the FIG. 1 compiler 10 accepts as an input (I) 12 a high-level language program and operates on it to an end of generating an output (O) 14 comprising an output target language program 16. In particular, the compiler 10 comprises a front end (FE) 18, a symbol-information data structure(s) (SIDS) 20 for recording information about symbols in an input program, an intermediate language generator (ILG) 22, an optimizer (OPT) 24, and a back end (BE) 26.

The FIG. 1 front end 18 typically converts the input program 12 to a (possibly) different internal form (IF) that may be conveyed (arrow 28) to the intermediate language generator 22. As part of the preparation of the internal form, the front end 18 may save information (arrow 30) in, and possibly retrieve information (arrow 32) from, the symbol-information data structure(s) 20. These symbol-information data structures, if they are used, may either be separate from or adjoining to the intermediate form.

Note that the intermediate language generator 22 produces intermediate language instructions (IL) from the internal form of the program, possibly consulting (arrow 34) the symbol-information data structure(s) 20. The intermediate language instructions are typically more similar to the output language (O) than to the input language (I). The intermediate language form of the program may be conveyed to the back end 26 either directly (arrow 36) or by way of the optimizer 24 (arrows 38 and 40). If the intermediate language (IL) form of the program is conveyed to the optimizer (OPT) 24, then the optimizer produces a functionally equivalent and preferably faster or smaller version of the program, typically again in the intermediate form. This version of the program may then be conveyed (arrow 40) to the back end 26. To this end, the optimizer 24 may be in communication (arrow 42) with the symbol-information data structure(s) 20.

Once an intermediate language form of the program is received by the back end 26, either directly (arrow 36) or after optimization (arrow 40), the back end 26 converts the program to a functionally equivalent version expressed in the output language.

It is explicitly noted that the output program may be in the same language as I, IF, or IL, even though it is typically in a form distinct from all of these.

Note finally (but most importantly with respect to the present invention), that the FIG. 1 input (I) 12 comprising the high-level program for operation thereupon by the compiler 10, is itself a component of an input block 44. The input block 44, in turn, comprises a subject program structure (SP) 46 sequenced to a program augmentation capability (PAC) 48, in turn, sequenced to the modified subject program structure (MSP) 50.

In net assessment of the FIG. 1 prior art compiler 10, we observe that a program augmentation capability 48 is outside of, and independent of, the compiler operation. Our invention may be sharply contrasted with an approach/sequence, as the present invention comprises a unique integration of program augmentation as a compiler technique incorporated in the code optimization, or in direct proximity thereto.

We now turn our attention to FIG. 2, which shows a block diagram of a preferred compiler apparatus 52 of the present invention.

An important advantage of the FIG. 2 compiler apparatus 52 is that it can optimally incorporate invariant conventional components of the FIG. 1 compiler, mutatis mutandis, thus securing great efficiencies of transformation and implementation, yet readily accommodating necessary changes reflective of the present invention. Accordingly, the following initial disclosure of the FIG. 2 structure and operation may be presented as a paraphrase of the FIG. 1 discussion, above.

In overview, the FIG. 2 compiler apparatus 52 can accept as an input (I) 54 a high-level language program (IF) 56, and can operate on it to an end of generating an output (O) 58 comprising an output target language program 60. In particular, the compiler apparatus 52 comprises a front end (FE) 62, a symbol-information data structure(s) (SIDS) 64 for recording information about symbols in an input program, an intermediate language generator (ILG) 66, an optimizer (OPT) 68, and a back end (BE) 70. These entities can all be realized by conventional components.

The FIG. 2 front end 62 preferably converts the input program 56 to a (possibly) different internal form (IF) 72 that may be conveyed to the intermediate language generator 66. As part of the preparation of the internal form, the front end 62 may save information (arrow 74) and possibly retrieve information (arrow 76) about the program and symbol information structure(s) 64. These symbol information structures, if they are used, may either be separate from, or adjoining to, the intermediate form.

Note that the intermediate language generator 66 can produce intermediate language instructions (IL) from the internal form of the program, possibly consulting the symbol information structure(s) (arrow 78).

The intermediate language instructions are typically more similar to the output language (O) 58 than the input language (I) 56. The intermediate language form of the program may be conveyed to the back end 70 either directly (arrow 80) or by way of the optimizer (arrows 82 and 84). If the intermediate language (IL) form of the program is conveyed to the optimizer 68, then the optimizer produces a functionally equivalent and preferably faster or smaller version of the program, typically again in the intermediate form. This version of the program may then be conveyed to the back end 70 or may be subject to some number of additional optimization passes (arrow 86). To this end, the optimizer 68 may be in communication (arrow 88) with the symbol-information data structure(s) 64.

Once an intermediate language form of the program is received by the back end 70, either directly (arrow 80) or after optimization (arrow 84), the back end 70 converts the program to a functionally equivalent version in the output language.

It is explicitly noted that the output program may be in the same language as I, IF, or IL, even though it is typically in a form that is distinct from all of these.

In sharp contrast to FIG. 1, however, the FIG. 2 compiler apparatus 52 comprises a critical and novel salient, namely an explicit inclusion of a program augmentation capability 90 located in association with phases of a standard compilation process, in particular, as a compiler technique incorporated in a code optimizer, or in direct (spatial, temporal) proximity thereto. This point is now elaborated.

First of all, it is noted that the program augmentation capability subserves e.g., differentiation (including symbolic
automatic differentiation), solution of ordinary differential equations by Taylor series in which the Taylor series can be automatically generated, or generation of Hessian matrices.

In and of themselves, program augmentation capabilities are known conventional techniques. See, for example, L. B. Rall, *Automatic Differentiation: Techniques, Applications, and Implementations*, in Lecture Notes in Computer Science, Vol. 120, Springer-Verlag, Berlin, 1981.

As just alluded to, the program augmentation capability of the present invention is located (temporarily, spatially) in association with phases of a standard compilation process. For example, and to articulate what we define as phases, the FIG. 2 embodiment locates this capability 90 subsequent to the intermediate language generator 66 and antecedent to the optimizer 68.

FIG. 3 has a variation of this concept. Here, a program augmentation capability 92 is located intra the optimizer 68.

FIG. 4 shows a further variation: here, a program augmentation capability 94 is located subsequent to the front end 62, (which preferably collects symbol information (arrow 76)), and antecedent to the intermediate language generator 66.

An important advantage of the present invention may now be readily discerned. FIG. 5 repeats the specifics of the FIG. 2 embodiment, but further comprises additional, enhanced optimization (indicated by a hashed-box optimizer 68) in which enhanced optimization may be specifically dedicated to directly handling the output of the program augmentation capability 90. For example, the optimizer 68 may be extended for handling differential dependencies when the program augmentation capability comprises symbolic automatic differentiation.

As an example of the present invention, input and target programs for a function Idf(Vgs, Vds, Vxx) appear in Appendices A and B, respectively. The target program in Appendix B was automatically generated from the input program in Appendix A by the compiler apparatus 62 described herein. Both the input and target programs were in Fortran; however, as stated above, the input and target programs could have been implemented in any computer language. Line numbers were added to Appendices A and B to identify input and results.

The automatically generated target program for Ids; i.e., the partial derivatives of Ids with respect to the independent variables Vgs, Vds and Vxx appearing in Appendix B, illustrates several features of the optimization described in this embodiment. These include a differential algebraic compiler that can operate upon an input program that contains conditional branching (line 7 in Appendix A) and subroutine function calls (line 8 in Appendix A), and the absence of redundant calculation in the automatically generated target program by the substitution of common subexpression with new variables (e.g., variables t7, t91, t13, t461, t661 in Appendix B). Subroutines in the input program may comprise Fortran code or tabulated data. For tabulated data, derivatives are obtained by functions that numerically approximate the tabulated data. It is noteworthy that partial derivative terms that are always equal to 0 are automatically removed and factors that are always equal to 1 are automatically removed.

Other embodiments of this invention include compilers designed to generate target programs for arrays of derivatives such as those found in Jacobian or Hessian matrices and power series expansions, as noted above.
APPENDIX B-continued

\[ i_{40} = 0.1D^3 e_{eff} \]
\[ m_{eff} = 500(1D^2 e_{40} e_{30}) \]
\[ i_{40} = 1/(1D^2 e_{30} e_{40}) \times 2 \]

\[ D_{m} e_{40} D_{e} g = -50D^4 * D_{m} e_{40} D_{e} g + i_{40} \]
\[ D_{m} e_{40} D_{e} x = -50D^4 * D_{m} e_{40} D_{e} x + i_{40} \]
\[ D_{m} e_{40} D_{e} v = -50D^4 * D_{m} e_{40} D_{e} v + i_{40} \]

5. A method according to claim 1, wherein said standard compilation process comprises a front end feeding into an intermediate language generator, the method comprising a step of locating said augmentation means subsequent to the front end and antecedent to the intermediate language generator.

6. A method according to claim 5, wherein said augmentation means comprises automatic differentiation.

7. A method according to claim 1, wherein said standard compilation process comprises an intermediate language generator feeding into an optimizer, comprising a step of locating said augmentation means subsequent to the intermediate language generator or and antecedent to the optimizer.

8. A method according to claim 7, wherein said augmentation means comprises automatic differentiation.

9. A method according to claim 9, wherein the standard compilation process comprises constructing a system comprising:

1) a front end (FE) for initially processing input program;
2) a symbol-information data structure (SIDS) in communication with the front end for recording information about symbols in an input program;
3) an intermediate language generator (ILG) in communication with the front end and the data structure for producing intermediate language instruction;
4) an optimizer (OPT) in communication with data structure and the intermediate language generator;
5) said augmentation means interposed with the optimizer; and
6) a back end (BE) in communication with the optimizer and the intermediate language generator for translating program into target code.

10. A method according to claim 9, further comprising providing an optimizer for handling an output of said augmentation means.

11. A method according to claim 10, wherein the optimizer is extended for handling differential dependencies when said augmentation means comprises symbolic automatic differentiation.

* * * * *