CRAY PASCAL

by

N. H. Madhavji and I. R. Wilson

Department of Computer Science

University of Manchester

M13 9PL U.K.

Abstract

This paper presents an investigation of the design decisions taken in the implementation of a compiler for Pascal on the CRAY-1 computer. The structured nature of Pascal statements and data structures is contrasted with the 'powerful computing engine' nature of the CRAY-1 hardware. The accepted views of Pascal as a simple one-pass language and the CRAY-1 as a vector processor are laid aside in favour of a multi-pass approach, taking account of the machine's scalar capabilities. The project as a whole, aims to produce highly efficient run-time code for applications likely to be programmed in Pascal. Some statistics are given to indicate the nature of such applications.

KEY WORDS: Compilation, Pascal, code optimisation, vector processors, CRAY-1

1. Introduction

CRAY Pascal - a multipass language on a scalar machine! This expletive is not intended to deny the simplicity of the design of Pascal or the vector processing capabilities of the CRAY-1. The power of this computer is often related to its performance improvement over earlier powerful scalar machines such as the CDC7600, for example Baskett and Keller[1] give a brief description of the machine and quote a ratio of a factor of two to three for scalar processes and two to ten for vector processes. However, as these authors state, viewing this machine purely as a vector processor does not do justice to its general purpose capabilities. Further, the extension of the use of vector processors into more general purpose environments, such as universities, shows benchmark results which resist improvement due to a base load of non-vectorisible work [2]. This becomes relatively more significant as the vectorisible component is more efficiently processed.

Pascal compilers are commonly based on one of the three compilers [3,4,5] originally implemented at Zurich, each for a different level of language provision. These compilers are single pass, with a recursive-descent recognition directly producing code for an interpretive or real machine. This reflects the simple LL(1) nature of the syntax and semantics of the language. However, a high degree of code optimisation necessitates knowledge of the complete program being compiled, which in turn implies a multipass compilation technique, see Wulf et. al. [6]. This is also recognised by Faiman and Kortesoja [7] who describe a Pascal compiler for the DEC10, which is among the notable exceptions to the simple Pascal compilers described above.

A statistical investigation of the nature of Pascal programs was carried out at an early stage of the CRAY Pascal project. This highlights the scalar nature of some Pascal programs, see Brookes [8]. Traditionally, most use of Pascal has been for non-numeric applications, such as system programming. Thus, gathering a significant representative sample of non-numeric programs involved little effort - whereas a corresponding sample of numeric applications was more difficult to obtain. A discussion of the results appears later, but it is interesting to note that there is statistical support for an intuitive feeling that the two application areas are different. This suggests that the vectorisible content of Pascal programs may be smaller than typical CRAY FORTRAN programs and that an examination of the relevance of other CRAY-1 features may be productive in this context.

2. CRAY-1 Features Concerning Pascal

A summary description of the CRAY-1 hardware is given in the Appendix, but some specific features concerning Pascal are of interest here. With high memory-to-register and register-to-register transfer rates, a Pascal program on the CRAY-1 is expected to achieve high execution speeds. However, the ratio of memory-to-register to register-to-register transfer speeds is 1:11. Therefore, it is important to minimise the number of store/loads, in a procedure, by devising a suitable register allocation scheme and by interleaving instructions so as to minimise 'hold' conditions arising from store accesses. This interleaving carried out by the compiler is called 'scheduling'.

With reference to the scalar capabilities of the CRAY-1, there are sixty-four intermediate B/T registers each. Hence, there is a possibility of utilising these registers as cache memory. Surprisingly, it will be shown that this number is more than the average number of variables declared in a procedure and comparable to the total number of variables declared in a program. Consequently, use of B/T registers to hold variable values is a major factor in reducing the number of store/loads required in a program.

With reference to the vector capabilities, there are eight vector registers, each of which has sixty-four 64-bit elements. While these are obviously utilised for vector processing, it is conceivable to utilise them as an 8 * 64 scalar cache memory. A closer investigation suggests that the benefits thus gained, compared to store/loads,

may not be significant, especially when the selection of a particular vector register is difficult and an A-register load is required. Also, a store/load may not result in a significant delay if data transfer overlaps with concurrent processing. This is a scheduling problem.

In small memory sized computers, generally there has to be a trade-off between run-time efficiency and compiler size/speed. Run-time efficiency generally induces complexity, which in turn increases compiler size and representation data size. In contrast, the 8 mbyte typical memory size of the CRAY-1 enables an optimising multi-pass Pascal compiler to be feasible. In general, this should assist in run-time efficiency of Pascal programs on the CRAY-1.

In addition to the parallelism implied in vector operations and autonomous store accesses, the CRAY-1 has independent pipelined functional units which may carry out several arithmetic and logical operations concurrently. The pipelining also permits each unit individually to accept functional requests at 12.5 nsec clock intervals even if the functional operation occupies several clocks. To take advantage of this. the compiler must further interleave the instructions by choosing registers and scheduling appropriately.

The high proportion of control transfers inherent in Pascal programs discussed later, suggests careful examination of the instruction buffering mechanism on the hardware. This is significant in size, the 256 instruction units being approximately five times that on a CDC7600. It also has a minimal structure consisting of four separately loaded buffers, but has no intelligence regarding choice of release, look-ahead filling or heuristics and each buffer must be treated as a contiguous unit. Clearly, loops which fit in total into these buffers and cover no more than four disjoint address spaces, will benefit from a lack of hold conditions arising from instruction store fetches. However, any deviation from these rules pays the penalty of apparently increased time to process a control transfer from a minimum in the case of a no-transfer conditional of 2 clocks to a worst case buffer 'thrashing' of 25 clocks.

3. Pascal Features Concerning the CRAY-1

3.1. Procedure Abstraction

With the provision of procedural abstraction facility in Pascal and a wide acceptance of structured programming principles, a large usage of procedures is expected in Pascal programs to be run on the CRAY-1, and indeed is later demonstrated. Hence, it is important to have an efficient procedure calling sequence. Also, in some cases it may be possible to perform macro-substitution. This could be a textual substitution at an early stage, or code substitution at a later stage. The principle advantages of the substitution would be a continuity in the usage of the scratch B/T registers and the elimination of the procedure entry/exit code sequence. On the other hand, a main disadvantage could be that there is no localisation of variables on which B/T cache register allocation is dependent. Thus, some registers could be artificially locked out.

Further, an increased size of the compiled code due to multiple substitution may cause a frequently obeyed sequence to overflow the instruction buffers.

In the CRAY-1 Pascal compiler (CP), the substitution of suitable procedures is neither performed by a preprocessor nor by the code generator. Instead, it is performed at the internal representation level, where there is a knowledge of the properties of the procedure concerned, and the cache register allocation scheme has already been applied.

3.2. Strongly Typed

Pascal is a strongly typed language. This enables compile-time selection of a suitable type of register for an operand, since conversion, sub-range information, etc. is known at compile-time. Consider the following declarations as an example:

```
var  I : 1..10; (* index variable *)
     B : boolean;
```

A 24-bit A-register is more suitable than a 64-bit S-register, given the bounds of variable I; whereas, an S-register is more suitable for variable B, since logical operations on the CRAY-1 are performed in these registers. This contrasts with the lack of such information in FORTRAN.

3.3. Recursion

While recursive procedures entail a stack mechanism, it will be shown in later discussion that recursive procedures are a small percentage of all procedures. For recursion, a structured machine, Morris and Ibbett [9], with a hardware stack is more suitable. However, the B/T register allocation scheme designed for the compiler emulates a structured machine. This scheme is based on processing a table of information on each procedure. After reserving some registers to reduce recursive call overheads and scratch registers to prolong the life of loaded operands, an appropriate number of B/T's are allocated to the main program and noted in the procedure/function table entry. The sub-procedure call tree is analysed to allocate registers at each node, with the last allocated to its parent node as a base. As a recursive procedure may call a static one, all the non-static procedures are examined for the calls of static ones, and a list of calls formed. Each element in the list is now processed as an independent tree.

3.4. Structured Objects

The ability to manipulate complete structures as comparisons and assignments suggests the possibility of vector operations, though not vectorisation in the sense of parallel arithmetic calculations.

4. The Nature of Pascal Programs

The analysis of the static nature of Pascal programs mentioned in the introduction, was carried out by modifying the Belfast ICL1900 Pascal compiler, described by Welsh and Quinn [10]. By replacing the code generation with frequency counting statements, a file of statistics on various programs could be built up. Brookes [8] describes this process in detail, but Table 1 shows some relevant measures.

	Pascal non-numeric	Pascal numeric
% assignment	36.9	39.3
% call	30.1	24.3
% if	12.6	10.9
% loops	3.8	10.0
% global variables	38.0	26.0
av. leading spaces	14.2	11.3
count of arrays	13.1	39.3
for : while&repeat	0.9	10.1
Min. % assignments	22.5	31.8
Max. % calls	55.4	32.3
INTEGERs : REALs	42.5	1.1
Max. av. leading spaces	29.9	12.8

Table 1 - Summary of Static Analysis Results

With reference to Table 1, it may be argued that the sample was not representative and that anything may be proved from an analysis of such measures. However, it may be reasonable to conclude that the more common non-numeric applications have fewer declared REALs, use fewer arrays, are more heavily procedured, use testing rather than counting loops and, by virtue of a greater level of indentation, would appear to be more logically complex. Such tentative conclusions would only be fully confirmed by a dynamic analysis of running applications on a CRAY-1, which the authors may not yet perform, though a similar analysis on compilers only, by Shimasaki et. al. [11], gives some support.

Taking the suggested nature of applications as a starting point, it is worth noting that the total number of scalars, for all procedures and the main program but excluding parameters, was 87.83 per program. The number of single word object parameters per procedure, was appproximately 4 and there were less than two declared scalars per procedure. Further, only approximately twelve percent of procedures appeared to be recursive and the ratio of all structured accesses to variable accesses was approximately 0.8.

It is suggested that the application profile emerging from this discussion, is one with a heavy use of procedures, which are largely non-recursive, with a significant proportion of control constructs and of structure accessing. Further, the number of variables is significantly fewer than one might expect in a language which is not block-structured and whose declarations are not mandatory, for example FORTRAN.

The need for careful optimisation of procedures is discussed elsewhere. The control construct nature of programs (gotos were non-existent) is beneficial in demonstrating control-flow, but emphasises the need for optimisation of jumps at co-incident construct boundaries. This is particularly true, in view of the control transfer times mentioned in Section 2.

The significant use of structure accesses suggests close attention be paid to common sub-expression analysis, as avoiding re-evaluation of recurring address calculations and indirections would be as or more critical than avoiding explicit arithmetic re-evaluations. Particularly as store access times

for indirection are significantly greater than functional arithmetic times. This type of optimisation is sometimes ignored on machines with sophisticated operand buffer mechanisms and hardware controlled cache stores, for example the local Manchester machine MU5 [9].

One other feature of Pascal programs which is of interest, is the distribution of literal sizes. The scalar enumeration types and subranges of Pascal encourage the implicit and explicit use of small positive constants, and a separate investigation by Wilson [12] suggests a distribution as shown in Figure 1.

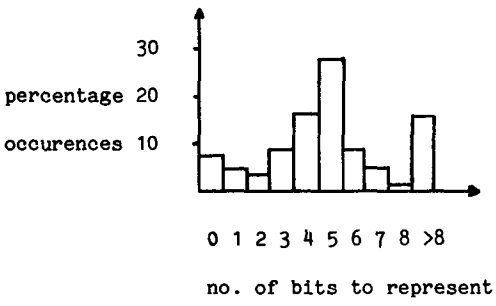

Figure 1 - Distribution of literal sizes

This distribution corresponds to the provision in the CRAY-1 hardware of a load 6-bit literal instruction.

5. Compilation Processes

It has been suggested by Fischer [13], that compilation itself is a vectorisible task, as a vector of text is matched with internal vectors (syntax, property lists, code skeletons etc.) to produce a vector of instructions. However, the complexities involved in optimisation and the limited nature of the vector operations available on current machines, do not appear to make practical use of this technique directly viable. This does not mean that the vector hardware is eschewed - three categories of use are apparent, viz. search operations, as an additional cache and for structured comparisons and assignments.

These techniques make it desirable to use operand manipulation in preference to control flow logic, where possible. Particularly in view of the jump times given above and the expected benefits of scheduling store/load waits. The recursive-descent technique, employed by most Pascal compilers, might appear to be critical in this respect. However, an analysis of the time spent in the recognition phase of the simplest of Pascal compilers by Oliviera [14] (see Table 2), shows this to be much less important than name look-up, for example. It should be noted that the measures in this table are, in fact, distorted by inefficient symbol table manipulation and the simplistic code generation required by the interpretive system which was analysed.

	percentage of compilation time
scanning	26
itemisation	9
symbol table	51
syntax recognition	2
error recovery	3
code generation	9

Table 2 - Performance of a simple compiler

Because of the necessity to open and close the scope of identifiers frequently (as record fields are selected and with statements applied), most Pascal compilers use a tree-structured symbol-table. This has the benefit of a logarithmic look-up time, but may involve significant control flow logic and typically five levels of tree-node (i.e. five store accesses and indirections). An analysis of hash table techniques given in Table 3, shows that the use of vector registers as hash tables, with any reasonable hash formula, would reduce the store accesses by a factor of four (to approximately one), while simplifying the control logic in the look-up process. Note the importance of this improvement due to the 1:11 ratio of register speed to memory speed mentioned earlier. In practice, record-field identifiers do not take advantage of this hashing, to reduce the overhead of scope open/close operations.

hash formula	av. no. of store accesses
a + b + c . . .	1.26 - 1.50
((a*2 + b)*2 + z)*2 + n	1.11 - 1.61
(. . .((a*2 + b)*2) . . .+ z)*2	1.73 - 2.16

Table 3 - Store Accesses Per Symbol Table Search

(here the name is abcd..z, length n and the hash value is taken modulo 64)

6. The Internal Representation

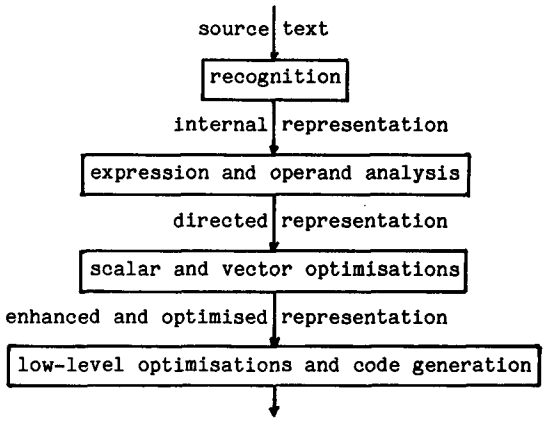

Figure 2 - Overall Operation of the Compiler

The internal representation referred to in Figure 2 is produced during recognition and is in five parts:

Procedure Table, Constant Pool, Property Lists, Statement Tree and Triple Vector.

The procedure table indicates whether a procedure is recursive, performs I/O, accesses non-locals (side effects), how many local variables of different types are present, etc.. The recursive nature is determined by a set of rules which guarantee that a recursive procedure is so marked. At the end of recognition, the register allocation algorithm described earlier is applied to associate the intermediate B/T registers to the procedures.

Some usage count information is retained in the procedure table. However, the structured nature of Pascal programs is used to avoid the necessity of GOTO analysis and determining areas of usage of variables as required in FORTRAN subprograms, which are textually larger.

It is hoped that the discrimination of different kinds of procedures (e.g. recursive, non-local accessing, etc.) will permit call overheads comparable with machine coding standards. This may not obviate the need for macro-substitution, where possible.

The control construct nature of a program is represented in a bracketted form within the expression triples, to assist low-level optimisations. However, the separate existence of the statement tree mitigates the disadvantages of the often quoted fixed nature of triples, Aho and Ullman [15], while maintaining the advantages of compactness. The level of operand representation is chosen such that two references to the same object (e.g. two matching variant fields or two named constants with the same value) are identical. This implies the use of actual offsets rather than property list addresses - though these are retained where a global analysis is required (e.g. vectorisation). This approach also contrasts with the register rather than operand representation used in CRAY-1 FORTRAN (CFT).

7. Optimisation of Structured/Dynamic Objects

Scalar optimisations are performed on the internal representation of a program (discussed in the previous section). In general, a program is optimised one procedure at a time. Within a given procedure, the flow of control information is appropriately represented in the triple vector. This drives the optimiser according to the properties of a control construct, and according to nested and sequential constructs.

On recognition of a control construct, a local table is initiated. This table holds the information about the operands in the construct, while the construct is being processed. Hence the table is referred to as the <operand,reference> (or <opd,ref>) table. Clearly, the <opd,ref> table behaves like a stack, as constructs are opened and closed.

An <opd,ref> table entry generally consists of an operand and its properties. The properties include the class of the operand; the class of register use; the reference of the operand in the triple vector; the availability, category, dependency, state and the usageorder of the operand and the dynamic linkages of the operand. This information, which is gathered while processing a construct, is also useful during the later phases of the compilation process (e.g. register allocation).

Associated with each triple is a row of entries in the usage-chain column for the left operand, the right operand and the complete triple. As each operand position can be distinguished within a row, the operand entries in the <opd,ref> table enable different operand-value usage-chains to be constructed. These chains are invaluable for detecting common sub-expressions, and are also useful during the later phases of compilation.

While a wide range of scalar optimisations are performed with these internal mechanisms, of particular interest are those which are performed as a result of the idiosyncrasies of Pascal: structured objects, dynamic objects, record

variants, pointers, VARiable parameters, functions and procedures with side-effects.

Unlike simple operands, complex operands (such as array elements, fields of records, complete structured objects, etc.) form separate triples. In order to detect common sub-expressions using such complex variables, the variables themselves participate in the detection process. For this purpose, structured objects maintain additional information in the <opd,ref> table.

7.1. Records and Dynamic Data Structures

For dynamic data structures, not only the references of individual elements but also the structural relationships have to be considered. The propagation and common sub-expression detection for these objects involve accessing the appropriate structures via static pointers. Consider the following dynamic variable access:

P↑.K↑.A

The internal representation of the relationships, emanating from the <opd,ref> table is as shown below:

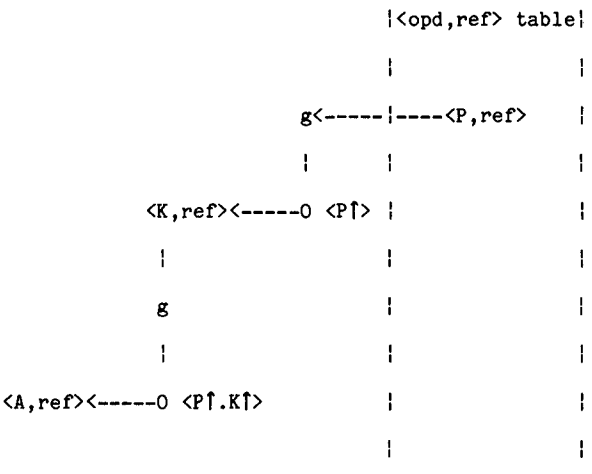

Figure 3 - <opd,ref> Table Dynamic Linkages

In the dynamic representation in Figure 3, <P,ref> is the entry for the static pointer P; g designates a GATE, to enable aliasing pointers to refer to a same dynamic object; 0 designates a dynamic record and <A,ref> is the field representation of object P↑.K↑.

Static record representation, unlike the dynamic one, is held directly in the <opd,ref> table. However, the representations for fields of a static record are held as dynamic linkages. Moreover, as there is no aliasing involved in this case, GATEs are not necessary.

7.1.1. Availability of Objects

On an assignment to a field, it obtains a right-hand-side reference. Further, the complete reference of the containing record is disabled, so that a next use of the record object is not detected as common to its previous definition. In essence, the record enters a new state. Consider the following example:

```
DOPRINT (P↑); (* value parameter *)
P↑.NAME := NEWNAME;
TEST := P↑ = Q↑;
```

As there is an intervening assignment to P↑.NAME, the detection of the second occurrence of object P↑ is inhibited. Instead, object P↑ obtains a new reference which can be propagated further. For an assigned field which is multiple levels within a structure, all the higher level containing structured objects are disabled. This higher level inhibition process is terminated on encountering a pointer. On the other hand, an assignment to a

complete record object implies copying of the knowledge of assigned-object fields. This technique enables the detection of the assigned-to object fields as common.

For pointer assignments, not only the right-hand-side reference is allocated, but structural changes also take place. For instance, new linkages are formed such that a common dynamic object is accessible via a common GATE. The implication of this is that, the usages of the same object referenced by aliasing pointers are detected as common.

Upon recognition of a triple for dynamic data structure, an appropriate structure is created, and if the structure is used (as opposed to assigned-to) then its reference is enabled. Subsequent use of this structure is detected as common to its first occurrence. The linkages of the structure exist in the local table for the duration of the current control construct. In theory, knowledge of suitable structures could be propagated into the outer loops, but this may introduce excessive compiler complexity.

Where side-effects are present, the precise nature of the affected pointers must be determined. For instance, assigning to a field could cause aliasing problems if another, apparently unrelated, pointer also has access to the field (e.g. non-local pointers). This is the case in the following example:

```
if <boolean-expression> then
 begin
   write(P↑.A);
   Q↑.A := maxint;
   Q := Q↑.NEXT;
   write(P↑.A)
 end(*if*);
```

Here, it may not be safe to detect the second occurrence of P↑.A as common, unless it is known that P and Q are not aliasing pointers. The dynamic linkages are of some assistance here.

In general, there are pointers whose access-right information is available, and these are termed 'GREEN pointers'; others are termed 'RED pointers'. The colour of a pointer can be deduced from the pointer entry in the local table. A GREEN pointer is allocated the highest dependency number (infinity); whereas a RED pointer is allocated a dependency number equivalent to its creation point in the triple vector.

Separate uses of RED pointers to an unknown alias structure, have no side-effects, but they may not be identified as common. As a result of an assignment to an object requiring an indirection through a RED pointer, all other RED objects are deleted. Thus, no affected RED object is wrongly detected as common. This is achieved by using the dependency values to limit the search region to the offending assignment point forward. In theory, this can be further optimised by utilising the type information gathered in the first pass. In addition, the assigned-to structure obtains the right-hand-side object colour.

7.1.2 Record Variants

An assignment to, or creation of a variant field may overwrite one or more existing variant fields, whose space ranges overlap the former field. With the knowledge of field displacement and size, overlapping fields can be deduced and erased. Since at a given time, only unique fields exist in the field representations, different variant fields of the same size and displacement are not distinguished internally. The effect of this technique may be compared to that of the optimising DEC10 Pascal compiler [7] by considering the following example text:

```
type  R = record
            A : integer;
            case B : boolean of
              true : (C1 : integer);
              false: (C2 : integer)
          end;
var  P, Q : ↑R;

     P := Q;
     P↑.C1 := 10;
     Q↑.C2 := 20;
     if P↑.C1 = 10 then ...
```

Whereas in the DEC10 compiler, in the last statement, P↑.C1 is equal to 10; in CP, P↑.C1 is equal to 20. Thus, CP enables the optimisation of record structures even in the presence of 'free unions'. This is particularly useful in some machine-dependent applications, where it may be necessary to map a value onto another one of a different type.

As the dynamic linkages (see Figure 3) simulate the run-time behaviour of dynamic data structures, it is possible to detect some run-time faults at compile-time, in a similar manner to assignments of variables with disjoint defined ranges. An example of the kind of dynamic data-structure fault detected, is accessing through a DISPOSEd structure.

7.2 Arrays

While fields of a record are represented in the dynamic linkages, it is not feasible to maintain linkages for array elements. The primary reason for this is that the subscripts themselves are variables, expressions, etc., making it not always possible to distinguish different elements. Thus, an element forms a separate entry in the local table.

On an assignment to an array element, its subscript expression may form an entry in the table and, if suitable, be detected as a common sub-expression. Nevertheless, an entry for the element is inserted in the local table. In order to prevent the detection of the complete array as common, the higher level structures are inhibited. Also, apparently different array elements are inhibited, since subscript expression values cannot always be determined at compile-time. This is achieved by giving a new reference to the array variable. This action is also taken when the complete array structure is assigned to. Thus, new array elements are not wrongly detected as common.

7.3 Variable Parameters and Function Assignments

Variable parameters are treated like assignments, since in general, they will be assigned to. The new reference of the affected variable can usually be determined after analysing the called procedure. This analysis induces significant complexity in the compiler mechanisms. A simpler alternative for processing the affected objects is to disable their references.

A similar complexity arises in disabling the non-local references made by a procedure after the point of call. The possibility of performing the necessary analysis is envisaged, and suitable information is collected for this purpose in the procedure table.

Clearly, for structured objects, this implies that both complete structure reference and element reference are inhibited. Thus the next use of the complete object or its element introduces the relevant new reference in the local table entry.

In the case of a function asssignment to an element of a structured object, the element reference is inhibited. As in other element assignments, the higher-level structures are also inhibited.

8. Code Generation

The enhanced and optimised representations mentioned in Section 6, are used in the final stages of compilation to perform code generation. This contains several areas with conflicting requirements. In particular, the optimal allocation of registers, interleaving of instruction sequences (to minimise waiting on the arrival of operands) and peephole optimisation of juxtaposed instructions, all conflict. Optimisation of register usage may clearly only be performed with real functional registers and the ordering of instructions is one factor in the choice of operand for register release. Contrarily, the allocation of real registers is an inhibiting factor on the scheduling process. Note the example below with operands from store, where fewer registers is a slower schema, but the cost of the use of an additional register is dependent on the surrounding code.

two register	start time	finish time
S1 = a	0	11
S2 = b	2	13
S1 = S1*S2	13 - wait for b	20
S2 = c	14	25
S1 = S1+S2	25 - wait for c	31

three register	start time	finish time
S1 = a	0	11
S2 = b	2	13
S3 = c	4	15
S1 = S1*S2	13 - wait for b	20
S1 = S1+S3	20 - wait for *	26

Figure 4 - 2/3 Register coding of a*b + c

CFT gives priority to scheduling by performing this with virtual registers, whereas CP allocates registers on the basis of a tree walk of the expression, leaving maximium freedom for later scheduling. Which technique is more beneficial is not immediately apparent and, no doubt, an

experimental analysis will lead to more interaction of the two requirements.

9. Conclusions and Acknowledgements

At the time of writing, the CP compiler is half-way through the development phase and has not been released for use. It is hoped to enhance the initial version of the compiler, with a vectorising section, for which hooks have been left, but no effort initially dedicated. This was a pragmatic decision, made on the basis of a lack of requirement in the bootstrap process, the interest in matching Pascal structuring to the CRAY-1 machine and the shortage of manpower.

Clearly, further statistical analysis of the use of structures and of dynamic data structures would be a desirable prerequisite to the design of further optimisation phases such as vectorisation. This analysis would preferably be of a dynamic rather than static nature.

We commenced the project fresh from a subset implementation for low-cost microprocessors whose store size is of the same order as the total CRAY-1 register size and also a P-code implementation. Thus, we are convinced that the only interesting numbers are 0, 1 and ∞.

Our thanks are due to CRAY RESEARCH, INC as a company, to Dick Hendrickson and his CFT team for intellectual and technical support and to CRAY-UK for their financial assistance.

10. References

[1] Baskett, F. and Keller, T.: An Evaluation of the CRAY-1 Computer. In: High Speed Computer and Algorithm Organization. Ed. by Kuck, D. J., et. al. Academic Press. 1977.

[2] Personal Communication with CDC and CRAY technical sales staff. 1981.

[3] Wirth, N.: Pascal-S A subset and its Implementation. ETH Zurich Report 12.

[4] Nori, K., et. al.: The Pascal <P> Implementation Notes. ETH Zurich. 1975.

[5] Wirth, N.: The Design of a Pascal Compiler. Software-Practice and Experience. Vol. 1, pp. 309. 1971.

[6] Wulf, W., et. al.: The Design of an Optimising Compiler. Elsevier. 1975.

[7] Faiman, R. and Kortesoja, A.: An Optimising Pascal Compiler. IEEE Trans. Software Eng. Vol. SE-6, No. 6, 1980.

[8] Brookes, G.: Compilation Analysis Techniques for Vector Processing Methods. MSc. Thesis. University of Manchester. 1981.

[9] Morris, D. and Ibbett, R.: The MU5 Computer System. Macmillan. 1979.

[10] Welsh, J. and Quinn, C.: A Pascal Compiler for the 1900 Series Computers. Software-Practice and Experience. Vol. 2, pp. 73, 1972.

[11] Shimasaki, M., et. al.: An Analysis of Pascal Programs in Compiler Writing. Software-Practice and Experience. Vol. 10, pp. 149, 1980.

[12] Wilson, I. R.: Pascal for School and Hobby Use. Software-Practice and Experience. Vol. 10, pp. 659, 1980.

[13] Fischer, C.: On Parsing and Compiling Arithmetic Expressions on Vector Computers. ACM Trans. on Prog. Lang. Vol. 2, pp. 203, 1980.

[14] Oliviera,J.: Pascal on Small Micro Computers. MSc. Thesis. University of Manchester. 1981.

[15] Aho, A. and Ullman, J.: Principles of Compiler Design. Addison Wesley. 1977.

Appendix

CRAY-1 Architecture

Figure 5 is reproduced from the CRAY-1 Hardware Reference Manual (CRAY RESEARCH INC HR0808) and shows the main architectural units of the CPU and the connecting data paths. The principle relevant features of the machine are:

- . up to 4M 64-bit words of memory
- . 137.5 ns memory access time
- . 12.5 ns clock period
- . 80 mflops instruction rate
- . 3 address instructions referring to registers
- . independent pipelined arithmetic units
- . 16 scalar/address registers
- . 128 scalar/address cache registers
- . 8 * 64word vector registers
- . vector instructions
- . 4 * 64parcel instruction buffers
- . overlapped instruction execution

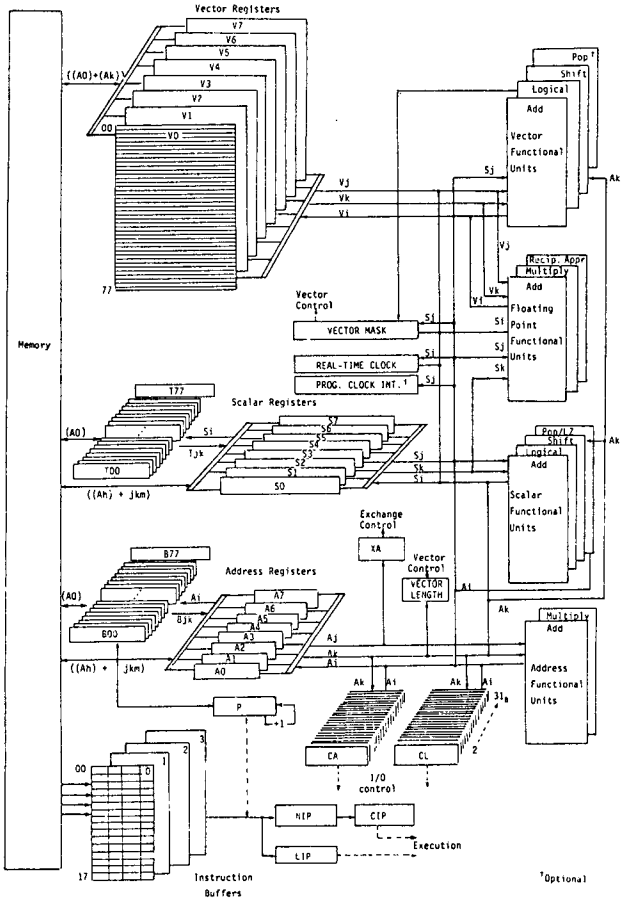

Figure 5 - Structure of the CRAY-1 Processor