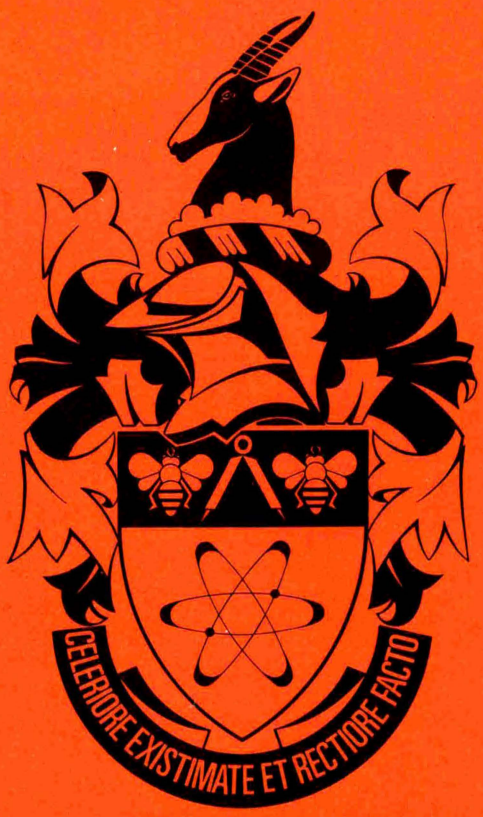


Quaestiones Informaticae

Vol. 2 No. 1 November, 1982

Quaestiones Informaticae

An official publication of the Computer Society of South Africa
'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika

Editor: Prof G. Wiechers
Department of Computer Science and Information Systems
University of South Africa
P.O. Box 392, Pretoria 0001

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R6	$7	£3,0
Institutions	R12	$14	£6,0

Quaestiones Informaticae is prepared for publication by Thomson Publications South Africa (Pty) Ltd for the Computer Society of South Africa.

NOTE FROM THE EDITOR

After an absence of two years we are happy to announce that we are now in a position to continue the publication of Quaestiones Informaticae. The first Volume of QI consists of three numbers, and appeared during the period June 1979 till March 1980 under the editorship of Prof Howard Williams. Because Prof Williams took up a post at the Herriot-Watt University in Edinburgh, he had to relinquish his position as editor. The Computer Society of South Africa, which sponsors the publication of QI, appointed me as editor, whereas Mr Peter Pirow took over the administration of the Journal. The editorial board functions under the auspices of the Publications Committee of the CSSA.

The current issue is Number 1 of Volume 2. It is planned to publish altogether three issues in the Volume, with most of the papers coming from the Second South African Computer Symposium on Research in Theory, Software and Hardware. This Symposium was held on 28th and 29th October, 1981. At present it appears that most of the material published in this Journal comes from papers read at conferences. We invite possible contributors to submit their work to QI, since only the vigorous support of researchers in the field of Computer Science and Information Systems will keep this publication alive.

G WIECHERS

November, 1983

Micro-Code Implementation of Language Interpreters

P. P. Roets
National Research Institute for Mathematical Sciences,
Council for Scientific and Industrial Research

Abstract
An evaluation is made of the improvement of language interpreters by coding recurring sequences of instructions in micro-code. The results of an experiment with PASCAL indicate that the increase in speed expected may not always be realised but that, somewhat surprisingly, micro-code may be used to reduce the effort of implementing an interpreter.

1. Introduction

There are several reasons why, even to this day, interpretative language systems remain an essential tool in the application of computer systems. The most important reason is the excellent inter-active support capabilities that are made possible by the interruptability of the execution process. As may be gleaned from the many implementations of the BASIC language, the interruptability of the program execution allows not only a unique way of debugging but also makes possible the integration of text editors, compilers and debuggers. Secondly, it may be claimed that interprative systems are relatively compact, although this compactness depends to some extent on the coding system used for the input to the interpreter.

In the case of BASIC the input often consists of unprocessed character text generated by the text editor, a compressed version of the text, or a semi-compiled form of the program such as Reverse Polish Notation. In the practical example to be used in this paper the program is compiled into the object language of a pseudo stack machine. Regardless of the internal coding used, the memory requirements may be reduced by sharing a single copy of the interpreter amongst several users. Significant savings can thus be made when a large interpreter, such as a hardware simulator, is used concurrently by a number of processes.

The major drawback of an interpretative system is its slow execution speed. Compared with a compiled version of the program, an interpretative system may require from three to four times the central processing cycles for completion of the calculation.

There are two major factors contributing to this slow speed. Firstly since the compilation of the program has not been completed, a number of cycles are expended in accomplishing this. These processes may include a complete analysis of the source program, certain (usually semantic) elements of the compilation process such as the evaluation of identifier-value associations, or code-generator processes such as address calculation. It must be borne in mind that if the source program contains loop control structures, these calculations are repeated at every iteration. From this we can conclude that any speeding up of the interpretation process can only linearly improve the execution time, while a full compiler can, through optimization, improve the execution time at a better than linear level.

Secondly, an interpretative system expends a number of cycles in decoding the next instruction to be executed. In the full compiler system this cost is limited to one instruction fetch per instruction. This advantage is somewhat counter-balanced by the fact that the fully compiled version of the program usually contains a greater number of (simple) instructions.

Having described the working environment, we are now in a position to evaluate the probable improvement of language interpreters through micro-coding.

2. The Micro-Program Environment

The computer designer uses the micro-programming technique to introduce a degree of freedom into the design. By employing a micro-coded control structure the specification of the user level instruction repertoire can be developed in parallel with the implementation of the actual hardware. In recent years, however, micro-code has been used extensively as a method for the economical post-production correction and expansion of the user-level instruction capabiliites of a computer; on most high-speed machines, for instance, a very large percentage of user level instructions are implemented as a single micro-instruction.

The micro-program differs from the assembly level program essentially in terms of the asynchronous and parallel actions of the instruction functions.

The assembly (user) level instruction usually specifies a single operation (e.g. a transfer of data) and is isolated from all asynchronous activities, such as interrupts. On the other hand most micro-instructions allow the specification of a number of simultaneous actions and have explicit parameters for controlling asynchronous activities. In addition, the micro-program is required to consider certain timing constraints and take steps to optimize execution of a sequence of micro-program instructions. This parallelism and time dependence in the design of the micro-level machine often results in micro-instructions having counter-intuitive side-effects, a problem area about which the user of the equipment is best kept in ignorance. It is, however, exactly this parallelism and timing freedom that could be used to create a different or expanded user level machine and thus provide a mechanism to improve the execution time of an interpretive system.

The degree of parallelism available in a micro-instruction is directly related to the number of bits (width) of a micro-instruction. On large mainframes a width of more than 100 bits is employed, and in addition some activities, such as effective address calculation, are implemented in hardware. This has the effect of reducing the average ratio of micro-instructions per user level instruction to nearly one-to-one. In older and slower machines this ratio is much larger.

For completeness, it should be remarked that the physics and electronics of computer design limit the minimum time required for execution of a micro-instruction, and thus future micro-programmed architecture will employ wider micro-words rather than faster micro-execution cycles.

In [1] these properties of micro-programs are discussed in some depth, and in addition a survey is given of the programming tools available to the programmer.

3. Eth-Pseudo Stack Machine [2]

Much of the popularity of PASCAL is due to the unrestricted availability of good quality compilers for the language. One of the relatively machine-independent methods of distributing PASCAL is through the so-called P-code version which consists of the source code and a compiled version of the compiler. The object language used is that of a pseudo stack machine. A compiler for a particular machine can then be bootstrapped by either writing a code generator to transform the output or

implementing an interpreter to simulate the stack machine. Historically the second approach was favoured, and the resulting interpretative system is often used for production. Most of the implementations on micro-computers are interpretative (due to hardware restrictions) except for the micro-coded implementation known as the 'PASCAL Micro Engine'.

The pseudo stack machine has a relatively simple structure, having a push-down stack used for temporary values and a frame pointer relative to which declared variables are allocated. Dynamically allocated variables are stored in a separate heap. Operations are available for loading and storing variables relative to any of the nested levels of the program, i.e. using the associated linked frame pointers. All arithmetic operators operate on a number of values obtained from the pushdown stack, and push the result back onto the stack.

All constants, including even the sixteen-byte set and string constants, as well as the offset and indexing constants, may be considered immediate operands of pseudo machine instructions.

In the following sections a few examples are given of specific pseudo instructions together with the interpreter implementations.

4. Replacement of the Next-Instruction-Loop

As is the case for all interpreters, a fixed sequence of instructions is used to proceed to the simulation of the next instruction. In the case of the Perkin-Elmer series of machines this loop consists of the following four instructions.

```
        B     NEXTINS          BRANCH TO LOOP
        .
        .
NEXTINS LB    RT,O(IC)         GET PSEUDO OPCODE
        LHL   RT,TAB(RT,RT)    GET OFFSET FROM BRANCH TABLE
        B     BASE(RT)         AND GO TO SIMULATOR
```

Such a loop is an obvious candidate for replacement by a series of micro-instructions, since in this way it would be possible to eliminate three of the memory accesses required for instruction reads.

The width of the host micro-instruction makes it possible to code the above instructions as $3+3+2+3 = 11$ micro instructions [4]. Creation of a new user level instruction PXNI requires 10 micro instructions plus an overhead of 5 micro instructions to activate it.

Contrary to expectations the activation protocol nullifies the slight gain achieved by implementing the transfer of control by micro instructions.

If we estimate the average execution time of an instruction simulator as equivalent to 16 micro instructions, the maximum gain to be obtained would be

$$1/_{(11+16)} = 4\%.$$

This estimate is borne out by the results given in Section 7.

Since the activation of a user-created micro instruction carries an overhead of 5 micro cycles, the replacement of the loop does not warrant the use of micro-code. Additional recurring sequences of code must be implemented as part of the micro instruction.

5. Replacement of Immediate Operands and Addressing Constants

The instruction format of the PASCAL stack machine consists of three fields: I, P, Q. The P field serves to identify the stack level for accessing variables, while the Q field may be either an addressing offset or a program constant. In some instances the P or both the P and Q fields contain null values. By encoding a set of flags in the branch table it is possible to do the accessing of P and Q during the micro-coded transfer of control between instruction simulators, and this has a dramatic effect on the length of these executors.

Example: LOI P,Q

Load an integer value from displacement Q above the pth frame pointer. The Q field may have a length of either 8 or 16 bits.

Assembly level coding consists of the following 13 instructions.

```
$LOI  LB    P,O(IC)     GET VALUE OF P
      LHL   Q,1(IC)     AND Q (HALFWORD)
      AIS   IC,3        UPDATE INSTRUCTION COUNTER
      B     LOI
$LOIS LB    P,O(IC)     P
      LB    Q,1(IC)     Q(BYTE)
      AIS   IC,2
LOI   BAL   RT,BASE     FIND FRAME POINTER
      AR    Q,IR        CALCULATE ADDRESS
      L     IR,O(Q)     AND GET IT'S VALUE
      ST    IR,O(TOS)   PUSH ONTO STACK
      AIS   TOS,4
      PXNI  0
```

When the address calculation is done in micro-code as part of the instruction transfer mechanism, these 13 instructions are replaced by the following

```
$LOI  EQU   *
$LOIS L     IR,O(Q)
      ST    IR,O(TOS)
      AIS   TOS,4
      PXNI  0
```

The same method (doing address calculations based on flag bits in the branch address word) could be applied at the assembly level. The cost of decoding is such that the total execution time would increase. Since the flag bits are picked up as part of the branch address, this can be done at the full micro-cycling speed by incorporating the address decoding in the micro-instruction. This is an instance where space is traded for time, and the loss recovered by using the higher speed of the micro-program.

6. Stack Management

The final wasteful activity of the pseudo machine instruction executor that may be considered for inclusion into the micro-code, is related to the stack manipulation instructions.

All arithmetic operators use a number of values from the stack and return the result to the top of the stack. Owing to the fast execution of micro instructions, it is feasible to leave results in machine registers and cause the transfer instruction to adjust the contents of the registers to contain the correct number of values for the next pseudo instruction. The number of stack values required could conveniently be encoded as part of the branch address to the instruction simulator. The number of values returned by the simulator is returned as a parameter of the transfer instruction.

Example:

ADD integer instruction

The original simulator form:

```
$ADI  SIS   TOS,4       POP VALUE
      L     IR,O(TOS)   FROM STACK
      A     IR,-4(TOS)  ADD AND
      ST    IR,-4(TOS)  STORE BACK
      PXNI  0
```

can then be changed to

```
$ADI  AR    IR,IE
      PXNI  1
```

It is relatively expensive to decode the adjustments that must be made to the stack (it requires 6 micro instructions) and thus any speed gain due to this technique is attributable entirely to the saving of transfers to and from main memory. Although an appreciable number of 'pushes' and 'pops' are saved, the current implementation is non-optimal, since instruction sequences with the following structure:

```
PUSH VALUE1
PUSH VALUE2
ADD
```

would cause VALUE1 to be stored in main memory before execution of the second instruction. This value would be retrieved before the ADD operation is executed. It is very difficult to eliminate these transfers since the micro-code does not allow information to be retained between successive activations. Indeed, it is estimated that decoding the complete state of the stack would be so costly as to cancel any possible time savings.

7. Implementation Results

If the common instruction sequences are replaced by a micro-coded equivalent, this should reduce resource requirements in two respects.

The memory requirements of the instruction simulator were reduced by 45%. This may well be an impressive figure. However, the executors are only a small component of the total memory requirement, which primarily consists of the storage required by the program being interpreted.

The more important objective of a micro-code implementation is to reduce the execution time requirement. Using the interpreter to compile the PASCAL compiler, the following clock times were obtained.

Original assembly level interpreter	342 s
After replacement of central loop	321 s
Adding constant and addressing calculation and stack management	297 s

This reflects an effective improvement of about 10 %.

8. Discussion and Conclusions

The effective use of micro-coded instructions is obviously dependent on a great many factors. The width of the micro instruction plays an important part, since this determines the ratio of micro instructions per assembly level instruction. When this ratio is small, there may very well exist little opportunity to improve performance by micro-coding segments of an interpreter. This also indicates that the instructions to be replaced must be selected with great care. On the specific machine available to the author, micro-coding of floating point operations would be much more profitable, owing to the width ratio and timing characteristics. We may also conclude that the use of micro-code in a smaller and slower machine will have a more dramatic effect.

In [5] a micro-programmed implementation of APL is described. The speed improvement ratio of some 2 to 3 can be attributed to two elements: the greater structural disparity between the structure of the APL and the IBM System/370, and the capability of the micro-architecture to support a restart of its execution after an external interrupt.

A less obvious use of micro-code would be to simplify the development task. Had the original interpreter been implemented using the final micro-coded special instructions, the time required for the development project would have been reduced considerably.

Finally, in some applications one may expect to achieve dramatic savings in storage requirements.

References

[1] S. Dusguata, Some aspects of high-level microprogramming, Computing Surveys, **12**, (1980) 295-323.

[2] K. V. Nori, et a, The PASCAL 'P' Compiler Implementation Notes, Revised Edition (E.T.H. Zürich, 1976).

[3] 3220 User's Manual, Document 29-963, (Perkin-Elmer, Computer Systems Division, Ocean Port NJ, 1979).

[4] Model 3220 Microprogramming Reference Manual, Document 29-694, (Perkin-Elmer, Computer Systems Division, Ocean Port NJ, 1979).

[5] A. Hassit, and L. E. Lyon, An APL emulator on System/370, IBM System Journal **4**, (1976) 358-378.

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to: Prof. G. Wiechers at:

Department of Computer Science
University of South Africa
P.O. Box 392
Pretoria 0001
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter l, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

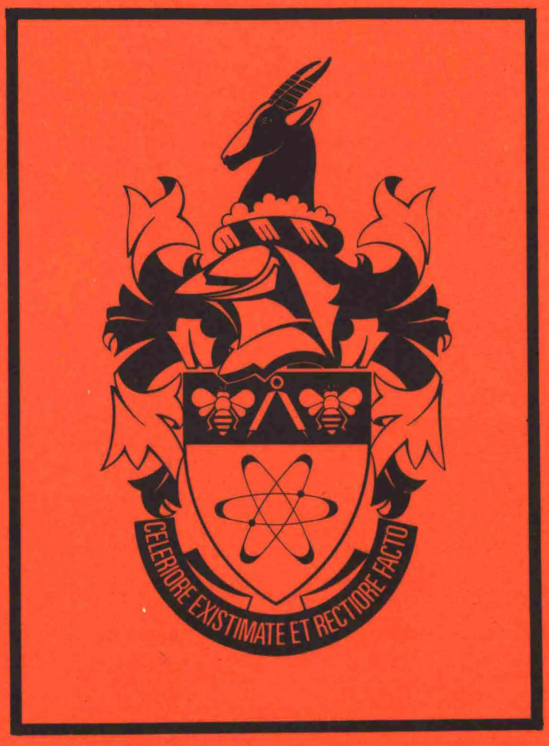
Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.

Quaestiones Informaticae



Contents/Inhoud

*Presented at the second South African Computer Symposium held on 28th and 29th October, 1981.