Run time interaction with FORTRAN using mixed code

T. S. Ng and A. Cantoni

Department of Electrical Engineering, University of Newcastle, New South Wales, 2308 Australia

This paper describes an interpretive function generator, COMPIL, for use with the PDP-11 FORTRAN COMPILER and Object Time System. The implementation uses a 'mixed code' approach, which enables the exchange of run time for program space. Other benefits include improved debugging facilities and extended interactive facilities such as the ability to define and link functions at run time. This latter feature is especially useful on the PDP-11 as compilation and linking of function subprograms takes an appreciable time.

(Received March 1974)

1. Introduction

The interpretive code generator, COMPIL, can be used in two modes. The first one allows users to generate up to a maximum of eight functions at run time: the source entered via a user keyboard or some other input device is compressed into pseudo-instructions by COMPIL. During execution when a function is called, COMPIL again takes over control, interprets the compressed code to execute the function. In the second mode, COMPIL compiles FORTRAN subroutines into pseudo-instructions and sets up linking information for Linker-11, thus enabling a mixed code program to be produced and executed.

In both modes the potential for run time and program space trade-off offered by 'mixed code', (Dawson, 1973; Dakin and Poole, 1973), can be exploited. The first mode also offers the additional bonus of run time function definition and linking. An example of this latter feature, in a more generalised form, is included as an integral part of the MULTICS System (Daley and Dennis, 1968) in which a combination of hardware and software is used to achieve run time linking of segments.

The ability to define functions at run time is found to be a useful feature for certain program packages that require functions as input data.

2. Design principle

The PDP-11 system installed in the Electrical Engineering Department at the University of Newcastle has a 28K word main memory, a 256K word fixed head disc and a 147K word per tape dual Dectape auxiliary memory system. In view of these facilities available at present, the following design rules were adopted in developing COMPIL.

1. Core space occupied by COMPIL should be kept to a minimum.
2. It should provide powerful error diagnostics.
3. Commonly used FORTRAN statements should be incorporated and should be implemented in an open ended manner so that new statements could easily be added.
4. It should be self-contained and in no way interfere with other system software.

3. COMPIL statements

The COMPIL language includes standard FORTRAN Logical and Arithmetic Assignment, GOTO, Logical and Arithmetic IF, DO, SUBROUTINE CALL, RETURN, EQUIVALENT and DIMENSION statements. COMMON is permitted if the interactive code is generated before linking as in the second mode of operation. In addition, the following statements have special meanings:

1. FNI (Arg1, Arg2, . . ., Arg*n*)—I is an integer from 0 to 7. This fixed function name statement identifies the function being defined at run time and serves as a dummy entry point when functions are called.
2. ENDF—This signifies the end of a function and causes exit from COMPIL during function definition at run time.

Though only a limited number of statement types are allowed, new statements can easily be added by changing the statement nmemonic table and adding the corresponding section of code.

4. Run time function generation

Functions can be defined via user keyboard or other input devices at run time through COMPIL. The user can call the function generator in his main program by incorporating the statement

```
CALL COMPIL (DEVICE, NAME)
```

where 'DEVICE' is the input device and 'NAME' is the file name. The second parameter is optional for nondirectory devices such as keyboard and papertape.

In response to the calling statement, COMPIL will read from the specified file the user defined functions. Syntactic errors are checked extensively to ensure that the least number of errors survive till execution time as each statement is read. If errors are detected, a message will be printed on the user console together with the statement in error requesting the user to make the appropriate correction. When the input device is keyboard, COMPIL will print out

```
READY
*
```

and wait for the function to be typed in. At the beginning of each line, COMPIL will type out * to indicate it is ready to accept further input strings.

There is no restriction in the format of function statements. The user is allowed to define his functions either as function subprograms or subroutine subprograms. In the former case, the user has to assign his function value to a specified variable FV$. If this variable does not appear in the function definition, COMPIL will assume the function to be a subroutine subprogram and the appropriate action will be taken during execution time. For example,

```
READY
*FN2(E, F)
*K = E**3/F
*FV$ = E − F + SIN (K)
*RETURN
```

is a function subprogram with name FN2 while

```
READY
*FN0(A, N, C)
*DIMENSION A(N)
*D = 1
*DO 10 I = 2.N
*10 IF(ABS(A(I)).GT.ABS(A(D)))D = I
*C = A(D)
*RETURN
```

is a subroutine named FN0 searching for the largest element of an array A(N).

5. COMPIL as a compiler

A version of COMPIL can be used in conjunction with the FORTRAN compiler to produce a mixed code program: Subroutines compiled via COMPIL are compressed into sections of pseudo-instructions. Linking information is set up in such a way that whenever subroutines of interpretive codes are called, the run time system of COMPIL will take over control and interpret the subroutines. When COMPIL is used in this mode, all linking is done through Linker-11.

6. Conclusion

It is realised that COMPIL involves a substantial run time overhead (1·8K word) which means that the core run time trade-off offered by mixed code is exploited only when many or long subroutines are used.

The run time function definition feature is found to be quite useful for certain program packages since on the present PDP-11 system the process of compiling and linking new functions is time consuming and inconvenient.

References

DAKIN, R. J., and POOLE, P. C. (1973). A mixed-code approach, *The Computer Journal*, Vol. 16, No. 3, pp. 219-222.

DALEY, R. C., and DENNIS, J. B. (1968). Virtual Memory, Processes, and Sharing in MULTICS, *CACM*, Vol. 11, No. 5, May 1968, pp. 306-312.

DAWSON, J. L. (1973). Combining interpretive code with machine code, *The Computer Journal*, Vol. 16, No. 3, pp. 216-219.

Testing overflow algorithms for a table of variable size

W. B. Samson

Dundee College of Technology, Bell Street, Dundee DD1 1HG

Large scatter tables which vary in size with time present a problem from the point of view of overflows. This paper describes a program which simulates overflows and indicates which table sizes are to be avoided when the size of the table is to be altered.

(Received February 1975)

1. Introduction

Ecker (1974) has shown that for a given table size a quadratic hash or some related overflow method can be chosen to give a period of search equal to the table size. If the table size varies with time, as may well happen in the case of a scatter table, then it is not usually practicable to alter the overflow algorithm to suit. However, it will usually be possible to adjust the table size by a small amount to give a period of search which is adequate to contain any overflow that is likely to occur.

2. Overflow testing program

The program described below computes the period of search, which is defined as the number of entries which appear in an overflow sequence before any entry is encountered twice, and the capacity for overflow which we define as the number of positions encountered for the first time in an overflow sequence before an endless cycle is entered.

e.g. Table size = M = 10 positions

Overflow algorithm:

$$r\text{th position in sequence} = (1 + \tfrac{1}{2}r^2 - \tfrac{1}{2}r) \bmod M \quad (1)$$

r	*Position*	*Remarks*
1	1	First time encounter
2	2	First time encounter
3	4	First time encounter
4	7	First time encounter
5	1	Second time encounter ∴ Period = 4
6	6	First time encounter
7	2	Second time encounter
8	9	Last first time encounter ∴ Capacity = 6
9	7	Second time encounter
10	6	Second time encounter
11	6	Third time encounter

The period of search in this table is 4 and the capacity for overflow is 6.

A program was written to simulate overflow for various table sizes and overflow algorithms. The following table shows results for table sizes in the region of 1,200 positions with overflow algorithm (1).

Table size (positions)	*Period of search*	*Capacity for overflow*
1,196	63	336
1,197	49	160
1,198	301	600
1,199	65	330
1,200	53	352
1,201	601	601
1,202	302	602
1,203	203	402

In the case of a table size of this order it is clearly more sensible to choose 1,201 positions than 1,197 positions from the point of view of overflows.

3. Conclusions

When one is dealing with a variable table of large size, it is advisable to simulate overflows for table sizes in the region of the preferred table size to find the most favourable size for overflow considerations.

Acknowledgements

I am grateful to Dr. R. H. Davis and Mr. J. R. Lowe for helpful discussions leading to the work described here and to the staff of Dundee College of Technology Computer Centre for their help in preparing and running the program.

Reference

ECKER, A. (1974). The period of search for the quadratic and related hash methods, *The Computer Journal*, Vol. 17, No. 4, pp. 340-343.