SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 6, 109–116 (1976)

A PASCAL Compiler for PDP 11 Minicomputers

C. BRON AND W. DE VRIES
Department of Electrical Engineering, Twente University of Technology, P.O. Box 217, Enschede, Netherlands

SUMMARY

In this paper the development of a cross-compiler running on the central computing facility is described. The compiler transforms PASCAL source code into object code for the PDP 11 family. The arguments for higher level languages on minicomputers and the choice made for PASCAL are discussed. It is shown that only a minor effort in terms of manpower is required if such a development is based on an existing compiler that is suited to the purpose of adaptation. Even without large amounts of optimization the code produced is both compact and efficient. Some attention is paid to requirements that should be fulfilled in portable compilers. The paper ends with a discussion of some strong points and weak points of the PDP 11 architecture.

KEY WORDS PASCAL Cross-compiler PDP 11 Portable compiler Code generation Machine architecture

INTRODUCTION

It has been argued[1,2] that programming in machine code or assembly language should become obsolete as knowledge of compiler construction and implementation by means of bootstrapping procedures increases. For users of minicomputers, equipped with a moderate amount of core store, and quite often without a random access backing store, higher level programming facilities are usually not available, or at best some small FORTRAN or BASIC subset (if these languages can be described at all by the term 'higher level'[3]).

The reasons are self explanatory: small systems are not able to support the compilers necessary to translate higher level languages of any sophistication into acceptable machine code. However, if one decides to perform compilations for minicomputers off-line on a large computer, such restrictions suddenly vanish.

At Twente University of Technology (Netherlands) at present seven PDP 11's are installed, ranging from 11/10 via 20 and 40 to 11/45. A DEC 10 as a central computing facility is due to arrive by summer 1975.

In the following the development of a PASCAL compiler for the PDP 11 series running on the DEC 10 is described.

PHILOSOPHY

When providing an implementation of a programming language in an environment with existing experience in assembly language programming it is a prime requirement that the higher level language should perform in such a manner that potential users are not confronted with the choice between the ease of the high level language and the speed of the hand coded program.

Received 27 January 1975
Revised 1 July 1975

Whereas the speed of the generated code is of extreme importance, the compilation speed is almost irrelevant, since programs to be compiled will be of moderate size, and the size of the compiler is not critical if it can be run on an external configuration.

A second requirement for the implementation is that the generated code should be compact, in order to fit into small memory sizes. These two requirements together express that the generated code should be like the code that would be written by a responsible, reliable assembly languages programmer (if such programmers exist at all!).

Thirdly, the implementation should not adopt a paternalistic attitude towards the programmer. He will be allowed to make mistakes that are not detected by the implementation, and some of these may be fatal ones.

No automatic dynamic checks are built into the code where the assembly language programmer would not build in these checks himself. However, any help that the *compiler* may provide in signalling errors should be most welcome!

Fourthly, the language should be available on the central computer, thus enabling programmers to verify their products in an environment more suitable than the real-time environment for which they are intended to be applied.

Finally, the language should cater for products of high quality. (It is particularly in the latter respect that, for example, FORTRAN fails miserably.)

CHOICE OF LANGUAGE

When selecting a language for a purpose as sketched above a number of conflicting interests may be involved. We will not try to weigh the languages considered (FORTRAN, a subset of ALGOL 60, CORAL 66,[4] ALGOL W, PASCAL), but solely list some of the arguments in favour of PASCAL,[5,6] in this particular instance, realizing that no language is perfect.[7]

(1) It provides for a form of data structuring (*viz.* records and pointers) that both FORTRAN and ALGOL 60 are lacking.

(2) It is a simple (although non-orthogonal) language.

(3) It is sufficiently close to ALGOL (and even to FORTRAN, if you wish) in order to allow users to switch, without providing extensive language (re)training.

(4) Previous experiences[8-10] have shown PASCAL to be readily implemented for a variety of machines.

(5) A PASCAL compiler, generating code for a hypothetical stack computer, and a description of that computer are available. This compiler itself was described in PASCAL[11] (P-compiler).

(6) PASCAL is already available on the DEC 10,[9] opening the possibility of constructing a compiler by doing a minimal amount of rewriting of the compiler mentioned under (5).

(7) The language PASCAL is designed in such a way as to cater for a straightforward implementation on most present day computers, without introducing appreciable overheads.

(8) The register and addressing structure of the PDP 11 would lend itself most readily to the implementation of a language which is described in terms of a stack machine.

(9) The availability of a compiler, developed 'in house', seems a good starting point, from which requirements from individual users may be built in.

(10) (3) and (4) together illustrate that when the choice for PASCAL is made the portability issue is not great. Inward portability is ensured by the ease of recoding from (for instance) FORTRAN into PASCAL. Outward portability will be increasing along with a rapidly increasing number of PASCAL implementations.

IMPLEMENTATION ASPECTS

It has been decided to base a compiler for PDP 11 code on the compiler written for the hypothetical stack machine (P-machine). It appeared impractical to transform the code for the P-machine directly into PDP 11 code for the following reason. In the P-machine all elementary data types ranging from 'boolean' to 'real' and 'powerset' are mapped onto one unit of storage. If addresses occurring in the P-machine's code are to be transformed into addresses for the target machine, then this can only be done if all elementary data types occupy an equal number (not necessarily 1) of storage units. On the PDP 11 with its small word size of 16 bits clearly the choice of a sufficient standard data size would have defeated our main objectives: compactness and efficiency of data representation.

A similar argument may be applied to the mapping of the P-machine's code, but here the problems could have been circumvented by the creation of a (lengthy) address correspondence table. So we decided to adapt the compiler itself, selecting for each data structure of the language the most attractive mapping in the target machine, and making for the code to be generated an optimal choice between in line code and subroutine calls.

An additional benefit in *adapting* the compiler was the possibility of code optimization (in essence this means an extension of the instruction set of the P-machine) and implementing certain features that were not present in the original version of the compiler. (In essence this means the liberty to extend the language so as to suit private needs.)

Examples of the first kind are: the elimination of runtime lower bound correction in array subscription, and optimization of the for-loop code, making use of the PDP 11 instructions that increment or decrement specified operands by 1.

Examples of the second kind: the implementation of formal procedure/function parameters by requiring full parameter specifications and an extension of the result type of functions to any type.

The latter extension allows the user to construct (prefix-)expressions out of any data type (e.g. complex, double length integer, three-dimensional point, etc.).

Execution efficiency has mainly been obtained by selecting data mappings in close accordance with the structure of the target machine.

The block structure does not affect the speed of reference to local objects, objects declared in the outer block or objects passed as parameters.

The sole overhead on the addressing mechanism (small as it is) is placed on the accessing of objects on intermediate block levels.

Compactness of code has been accomplished by making use, wherever possible, of auto-increment or auto-decrement addressing.

The speed of procedure call/return is demonstrated by the following summary:

Apart from the PDP 11 subroutine call and return instruction we find:

(1) In the calling sequence: 1 word.

(2) In the procedure body: 3 words at the entrance and 3 words at the exit.

(3) Counting instruction times and memory cycles, the total cost of procedure call plus return is estimated at 15 μsec, a figure that compares very favourably with known implementations of block structured languages.[12]

Thanks to the static structure of PASCAL the parameter organization could be kept simple, parameter checking being performed at compile time.

A runtime package of approximately 1,000 words provides for code compaction where this will not lead to loss of execution efficiency.

Due to the subroutine structure of the target machine code sequences of more than 4–5 words may well be replaced by subroutine calls. A further role for the subroutine's package is to hide the differences in an instruction set within the PDP 11 range from 10–20–40–45. The code generated for each of these configurations will be approximately the same, but the subroutine package (fed as input to the compiler) will be adapted to the configuration.

Of the runtime package only those routines that are actually required by the code are 'generated' along with the compiled code. The decision to separate the 'return-address stack' from the data stack allows subroutines to operate on 'the top of' the data stack.

The user interface is deliberately kept simple. The generated code is relocatable, entirely independent of assumptions about resident software, and requires a contiguous memory area of sufficient size. This size will only be a few K words for modest programs, such that even the barest of configurations will be able to support PASCAL object code. The compiler itself, although not intended to run on the PDP 11, and therefore not 'compacted' at source level, consists of over 4,000 lines of source code and will compile to $\sim$28K words of PDP 11 code. For data allocation any contiguous area will do as well, but no space will be wasted if code and data area are juxtaposed. In other words, to load a PASCAL program just *one* memory area need be specified. The mapping of data structures and parameters onto PDP 11 memory is such that it should be trivial to interface PASCAL procedures with data supplied by a real-time environment.

AN APPRAISAL OF PASCAL

In this section we do not discuss the merits of a programming language, but simply make a few remarks that arose during the project. These remarks suggest possible extensions, the implementation of which might be undertaken in the future.

In the previous section we have already mentioned formal procedures/functions and a generalization of the function concept.

The compile-time knowledge of the size of data structures and the restriction of goto statements within a block do cater for a high degree of runtime efficiency, although the advantages have not been exploited fully: non-procedure inner blocks could have been incorporated in the language without any cost at all! Structured values (i.e. values that may be taken on by records or arrays) have a place in the language, and are already there if one considers how parameters are built on top of the stack.

It should be possible to pass arrays as parameters for which the structuring *is* but the bounds are *not* fixed at compile time.

It is unclear why arithmetic routines such as '*sin*' and '*ln*' should be supplied as standard procedures. In an acceptable implementation they may as well be declared as source code procedures.

SOME NOTES ON THE P-COMPILER

The PASCAL compiler for the PDP 11 was developed by adapting the P-compiler.[11]

The total effort spent on the development of the PDP 11 compiler amounts to less than 1 man year, the major part of which was contributed by the second author, who had no

previous experience in compiler construction, and hardly any in dealing with large programs.

Apparently the development of a compiler by modification of a 'master copy' is a highly successful approach. Nevertheless, we would like to make a few remarks that might be useful to those who would like to undertake similar projects.

(1) The *virtue* of the interpreter is the concise description of the instruction set of the P-machine.

An implementation of the interpreter on a machine other than one already providing an efficient PASCAL implementation gives rise to unacceptable overheads. (And then, why bother ?) The effort of generating assembler code in the P-compiler and loading that code in the interpreter does not seem worthwhile. Instead, the code might have been generated in such a form as to be more amenable to transformation for machines with different data formats.

(2) The documentation provided with the P-compiler and interpreter was mainly concerned with remarks on the object machine, the assembly code, the tapes containing the code and the character sets.

No documentation at all was provided with the compiler itself, as if the latter was to be considered as a box. This remark is not meant as a criticism of Mr. Amman's work, who prepared the compiler within a short period, but it points out one of the prerequisites if the process of compiler adaptation is to be successful. What we have in mind here is a document such as Reference 13 in which a clear description is given of an ALGOL W compiler.

Fortunately there were several aspects that enabled us, even without the suggested documents, to modify the existing compiler. To list these:

(a) The compiler is built up out of syntactic subroutines, definitely the most adaptable and transparent structure.

(b) The compiler proceeds in one pass, obviating the need to understand an interface between passes.

(c) The compiler is well written in a suitable language and therefore reasonably 'legible'.

Nevertheless, the missing documentation could have augmented the compiler in the following respects:

(i) A more extensive description of the data structures and their interdependence than can be derived from the scant comments in the compiler text.

(ii) A motivation of certain implementation choices to prevent implementors from starting these considerations all over again.

(iii) An outline of the philosophy for error reporting, such that any modifications may comply with that philosophy.

(iv) A description of the interfaces of the compiler procedures with their calling environment, i.e. the separation of responsibilities, in particular the extent to which the source code will have been scanned by each syntactic subroutine.

(v) The P-compiler, if intended for modification, could have been equipped with provisions in order to facilitate the generation of code for machines whose structure dictated different mapping for the data types.

(vi) On the positive side, let us remark that a master compiler generating code for a hypothetical machine does take away many of the worries about language mapping from the implementor, i.e. he does not need to worry about them if he does not want to.

NOTES ON THE ARCHITECTURE OF THE PDP 11

(This section is the sole responsibility of the first author.) Although it is not customary in the literature to discuss the merits of a hardware design in terms of its fitness to software implementation we feel urged to do so. The main reason being that a sufficient amount of such discussion may in the long run influence future designs.

First we note that the PDP 11 and PASCAL form a nearly ideal marriage. The remark is probably neither unique for the PDP 11 nor for PASCAL, but it shows that machines and languages, designed with an eye for each other, are able to meet each other somewhere halfway.

We will split our further comments into a section of praise and a section of criticism.

Praise

(1) The subroutine mechanism is a beauty. It is not unlike the one in some other stack-oriented machines (e.g. KDF 9, EL-X 8, DEC 10) but it should be pointed out anyway. It does the essential thing, *viz.* stacking of the return address, but nothing more.

This absence of further actions is *essential*, because it makes the subroutine mechanism *cheap*, and leaves any particulars of setting up an addressing environment to the software. We would like to make this remark because in a stack- and ALGOL-oriented machine like the B 6700 the only subroutine mechanism is the ALGOL-procedure call, which is much too circumstantial.

(2) The peculiar way in which the top of the return-address-stack (in the PDP 11) may be implemented as a specified register should *not* be recommended as a general vehicle for parameter transmission, but does serve well to pass compile time constants as parameters to subroutines that emulate instructions of the P-machine. Using this mechanism an utmost compaction of compiled code can be obtained.

(3) We do not need to comment on the addressing structure in general. It is an excellent example of how a short 'address field' in the instruction format may be employed to advantage. In this respect the PDP 8 with its asymmetric treatment of addresses is an open invitation to unclean programming.

Previously we discussed the advantages of not being pinpointed to one hardware stack, but having the liberty of implementing several.

(4) Indexing with both positive and negative index displacements is another feature that should be valued highly. It gives the implementor the freedom to address relative to a pointer, having come to rest at a natural location, instead of residing at a particular end of a store area.

This comment should have been superfluous, but, alas, too many of today's machines do *not* allow this freedom of indexing.

Criticism

(1) Whereas the addressing structure of the PDP 11 is quite advanced, the way in which the conditions arising on account of operations (positive, zero, carry, overflow, etc.) are handled is as old-fashioned as one can think of.

The only way in which these conditions can be 'tested' is by a large variety of conditional branch instructions. In no practical way can the condition values be

made available. This may be a minor nuisance in particular applications, but it is a serious drawback in the general implementation of logical expressions. This aspect of the machine had to be 'programmed around' in an unnatural and unsatisfactory way.

Suggestion: replace the wealth of conditional branches by the same wealth of operations that transfer the result of the specified combination of conditions in a standard form (say: 0 or 1) to a specified destination operand.

In order to allow branching, only 1 (or 2) conditional branches are required (say: a BRANCH FALSE and a BRANCH TRUE).

(2) We note an inconsistency in the treatment of the operand in the instructions JUMP and JUMP TO SUBROUTINE. This inconsistency is found in many present-day machines. The notion of *jumping to* makes the operand appear as a destination, whereas it actually is a source-operand in the assignment:

program counter := source.

Viewing the operand in this way the specification of an operand in register mode is quite natural.

Register deferred mode then adds a level of indirection. We have now been forced to use an auto-increment deferred mode to call procedure parameters because this mode has a higher degree of indirection.

(3) We found hardly any use for indexed deferred mode (in spite of its potential for parameter access) and lacked frequently the possibility of obtaining the address which is calculated in the course of indexed mode. For address calculation we had to resort to ADD instructions more than we liked. We wonder if other implementors would suggest, as we do, to decrease the level of indirection for indexed mode operands ?

(4) Byte-addressing is of limited usefulness when word boundaries pervade the use of the store so strongly.

CONCLUDING REMARKS

As work on the compiler described in this paper was in progress, another PASCAL project for the PDP 11 was brought to our attention.[14] Although at this time it is impossible to compare the two implementations, each appears to have merits of its own. Whereas Feiereisen's implementation is of use for stand-alone systems that are sufficiently equipped, our approach is directed to application on even the barest 11/10 configurations. The advantages of a cross-compiler approach, yielding adaptability and accessibility of the implementation, should not be underestimated.

Our approach has also made clear that the amount of labour involved in the construction of a compiler for a non-trivial programming language by adaptation of a master compiler may be considered negligible compared to current investments in *ab initio* developments. Prerequisites are that such a master compiler be written in a clear style, in a suitable language and with future adaptation in mind.

It has also been argued that a suitable choice of language and the absence of strict space requirements on the compiler itself allow for the generation of both compact and highly efficient code.

REFERENCES

1. C. Bron, 'Machinetaal—dode taal (machine language—extinct language)', *Informatie*, **14**, 376–382 (1972).
2. C. A. Lang, 'Languages for writing system programs', *Univ. Math. Lab., Cambridge* (October 1969).
3. E. W. Dijkstra, 'The humble programmer' (Turing lecture), *Comm. ACM*, **15**, 859–866 (1972).
4. *CORAL 66, Official Definition of*, HMSO, London, 1970.
5. N. Wirth, 'The programming language PASCAL', *Acta Informatica*, **1**, 35–63 (1971).
6. N. Wirth, 'The programming language PASCAL (revised Report)', *Berichte der Fachgruppe Computer Wissenschaften ETH Zürich*, **5** (1973).
7. A. N. Habermann, 'Critical comments on the programming language PASCAL', *Acta Informatica*, **3**, 47–57 (1973).
8. P. Desjardins, 'A PASCAL compiler for the XEROX Sigma 6: ACM SIGPLAN', *Notices*, **8**, No. 6, 37 (June 1973).
9. G. Friesland, C. O. Grosse-Lindeman, F. H. Lorenz, H. H. Nagel and P. J. Stirl, 'A PASCAL-compiler bootstrapped on a DECSystem 10', Proc 3. GI-Fachtagung über Programmiersprachen, Kiel, 1974, in *Lecture Notes in Computer Science* (Ed. B. Schlender and W. Frielinghaus), **7**, 101 (1974) (Springer Verlag, Berlin, Heidelberg, New York).
10. J. Welsh and C. Quinn, 'A PASCAL compiler for ICL 1900 Series computers', *Software—Practice and Experience*, **2**, 73–77 (1972).
11. U. Amman, 'The method of structured programming applied to the development of a compiler', in *Proc. ACM International Comp. Symp. Davos, 1973* (Ed. A. Günther, B. Levrat and H. Lipps), North Holland Publishing Co., 1974, p. 93.
12. B. A. Wichmann, *ALGOL 60 Compilation and Assessment*, Academic Press, London–New York, 1973, Table 19.
13. H. Bauer, S. Becker and S. Graham, 'ALGOL W implementation', *Report CS.98, Computer Science Department, Stanford University* (March 1968).
14. L. Feiereisen, 'Implementation of PASCAL on the PDP 11/45', *DECUS Conference*, Zürich, September 1974, p. 259.
15. D. A. Bell and B. A. Wichmann, 'An ALGOL-like assembly language for a small computer', *Software—Practice and Experience*, **1**, 61–72 (1971).
16. C. Bron, 'On complete specification in ALGOL', *Technical Report CB 54b*, Twente University of Technology (1972).