N71-23435
NASA CR-117901
Syracuse Univ.

COMPUTER ENHANCEMENT
through
INTERPRETIVE TECHNIQUES

Semi-Annual Status Report

for the

National Aeronautics and Space Administration
Goddard Space Flight Center

under

Grant NGR 33 - 022 - 125

by

Garth H. Foster
Principal Investigator

I. Introduction

This study has as its thesis the improvement in the usage of the digital computer through the use of the technique of interpretation rather than the compilation of higher ordered languages.

Nowadays more and more computer programs in the scientific and commercial sectors are being written in higher level languages such as FORTRAN, ALGOL, PL/I, and COBOL. Such programs are compiled or translated to the machine language of a specific machine and run in a production environment, generally that of multiprogramming.

The rationale of this study is that there are three areas where interpretive techniques could enhance the performance of computers. The first would be in those instances where interpreters could best compilers in execution speeds. Investigating such a possibility implies the restriction of the problems to areas in which both techniques could be applied and of course the use of higher level languages in coding the problems. We shall discuss this further shortly.

The second way in which utility could be provided by interpreters is that of trading machine cycles in execution speed for space in the run time code stream. The third way in which interpretation techniques would be of value would obtain if the implementation of an interpreter of a given language provides more effective use of programmer time in the development of software and for problems which are to be run once or only a very few number of times. In this context it is envisaged that a given language would have two (and perhaps more) implementations; one would be an interpreter on which the program development would be done and the other would be a compiler in which the production work would be done. If the problem is to be run few enough times, then the interpreter only would be used. Here the number referred to as a few depends upon the size and complexity of a program, the execution and compile time in addition to the interpreted run time, the cost of the program development, and the number of compilations used before the program may be run usefully for the first time. The three points of view relative to interpretation given above sketch a range of capabilities ranging from direct superiority to sometimes usefulness. We now turn attention to detailing investigations in these areas.

II. Fundamental Choices

The equipment and machine configuration on which this study is being conducted is an IBM System 360 Model 50 - I (512 K bytes) with 2 2314 disk units. The operating system is the Syracuse University Operating System (SUOS) a modification of MVT II release 18.6 using a HASP-like spooling program to provide spooling and allocation of ports to interactive problem processors.

The interpretive system considered for this study is APL\360 (A Programming Language for the System 360). More than just being the time sharing available at this institution, APL was chosen for several specific reasons. First, by the nature of an interpreter the input source string is interpreted requiring syntax analysis and run time elaboration of every statement every time it is encountered. The implication is that only if the language is imbued with powerful language primitives and compact constructs, can there be a hope of absorbing the overhead of interpretation. It is our judgment that APL comes closer to this objective for a reasonable variety of problems than other available computer languages.

Next, the more condensed the source string the interpreted language has, the higher the ratio of the size of the run time object program of compiled code to the length of the source to be interpreted. This leads to better space trade offs for computer cycles lost in interpretation. The terse nature of "good" APL code makes it a natural choice in this context.

Finally, the spectrum of language processor implementation ranging from interpreters to compilers has blurred with increased importance placed on binding variables closer to execution, tracing and debugging aids, and incremental compilation. Thus, we do not exclude the possibility of "smart" interpreters which enlarge the segment of the input code string skanned in determining the environment for interpretation. This would not be compilation since no code would be saved and the process is so data and code sequence dependent that it can not be considered compilation. In this respect P.S. Abrams [1] has already established the power of such an approach. The advent of large scale microprogrammed computers, particularly those with writable control stores, leads to the possibility executing a higher level language as the native language of the computer rather than machine language. The structure of APL suggests that implementation of it in such a computer as a native language is worthy of further exploration.

In suggesting the principal compiler language we have chosen FORTRAN IV which for IBM computers the choice has been FORTRAN IV -H (Opt 2). FORTRAN is probably the most widely used language in this country and the period of development of compilers for that language suggests a wealth of experience from which improvements have come. Other versions of FORTRAN including those kept in-core for load and go operation will be referred to. PL/I may be considered although such reflection has not been extensive at this time.

III. Relative Raws Speed

Initial efforts were to examine some of the powerful APL programming constructs from which more complicated programming expressions could be built. If the interpretive system can not compete on this level, then it will not be able to compete on a more macroscopic level. Reduction, inner and outer products are three of the more obvious operations to investigate.

The reduction expression ×/⍳56 for example generates the integers 1 through 56 (if the ORIGIN of indexing is 1) and the pair of symbols ×/ causes all of the number to be multiplied together. Clearly this is equivalent to 56 factorial written in APL as !56

As a side comment this is the largest factorial which may be calculated precisely in the System 360.

To execute 56 factorial as ×/⍳56

APL required an average of 3.9 60ths of a second of CPU time (but not console time) to execute. On the other hand FORTRAN IV H(OPT=2) required the following times (60ths of a second) to:

	COMPILE LOAD and GO		GO CPU time	
VARIABLE TYPE	WITH PRINTING	WITHOUT PRINTING	WITH PRINTING	WITHOUT PRINTING
I*4	746	611	20	15
R*4	769	608	21	22*
R*8	741	591	21	17

On the other hand, looking at the summation of the first 7500 integers (coming close to the limits placed upon us by the standard 36K byte workspace) which in APL notation is: +/⍳7500 takes 165.6 60ths of a second on the average (over 10 trials).

The comparable figures in FORTRAN are:

FORTRAN VARIABLE TYPE	COMPILE LOAD and GO 60ths of a second		GO step only 60ths of a second	
	WITH PRINTING	WITHOUT PRINTING	WITH PRINTING	WITHOUT PRINTING
I*4	744	600	17	16
R*4	605	614*	48	48
R*8	635	644*	47	48

In the first instance APL appears to be about 5 times faster than the GO step for FORTRAN whereas in the second case the GO step in FORTRAN is anywhere from 3.5 to 9.75 times as fast as APL.

Several observations are in order.

1) In APL in both of the reduction cases cited all of the data is generated and temporarily stored and then the multiplications or additions are performed. The compiled code on the other hand calculates the product or sum as a part of a DO loop, thus using less transient core space than APL. This is an inherent price due to the interactive nature of the system and protecting the workspace environment in the implementation of the interpreter. In other words once the code has been passed APL does not back up. Where the data is present in the environment and not required to be generated, the overhead of interpretation many be spread somewhat further. This is true even when data has to be generated. For example consider that two APL reduction expressions +/⍳2000 and +/2000⍴1, which sum the first 2000 integers and 2000 ones, have execution times which average 46.7 and 44.1 60ths of a second, respectively. Clearly the cost of generating 2000 different

integers is not much higher than generating 2000 constant values of one. Since the time for $+/\iota$ averages 1.4 60ths of a second, thus we see that there is a small amount of overhead but when we sum the first 7500 integers rather than the first 2000 we do roughly 3.75 times as many operations at an expenditure of 3.6 as much time.

2) There are improvements in both compile, load and go as well as just the go step in almost all cases when there is no printing required in the FORTRAN program formulation. Those cases in which no improvement is seen in the timings are marked with an asterisk; these probably follow a similar pattern but it has been masked by system timer inaccuracies. The reduction in times are in areas in which FORTRAN has its closest approach to being interpretive, that is in I/O and its associated format control.

Thus compiled FORTRAN programs can suffer some interpretive degradation when a great deal of output using many formats is required. The closer that we seek to having control at run time the more willing we have traditionally been willing to give up execution speed.

3) The compile times for a fixed FORTRAN variable type are somewhat constant as one might expect and one might ask how should these times be considered, relative to the GO step. We might charge the compile and link editing times off against a number of program runs and ask where the break even point would be for

$$\frac{1}{N}\left(\begin{matrix}\text{Compile, Load and Go}\\ \text{time}\end{matrix} - \begin{matrix}\text{Go}\\ \text{time}\end{matrix}\right) + \begin{matrix}\text{Go}\\ \text{time}\end{matrix} = \begin{matrix}\text{APL}\\ \text{time}\end{matrix}$$

so

$$N \leftarrow \lceil \left(\text{Compile, Load and Go} - \text{Go}\right) \div \text{APL} - \text{GO}$$

The positive N would then be a figure of merit. In the first case a positive N does not exist and in the second illustrative case N is 5.

In the previously mentioned side cases of +/⍳2000 and +/2000⍴1 (having APL times of 46.7 and 44.1 60ths of a second) we have comparable FORTRAN times (with printing) of:

	+/⍳2000		+/2000⍴1	
VARIABLE TYPE	COMPILE, LOAD and GO 60ths	GO 60ths	COMPILE, LOAD and GO 60ths	GO 60ths
I*4	723	21	741	22
R*4	715	26	761	25
R*8	646	22	734	21
N =	26		31	

4) It should be noted in all cases cited if REAL variables are required there are some advantages in using R*8 even when R*4 will suffice.

A number of preliminary steps in examining the efficiencies of inner and outer product evaluations in APL compared with comparable "mini"-programs written in FORTRAN have been undertaken.
Typical of these is the inner product represented by the execution of the expressions

```
D ← 3 3 ⍴ ⍳ 9
```

and then timing

```
D +. ⌊ D
```

In APL this timed to 2.4 60ths of a second. In FORTRAN 4 the comparable times are

FORTRAN VARIABLE TYPE	COMPILE LOAD and GO (60ths)	GO only (60ths)
I*4	684	19
R*4	854	19

Once again there does not exist an N such that if the compiled version were run N times the compilation overhead could be absorbed.

The inner product does not have significance as far as we know, (D⌊.+D gives the shortest 2 leg trip through a distance graph), but was chosen to use simple functions either readily available in a computer's machine language or easily synthesized.

IV. Core Savings at Run Time with an Interpretive System.

For the sake of comparison we again consider a rather trivial programming problem, that is: Write a generalized routine which for arbitrarily named parameters takes the value of R and adds 5 to it and assignes the value of the reult to variable Z.

In APL this would be written:

```
    ∇ Z ←F   R
[1]   Z ←5 + R
    ∇
```

The total amount of space required is 68 bytes, 40 of which is header overhead.

Disregarding the fact that the APL expression works independent of whether the argument of F is a scalar, vector, matrix or an array of higher rank and the size of programs be quoted is for scalar R only, we have for similar programs written in assembler (BAL), FORTRAN IV G, WATFIV (in core extended FORTRAN), PL/I F, PL/C (an in core PL/I subset) the following core map sizes:

Processor	Size in bytes
Assembler	26
FORTRAN IV G	120
WATFIV	140
PL/I (F)	510
PL/I C	550

The differences of the coding sizes between assembler and the FORTRANs may be presumed to arise from tighter code and complete control in function calling and handling of parameters. The added sizes of the use of PL/I processors results in part from a difference in language philosophy and such considerations as default parameters. Part of those size differences however come from the fact that symbol table maps and other conveniences for program tracing, debugging, and maintenance are generated for the PL/I processors. Such conveniences are usually a facility found in interpretive systems and this should be kept in mind when either trying to write off the extra size in the compiler based system or when keeping in mind the size of the interpreter generally residing in core.

Even at that, the run time packages can be significant for ordinary programs; however, for FORTRAN programs equivalent to the APL expressions given earlier the run time load modules are overbearing. For example, the FORTRAN program for $x/\iota 56$ has the following program and load module sizes.

FORTRAN VARIABLE TYPE	FORTRAN PROGRAM SIZE (BYTES)	LOAD MODULE SIZE (BYTES)
I*4	320	20,696
R*4	304	20,680
R*8	312	20,688

Load modules of sizes similar to these hold for the other examples including the program equivalent to $Z \leftarrow 5 + R$.

While the surface has only been scratched it appears that on the microscopic level APL has a good chance to compete in code space density considerations.

Overall, if APL succeeds in competition with a compiled language it will be partly because those of us in computing activity have not questioned sufficiently the overhead costs that present batch systems have in their operation.

The one program which we have been supplied by NASA which has been coded both in FORTRAN and APL and which may be taken to be typical of work required at Goddard Space Flight Center has been found to be exceptionally long and inefficient in its APL form; that is to be rewritten while the FORTRAN program is learned.

A paper, "The Use of APL to Investigate Sequential Machines" [2] considered a number of programs which had already been developed by other researchers to study a number of aspects of logical and sequential machine theory. Even when modeling the orginal programs down to the detail of the IO formats and console interactions the APL programs were anywhere from 2.5 to 6 times as dense on the source statement level than the BASIC and FORTRAN programs. This is of little meaning however, although it is a source language comparison such as this which is usually made. If even slight incompatibilities are allowed in IO, then some ratios go as high as 42 to 1. Further observations on a more detailed level will be made about selective programs.

V. <u>Other Considerations</u>

A part of our effort has been to study some microprogrammed processors to examine the possibilities of imbedding an APL executing engine in such a machine. This could either be a standard APL implementation or one of the "intelligent" variety as described by Abrams.

The primary line of investigation has been to examine a processor which is under design and construction by Burroughs. Other than to report our thoughts in these directions it is yet too early to make a more definite statement.

References

[1] P. S. Abrams, "An APL Machine", Ph.D. Dissertation, Stanford University, SLAC REPORT NO 114, Feb. 1970, AD 706 - 741.

[2] G. H. Foster, "Using APL to Investigate Sequential Machines", NEREM-70 70 C 63 NEREM Technical Applications Sessions, pp. 120-128.