Proceedings of the ACM SIGMINI/SIGPLAN Interface Meeting on Programming Systems in the Small Processor Environment, New Orleans, LA, March 1976. ACM SIGPLAN Notices, vol. 11, no. 4, April, 1976.

OPTIMIZED MACHINE-INDEPENDENT EXTENDED FORTRAN FOR MINICOMPUTER PROGRAMMING

Gordon S. Novak Jr.
Tracor, Inc.,
and The University of Texas at Austin

ABSTRACT

A FORTRAN compiler is being developed to aid in writing systems and application programs for minicomputers. The compiler is machine-independent in the sense that it can be made to generate code for a new machine with relatively little effort. The FORTRAN language accepted by the compiler is extended to facilitate the writing of "systems" programs. Particular emphasis is placed on optimization of the generated code for efficiency of storage utilization.

INTRODUCTION

The advantages of programming in a higher-level language such as FORTRAN are well known. However, the use of FORTRAN for programming minicomputers often turns out to be impractical. The compilers supplied by manufacturers of minis typically accept only a poor and nonstandard FORTRAN dialect, produce inefficient code, and require a large run-time library. The few good compilers require a large machine configuration to run the compiler (and sometimes to run the object program). Programs must be tailored so closely to the particular mini that they become difficult to transport.

In order to overcome these problems, we are developing at Tracor a machine-independent FORTRAN compiler. The compiler itself is written in standard FORTRAN for a medium-to-large scale computer (initially a UNIVAC 1108). It will be machine-independent in the sense that it can be made to generate code for a new minicomputer with about one man-month of effort. The FORTRAN language accepted by the compiler is a good one, with a number of extensions beyond standard FORTRAN (X3.9,1966) to make it easier to write "systems" programs. (Most of these extensions are likely to be included in the revised FORTRAN standard when it is released.) The object code produced by the compiler will be highly optimized; in many cases, it will produce code which is better than that of assembly language programmers.

The compiler will produce assembly language code as output. This will allow the programmer to further optimize some sections of the code if necessary. A machine-independent assembler and an optimizing link-editor, also running on the large computer, will be available to produce absolute object code.

The syntax-scanning phase of the compiler has been completed, and work on the optimization phase is in progress. We hope to begin generating object code in mid 1976.

Cross-assemblers for minicomputers are fairly common. (Gray,1974) reports a project to produce code for minicomputers from the optimized intermediate-language output of the FORTRAN compiler of the Honeywell 635.

FORTRAN LANGUAGE EXTENSIONS

The FORTRAN language is a good one for systems programming because there is a clear and efficient mapping from the language constructs of FORTRAN into the instruction sets typically provided by present-day computers, including minis. Further, FORTRAN can be compiled directly into machine code, making very few assumptions about its operating environment. The language extensions we have made are designed to give the programmer some of the control and flexibility available with assembly language, and to avoid cumbersome constructions. Many of these language extensions were inspired by UNIVAC 1108 FORTRAN V.

The PARAMETER statement associates a constant value with a symbolic name, much like the EQU directive often available with assembly languages. This can be used to parameterize programs at no cost in object program efficiency. For example, one could write

```
PARAMETER LTABLE = 200, PI = 3.1415926
DIMENSION TABLE (LTABLE, 5) .
```

The DEFINE function is an extension of the FORTRAN arithmetic statement function; a reference to a DEFINE function will produce in-line code for the function at the point of reference. DEFINE functions may be used as a sort of extended EQUIVALENCE to equate simple variable names with partial-word quantities. For example,

```
DEFINE ITYPE = FLD (7,8,ITABLE(3))
...
ITYPE = 6
```

will define ITYPE to be the lower eight bits of the word ITABLE(3); the statement ITYPE = 6 will generate code to store a 6 in the lower eight bits of ITABLE(3), while leaving the rest of the word unchanged.

The use of ITYPE in the above example is an example of a SYMMETRIC function. A SYMMETRIC function is one which has a storage characteristic, and may appear on either side of the equal sign in an arithmetic statement. Examples of SYMMETRIC functions are the intrinsic functions FLD (partial-word field) and CHAR(i,s) (character i of the string s). A very large "array" could be implemented by a SYMMETRIC function which would handle the array as virtual storage under software control. References to the virtual storage function would be written in exactly the same form as references to an array.

Another language extension is the ability to specify run-time binding of variables. Binding normally occurs in FORTRAN when the dummy argument names of a subprogram are bound to the actual arguments specified in a reference to the subprogram. Run-time binding allows such binding to be done under program control. In systems programs, for example, it is frequently necessary to request space for a table entry from a storage management routine. The address of the table entry is kept in an index register, and parts of the table are referenced as EQUated symbols relative to the index register. The same thing can be accomplished in FORTRAN by using DIMENSION, EQUIVALENCE, and DEFINE functions to define the structure of the table entry, then using run-time binding to bind the table entry to the address gotten from the storage management routine. Binding and unbinding are accomplished by the statements

```
CALL BIND (a, b, n)
CALL UNBIND (a, b, n)
```

where "a" is the name of the array to be bound, "b" is the location it is to be bound to, and "n" is the length of the table. These forms provide compatibility with standard FORTRAN, where actual transfer of the arrays can be used for program checkout. The abbreviated form

```
CALL BIND (a,b)
```

is accepted, and UNBIND is unnecessary. No storage is reserved for a name which appears as the first argument in a CALL BIND statement.

Type statements and the IMPLICIT statement may specify the number of machine words per storage unit. For example, the statement

```
IMPLICIT REAL*3(A-H,O-Z), INTEGER*1(I-N)
```

would define the normal FORTRAN name rule using three words for each REAL quantity and one word for each integer quantity.

Assembly language code for the target computer may be included within the FORTRAN program. While this is distasteful from the standpoint of program transportability, it is necessary for systems programming to be able to specify the idiosyncratic instructions required for I/O and interrupt handling. Hopefully this feature will allow all but a few instructions to be written in FORTRAN, rather than encouraging the inclusion of assembly code for functions which could be expressed in FORTRAN. As an alternative to using assembly language within the program, the code generator could be made to intercept certain subroutine calls and generate the appropriate instructions. For some instructions (e.g., enabling and disabling interrupts) this approach might be desirable.

ADAPTING THE COMPILER TO A PARTICULAR MACHINE

The word size, character size, character set, number of words per unit for each data type, and similar parameters of the target machine are defined by tables within the compiler. Other parameters (such as the "quote" character for use in FORMAT statements) make it easy to make the compiler accept some nonstandardized features of other FORTRAN dialects.

All constant computations during compilation are done in double precision, so that code may be compiled for machines with word lengths up to 72 bits. Small hollerith constants are converted to integer constants (using the character set of the target machine) so that they can participate in optimization.

Provision is made for keeping the various kinds of program storage separate in the generated code, so that programs may be compiled for a machine with Read-Only Memory and Random Access Memory (ROM/RAM). In a machine with volatile RAM, all constants must be stored in ROM; thus, it becomes a programmer error to assign a value in an arithmetic statement to a variable which appears in a DATA statement. Generation of this error message is controlled by a ROM/RAM architecture flag in the compiler control tables.

In order to adapt the compiler to produce code for a particular machine, it is necessary to add machine dependent subroutines to perform the functions of assembly language parsing, preoptimization, code generation, postoptimization, and assembly language generation. Each of these functions is described below.

Assembly language parsing is required to allow assembly language code to be included in the FORTRAN program. It is necessary for the compiler to parse this code to find out what variables are referenced and defined and what registers are used, so that the optimizer can produce correct code.

The function of the preoptimization phase is to identify the significant subexpressions (generally, those which must be explicitly computed in an addressable register) for the optimizer. This is needed because some parts of an expression may be computed "free" on a particular computer; these parts should not be considered in common subexpression analysis. For example, consider the addressing for the expression X(I), where X uses three words per storage unit. The expression for the effective address of X(I) is

$$A(X) - 3 + I*3$$

where A(X) represents the starting address of the array X. On some machines (e.g., the INTERDATA series), a full machine address can be specified in an indexed instruction; in such a case, the address field of the instruction would contain A(X) - 3, and the significant subexpression would be I*3. On other machines (e.g., the NOVA), only a small addressing offset can be specified; in this case, the address expression would be -3, and the significant subexpression would be A(X) + I*3. On very small machines (e.g., the PDP-8), the whole expression would have to be computed and used for indirect addressing.

The code generation section will generate code for the target machine from the intermediate language output of the optimizer. Hopefully, this section will mostly consist of a standard table-driven code generator, with some special routines to recognize cases (such as incrementing an integer by one) where particular instructions can be used to advantage.

The postoptimization phase will operate on the output of the code generator to detect small local sequences of code which can be improved. On the NOVA, for example, a test on an accumulator can be merged with an instruction which computes a value in that accumulator. Such cases can be detected most easily after code generation.

The assembly language generation phase will produce assembly language from the generated code for the compilation listing and for the compiler output. Assembly language compatible with the target machine's assembler can be produced if desired. Comments will be generated for inclusion in the assembly language; this is especially important when code is moved by the optimizer and variables are not materialized (stored in a unique memory location).

Although there is a considerable amount of code to be written to adapt the compiler to a new minicomputer, hopefully the availability of standard subroutines to do most of the work will enable us to meet our goal of being able to do it with a man-month of effort.

OPTIMIZATION

There is a fairly well-developed theory and set of techniques for global program optimization; see, for example, (Schaefer, 1973), (Aho and Ullman, 1973), and (Cocke and

Schwartz, 1970). These techniques can produce much better code than that which is produced by most production compilers, even on large machines. This is primarily because the computer manufacturers forego some optimizations to increase compilation speed. For minicomputer and systems programming, however, it is profitable to accept the relatively small increase in compilation time required for good optimization - especially when the alternative is writing the program in assembly language. We shall describe below some of the optimizations which we are implementing. Many of these are standard techniques which are described in the references listed above.

The syntax analysis phase of the compiler constructs a flow graph which represents all the possible paths the program may take during execution. Analysis of this flow graph can produce many valuable types of information: parts of the program which can never be reached during execution (these may be deleted and a diagnostic message given); points at which variables are defined and points at which they are referenced; paths along which a variable is busy (defined at the beginning of a path and referenced at one or more points within the path); places where a variable is referenced before it is defined (this may be a programmer error). When a variable is not used after it is defined, the statement which defines it may be deleted. (It is amazing how often such cases can be found in "checked out" programs.) Common subexpressions can be detected and computed only once; this is especially important for minis, where even "simple" computations such as subscript calculations can require many instructions and registers. The use of an expression optimizer and the machine-dependent pre-optimization phase facilitate the detection of common subexpressions.

Where possible, constants are propagated and combined. Thus, the statements

```
PI = 3.1415926
...
TH = DEG*PI/180.
```

would be compiled as

```
TH = DEG*.0174533 .
```

Assuming that PI was not redefined, the statement defining PI would be compiled as if it were a DATA statement; if PI were only used with other constants where it could be combined, it would not appear in the compiled program at all.

In the case of minicomputers, minimization of the amount of storage required is very important. Analysis of the paths in a program where a variable is busy permits storage optimization in two ways: by avoiding materialization of variables and by overlapping variables which are never busy at the same time. Often, a variable is used as a temporary over a short section of code, and is not referenced thereafter. For example, to exchange the values of two variables, a programmer might write:

```
TEMP = X
X = Y
Y = TEMP .
```

On a multi-register machine, this exchange could be performed using two registers; if TEMP were not referenced again before being redefined, there would be no need (for this code) to assign a storage cell to TEMP and store a value into it. Avoiding the materialization of other variables (such as DO loop induction variables) can save considerable execution time as well as space.

Even when it is necessary to save the values of variables in core, storage can be saved by using the same storage locations for variables which are busy at different times. Many variables are active for only a short time, independent of the length of the program in which they appear. Assigning unique storage locations to such variables results in a very low storage utilization efficiency. In one case where we applied a storage overlap algorithm to a few hundred lines of hand-generated assembly language for a mini, the temporary storage required was reduced from 200 words to 60.

The compiler is able to determine the storage which is used for program temporaries - simple variables which are local to a subprogram and are always defined before they are referenced. These temporaries are grouped under a special location counter for use by the link editor. Using the same graph analysis techniques used by the compiler, the link editor can determine which subprograms are active at mutually exclusive times and overlap their temporary storage areas.

The storage-overlapping optimizations described above could make a program very difficult to debug; hence, they can be inhibited for individual variables or for the entire program by compiler options and special declaration statements.

CONCLUSION

If the promise of abundant inexpensive computation by mini- and micro-computers is to be realized, we must have software development tools to help us write code for these machines. Programming in assembly language is intolerably expensive for machines whose primary attraction is their low cost. A machine-independent FORTRAN compiler can minimize the software development cost, allow programs to be transported easily, and free the user from dependence on a single computer manufacturer.

REFERENCES

1. Aho, Alfred, and Ullman, Jeffrey. The Theory of Parsing, Translation, and Compiling, Volume II. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

2. Cocke, John and Schwartz, J. T. Programming Languages and Their Compilers. Courant Institute of Mathematical Sciences, New York University, 1970.

3. Gray, Philip N. "A Host Computer System for Software Development," Proc. 1974 ACM National Conference, pp. 498-500.

4. Schaefer, Marvin. A Mathematical Theory of Global Program Optimization. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

5. UNIVAC 1100 Series FORTRAN V Programmer Reference, UP-4060, UNIVAC Division of Sperry Rand Corporation.

6. U.S.A. Standard FORTRAN, ANSI Standard X3.9 (1966).