University of London
Imperial College of Science & Technology
Department of Computing and Control

Pascal-orientated computer design

by

E.A. Schmitz

April 1980

A thesis submitted for the degree of Doctor of Philosophy of the
University of London and Diploma of Membership of Imperial College.

Abstract

This thesis is concerned with the problem of language orientated computer design from the perspective of intermediate language machines, i.e. the abstract machines defined by intermediate forms of compilation which can be, afterwards, either translated to target machine code or interpreted (by software or microprogram).

Two kinds of intermediate language machines are considered: the first is designed around a particular memory structure and the second is a more general machine which can be either software or hardware interpreted, or may be further translated prior to interpretation.

The first case examines the problem of designing an intermediate language machine for a subset of Pascal in which a special hardware memory structure is provided to match the requirements of the source language data and control structures. Since the mapping of the *full* Pascal data structures to a hardware mechanism is very complex an alternative solution using a descriptor mechanism is then presented.

The second case starts with an empirical study of Pascal programs in which a wide range of data about static form and dynamic behaviour of Pascal programs is collected and discussed. This data is afterwards used to evaluate the Pascal P4 intermediate language machine. From this evaluation the most expensive source language constructs are detected and alternative intermediate language primitives are suggested leading to an improved P4 machine.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Prof. David J. Howarth, whose insight in computer architecture combined with sure guidance helped me during all stages of this work.

I would also like to thank Bill Izatt for many stimulating discussions; Greg Pugh, Stuart McRae and Ed Davies for providing indispensable material and suggestions in the study of Pascal programs, and my fellow research students for suggestions and discussions.

This work was supported by a grant from the Conselho Nacional de Pesquisas and the Universidade Federal do Rio de Janeiro, Brasil.

Table of Contents

Appendix

Index of tables

Index of figures

Thirty spokes are made one by holes in a hub

By vacancies joining them for a wheel's use;

The use of clay in moulding pitchers

Comes from the hollow of its absence;

Doors, windows, in a house,

Are used for their emptiness;

Thus we are helped by what is not

To use what is.

Lao Tzu, 'Tao Te Ching', (XI)

Translated by W. Bynner

Chapter 1 - Introduction

1.1 Positioning the problem

The application of a computing system to the solution of a problem expressed in a high-level source language such as Fortran, Cobol or Pascal involves two different operations applied in sequence. Firstly, the source language statements require translation to some intermediate form of statements, an operation normally known as compilation. Secondly, the intermediate form statements require interpretation, a phase of operation known as execution or run-time.

In attempting to orientate the design of a computer towards the solution of problems expressed in a high-level language, we are therefore concerned with the overall cost-effectiveness of the translation and interpretation processes. The balance sought between these operations will clearly vary in different user environments. In a predominantly development environment such as a laboratory servicing classes of undergraduate students, the translation or compilation phase will be dominant. In a production environment, as in a data processing system in a bank, repeated interpretation of a limited number of programs will be the dominant process.

Given a particular source language, the translation and interpretation phases will be affected in different ways by the nature of the intermediate form of the program (the output of the translation and input for the interpretation) and by mechanisms used to achieve the transformation; either or both of these factors can be varied by the designer to achieve an optimum effect. We shall consider in the following chapters examples of optimization of each of these factors independently in one case considering the intermediate form of the program to be the instruction code of a computer with a store-processor structure designed to match the requirements of the source language, and in the other considering the nature of the intermediate form without analysing in detail the mechanism of the interpretation of this intermediate form.

The nature of the intermediate form of programs is influenced by the use to be made of the intermediate form. Example of different properties of intermediate forms, dictated by use, are:

1-source language independence: a common example is the machine code of a general purpose computer, compilers of all source languages producing a common machine code, subsequently interpreted by hardware at run-time.

2-interpreter independence: an example used subsequently in this thesis is the P4 intermediate form of Pascal source programs. The result of the compilation may be directly interpreted (for example by software or microcode interpretation on a variety of computers having different machine code or microcode properties) or may be subjected to further translation to a group of machine codes which are subject to hardware interpretation.

3-source language and interpreter independence: the classic example of such a universal intermediate form is the UNCOL form (Str58a).

4-source language dependence permitting (although not demanding) the use of a specialized interpreter : in this case the intermediate form acts as a natural interface in the overall system designed to execute programs expressed in a particular source language.

The work reported in this thesis partitions the overall problem and considers only intermediate forms having the properties of category 4 above. We postulate, therefore, an intermediate form specifically related to a single source language and which may be interpreted afterwards by either a particular hardware mechanism or (possibly after further translation) by a more general purpose hardware or microcode mechanism.

The translation phase is also affected, of course, by the mechanism of interpretation of the translator or compiler. It may be that the mechanism is identical to that used to interpret the intermediate code (eg. it is common to use the same machine for compilation and execution). If it should also be the case that the translator is written in the original source language, then overall optimization is achieved in the compilation phase by the optimization of the run-time characteristics. This is the case of one example considered in detail in this thesis, namely, the Pascal P4 compiler. If the translator cannot be written in the source language under consideration, the compilation phase becomes a separated exercise in the design of a system to handle this different form of problem specification. We shall not pursue further in the present thesis a separate study of the execution of the compilation phase; we shall however use "ease of translation" as a critical input to the design of the intermediate form.

We shall consider two different types of intermediate forms, one designed to be interpreted by a specifically designed hardware mechanism and one designed for more general interpretation. In the latter case, it would be desirable to analyse the effectiveness of different forms of interpretation, which would require a separate study in itself if we were to consider the variety of machinery available from different suppliers. In the present study, we limit our analysis of effectiveness of the intermediate form by analysing the limiting performance of a hypothetical interpreter, the limiting case being dictated by store occupancy and frequency of access to code and data. No attempt has been made to complete a thorough analysis of the effectiveness of the intermediate form when translated and/or interpreted on an existing general purpose computing system such as the IBM 370.

1.2 - Delimiting the problem

Our study is concerned with the design of high-level language orientated computers. We have approached the problem by considering different forms of intermediate languages which can afterwards be either interpreted directly or suffer a further translation to a different form to be executed. We have transformed this problem to the problem of finding an intermediate form defined by two constraints: it should offer a simple translation and its interpreter should be efficient at run-time.

The source language chosen for this study is Pascal. Pascal is a block structured high-level language providing a wide variety of control and data structuring methods which makes it suited for writing well-structured programs. Pascal has been experiencing widespread support and is being used in the programming spectrum ranging from teaching basic programming principles to sophisticated system applications. This fact confirms the correctness of the principles used by the designer N. Wirth, and gives support to the study of Pascal orientated machines. Part of the success of Pascal can be attributed to its well-defined and consistent definition, both at the syntactic and semantic levels. Good references to these can be found in Wir71a, Hoa73a and Jen75a.

The efficiency of the abstract machine defined by an intermediate form can be evaluated according to the resources consumed by the machine when executing a benchmark. The use of the resources implies a cost measure which is a function of a set of cost parameters. The cost parameters chosen in our case must be, preferably, implementation independent; this lead us to choose parameters based on memory utilization instead of time or speed which are dependent on low-level implementation and operating system behaviour. The set of cost parameters has static and dynamic components, the static ones measuring code and data occupancy and the dynamic parameters measure the information traffic during program execution. More explicitly the cost parameters used, denoted by a_i $(1 \le i \le 6)$ are the same as the set used by Wortman (Wor72a) in his study of a Student-Pl machine, and can be defined as:

a1-the number of bits required to represent instructions
a2-the number of bits required to represent data
a3-the number of memory references to fetch instructions at run-time
a4-the number of memory references to access data (load or store) at run-time
a5-the number of bits of instruction fetched during program execution
a6-the number of bits of data accessed during program execution.

The evaluation of these cost parameters involves using a given estimative of the workload to which the machine is going to be submitted. There are two approaches to this measuring:

i-direct measurement: in this case the compiler (from the source language to the intermediate form) and the interpreter are modified to provide direct monitoring information about data and code usage when running the benchmark.

ii-indirect measurement: the composition of the benchmark is analysed in terms of source language constructs (Wor72a). With the aid of the code generation patterns for these constructs and the knowledge of the static and dynamic frequency of appearance of these constructs the cost measure can be estimated without running the benchmark. This method can be very useful in initial design phases if the frequency of usage of these constructs is known since it offers the advantage of accessing performance without the need for constructing a compiler and interpreter.

1.3 Related work

One of the first reported contributions to the study of intermediate languages is given by Randell and Russell (Ran64a) in their description of the intermediate language machine for Algol-60 to run in the KDF-9. Their proposals had an influence in the Burroughs B6700 and some ideas are found in the Atlas computer.

The use of abstract intermediate language machines as one way of writing portable compilers is a common practice in compiler writing. The compiler is divided in two parts: the first of which is source language dependent and the second of which is interpreter or machine dependent; the interface between the two parts being an abstract machine. This approach is used in several compilers, such as BCPL, Algol 68C and Pascal _P_. Compiler portability is achieved by writing one translator from the intermediate code to the target machine code. The BCPL intermediate code machine is called OCODE (Ric71a) and is a zero address stack based machine. Since BCPL allows access only to variables in the current procedure or global variables, the addressing mechanism needs only two registers: one to the base of the local stack and one to the global area. Further simplifications in the OCODE machine come from the fact there is only one data size in BCPL and the language does not allow dynamic creation of objects. The Algol 68C (Bau73a) Z-Code machine is a one address machine with a set of registers. The set of instructions provide register-register and register-storage instructions and is orientated to interpretation in the IBM 370.

The Pascal P4 compiler is a portable compiler for a subset of standard Pascal (Jen73a, Nor74a). The compiler generates code for an abstract intermediate language machine. The P4 machine is an almost pure stack machine whose design constraints where both simplicity of compilation and interpreter efficiency. The machine is considerably more complex than OCODE due to Pascal rules for variable accessing, dynamic creation of objects and different data sizes. This compiler has been implemented in a wide range of machines from Cray-1 to microcomputers. A more detailed description of it is made in chapter 5.

N. Wirth describes in his book "Algorithms+Data Structures= Programs" an interpreter for an intermediate language used in the compilation of PL/0. We have used this work as a basis for our experiment with the extended PL/0 machine described in chapter 2. The PL/0 machine described by Wirth is a simple, pure stack machine reflecting some of the ideas of the P4 machine.

The "Basic Language Machine" is an attempt at designing a computer architecture to suit a given language. The approach taken by Iliffe (Ili68a) is the design of a conceptual storage structure to meet the requirements of a language to be used in systems and application tasks. The storage is organized as a tree in which the storage elements are grouped together in sets of various types. The sets are linked to each other by structural information called "codewords", which are also grouped into sets. There are some real machines which have incorporated concepts derived from the study of intermediate language machines. The Burroughs 6500 is the first realization of one architecture to solve the weaknesses of conventional architectures for handling languages like Algol-60 where dynamic storage allocation is a language property. The main problem posed by Algol-60, i.e. the formation of addresses at run-time and the maintenance of procedural history are elegantly solved by the use of the display and the B-6500 stack organisation (How76a, Hau68a).

Two of the main design objectives in the design of the ICL-2900 were related to the matching of high-level language characteristics: efficiency in handling code from several high-level languages and capability for handling dynamic code and data structures. In other words, the ICL 2900 was designed to act as an intermediate language machine for various source languages, (Buc78a and How76a).

The Burroughs B1700 system (Wil72b) is aimed to work as a universal intermediate level machine. The machine does not possess a fixed instruction set but allows the possibility of every application defining its required instruction set and addressing primitives into what is called a S-language, which is then interpreted by changeable microprogrammed emulators.

Wortman presents in his Ph.D thesis (Wor72a) one instance of the whole process of language oriented computer design. The source language is Student-PL, a dialect of Pl-1, used for teaching purposes at Stanford University. The work has two main parts: first is the definition and refinement of a Student-Pl machine; second a comparison of the efficiency of this machine against the IBM 360. Wortman's evaluation technique is an extension of Wichman's method (Wic69a, Wic70a, Wic71a) for comparing Algol-60 implementations. Wichman's work indicates that terms of the cost measure used in the evaluation can be associated with statements in the source language and that it is a useful way to characterize machine performance. Wortman extended this work by relating the cost parameters used in the evaluation with language fragments which often constitute only parts of statements. The defintion of which fragments to use in the evaluation depends on the source language, the cost measure function being used and the implementations being compared. The basic idea is to choose enough small fragments so that every cost parameter can be uniquely associated with a set of language fragments. According to Wortman, there are two conditions for choosing the fragments:

a-each fragment must be mapped to a non-overlapping sequence
of object code instructions
b-it should not contain data dependent loops.

This evaluation technique will be used in chapters 5 and 6 of this thesis.

1.4 Method

We present in this thesis two approaches to the study of Pascal orientated intermediate language machines. The first is presented in chapter 2 and the second is covered in chapters 4, 5 and 6.

The first method deals with the derivation of an intermediate language machine, whose primitives are built using a specifically designed memory structure. We start by considering two aspects of the problem: the derivation of intermediate language primitives for control statements and primitives for language data structures. The advantage of this approach is that we can work with two different aspects of the source language; in the first case we study the problem of implementing procedure calls, loop handling and expression evaluation without considerations about data; in a second stage we study the problem of mapping data structures independent of control sequence. In both cases the requirements of both control and data structures are translated in terms of abstract data structures. These abstract data structures are then translated in terms of special-purpose memory systems, such as specially designed shift-registers or random-access memories with automatic indexing capabilities. This technique can be seen as an attempt to bridge the gap between the definition of intermediate languages and the possibilities of large-scale integration for implementing complex but repetitive hardware structures. One way of achieving simpler intermediate forms is by the use of more sophisticated hardware memory structures, which is an exact parallel of the case of introducing hardware primitives for real arithmetic instead of the painstakingly interpretation of these using simple arithmetic for integers.

A method for deriving intermediate language machines which are more general purpose and more independent of its mode of interpretation is presented in chapters 4, 5 and 6. The method is based on two simple principles: the first is that the process of deriving an intermediate form is essentially iterative, i.e. it may have to be repeated several times until the desired results are obtained; the second is the principle that the intermediate language machine has to be evaluated and improved according to the workload to which the intermediate language machine is going to be submitted. This method has four distinct phases:

i-start with the definition of an "easy to compile" intermediate form. In our case we start with one already defined intermediate form - the P4 intermediate form used by the Pascal P4 compiler.

ii-a study of an advanced workload is made, from which the characteristics of the advanced workload are collected. This information is to be used both for evaluation and improvement of the intermediate form machine. The key to this evaluation technique is the concept of language fragments (Wor72a).

iii-the evaluation of the cost measure is made using the static and dynamic distribution of fragments obtained in phase ii. Determine which are the language fragments which use most of the resources.

iv-alternative intermediate form primitives for the mapping of these fragments are suggested. The data about frequency usage of language fragments is now used to evaluate the effect of these alternative strategies in the cost measure. This process can be repeated several times until a satisfactory cost measure is obtained. In a latter phase, not developed in this work, the data about fragments usage can be used to compare the resulting intermediate form with other forms using different hardware bases.

1.5 Thesis composition

In chapter 2 we consider the problem of deriving an intermediate language machine for a subset of Pascal. The control and data structures are analysed and implemented with special purpose hardware mechanisms. The source language is an extended version of the PL/0 language with additional control and data structures.

Chapter 3 presents a study of the problem of defining intermediate language primitives for the implementation of Pascal data structures. The resulting technique is a descriptor mechanism using a set of descriptor operators and descriptor formats.

Chapter 4 presents the result of our analysis of form and behaviour of Pascal programs. From this study, among other data, we have collected static and dynamic properties which permit evaluation of different implementations of Pascal orientated machine, and it will be used to evaluate and improve the Pascal P4 intermediate language machine.

In chapters 5 and 6 we study one particular intermediate language machine - the P4-machine. We start by evaluating the P4 machine using the data collected in chapter 4. The result of this evaluation is used to detect the areas of the P4-machine which use most of the resources. Alternative constructs are suggested and the overall improvement measured.

2 - The EPL/0 machine

2.1 - Introduction

The implementation of a high-level language on a real or abstract machine via a compiler involves the implementation of two different language aspects:

a - the language control structure, comprising the set of primitives for mapping selection, repetition and procedure call statements.

b - the language data structures, comprising the representation of primitive data types and the provision for implementing the methods for data structuring provided by the language.

Implementing each one of the features in the first group above involves the use of some storage space; which has some properties defined by the language rules and some defined by the method chosen by the implementor to execute the translation procedure. For example, the implementation of procedure and function calls need some area of storage to be used to store control information. The type of information and its structure depend on language rules - e.g. it will depend on whether the language allows procedures to be nested or recursive or permits procedures to be passed as parameters (McKe75a). These features would imply, for example:

a - if no recursion is allowed, storage for data areas can be allocated when the program is loaded.

b if no nesting is permitted, the addressing will be reduced to local and global variables.

c - if the language does not allow parametric procedures, all calls will preserve the current scope of addressing apart from the data area of the called procedure, i.e. only one change in the scope is made.

We can imagine that the area of storage used for the support of procedure implementation forms a data structure, whose actual form depends on language rules and compiler strategy. The same concept can be applied to the rest of the control primitives. The storage area needed for the implementation of language data structures can also be thought of as a data structure. As in the case of procedures, there are several language parameters which can influence the run-time data structure, including:

a - if the type definitions can be nested

b - if the size of data objects can vary at run time.

c - if new objects can be generated at run time.

One approach to the problem of language oriented computer design is through the direct hardware implementation of the run time control and data structures. A good example is the display mechanism of the Burroughs B6700, which is a hardware implementation of part of the data structure required for keeping the run-time addressing environment in a block structured language. This partial data structure is an array of addresses pointing to the data areas accessible to the running procedure (or block) which is implemented as a set of fast registers. This idea could be extended to cover not only the control activities (like return addresses, memory allocation) but also storage and access of structured variables, e.g. arrays could be stored and accessed in a special memory for arrays, records would have their special memory etc.

This chapter describes one exercise in language oriented machine design, based on the ideas presented above. The work consists of four parts:

a - the design of the specialized memory structures to meet the language requirements.

b - design of the intermediate language which uses the specialized memory primitives.

c - write a compiler and interpreter to verify these concepts

d - run a test batch to collect some machine statistics and study its behaviour.

We have chosen for this experiment a subset of Pascal called PL/0 (Wir76a) which was extended to provide additional control statements and data types. We have chosen this extended version of PL/0 as it provides a realistic language on which to demonstrate the design method, while remaining simple enough to be analysed and tested without excessive effort.

2.2 - The PL/0 language and its extensions.

The PL/0 language was created by N.Wirth (Wir76a) for the purpose of teaching compiler techniques. The design constraints for this language were that it should be "small" enough for its compiler to be presented in a book and sufficiently complex to expose the basic concepts of compilation. It can be thought as a simplified version of Pascal designed for compiler teaching purposes.

The original version of PL/0 contains the basic control statements for selection and repetition: if-then and while statements. It also provides assignments and procedure calls. The procedure definition can be nested and procedures can be recursive. The only data type offered is integer.

The original PL/0 definition has been extended in the present work and it will be referred to as EPL/0 from now on to differentiate it from the original version.

The modifications introduced are:

a - inclusion of the for, repeat and case statements.

b - the procedure definition can specify value parameters.

c - two new data types: array and stack.

Arrays are of type integer and the lower bound is always zero. Stacks, corresponding to the common last-in first-out structure have a base type integer. A variable which is declared of type stack can appear either in expressions or in the left-hand side of assignments; if inside an expression every reference to it implies that an element in the top of this stack is read out, while if in the left hand side the result of the expression is inserted on top of the stack. There is also a primitive called - empty(s) which returns the value 1 (there are no booleans in PL/0) in the case when the stack denoted by the parameter s is empty, otherwise returns 0.

The main reason for the introduction of stacks as a language feature (which makes it not a true subset of Pascal) is for testing the possibility of designing memory structures for matching abstract data structures. The only way of using this data structure in a language like Pascal is to introduce it as a type to be used by the programmer at variable definition time.

A syntactic flowgraph of the extended version of PL/0 is presented in Appendix 4.

2.3 - The extended PL/0 machine

The design of the extended PL/0 machine followed the principle, outlined in section 2.1, that the structure of a machine oriented towards a high level language can be derived from the analysis of the run time requirements for implementation of that language. The run time requirements are first expressed in terms of abstract data structures which are then implemented by specialized memory devices. Each one of these specialized memory devices will have a set of primitives upon which the instruction set is defined, with each instruction expressed as a combination of these primitives.

The first step in our design for an extended PL/0 machine is the study of the requirements for the memory system. We have already identified two basic structure classes: the language data structures and the control structures, so we assume that the machine needs two memory systems:

a - the data memory: to store the local and global variables simple or structured and support their access methods.

b - the control memory: to store the control information needed for supporting data and code addressing in a nested, recursive procedure environment and provide a mechanism for expression evaluation.

2.3.1 - The data memory system

The data memory system implements the data structures needed for supporting the storage and access of simple and structured variables. The problems arising from the use of this memory system in a dynamic allocation scheme do not affect primarily the memory structure, but only its addressing, which will be dealt with in the next section.

Extended PL/0 has one simple type - <u>integer</u> and two structured types <u>array</u> and <u>stack</u>, both of base type integer. This lead us naturally to the subdivision of the data memory system in three subsystems:

- the integer memory to store integer variables
- the array memory to implement array variables storage and access.
- the stack memory to implement stack variables storage and access.

2.3.1.1 - The integer memory

The integer memory, referred to as IM is, in abstract, an array of integers, the array upper bound being the memory size. An array is mapped directly in hardware to a random access memory. There are two primitives defined for accessing the integer memory:

- readint (absadr): read the contents of the integer memory whose address is absadr.
- writeint (absadr): write the contents of the memory buffer into the address absadr.

The absolute address utilized by the primitives is the result of the translation of the address couple in the display memory, discussed in section 2.3.2.5. The integer memory and its primitives are presented formally using Pascal notation in table 2.1.

2.3.1.2 - The array memory

The array memory stores the local and global variables of type array and provide a simple mechanism for array element access. Individual components of an array variable are denoted by a selector of the form: x[i]. An access to an array element involves two main actions:

a - check if index _i_ is in the array range

b - evaluate the address of x[i].

In some machines a dedicated register, the index register, is used to mechanize the second operation. This operation involves setting the index register to the value of the index _i_ and loading the value of the array base address to the accumulator, the value of the address of x[i] being automatically generated. The actions for range checking the index are left to the programmer.*

A special memory system can be devised in such a way that both operations are executed by the memory system itself. One solution is the specification of an array memory with the following components: (Fig 2.1)

a - one random access memory AD for storing the array data.

b - a random access memory MA for storing the addresses of arrays in AD.

c - one active component for adding and comparing addresses.

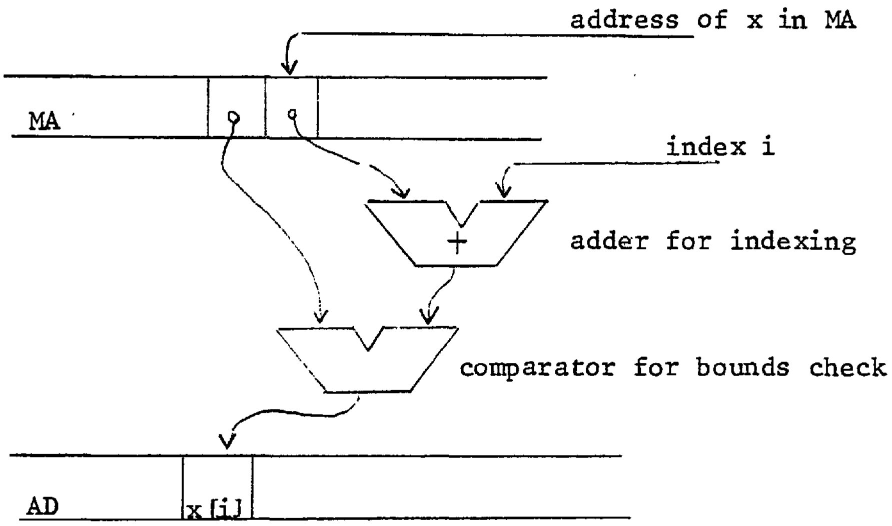


Figure 2.1-Array memory lay-out

The operation is as follows: the address of array x is an entry to MA; the system reads the address of array x in AD and reads the base address of the successor of x in MA, which enables the system to execute

---

* Footnote Machines with descriptor mechanisms can do both actions simultaneously, see chapter 3 for discussion of this alternative solution .

bounds checking. A formal description of the array memory is given in table 2.2.

2.3.1.3 - The stack memory

The design of the stack memory followed the same ideas used in design of the array memory. Stacks are usually simulated in random access memories as arrays with implicit indices, which are incremented or decremented according to the operation being executed. Complex problems of space management appear when more than two stacks are required to share a limited memory area (Knu68a).

A single stack can be implemented by a shift-register on which the primitives shift-right and shift-left are used to map the stack primitives pop and push. However, a problem appears when more than one stack has to share the same physical device, since the normal shift-register has only its extreme elements accessible. One technique for solving this problem is by defining a shift-register in which every position is addressable and the primitives shift-right and shift-left are changed to:

a - push from x: elements with address greater or equal to x are shifted one position. The position x is written with the value of the memory buffer mbr.

b - pop from x: the position x in the shift-register is read out and all elements with address greater than x are shifted one position.

As in the case of arrays, two operations have to be implemented when accessing an element of a stack variable:

a - locate the top of stack in the stack memory

b - check if the stack is empty

For performing these two operations another memory, the mark-stack

MS is introduced in addition to the storage for the stack variables SM. The mark-stack or MS is a simple array of pointers each of which contains the address of a particular stack variable. The address of the top element of a particular stack is obtained by relocating the stack address as known in the program in the mark-stack; i.e. the variable index in the primitives described in table 2.3 can be defined as:

index: = MS [stackaddress]; where stackaddress is obtained after relocation of the stack address couple in the display memory.

Since after any stack operation occurs, all the addresses of other stacks will change, a primitive operation, called fixmarkstack, is to be incorporated in MS to execute this correction automatically.

Table 2.3 defines the memory subsystems for the stack memory. Since we are using a sequential language for describing parallel operations, stacks in table 2.3 are represented as arrays with the sequential for loop in the primitives pushfrom, popfrom and fixmarkstack denoting an operation which is to be executed in parallel. Table 2.3 also defines three high-level primitives which will be used directly in the instruction set:

a - pop(absadr): to read a stack element

b - push(absadr): to write on a stack variable

c - empty(absadr): to test if the stack is empty

2.3.2 - The control memory system

The basic requirement for the control memory is the supporting of the implementation of the control structure of the language. This implies in mechanisms for the support of the selection (if-then and case statements), repetition (while, repeat and for statements) and abstraction statements (procedure and function calls). The selection and repetition statements require very simple control mechanisms, which are reduced to expression evaluation and code jumps. Expression evaluation is needed in assignment and procedure calls (in parameter passing) and a special memory is assigned to it.

Most of the other requirements for the control memory stem from the fact that extended PL/0 procedures can be nested and recursive. Procedure calls in a static language like Fortran can be implemented very simply by storing the return address in the body of the called procedure - a solution which can not be used in EPL/0 since procedures are recursive; the mechanism for implementing return address requirements is the procedure linkage memory.

Other requirements come from data addressing in an environment where procedures can be nested and recursive. Two different mechanisms are needed -the first to manage the creation of procedure data areas called the mark-data-area memory; the second for the support of data addressing mechanism and it is called the display memory.

In the description of the control memory system we shall make frequent use of the stack data structure. To simplify the description we shall use a type stack having the same properties as the type stack in EPL/0.

The control memory system contains four memory subsystems:

1 - the expression evaluation memory

2 - the procedure linkage memory

3 - the mark-data memory

4 - the display memory

2.3.2.1 - The expression evaluation memory

The data structure needed for implementing expression evaluation depends on the algorithm chosen by the implementor to translate expressions (and obviously the language rules). The simplest form of translating expressions is by transforming the expression to a reverse polish format which is then executed by using a stack and reverse polish (postfix) operators in the order code. In normal stack machine implementations the evaluation stack coalesces with the data storage stack, although this is not a necessary condition since the data storage stack is used as a stack for block allocation of variables which are subsequently accessed, not in the last-in first-out manner but in a random-access way. We assume the expression evaluation memory EM to be a stack on which the following primitives are defined:

a - pushem: push mbr in expression memory

b - popem: read top of expression stack to mbr.

c - literal (value): push value to expression stack

d - operator (op): execute operation defined by op with the two elements in the top of EM and return result to EM.

A definition of the expression memory and its primitives is presented in table 2.4.

2.3.2.2 - The procedure linkage memory LM

Every time a procedure is called, the information about the return address - which is the actual value of the program counter PC (in the extended PL/0 machine) must be saved. In a normal implementation this information would be saved in the activation record of the called procedure and restored on return. Return addresses of procedures are naturally accessed in a last-in first-out manner, so the abstract data structure in this case is the stack.

The linkage memory can be used for loop control, and in the extended PL/0 implementation it is used for storing information used in the execution of the for statement.

2.3.2.2 - The mark-data-area memory

Extended PL/0, like Pascal, allows procedures to be recursive.

This implies that procedure data areas can not be allocated at load time; instead they must be allocated at procedure entry time and released at exit time. Since the last data area to be allocated is the first to be released the addresses of active data areas form a stack which is called MD in the EPL/0 machine.

In the extended PL/0 machine there are three different data memories, one for each type, so each entry in the stack which holds the address of the active data areas should contain three pointers, one to each of the specialized data memories: integer, stack and array. The format of each entry can be defined by the type declaration datapointer in table 2.6. We also define a register TOP to point the start of the free space in the data memories. The management of the data areas is executed by the primitives pushmd and popmd.

2.3.2.5 - The display memory

The addressing of global data areas, i.e. addressing variables of procedures in levels of nesting less than the current procedure, can be achieved via the display. The display is simply an array of pointers to the areas of the procedures which are accessible to the one which is currently in execution. If the maximum level of nesting allowed is maxnest then a simple display can be defined as:

```
DM: array [1..maxnest] of datapointer;
```

When a procedure is called the display memory must be updated. If the language does not allow procedures to be passed as parameters, then this implies that the called procedure must be in the scope of the caller. The implication is that the display is already set, apart from the entry corresponding to the data area of the called procedure. If the lexical level of the caller is m and the lexical level of the called procedure is n ($1 \leq n \leq m+1$) then the actions at procedure entry and exit are:

at entry: save DM[n] (the entry to be changed)
replace DM[n] by current stack marker
at exit: restore DM[n] to old value.

Since the first exit must correspond to the last entry, then a stack is the adequate structure for storing the entries of the display to be saved. The simplest solution is to define each entry in the display to be a *stack*, such that the contents of each element can be automatically saved. With this refinement the definition of display becomes:

```
DM: array [1..Maxnest] of
    stack of datapointer;
```

The set of primitives operating on *DM* (table 2.8) simplifies the display maintenance, which becomes:

at entry: execute 'pushdisplay(n)' where n is the level of the called procedure, this will automatically save the old display entry and load the address of the free area to DM[n],
at exit: execute popdisplay(level) to restore old data area pointer.

Each extended PL/O variable is defined by a triple (type, level, offset), with the variable type embedded in the instruction format. The absolute address for an element of type integer is formed by the primitive:

```
procedure intaddress(level, offset)
begin (*evaluate absolute address*)
absadr: = DM[level]. intpointer + offset
end;
```

The same applies for stacks and arrays.

The extended PL/O instructions set (table 2.8) is built with the set of memory primitives defined for data and control memory systems. Fig. 2.2 shows the relation between the various system components.

2.4 - The experiment

A complete system was designed to test, simulate and perform measurements in the extended PL/0 machine defined above. The system consists of an extended PL/0 compiler and interpreter both written in Pascal. The compiler translates extended PL/0 programs to EPL/0 machine code. Monitoring instructions embedded in the compiler and interpreter are used to collect data about source text composition, the code generated and run-time machine characteristics.

Two experiments were made in the simulated version of the EPL/0 machine:

a - a set of 28 procedures was collected from Wir76a and Wir73a. These procedures were coded in EPL/0, compiled and run in the system.

b - a subset of the test batch above, consisting of five sort procedures was coded and run in the original PL/0 machine, as defined in Wir76a. Both machines are compared in terms of the number of instructions needed for coding and running the sort algorithms.

The information resulting from the above experiments, although of limited scope, can be used to check the correctness of the design principles and suggest improvements in the EPL/0 machine to match language usage requirements.

2.5 - Results

The data gathered in the first part of the experiment can be divided in three groups:

a - data about EPL/0 program composition: tables 2.9 and 2.10 display the information about frequency of use of source language constructs and operators, while table 2.11 displays EPL/0 procedure data area size statistics.

b - data about the code generated - tables 2.12 and 2.13 contain the static and dynamic distribution of EPL/0 machine instruction usage.

c - data about EPL/0 machine memory behaviour: table 2.14 displays the frequency of use of each one of the components of the EPL/0 memory system.

The results of the experiment of running five sort procedures in both the EPL/0 machine and the original PL/0 is presented in table 2.15.

2.6 - Conclusions.

This chapter describes an attempt to design an intermediate language machine around a specialized memory system. The memory system primitives are defined to match the source language data and control requirements.

The following points should be noted concerning the mapping of language data structures:

a - there is a definite improvement in data access efficiency combined with a simplification in code generation.

b - the use of a special memory for types which are not in the language definition, such as stacks, although offering some implementation difficulties can bring considerable gains. However, to be used efficiently, these types must be embedded in the language.

c - there is an overhead in the control memory system incurred in the management of different data memories. As a consequence the data space required by the display and mark-data memories is trebled in the design considered here.

d - there will be difficulties when we try to apply this concept to the full data structuring methods provided by Pascal. This is due to fact that Pascal types can be nested, implying that the access technique for one type cannot be used for another.

In the case of the control memory we observe that:

a - there is a significant improvement in compiler simplicity and in the size of the generated code achieved by the use of a memory system orientated towards the control requirements of the source language. In this specific case, the control memory has two main components one for expression evaluation and a second for supporting the procedure call mechanism; these are the most used language features (see table 2.9).

b - the components of the control memory can be implemented by cheap, large sequential memories thus providing a fast and simple solution to control structure mapping without any loss in generality.

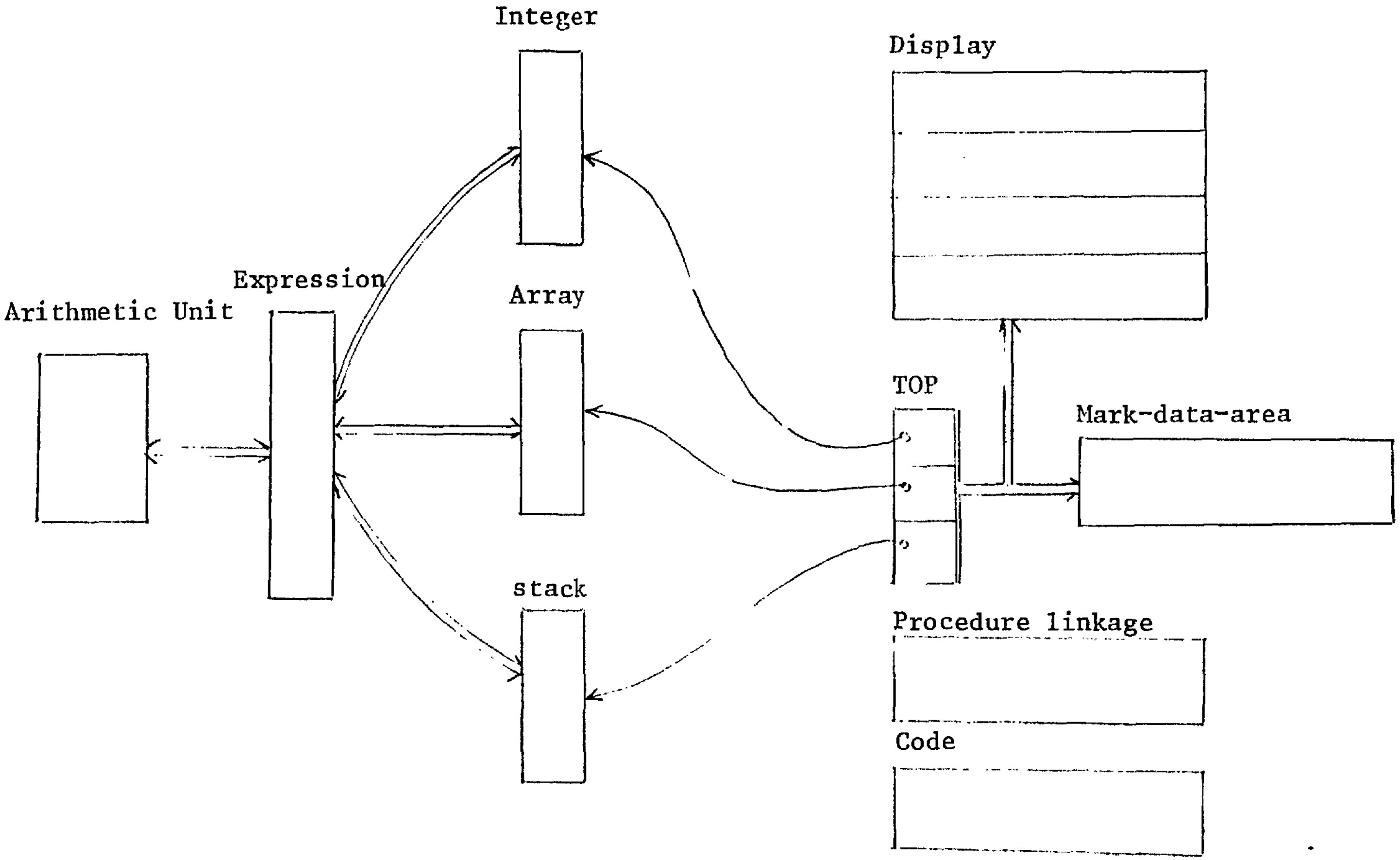


Fig. 2.2 - EPL/O machine memory system
conventions:
——— pointer
═══ data flow

Table 2.1 - Integer memory definition

```
IM: array [0..maxintaddress] of integer;

procedure readint (absadr: 0..maxintaddress);

    begin

      mbr: = IM[absadr]

    end;

procedure writeint (absadr: 0..maxintaddress);

    begin

    IM[absadr] := mbr

    end;
```

Note: mbr is a special purpose register - the memory buffer register.

Table 2.2 - Array memory definition

```
AD: array [0..maxarrayaddress] of integer;

MA: array [0..maxnoarrays] of 0..maxarayaddress;

procedure readarrayelement (arrayaddress, index);

     begin

     elementaddress: = MA[arrayaddress] + index; (*form real address*)

     arraybound: = MA[arrayaddress +1]; (*bound is next array start*)

     if (index < 0) or (elementaddress ≥ arraybound)

      then

        error

      else

        mbr: = AD[elementaddress]

      end;
```

Table 2.2 (Cont'd)

```
procedure writearrayelement (arrayaddress, index);
 begin
   elementaddress: = MA[arrayaddress] + index;  (*form address*)
   arraybound     : = MA[arrayaddress + 1];      (*bound*)
   if (index < 0) or (elementaddress ≥ arraybound)
     then
       error
     else
       AD[elementaddress]: = mbr
 end;
 procedure updatemark (arraysize,offset);
 begin
  MA[offset] + arraysize (*offset is address of next array in MA*)
 end;
```

Table 2.3 - Stack memory definition

```
SM: array [0..maxstackaddress] of integer;
procedure pushfrom (index);
  begin
   for control: = maxstackaddress
     downto index + 1
     do SM[control] : = SM[control-1];
   SM[index ] : = mbr
end;
procedure popfrom (index);
  begin
    mbr: = SM[index];
    For control: = index
      to maxstackaddress-1
      do SM[control]: = SM[control+1]
    end;
```

Table 2.3 (Cont'd)

```
procedure fixmarkstack (stackaddress);
  begin
  for control: = stackaddress
   to maxmarkstackaddress
    do SM[control]: = succ(SM[control]);
  end;
procedure pop (absadr);
  begin
    index: = MS[absadr];      (*get address of top of the stack*)
    popfrom (index);          (*mbr has data*)
    fixmarkstack;             (*correct mark-stack*)
  end;
procedure push (absadr);
  begin
    index: = MS[absadr];      (*address of top of stack*)
    pushfrom(index) ;         (*inser mbr in top of stack*)
    fixmarkstack;             (*correct mark-stack*)
  end;
procedure empty (absadr);
  begin
    if MS[absadr] = MS[absadr+1]
     then
       mbr: = 1               (*stack is empty*)
     else
       mbr: = 0
  end;
```

Table 2.4 - Expression memory definition

```
EM: Stack of integer;
procedure pushem;
begin (*write mbr in top of stack*)
  EM: = mbr
end
procedure popem;
begin (*read top of the stack to mbr*)
  mbr: = EM
end;
procedure literal (value);
begin (*load a literal constant on evaluation stack*)
  EM: = value
end;
procedure operator ( op);
begin (*execute operation defined by op*)
  acl: = EM;          (*read first operand*)
  ac2: = EM;          (*second*)
  case   op of :
  + : :   EM: = acl + ac2 ;
  -   :   EM: = ac2 - acl ;
  *   :   EM: = acl* ac2  ;
  /   :   EM: = ac2/acl   ;
end;
end;
```

Table 2.5 - Procedure linkage memory definition

```
LM: stack of integer;
procedure pushlm;
  begin (*save PC in linkage memory*)
    LM: = PC
  end;
procedure poplm;
  begin   (*restore PC*)
    PC: = LM
 end;
```

Table 2.6 - Mark data-area memory definition

```
type datapointer = record
                    intpointer: 0..maxintaddress;
                    stapointer: 0..maxstackaddress;
                    arrpointer: 0..maxarrayaddress
                   end;
TOP: datapointer;
MD : stack of datapointer;
procedure  pushmd;
  begin (*save current pointers to free data space*)
    MD: = TOP
  end;
procedure popmd;
  begin (*restore pointer to free space*)
    TOP: = MD
  end;
```

Table 2.7 - Display memory definition

```
DM: array [1..maxnest] of
       stack of datapointer;
procedure pushdisplay(level)
  begin (*save current entry in the display and update*)
DM [level] : = TOP
  end;
procedure popdisplay(level);
  begin (*return old entry in level*)
    scratch: = DM [level]
  end;
```

Table 2.8- EPL/0 instruction set.

```
1-LODINT     1,a    :begin (*load integer to expression memory*)
                       readint(absadr) *
                     end;

2-LODSTA     1,a    :begin (*load stack element to expression memory*)
                       pop(absadr)
                     end:

3-LODVEC     1,a    :begin (*load array element to expression memory*)
                       popem;
                       readarrayelement(absadr,mbr)
                     end;

4-STOINT     1,a    :begin (*store top of EM in integer memory*)
                       writeint(absadr)
                     end;

5-STOSTAL    1,a    :begin (*store top of EM in stack memory*)
                       push(absadr)
                     end;

6-STOVEC     1,a    :begin (*store top of EM in array memory*)
                       popem;
                       writearrayelement(absadr,mbr)
                     end;

7-DOUPEN       a    :begin (*do loop enter sequence*)
                       if IM[ctladr]>limit **
                         then PC: = a; (*a contains out of loop address*)
                     end;

8-DOUPRE       a    :begin (*do loop tail sequence*)
                       IM[ctladr] : = succ(IM[ctladr]);
                       PC: = a
                     end;

9-STOCS             :begin (*store top of EM in linkage memory*)
                       temp: = PC;
                       PC: = EM;
                       pushlm;
                       PC: = temp
                     end;

10-LITCS       a    :begin(*store literal a in linkage memory*)
                       temp: = PC
                       PC: = a;
                       pushlm;
                       PC: = temp
                     end;
```

---

* Footnote absadr is the absolute address generated by the relocation of the address couple (1,a) by the display memory.

```
11-TSTACK   1,a   :begin (*test if stack is empty*)
                     if empty(absasdr)
                       then litem(1)
                       else litem(0)
                    end;

12-STOIP    1,a   :begin (*store integer parameter at offset a*)
                      writeint(TOP.intpointer+a):
                   end;

13-STOSP      a   :begin (*store stack parameter at offset a*)
                      push(TOP.stapointer+a)
                   end;

14-LIT        a   :begin (*load literal a at EM*)
                      literal (a)
                    end;

15-OPR        a   :begin (*execute arithmetic operation*)
                      operator(a)
                    end;

16-JMP        a   :begin (*jump to a*)
                      PC: = a;
                    end;

17-JPC        a   :begin (*jump if false to a*)
                      if EM = 0
                        then PC: = a
                    end;

18-CALL     1,a   :begin (*call procedure at level 1 address a*)
                     pushdisplay(1);
                     pushlm;
                    pushmd;
                    PC: = a
                    end;

19-RETURN   1     :begin (*return from level 1*)
                    poplm;
                    popmd;
                    popdisplay(1)
                   end;

20-ENTI       a   :begin (*allocate space for a integer variables*)
                     TOP.intpointer: = TOP.intpointer+a
                    end;

21-ENTS       a   :begin (*allocate space for a stack variables*)
                     TOP.stapointer: = TOP.stapointer+a
                    end;

22-ENTA       a   :begin (*allocate space for a array variables*)
                     TOP.arrpointer: = TOP.arrpointer+a
                    end;
```

** Footnote ctladr is the address of the for control variable and limit is the maximum value of the iteration; both of which are stored in the procedure linkage memory.

Table 2.9 - Sentence Distribution

SENTENCE	FREQUENCY	PERCENT
IF	38	9
WHILE	24	6
REPEAT	17	4
FOR	27	6
CASE	4	1
CALL	51	12
ASSIGNMENT	258	62

Table 2.10 - Operator Distribution

OPERATOR	FREQUENCY	PERCENT
+	48	25
-	38	20
*	15	8
/	9	5
OR	5	3
AND	2	1
NOT	6	3
=	8	4
<>	5	3
<=	10	5
>=	4	2
>	30	16
<	9	5

Table 2.11 - Data Area Size Distribution(Static)

BLOCK SIZE*	FREQUENCY	PERCENT
0	3	5
1	2	3
2	10	15
3	23	35
4	10	15
5	10	15
6	2	3
7	3	5
8	2	3
9	0	0
10	0	0

*Total number of declared variables and parameters in a procedure.

Table 2.12 - Instruction Distribution(Static)

INSTRUCTION	FREQUENCY	PERCENT
LODINT	405	19
LODSTA	11	1
LODVEC	82	4
STOINT	199	10
STOSTA	14	1
STOVEC	72	3
DOUPEN	24	1
DOUPRE	24	1
DODOEN	3	0
DODORE	3	0
STOCS	27	1
LITCS	27	1
TSTACK	12	1
STOIP	16	4
STOSP	0	0
STOVP	0	0
LIT	313	15
OPR	214	10
JMP	115	6
JFC	0	0
JPC	91	4
CALL	50	2
RETURN	65	3
ENTI	65	3
ENTS	65	3
ENTV	23	1
ENTD	65	3
DISP	33	2
ABORT	4	0

Table 2.13 - Dynamic Instruction Frequency

INSTRUCTION	FREQUENCY	PERCENT
LODINT	3143	29
LODSTA	35	0
LODVEC	861	8
STDINT	823	8
STOSTA	58	1
STOVEC	550	5
DOUPEN	264	2
DOUPRE	216	2
DODOEN	44	0
DODORE	36	0
STOCS	56	1
LITCS	56	1
TSTACK	34	0
STOIP	209	2
STOSP	0	0
STOVP	0	0
LIT	1079	10
OPR	1546	14
JMP	291	3
JFC	0	0
JPC	688	6
CALL	123	1
RETURN	124	1
ENTI	152	1
ENTS	152	1
ENTV	23	0
ENTD	152	1
DISP	60	1
ABORT	0	0

Table 2.14 - Memory Access Distribution

MEMORY	FREQUENCY	PERCENT
INTEGER	4735	13
STACK	127	9
ARRAY	1434	4
EXPRESSION	10662	29
DISPLAY	6835	19
MARK-DATA	992	3
LINKAGE	796	2
CODE	10779	29

Table 2.15 - Comparison of two PL/0 machine versions.

Machine	Code size	Instructions executed
Original PL/0	536	4788
EPL/0	398	2803

3 Descriptors and the implementation of data structures.

3.1 Introduction

According to N. Wirth (Wir76a) a well-structured program can be thought as consisting of two different parts: a data structure and the algorithms which work upon it. In the same manner, the problem of mapping a high level language like Pascal to an architecture can be subdivided in two problems:

- the mapping of language data structures.
- mapping of the control statements.

The problem of finding primitives for control structure does not present major difficulties. Deriving primitives of language data structures which can be efficiently mapped to hardware is more difficult, not only because data structures are more complex than control structures but also because the former is interwined with the addressing method of the language with its problems of scope, blocks etc.

There are two main techniques for mapping language data structures to machine architecture:

a. A hardware solution, which is characterised by the design of special purpose memories to meet the required data structure primitives. This is the line of solution used in the PL/0 machine discussed in Chapter 2. This approach can only be used when the language data structures are not very complex - like PL/0. But, when we consider a language like Pascal two main obstacles appear:

   i. there are several forms of structuring data. This leads to several different memories whose management is very costly.

   ii. the data types can be nested. The memory structure to cope with this case would be very complex.

b. A software or logical solution which involves a memory 'emulation device', i.e. a device capable of transforming the linear memory of present day computers into a structured space with the required properties of the data structure definition. This can be achieved using descriptors.

As a starting point in our study of descriptors we decided to investigate the ICL 2900 descriptor mechanism. (ICL76a). The ICL 2900 was designed originally to act as a target language machine, i.e. to match the needs of the intermediate forms of various compilers (Buc78a). Since it is also one of the more advanced architectures in the market, it was thought profitable to study it and to obtain the maximum feedback from its design.

However, the ICL 2900 descriptor presents several problems to the implementor of Pascal. These problems are discussed in (Iza79a and Ree77a) and can be briefly summarized as:

a. the 2900 descriptor being a 64-bit entity gives a low value to the quotient (data bits/descriptor), i.e. descriptors occupy too much space.

b. the 2900 scheme is not general enough to Pascal requirements, e.g. the descriptor 'size' fields can describe only the basic machine types, making it impossible the define arbitrary size elements as a Pascal record.

Although an 'ad hoc' method can improve this ratio, it imposes a penalty on compiler simplicity. The standard architectural solution not only does not provide a simple method for the assignment of data structures (as a block) due to the fact that descriptors and data are mixed but also complicates the creation of dynamic data structures since structural information must be evaluated at generation time (through the procedure new).

This chapter describes a descriptor mechanism for mapping Pascal data types to computer memory. It consists of a set of type descriptors and three descriptor operators. The idea is transforming a valid Pascal name into a semantic expression which when evaluated at run time will give as a result the semantic attributes of the name: address and type.

The semantic expression consists of operands and operators. The operands are descriptors and the operators (which operate on descriptors giving descriptors) have a one to one correspondence with Pascal data selectors.

Our solution tries to cope with the 2900 descriptor problems cited above. The ratio (data/structural information) can be improved by attaching descriptors to types instead of to variables. Assignment and creation of data structures are simplified because data structures are laid down linearly in memory and descriptors are kept separate from data.

The main advantage of this method against the traditional compiler evaluation is the obvious simplification of the translation procedure which is one of the aims of the language-oriented computer design. This is achieved by delaying all the work related to address and type evaluation of data structure elements to run time.

3.2 Basic definitions

In some primitive machine architectures the semantic information about a type or variable is distributed throughout the code, without any structure. An organised technique for description of data at machine level is required.

Instead of having data about an array, for example, distributed in instructions like 'compare bounds' or 'load an element of size x', data about bounds and element type could be stored in a special position which is read each time an access to the array is executed. This position is here called the (array) descriptor.

Since all accesses to data structures of the same kind require the same set of operations, it seems natural to associate with the descriptor some implementation of the primitive access operations required for the particular structure. In the case above, the instruction 'compare bounds' is a primitive of all array access therefore it can be merged in a more general operation 'access array through descriptor' which would execute this checking automatically.

Hence, when discussing descriptors, it is useful to remember that the term connotes, with its semantic data, a set of basic operations used in data structure access.

In the implementation of language data structures using descriptors the latter will act as a bridge connecting abstract data structures to concrete computer memory. Since the terms type, data structure and descriptor are very frequent in this chapter, it is useful to start by stating their definition and associated symbols.

Type determines the class of values that may be assumed by a variable or expression. Structured type is a type defined in terms of other types. Data structure is a structured type together with some operations on that data type (Col78a)

Descriptor is a data object containing the semantic specification of a type or variable. The value of a descriptor, x, is displayed as

x=(n0=f0,n1=f1,....nk=fk),

where ni is the identifier of the i-th field and f0,f1,..fn represent the values of the descriptor fields in the same order as they appear in the definition. We refer to a descriptor field using the same dot notation as in the reference to a Pascal record i.e.:

x.ni = fi where ni is the identifier of the i-th field in the descriptor template definition.

Descriptor template defines the class of values that may be assumed by the descriptor. The template acts as 'type' for the descriptor. The definition of a descriptor template is made using the same notation used for Pascal records. When defining physical fields

bit [n] = array [1..n] of boolean

is used.

Descriptor operations are the set of basic addressing and type evaluating primitives working on descriptors. The description of these operations will be made using the same form as a Pascal procedure.

3.3 Descriptor objects

Descriptors are data objects. As data objects they have a name and a value. The descriptor value is a set of attributes which characterise some computer object, e.g. variable, file, procedure or another descriptor. Since this chapter is discussing data structures, 'descriptor' will hereafter denote descriptor for data objects only, excluding code descriptors, etc.

Descriptors are complex data objects. The basic units forming the descriptor are called descriptor fields. Each one describes one of the attributes of the object. A descriptor field can itself be a complex data object depending on the particular attribute being defined.

The descriptor template defines the set of values that a descriptor can assume by defining how many and what kind of fields the descriptor has. There are many ways to arrange the semantic information concerning an array (array bounds, element type, size, address) in different descriptor fields. We have chosen one field partition, which will result in the simpler algorithm for name translation, as shown in section 3.5. This format is not intended to be the final one, since many efficiency constraints could modify it. Among the factors which can influence partition are descriptor size and information traffic. A field partition whose target were to minimize the area occupied by descriptors would give a different descriptor template.

3.4 Descriptors for Pascal data types

We are concerned only with Pascal data objects. The data objects generated by Pascal are Pascal variables. A Pascal variable can be defined by its address and type so a Pascal variable descriptor has two fields, type and address, the former being usually a complex field. It can be seen that finding descriptors for Pascal variables can be reduced to the problem of finding descriptors for Pascal types. It is also useful to consider type descriptors as objects in themselves. Since types can be shared by variables, the same descriptor can be used in the definition of several variables.

In this chapter we discuss the definition of descriptor templates for seven different Pascal data types, of which three are simple types and four are structured.* In order to be able to describe semantically all possible type declarations we need at least one

*Footnote We shall not consider file types since it involves system dependent features.

descriptor template for each data type.

All non-recursive types, simple types and sets, can be described by a fixed format descriptor template. Arrays and records on the other hand, if one ties to put in their descriptor the entire semantic specification, cannot have a fixed descriptor representation. Fortunately Pascal restricts the type of operations on structured types. The only operation allowed is assignment of equal type structures which does not depend on any semantic attribute of the type apart from its size. As an example, the semantic data needed for an array x in its two forms x and x[i] are different. For the first case, only its size, whilst for the second information about bounds and element type is necessary. There is an exact parallel in the case of records.

This fact gives us the key to solving the problem of the recursive nature of these types. The descriptor template for arrays and records has (apart from its tag) only one field to hold the array or record physical size; separate templates are defined for array elements and record items.

Note: in order to get a more concise representation for descriptors, we will omit the field identifier in front of the descriptor field value.

Example: d(T)=(tag=sca,card=3) will be denoted by d(T)=(sca,3).

In the case of complex fields, parenthesis are used to give the correct hierarchy. Also, the mnemonic bit [n] denotes an implementation dependent field size.

We show below for each Pascal data type its associated descriptor template:

1-primitive types

Primitive types being predefined and static, there is no need for any semantic parameter in their definition. They are defined uniquely by their tag.

```
primitive-type-template=      record
                                tag: (int,char,bool,real)
                              end
```

for example, a declaration like:

```
type T = integer;
```

would create a descriptor for T denoted by d(T) as :

d(T)=(int).

2-scalar types

A scalar type is defined by a set of constant identifiers over which the Pascal standard functions pred(x) and succ(x) are defined. They can be implemented by mapping the constants on to a subset of integers 1..n, so their semantic description needs only the number of elements in the set. Their template is:

```
scalar-type-templates=        record
                                tag: (sca);
                                card: bit [n]
                              end
```

for example :

type T=(white, grey, black) would generate

d(T)=(sca, 3).

3-subrange types

The subrange type is defined by a pair of constants marking an interval over an already defined scalar type. Its template can be defined as:

```
subrange-type-templates=        record
                                  tag : (subr);
                                  lcon, ucon : bit [n]
                                end
```

4-set types

The set type can be semantically identified by:

```
set-type-template =             record
                                  tag : (set);
                                  card: bit [n]
                                end
```

where the field *card* defines the number of elements in the type over which the set is defined.

5-Array types

The semantic definition of the type array involves two templates. The first is a descriptor for the whole array:

```
array-type-template =           record
                                  tag : (arr);
                                  size:  bit [n]
                                end
```

where *size* is field to hold the array physical size, e.g. in bytes. We have a second one for the array elements:

```
array-element-template =        record
                                  index : simpletypetemplate;
                                  element : typetemplate
                                end
```

where *index* is any simple type template and *element* is any type

6-Record types

As in the array case, there are two templates defined for records.

```
record-type-template =          record
                                  tag : (rec);
                                  size:  bit [n]
                                end

record-item-template =          record
                                  tag : (fld);
                                  item:  typetemplate ;
                                  offset: bit [n]
                                end
```

where *offset* is a field holding the physical distance of the item from the beginning of the record.

7-Pointer type

Since pointers are defined over an already defined type, their semantic specification does not need any semantic fields (they are already in the pointed type).

```
pointer-type-template =         record
                                  tag : (ptr)
                                end
```

Example - We show below how the descriptors for some simple types are being absorbed into more complex ones. In the left column there is a Pascal declaration, and in the corresponding right column we find its descriptor.

In the following example we used the symbols :

d(<id>) - for the descriptor of id

d(<id>-e) - for the descriptor of the array element of <id>.

d(<id>-p) - for the descriptor of the type to which the variable <id>, a pointer, is bound

```
type
          alfa = 1 .. 10 ;              d(alfa)=(subr,1,10)
          beta = set of 1..6;           d(beta)=(set,6)
          gama = array  [alfa]
                        of char;        d(gama e)=(arl,(subr,1,10),chr)
                                        d(gama)=(arr,10)
          delta= record
                   x : alfa ;           d(x)=(fld,(subr,1,10),
                   y : beta ;           d(y)=(fld,(set,6),1)
                   z : gama ;           d(z)=(fld,(arr,10),2)
                   u : ↑ delta ;        d(u)=(fld,ptr,12)
                end;                    d(delta)=(rec,16)

          epsilon = array [alfa]  of data ;
                               d(epsilon¯e)=(arl,(subr,1,10),(rec,16))
                               d(epsilon)=(arr,160)
```

3.5 Descriptors for Pascal variables

Given a set of descriptor templates, one for each data types, we can generate descriptors for any Pascal variable. The descriptor for a variable is defined by two fields: a data attribute field which is the type descriptor to which the variable is bound and an address field.

The format of any variable descriptor can be defined as:

```
variable-descriptor = record
                                attribute : typetemplate;
                                address : bit [n]
                      end
```

for example suppose a declaration like:

```
var sigma: epsilon;
```

d(sigma)=(d(epsilon),address)

but as in the last section :

d(epsilon)=(arr,160)

and if sigma is bound to location 300 in memory, the value of its descriptor is :

d(sigma)=((arr,160),300).

3.6 Descriptor operators

Given this semantic description of a data structure we can get the descriptor of one of its elements, by using specific descriptor operators.

The operation executed by the 2900 array descriptors is an example of a descriptor operator in which given the array descriptor and an index it evaluates the array element descriptor.

After a structure is defined it can be accessed as a whole or in parts. The access to certain components of a data structure is made through the use of selectors. There is a selector corresponding to each structuring method, and in the same way, both can be recursively used.

When a single element which is part of a data structure is referenced, it is denoted by a series of selectors applied to the highest hierarchic name in the data structure. One way of thinking about a cascade of selectors is as constituting a series of operators applied on data types.

The main constraint in the design of the descriptor operators, was the need for a resulting simple translation algorithm to minimize the work done by the compiler when generating code for a Pascal name. The second restriction is one-symbol-look-ahead, which implies that the analysis of names must be done in a single scan from left to right. Additionally, during evaluation, the system should use the normal data stack, without any special features. At the end of the evaluation process the resulting descriptor should be at the top of the data stack. These conditions allow a very simple and structured technique for evaluating Pascal names, since all evaluations, both of expressions and descriptors are made on the same stack.

The simplest way of fulfilling the above condition is a simple one-to-one replacement of the Pascal '[' the array selector, '.' the record item selector and '↑' the pointer selector by three descriptor operators, which we call bracket, dot and arrow.

For example, a name like sigma[2].z, would be converted by the compiler into the Reverse Polish string

d(sigma) d(sigma¯e) 2 bracket d(z) dot .

where d(<id>) means 'load the descriptor of <id> to the stack'. In this case bracket would operate on d(sigma), d(digma¯e) and the value 2 to produce the descriptor of sigma [2], which combined with d(z) by the operator dot gives as result the address and type of sigma[2].z.

This means a transfer of the operations made by the compiler when generating code for sigma[2].z to runtime.

The descriptor operators assume a resulting descriptor with the format:

```
result:    record
             type: typetemplate
             address: bit [n]
           end
```

We use also the following functions:

Length (x) - is a function that when applied to the descriptor argument x returns the size (in bytes) of the element described by x.

value (x) - the argument x is a descriptor, the function returns the value of the object described by x.

lbound(x) - the argument is a simple type descriptor, the function returns the value of the lower bound of the type specified by the descriptor.

i-the bracket operator

The function of this operator is given an array descriptor x an array element descriptor y and an index value z generate a descriptor for the array element variable. Its operation can be defined by the following procedure (operands being assumed to be global) :

```
procedure bracket;
   {generate a variable descriptor for the array element}
     begin
       result.type := y.element;
       result.address:= (z-lbound(y.index))
                        *length(y.element)
                        +x.address
     end
```

This means the generation of a variable descriptor whose type is the element field of the array element descriptor and whose absolute address is the sum of the base address of the array with the product of the index by the array element size.

ii-the dot operator

This supplies as result the semantic characteristics of the item being selected inside a record. If x is the array descriptor and y is the record item descriptor then its operation can be defined as

```
procedure dot;
   {generate the descriptor for the record item}
     begin
        result.type := y.item;
        result.address := x.address+y.offset;
     end
```

This means the generation of a variable descriptor whose type is the same as the item descriptor and has as address the sum of the base address of the record with the item offset.

iii-the arrow operator

This supplies the semantic description of a pointer selected variable. As the pointer variable descriptor describes a pointer variable, whose contents point to a variable of type y, the result is the creation of a variable descriptor of the same type, having as address the contents of the pointer variable. This can be seen in the definition below:

```
procedure  arrow;
   {generate the descriptor of a pointed variable}
      begin
        result.type := y;
        result.address := value(x)
      end
```

3.7 Examples

This example shows how, given a name in its textual form with all the descriptors associated with it, we can form descriptors for its elements. In the following examples suppose the variable sigma is bound to memory location 300 and that descriptor evaluation is taking place on the same stack as the expressions. Figure 3.1 shows a graphic representation of the memory lay-out of this data structure.

Let us use the same type definitions as in section 3.4. Valid Pascal names, defined over a variable sigma of type epsilon are:

```
sigma
sigma[2]
sigma[2].x
sigma[2].z
sigma[2].z[3]
sigma[2].u↑.z
```

case 1

The name is sigma. The descriptor of sigma is:

d(sigma) = (d(epsilon),300) = ((arr,160),300)

which means that sigma is an array of size 160, starting at location 300. See figure 3.1. Note that no other semantic information is needed, since Pascal operations on data structures are limited to assignment.

case 2

The name is sigma 2 . Its descriptor is defined by the following reverse polish string:

d(sigma [2]) = d(sigma) d(sigma¯e) 2 bracket

Looking at the definition of bracket we can see that the result is ((rec,6),316), which agrees with figure 3.1-ii.

case 3

The name now is sigma [2].x. In this case we have:

d(sigma[2].x) = d(sigma[2]) d(x) dot

d(sigma[2].x) = ((rec,16),316) (fld,(subr,1,10),0) dot

d(sigma[2].x) = ((subr,1,10),316)

The record with its elements is shown in figure 3.1-iii.

case 4

The name is sigma [2].z. The result in this case is an array descriptor with the attributes of type gama and address 318.

d(sigma[2].z) = d(sigma[2]) d(z) dot

d(sigma[2].z) = ((arr,10),318)

Again, this is shown in figure 3.1-iii.

case 5

The name is sigma[2].z[3]. What we get now is the descriptor of a variable of type char at address 320, as in figure 3.1-iv.

d(sigma[2].z[3]) = d(sigma) d(sigma¯e) 2 bracket d(z) dot
d(z¯e) 3 bracket

which expression when evaluated from left to right gives

d(sigma[2].z[3]) = (char,320).

case 6

The name now is sigma[2].u↑.z. Suppose also that an instruction new (sigma[2].u) was issued before, allocating a record of type delta at position 1000 in memory.

d(sigma[2].u↑.z) = d(sigma[2]) d(u) dot d(u¯p) arrow d(z) dot

which will give as final result ((arr,10),1002).
See parts v and vi of figure 3.1.

3.8 Conclusions

We have derived in this chapter one technique for Pascal data structure implementation based on language considerations. This scheme is more general, less space consuming and simpler to use at compile time than the mechnism incorporated in the ICL 2900 architecture.

Before any efficiency evaluation of this mechanism can be made, several implementation considerations must be solved first:

a. the final descriptor format with number and size of fields.
b. how to implement descriptor operators - as zero address instructions or as one address instructions with the address field specifying the descriptor address.
c. the primitives use for load, store and move data via descriptors.
d. the method used for store and descriptors: as constants, variables in the code area etc.

In order to answer these questions and to evaluate the efficiency of this mechanism we must know first the usage patterns of Pascal data structures. An investigation of these usage patterns will be the subject of the next chapter. Considerations about implementation and efficiency of the descriptor mechanism will be presented in Chapter 6.

(i)

address	300	460		
name	← sigma →			

(ii)

address	300	316	....	444
name	sigma[1]	sigma[2]	....	sigma[10]

(iii)

address	316	317	318	328
sigma[2]	x	y	z	u

(iv)

address	318	319	.....	327
sigma[2].z	z[1]	z[2]		z[10]

(v)

address	328	.....	1000
sigma[2].	u		u
value	1000		

(vi)

address	1000	1001	1002	1012
sigma[2].u↑.	x	y	z	u

Figure 3.1

4-A Study of Pascal programs

4.1-Introduction

The main problem in language oriented computer design is to find which of the semantic primitives in the source programming language must be optimized when mapped to real hardware. A language is only a set of rules. It is possible to derive a multitude of machines to implement that set of rules. We are looking for a language oriented computer matching some efficiency criteria.

The efficiency criteria we are using is the one already defined by McKeeman (McKe67a), which is based in the amount of redundant information used by the language-oriented machine to store and run programs in the source language. The more information the machine uses the less efficient it is. The task of designing a machine for a given programming language can be defined by two constraints: the machine should allow the implementation of all the language constructs and be efficient in terms of information usage.

It is simple to conform to the first constraint, since any machine with a simple increment, test and branch on minus can be proved to execute any computable function. However, to minimize the redundant information required to store and run programs in the source language, we must know the characteristics of these programs in order to adapt the machine characteristics to the most frequent program patterns.

If one had the complete information about the actual programs behaviour it would be possible to design a machine which uses the minimum of redundant information to run a specific workload.

Unfortunately this is not possible in a real environment, since the components of the workload are not always the same and usually each program is being updated and changed as time passes. However, the patterns of the population of programs which constitute the workload can be achieved by a statistical analysis of a sample of programs representative of the whole population. Extrapolation from such a sample is possible since the population of programs will have some properties which will be imposed both by the type of application and the language rules.

There are several works in the area of analysis of behaviour of programs written in a high-language. Algol-60 was studied by Wichman (Wic70a), Chevance analysed Cobol (Chev78a), Knuth studied Fortran (Knu71a) and Alexander and Wortman analysed XPL (Ale75a). Wortman made a deep study of a dialect of PL-1 called Student-Pl in his doctoral thesis (Wor72a).

Since we are working in Pascal oriented machine architecture and there exists no case in the literature of a study, similar to the above, of Pascal programs it was necessary to conduct our own measurements.

This study has two main targets:

1-to collect characteristics of programs which can be used to design and improve Pascal oriented machines.

2-to obtain data which is general enough to enable the building of program models. These models could then be used to build synthetic workloads,to make predictions and evaluations of computer performance.

This chapter contains a description of the results obtained by the analysis of form and behaviour of well-structured Pascal programs. In selecting the sample of programs, we concentrated on system programs since it is reasonable to assume that they will consume most of the installation resources. The analysis includes textual structure, measurement of syntactic composition and usage of language fragments, both static (appearing in the object code produced by the compiler) and dynamic (executed at-run time).

4.2-The Experiment

The experiment sample consisted of 38 Pascal programs making a total of 65000 lines of text; out of the 38 programs we selected 23 for dynamic analysis.

The experimental tool used in the study was based on the Pascal P4 compiler which runs in the IBM-370 of the Computing and Control Department of Imperial College. See Pugh79a for more details about this implementation.

The Pascal compiler was modified to collect data about the currently compiled program composition and its code generation part also was modified to insert monitoring instructions in the intermediate text being generated. The analysis of the source text is made by procedures called at three stages of the compilation process:

Stage 1 - at the end of compiling a procedure

Two main routines are executed:

i- symbol table scan - gets data about declared entities in this procedure, more specifically labels, constants, types, variables procedures and parameters

ii- procedure body composition-collects data about frequency and size of statements.

Stage 2 - syntax phase

Based on the Pascal syntax definition given in the Standard Report (Jen74a), monitoring instructions are inserted in the text of the compiler to count the frequency usage of the parsing rules.

Stage 3 - code generation phase

We defined a set of Pascal fragments corresponding to some sequences of code generated for possible paths in the code generation process. In general, fragments constitute only parts of statements. Each time a particular fragment is found, an instruction "monitor fragment i" is inserted in the intermediate code and a static record of it is made.

At run time, the monitoring instruction, translated to 370 object code, updates an array in the stack of the running program. Programs are also modified such that they will output automatically, at the end of the run, a file containing the record of the dynamic usage of the fragments.

A flowgraph of the measurement system is shown in Figure 4.1.

4.3-Results

This section is intended to serve as a guide to the interpretation of the tables obtained as a result of the experiment. The results can be divided in three classes: text composition, syntactic structure, and code fragments usage.

4.3.1-Text Composition

The data about textual composition of programs contains the cumulative result of the 38 programs analysed. This data is divided again according to the declaration parts in Pascal texts: labels, constants, types, variables and procedures.

4.3.1.1-Labels

The distribution of labels in lexical levels is presented in table 4.1

4.3.1.2 - Constant declarations

Table 4.2 shows both the distribution of declared constants in lexical levels and type. Under the entry Scalar are counted all constants declared in a definition of a user defined scalar type. The third part of the table shows the distribution of the logarithm (base 2) of the value of the declared integer constants.

4.3.1.3 - Type declarations

Data collected about types consists of: type distribution by level, type distribution by form and composition of structured types. The data for the last case is presented in matrix form. The lines represent the form of the structured type and the columns the component type. Each matrix element is a frequency count of the occurence of a structured type of a given component type. In the record case, each field is accounted separately and pointer entries are for the pointed element type.*

4.3.1.4 - Variable declarations

We have lumped together local variables and value parameters in Table 4.4, since they are indistinguishable in the compiler symbol table. The same considerations as for the TYPE area apply.

4.3.1.5 - Procedures and functions

Table 4.5 presents the distribution of declared procedures by lexical levels and table 4.6 displays the same distributions for functions together with the distribution by result type.

Parameters - Tables 4.7 and 4.8 show the parameter distribution both by value and reference (var). Note that a parameter declared in a procedure at level n belongs to the level n+1. Since only 5 out of 2026 of the parameters are procedures their statistics is not displayed.

* Footnote A more detailed study of the composition of the type area appears in Sch79a.

Cardinality of arrays, subranges, records and scalars

Tables containing this information are presented in Appendix 1.

Procedure body composition

In table 4.9 we have the distribution of logical or syntactical size of procedures. Table 4.10 shows an equivalent distribution for statements. In the case of procedures, a size of n means that the compiler procedure parsing statements was called n times inside the procedure body. (The compulsory begin-end pair is not counted since it does not call for statement parsing)

The information in table 4.11 was collected in order to answer the question "what is the composition of a Pascal procedure in terms of statements?". A more accurate answer was required than the simple average of how many statements of a given kind were found. The result is a matrix giving the frequency count of the frequency of appearance of statements in procedures. E.g. 175 procedures were found with 2 if-statements inside. Only the non-zero entries are listed to increase legibility.

4.3.2 Syntactic Structure

A convenient way of describing the syntactic composition of programs is through a table showing the frequency of utilization of the syntax rules used in parsing. Table 4.12 shows this information. The format of this table is: the first column has the rule number, the second its frequency count followed by the percent against the total number of rules. The last column has the description of the rule as it appears in the Standard Report.

There are some simplifications in the set of rules presented and they are concerned mainly with redundant rules or some rules used in the lexical definitions of integer, identifier etc. Some of the rules which appear in the syntax definition are ignored by the compiler, but are still presented here although their count is made "a posteriori" (since they are redundant). As one example:

$$\langle \text{unlabeled statement}\rangle ::= \langle \text{simple statement}\rangle \mid \langle \text{structured statement}\rangle$$

is disregarded by the recursive descent top-down parser.

The meaning of the frequency count associated with recursive rules is as follows: 1-if the rule has the form:

$\langle x\rangle ::= \underline{t}\langle y\rangle \{,\langle y\rangle\}$ the frequency count is the number of times the terminal <u>t</u> was found in the text

2-if the rule has the form

$\langle x\rangle ::= \langle y\rangle\{,\langle y\rangle\}$ the frequency count is considered to be the number of times the non-terminal y is parsed. Example - looking at the syntax rules nos. 4 and 5 in table 4.12, we can conclude that out of the 1577 (1538+39) times the non-terminal ⟨label declaration⟩ was parsed, in 1538 cases no label was declared and in 39 cases the reserved word <u>label</u> was found.

4.3.3 <u>Pascal fragments usage</u>

Table 4.13 contains the distribution of usage, both static and dynamic, of code fragments. The static distribution refers to the whole sample - 38 programs while the dynamic is related to the usage pattern of a subset of the sample with 23 programs.

The main categories of fragments are:

1-Program entry/exit - fragments 0 and 1.

2-Block entry/exit - fragments 2 and 3.

3-Assignments - fragments 4 to 36. This class contains the frequency count of the code sequences generated for assignments.

They contain three classes depending on the right hand side being a constant, a variable or an expression.

4-Procedure calls - the first group of fragments refers to parameter passing by value, using the same categories as the assignment. The second group is for var parameters. Fragment 81 and 82 relates to procedure calls with or without parameters. The last group contains information about usage of the most frequent standard procedures.

5-Control statements - fragments 93 to 106.

6-Expressions - we have examined the code fragments used in code generation for expressions according to operator and the class (constant variable or expression) of the operands. We have also monitored the use of factors: constants, variables, user and standard functions.

7-Structured variable access - a set of fragments to monitor data structure access by the class of access used - record, array, pointer or file and type of the accessed element.

4.4 Conclusions

4.4.1 Program composition in general

The average program is about 1685 lines. Its declaration part has 16 constants, 17 types, 91 variables, 36 procedures and 6 functions. Each program has, on average, 3.2 external files.

For the average program, the Pascal P4 compiler generates 3456 intermediate code instructions - an average of 2.06 instructions per source line or 5.14 instructions per statement.

4.4.2 Constants

The constants appearing in the object code can be classified in: explicit and implicit. An explicit constant appears in the source text inside an expression as a literal or a constant identifier.

On the other hand, each time an array index (or a subrange) is computed (or assigned to) there is a reference to a pair of constants which do not appear explicitly in the text - the array range bounds. We call each element of a bounds pair a implicit constant. Statically the total number of references to explict constants is 16,899 against 10,080 implicit. Dynamically there are 953,346 references to explicit constants against 1,289,622 references to implicit constants.

The following points are interesting to note:

- the fact that implicit and explicit constant usage tend to balance each other both statically and dynamically

- almost the totality of implicit constants are of type integer as are about 40% of the explicit ones

- constant strings have a very high static use, about 24%, but their percentage of the total number of references to constants at run-time drops to only 4%

- the pointer constant nil, on the contrary, has a low percent of the total number of the static references, (6.3%) but its percent increases to more than the double (13.5%) at run-time. This is due to the fact that many loops for scanning lists and trees make use of a construct like: while pointer ≠ nil do begin .. end.

Table 4.14

Explicit Constants usage

Type	Static%	Dynamic%
Integer	38.9	40.2
Real	.1	-
Character	10.7	28.6
Boolean	6.5	3.0
Scalar	13.4	10.5
String	23.9	4.2
Pointer	6.3	13.5

Con'td Table 4.14

Implicit constants

Use	Static%	Dynamic%
Assignment	21.5	36.0
Value Par.	9.6	0.7
Array index	69.2	63.5

4.4.3 Variables

Variables are used mostly in expressions and in the left-hand side of assignments. Statically there is one reference to a variable in the left side against two inside expressions. At run-time we have one (store) reference against three in expressions (loads).

About 70% of the variables appearing in the text were entire i.e. they had no selectors. Statically we have 0.29 selectors/ variable, but this proportion rises to almost the double (0.57 selectors/variable) at run-time. This means that it is common to find more structured variables than simple variables inside the processing loops. This case can be noticed clearly in file buffer access, where only 296 static references were found whereas the dynamic count measured was 280,000. Since most of the programs were used in some form of symbol processing, we could have expected a high number of dynamic access to structures including reading and writing files, tree searching and insertion, table accesses, etc... The fact that some loops can be very tiny (e.g. while not (eof) do read(c)) but work on large pieces of data accounts for the large number of dynamic references to structured variables.

Table 4.15

Variable use

Construct	Static %	Dynamic %
Expressions	57.00	66.96
Assignment	28.54	22.80
Ref. parameters	10.11	5.83
With statement	3.32	2.03
For control	1.11	2.35

Static variable composition

Class	Static %
Entire	71.14
Indexed	10.17
Field design	11.25
Referenced	6.58
File buffer	0.87

Structured data access

Class	Static %	Dynamic %
Record field	38.96	40.13
Array element	35.24	25.10
Pointed element	22.78	18.13
File buffer	3.00	16.61

4.4.4 Procedures and functions

The compiler processed 1671 procedure declarations. Included in this count are 50 procedures with the attribute FORWARD and 82 external procedures. About 13% of these declarations were functions.

There were 2026 parameters declared, including parameters to external procedures, so the average number of parameters per procedure is 1.25. About 37% of the procedures and 17% of the functions had no parameters.

It is an accepted fact that the better structured a program is, the higher the proportion of procedure calls it has. It was reported by Tanenbaum (Tan78a) how the proportion of procedure calls inside the text changed from Fortran to block structured languages like XPL and SAL. In a typical Fortran program one might expect a ratio of 10:1 of assignments to procedure calls, while in XPL and SAL this ratio is between 3:1 to 2:1. As a consequence of the high level of programs in our sample, this ratio dropped to almost 1:1 (Including standard procedure calls).

The tables below show the results of the statement distribution and also the static distribution of executable statements found in several studies of program behaviour.

Table 4.16

Frequency distribution of statements

Statement	Static %	Dynamic %
Assignment	30.6	40.3
Call	29.2	24.3
If-then	12.3	29.4
Case	0.8	0.6
While	2.0	1.5
Repeat	0.7	0.4
For	1.2	0.4
With	2.9	2.9
Goto	0.2	0.1
Compound	12.16	-
Empty	8.13	

Cont'd Table 4.16

Executable statements in several languages (static %)

Statement	Fortran	XPL	SAL	Pascal
Assignment	51	55	47	38
Call	5	17	25	37
If	10	17	17	15
Loops	9	5	6	5
Goto	9	1	0	0.3

From the table above we can conclude that the increase in the proportion of procedure calls coincide with the decrease of assignments and gotos. But, the proportion of if's and loops tend to remain constant.

The composition in static terms is dominated by assignments, calls and ifs. The proportion of assignments, calls and ifs at run-time tend to be equal.

4.4.5 Assignments

Assignments tend to be very simple. A simple constant or variable in the right hand side accounts for 66% of the static assignments, this proportion falling to 57% at run-time.

There is a high proportion of assignments of constants, arising in part form initialization of variables as a consequence of the extensive use of procedures and local variables.

About 14% of the static assignments need a range check, but this proportion goes up to 37% at run-time.

The assignment of structured variables - of type record or array accounts 14% of the static assignments but only 3% of the dynamic ones. This is a consequence of the fact that most string usage is in initialization of printable titles.

4.4.6 Procedure calls and parameter passing

Procedures are the second most used language feature. We have included standard procedures in our statistics since 50% of procedure usage (both static and dynamic) is of intrinsic procedures.

The situation changes when considering function usage. Although statically user and standard function balance each other, there is a much higher use of standard functions at run-time - about 80% of all function calls. The predicates EOF and EOL dominates at run-time - 50% of total function calls, with SUCC accounting for the additional 30%.

The amount of effort expended in parameter passing is noteworthy. Parameter passing to user procedures is almost half of the assignments - statically 8500 assignments against 7600 parameters passed, dynamically 620,000 assignments against 3000,000 parameters passed. Since this count not include parameters passed to standard procedures and functions we can infer that parameter passing has the same level of usage as assignments.

User and standard procedures - Table 4.17

Class	Static %	Dynamic %
User	55	48
Standard	45	52

User and standard functions

Class	Static	Dynamic %
User	50	20
Standard	50	80

Parameter passing to user procedures and functions

Class	Static %	Dynamic %
Value	60	42
Reference	40	58

4.4.7 The selective statements - if-then, if-then-else and case

Statically these account for 15% of the executable statements, this proportion going up to 30% at run-time. This can be partly explained by the fact that in a if-then statement at run-time the if part will always be executed but the execution of the then part depends on a condition (the same consideration applies for the if-then-else) although statically they are counted as two separate statements. The average number of case labels in the case statement derived from table 4.12 is 7.3 labels per case statement.

4.4.8 The repetitive statements - while, repeat and for

Using the data present in the fragments table we can evaluate the average loop traversal for the repetition statements. See table 4.18.

Average loop traversal for repetitive statements - Table 4.18

Statement	Usage	Repetation	Traversal
While	26,164	145,958	5.6
Repeat	7,332	59,497	8.1
For	6,838	70,119	10.25

4.4.9 Abbreviations - WITH statement

We have found 793 WITH statements, accounting for 962 abbreviated variables. The estimated dynamic use of abbreviations at run-time is 60,465. Unfortunately we have not the data to know how many variables in the text were being abbreviated.

4.4.10 Expressions

Expressions tend to be very simple. The average number of operators per expressions is 0.21 (statically) i.e. 4 out of 5 expressions will have only one operand. The situation changes at run-time - statically there are 0.18 operators/operand but this quota rises to 0.33 run-time.

Logical expressions have a different pattern. They are used in control of selective and repetitive statements, so they include one relational operator or a conjunction of conditions. Statically we have 1.15 operators per logical expression. Also, 75% of the logical expressions have a form:

⟨variable⟩ ⟨relational operator⟩ ⟨constant⟩.

The present results can be compared with those already obtained by Tanenbaum and Alexander. The average number of operators per conditional expression is for XPL 1.19 and for SAL 1.22; which shows a good correlation with our results.

Alexander and Wortman have reported about the inefficiency of the recursive descent parser when analysing simple forms of expressions. This is also noticed in our case, where the compiler uses 30% of all the productions only for evaluation of precedence without any semantic purpose. The source of inefficiency lies in the fact that the recursive descent implements productions by real procedure calls - such that the parsing of a single constant or variable takes 3 procedure calls (with parameter passing etc..). In a machine without a support for procedure calls this can be very expensive.

Table 4.19

Usage of factors

Class	Static %	Dynamic %
Variable	51.25	62.00
Constant	43.41	37.30
User function	2.18	2.03
Standard function	2.15	8.29
Set expression	0.29	0.36

Operator distribution

Class	Static %	Dynamic %
Relational	52.99	61.77
Add group	23.76	19.10
Multiply group	12.74	11.02
Not (logical)	10.51	8.28

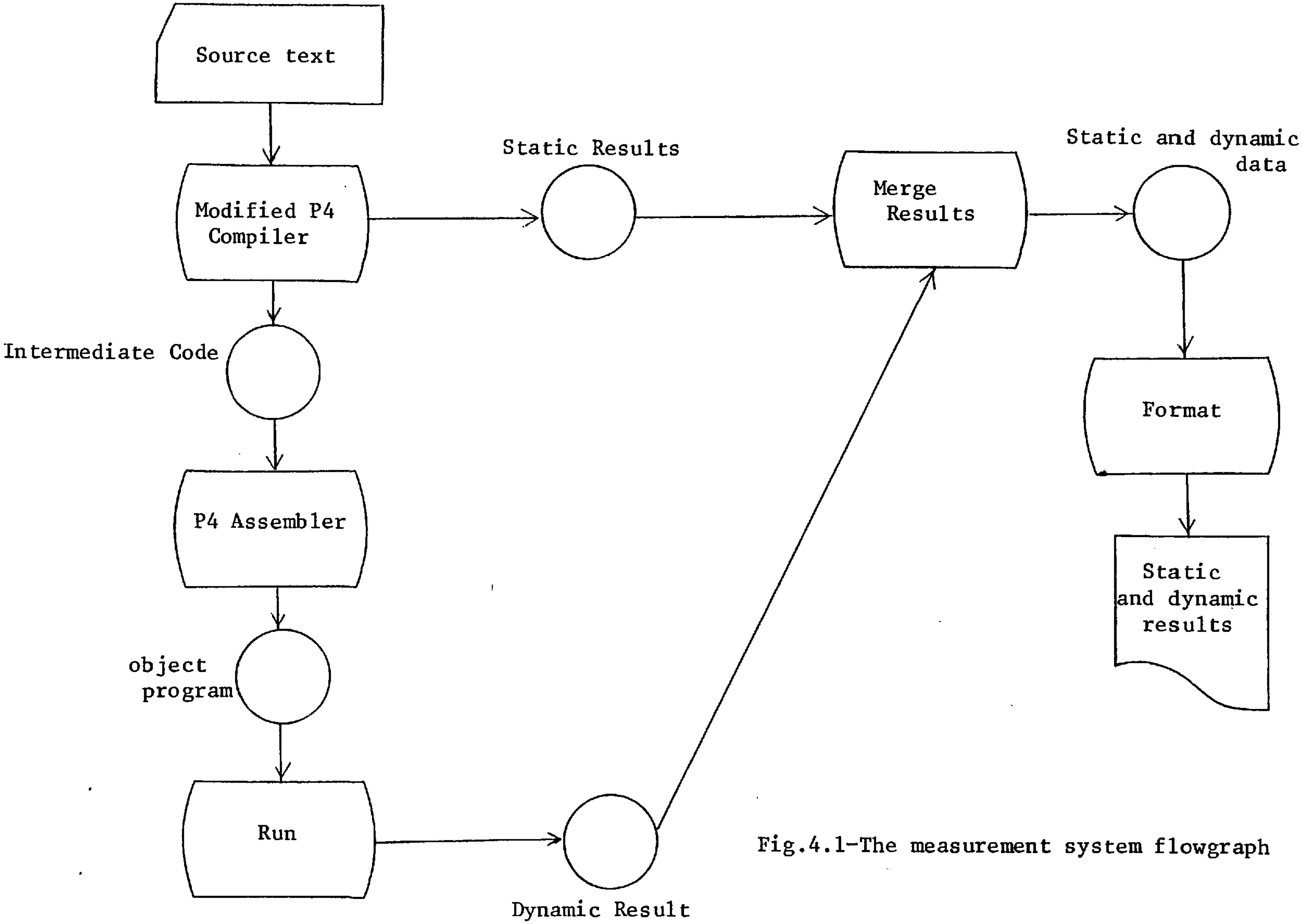


Fig.4.1-The measurement system flowgraph

```
*****************************
Table 4.1-Label declarations
*****************************

Level Distribution
    Levels              Count            Percent        Cumulative
         1                  7              16.67            16.67
         2                 12              28.57            45.24
         3                  7              16.67            61.90
         4                  4               9.52            71.43
         5                  1               2.38            73.81
         6                  7              16.67            90.48
         7                  3               7.14            97.62
        10                  1               2.38           100.00
Total of Labels =              42
```

```
*******************************
Table 4.2-Constant declarations
*******************************

Level Distribution
    Levels            Count          Percent      Cumulative
         1             1070            83.46           83.46
         2              179            13.96           97.43
         3               21             1.64           99.06
         4                7             0.55           99.61
         5                3             0.23           99.84
         6                2             0.16          100.00

Type Distribution
Types                 Count          Percent
Integer                 363            28.32
Real                      1             0.08
Char                     37             2.89
Boolean                   6             0.47
Scalar                  671            52.34
Array                   204            15.91

Value distribution of  integer constants
Size(bits)   Count

 1             40
 2             56
 3             61
 4             38
 5             24
 6             22
 7             40
 8             37
 9              5
10              7
11              8
12              5
13              2
14              6
15              4
16              6
18              1
31              1

Total of Constants =        1282
```

```
****************************
Table 4.3-Type declarations
****************************

Level Distribution
     Levels                Count           Percent        Cumulative
          1                  568             88.20             88.20
          2                   69             10.71             98.91
          3                    5              0.78             99.69
          4                    1              0.16             99.84
          6                    1              0.16            100.00

Type Distribution
Types                      Count           Percent
Integer                       11              1.71
Char                           4              0.62
Scalar                        84             13.04
Subrange                     120             18.63
Set                            8              1.24
Array                        140             21.74
Record                       176             27.33
Pointer                       90             13.98
File                          11              1.71


Structured type composition
          Int   Rea   Cha   Boo   Sca   Sub   Set   Arr   Rec   Poi
Subrange  118     0     1     0     1     0     0     0     0     0
Set         0     0     0     0     5     3     0     0     0     0
Array       3     0   110     0     1     9     0     1    14     2
Record    114     5    30    67    47   128     6   375    54   221
Pointer     2     0     0     0     0     2     1     3    82     0
File        2     0     0     0     0     0     0     5     4     0
```

```
******************************
Table 4.4-Variable declarations
******************************

Level Distribution
    Levels          Count         Percent      Cumulative
         1            711           17.99           17.99
         2           1719           43.50           61.49
         3           1076           27.23           88.71
         4            247            6.25           94.96
         5             90            2.28           97.24
         6             52            1.32           98.56
         7             47            1.19           99.75
         8              6            0.15           99.90
         9              4            0.10          100.00

Type Distribution
Types               Count         Percent
Integer               544           13.77
Real                   37            0.94
Char                   71            1.80
Boolean               422           10.68
Scalar                148            3.74
Subrange              785           19.86
Set                    62            1.57
Array                 664           16.80
Record                273            6.91
Pointer               812           20.55
File                  134            3.39


Structured type composition
           Int  Rea  Cha  Boo  Sca  Sub  Set  Arr  Rec  Poi
Subrange   783    0    1    0    1    0    0    0    0    0
Set          0    0    5    0   44   13    0    0    0    0
Array       23    0  559    0    6   11    0   19   36   10
Record     220   11   33  140   66  243   11  649  105  192
Pointer      0    0    0    0    0    0    2    5  805    0
File         3    0  104    0    0    1    0   15   11    0
```

```
***************************
Table 4.5-Proper Procedures
***************************

Level Distribution
    Levels          Count          Percent      Cumulative
         1            798            56.72           56.72
         2            403            28.64           85.36
         3            121             8.60           93.96
         4             31             2.20           96.16
         5             22             1.56           97.73
         6             23             1.63           99.36
         7              6             0.43           99.79
         8              1             0.07           99.86
         9              2             0.14          100.00
Total of Procedures =       1407
```

```
*******************
Table 4.6-Functions
*******************

Level Distribution
     Levels              Count          Percent       Cumulative
          1                122            57.01            57.01
          2                 67            31.31            88.32
          3                 11             5.14            93.46
          4                  8             3.74            97.20
          5                  2             0.93            98.13
          7                  3             1.40            99.53
          8                  1             0.47           100.00

Type Distribution
Types                    Count          Percent
Integer                     25            11.68
Real                         3             1.40
Char                         7             3.27
Boolean                     88            41.12
Scalar                       2             0.93
Subrange                    34            15.89
Pointer                     55            25.70
Total of Functions =              214

Total of procedures & functions =          1621
```

****

Table 4.7-Value Parameters

****

Level Distribution

Levels	Count	Percent	Cumulative
2	635	57.99	57.99
3	332	30.32	88.31
4	65	5.94	94.25
5	28	2.56	96.80
6	19	1.74	98.54
7	11	1.00	99.54
8	4	0.37	99.91
9	1	0.09	100.00

Type Distribution

Types	Count	Percent
Integer	134	12.24
Real	6	0.55
Char	23	2.10
Boolean	61	5.57
Scalar	66	6.03
Subrange	199	18.17
Set	31	2.83
Array	264	24.11
Record	79	7.21
Pointer	232	21.19

Structured type composition

	Int	Rea	Cha	Boo	Sca	Sub	Set	Arr	Rec	Poi
Subrange	198	0	0	0	1	0	0	0	0	0
Set	0	0	0	0	27	4	0	0	0	0
Array	2	0	258	0	0	0	0	0	3	1
Record	67	1	10	37	13	52	1	250	54	50
Pointer	0	0	0	0	0	0	2	0	230	0
File	0	0	0	0	0	0	0	0	0	0

****

Table 4.8-Var Parameters

****

Level Distribution

Levels	Count	Percent	Cumulative
2	533	63.99	63.99
3	172	20.65	84.63
4	108	12.97	97.60
5	12	1.44	99.04
6	4	0.48	99.52
7	4	0.48	100.00

Type Distribution

Types	Count	Percent
Integer	61	7.32
Real	3	0.36
Char	7	0.84
Boolean	105	12.61
Scalar	31	3.72
Subrange	44	5.28
Set	2	0.24
Array	124	14.89
Record	259	31.09
Pointer	110	13.21
File	87	10.44

Structured type composition

	Int	Rea	Cha	Boo	Sca	Sub	Set	Arr	Rec	Poi
Subrange	44	0	0	0	0	0	0	0	0	0
Set	0	0	0	0	0	2	0	0	0	0
Array	0	0	107	0	1	2	0	0	14	0
Record	154	39	87	323	46	602	1	553	222	112
Pointer	0	0	0	0	0	0	1	0	109	0
File	1	0	82	0	0	0	0	4	0	0

```
************************************
Table 4.9-Logical Size of Procedures
************************************

Size        Count       Percent         Cumulative
   0            6          0.38               0.38
   1          167         10.59              10.97
   2           64          4.06              15.03
   3           76          4.82              19.85
   4           99          6.28              26.13
   5           78          4.95              31.07
   6           86          5.45              36.53
   7           68          4.31              40.84
   8           62          3.93              44.77
   9           56          3.55              48.32
  10           62          3.93              52.25
  11           55          3.49              55.74
  12           42          2.66              58.40
  13           38          2.41              60.81
  14           36          2.28              63.09
  15           42          2.66              65.76
  16           24          1.52              67.28
  17           32          2.03              69.31
  18           26          1.65              70.96
  19           40          2.54              73.49
  20           22          1.40              74.89
  21           25          1.59              76.47
  22           33          2.09              78.57
  23           21          1.33              79.90
  24           24          1.52              81.42
  25           22          1.40              82.82
  26           13          0.82              83.64
  27           12          0.76              84.40
  28           11          0.70              85.10
  29           13          0.82              85.92
  30           24          1.52              87.44
  31            7          0.44              87.89
  32            8          0.51              88.40
  33            7          0.44              88.84
  34           15          0.95              89.79
  35            5          0.32              90.11
  36            9          0.57              90.68
  37            4          0.25              90.93
  38            8          0.51              91.44
  39            7          0.44              91.88
  40            6          0.38              92.26
  41            1          0.06              92.33
  42            4          0.25              92.58
  43            9          0.57              93.15
  44            6          0.38              93.53
  45            8          0.51              94.04
  46            2          0.13              94.17
  47            6          0.38              94.55
  48            1          0.06              94.61
  49            4          0.25              94.86
  50+          81          5.14             100.00
```

****

Table 4.10-Logical Size of Statements

****

Size	If	While	Repea	For	Case	With	Compo
0							
1	2		1				4
2	778	141	24	191	3	35	55
3	473		16	10	16	16	989
4	241	100	12	39	12	108	530
5	360	66	10	15	27	81	276
6	140	40	11	20	15	78	222
7	188	29	9	11	6	55	173
8	112	24	13	8	6	41	130
9	118	16	3	7	11	31	100
10	79	9	6	6	13	37	85
11	95	7	6	2	2	29	81
12	66	8	13	2	5	20	60
13	74	6	6	1	7	15	65
14	55	6	6	2	3	21	57
15	79	11	3	1	7	19	52
16	47	6	3		8	13	50
17	58	7	4		9	17	33
18	26	7	4	1	2	13	39
19	37	2	2	3	6	13	24
20	32	3	1		1	12	25
21	31	4		2	4	10	18
22	17	2	2		7	11	21
23	18	2		1		11	17
24	27	3	2	1	3	9	24
25	16	1	1		2	15	16
26	24	3	3	1		4	18
27	22	3	1	2	1	4	10
28	14	1	1			5	13
29	14	4	3		1	4	16
30	10	5	2	1	2	5	13
31	8	3	1	1	1	6	7
32	11		2			2	7
33	4	1	1		1	2	15
34	12	9	1		1	3	10
35	4	2	2		2	3	8
36	9	1				1	8
37	7	2			1	1	4
38	4	2	3		3	1	5
39	7				1	4	6
40	5	1			1	1	3
41	7	1					4
42	3	1				1	11
43	3		1	3	1	6	2
44	5						2
45	5				1		6
46					2	2	3
47	4				1	2	1
48	1				1	3	6
49	3	1				1	1
50+	64	16	4		17	15	56

Table 4.11-Frequency distribution of statements per procedure

Freq	Assig	Call	Goto	If	While	Repeu	For	Case	With	Compo
0	348	338	1537	619	1163	1432	1345	1433	1033	577
1	223	296	31	338	313	127	178	113	423	353
2	138	202	3	175	70	13	38	13	72	222
3	155	122	5	147	21	2	6	12	21	132
4	134	94		85	10		4	3	13	99
5	81	66		55		1	2	1	6	45
6	79	49	1	23		1			2	46
7	56	53		29			1		2	21
8	39	39		20			2	1		18
9	32	44		19						14
10	42	32		11			1		3	14
11	25	26		11					1	4
12	25	31		8						3
13	15	23		9		1		1		5
14	13	23		8					1	6
15	11	13		2						1
16	13	18		5						6
17	7	13		1						1
18	13	9								
19	10	6								
20	5	13		4						4
21	4	11		2						
22	1	5		1						
23	7	11								
24	5	7		1						
25	6	3								2
26	4	1		2						
27	1	3								1
28	2	1								
29	1	2								
30	1	3								
31	3	1								1
32		2								
33	2	1								
34		1								
35	1									
36	2	1								
37	1	2								
38	2									
39	1									
40	1			1						
41										
42	2									
43										
44										
45	3	1								
46										
47		2								
48		1								
49	1									
50+	12	8		1						2

****
Table 4.12-Syntax Rules Usage
****

Rule	Rule Count	Percent	Rule Name
1	38	0.01	<program>::=<program heading><block>.
2	38	0.01	<program heading>::=program<id>(<file id>{,<file id>})
3	1577	0.40	<block>::=<label dec part><const dec part><type dec part><var dec part> <pro/fun dec part><stmt part>
4	1538	0.39	<label dec part>::=<empty> |
5	39	0.01	label <label>{,<label>}
6	1481	0.38	<const dec part>::=<empty> |
7	96	0.02	const <const def>{,<const def>}
8	611	0.16	<const def>::=<id> = <constant>
9	1650	0.42	<constant >::=<unsigned number> |
10	24	0.01	<sign><unsigned number> |
11	1463	0.37	<constant identifier> |
12	3	0.00	<sign><constant identifier> |
13	373	0.09	<string>
14	1673	0.42	<unsigned number>::=<unsigned integer> |
15	1	0.00	<unsigned real>
16	1507	0.38	<type dec part>::=<empty> |
17	70	0.02	type<type def>{,<type def>} ;
18	644	0.16	<type def>::=<id> = <type>
19	3593	0.91	<type>::=<simple type> |
20	580	0.15	<structured type> |
21	101	0.03	<pointer type>

Table 4.12-Syntax Rules Usage (cont.)

22	94	0.02	<simple type>::=<scalar type> |
23	560	0.14	<subrange type> |
24	3298	0.84	<type id>
25	94	0.02	<scalar type>::= (<id> {,<id>})
26	560	0.14	<subrange type>::= <constant>..<constant>
27	374	0.09	<structured type>::=<unpacked structured type> |
28	206	0.05	packed <unpacked structured type>
29	339	0.09	<unpacked struc.type>::=<array type> |
30	194	0.05	<record type> |
31	20	0.01	<set type>|
32	27	0.01	<file type >
33	339	0.09	<array type>::=array [<index type>] of <component type>
34	339	0.09	<index type>::= <simple type>
35	339	0.09	<component type>::= <type>
36	194	0.05	<record type>::=record <field list> end
37	287	0.07	<field list>::=<fixed part> |
38	23	0.01	<fixed part> ; <variant part> |
39	35	0.01	<variant part>
40	987	0.25	<fixed part>::=<record section> {;<record section>}
41	1075	0.27	<record section>::=<field id>{,<field id>} : <type> |
42	0	0.00	<empty>

Table 4.12-Syntax Rules Usage (cont.)

<table>
<tr><td>43</td><td>57</td><td>0.01</td><td><variant part>::=case<tag field><type id> of <variant>{;variant}</td></tr>
<tr><td>44</td><td>31</td><td>0.01</td><td>tag field>::=<field identifier> : |</td></tr>
<tr><td>45</td><td>26</td><td>0.01</td><td><empty></td></tr>
<tr><td>46</td><td>151</td><td>0.04</td><td><variant>::=<case label list> : (<field list>) |</td></tr>
<tr><td>47</td><td>18</td><td>0.00</td><td><empty></td></tr>
<tr><td>48</td><td>151</td><td>0.04</td><td><case label list>::=<case label>{,<case label>}</td></tr>
<tr><td>49</td><td>180</td><td>0.05</td><td><case label>::=<constant></td></tr>
<tr><td>50</td><td>20</td><td>0.01</td><td><set type>::= set of <base type></td></tr>
<tr><td>51</td><td>20</td><td>0.01</td><td><base type>::=<simple type></td></tr>
<tr><td>52</td><td>27</td><td>0.01</td><td><file type>::=file of <type></td></tr>
<tr><td>53</td><td>101</td><td>0.03</td><td><pointer type>::= ↑ <type id></td></tr>
<tr><td>54</td><td>698</td><td>0.18</td><td><variable declaration part>::=<empty>|</td></tr>
<tr><td>55</td><td>879</td><td>0.22</td><td>var <var.declaration>{;<var.declaration>}</td></tr>
<tr><td>56</td><td>2879</td><td>0.73</td><td><var.declaration>::=<id>{,<id>} : <type></td></tr>
<tr><td>57</td><td>1671</td><td>0.42</td><td><proc/func dec part>::={<procedure or function declaration>;}</td></tr>
<tr><td>58</td><td>1407</td><td>0.36</td><td><procedure or function declaration>::=<procedure declaration>|</td></tr>
<tr><td>59</td><td>214</td><td>0.05</td><td><function declaration></td></tr>
<tr><td>60</td><td>1407</td><td>0.36</td><td><procedure declaration>::=<procedure head><block></td></tr>
<tr><td>61</td><td>512</td><td>0.13</td><td><procedure head>::=procedure<id> ; |</td></tr>
<tr><td>62</td><td>895</td><td>0.23</td><td>procedure <id>(<formal par.sec.>{;<formal par.sec.>})</td></tr>
</table>

Table 4.12-Syntax Rules Usage (cont.)

<table>
<tr><td>63</td><td>977</td><td>0.25</td><td><formal parameter section>::=<parameter group>|</td></tr>
<tr><td>64</td><td>900</td><td>0.23</td><td>var<parameter group>|</td></tr>
<tr><td>65</td><td>0</td><td>0.00</td><td>function<parameter group>|</td></tr>
<tr><td>66</td><td>5</td><td>0.00</td><td>procedure<id>{,<id>}</td></tr>
<tr><td>67</td><td>2021</td><td>0.51</td><td><parameter group>::=<id>{,<id>} : <type id></td></tr>
<tr><td>68</td><td>214</td><td>0.05</td><td><function declaration>::=<function head><block></td></tr>
<tr><td>69</td><td>35</td><td>0.01</td><td><function head>::=function <id>:<result type> |</td></tr>
<tr><td>70</td><td>179</td><td>0.05</td><td>function <id> <formal par sec> {;<formal par sec>}:<result type></td></tr>
<tr><td>71</td><td>214</td><td>0.05</td><td><result type> ::=<type id></td></tr>
<tr><td>72</td><td>1577</td><td>0.40</td><td><stmt part>::=<compound stmt></td></tr>
<tr><td>73</td><td>27766</td><td>7.05</td><td><statement>::=<unlabelled stmt>|</td></tr>
<tr><td>74</td><td>42</td><td>0.01</td><td><label> : <unlabelled stmt></td></tr>
<tr><td>75</td><td>18932</td><td>4.81</td><td><unlabelled stmt >::=<simple stmt> |</td></tr>
<tr><td>76</td><td>8876</td><td>2.25</td><td><struct. stmt></td></tr>
<tr><td>77</td><td>8505</td><td>2.16</td><td><simple stmt >::= <assignment stmt > |</td></tr>
<tr><td>78</td><td>8109</td><td>2.06</td><td><procedure stmt> |</td></tr>
<tr><td>79</td><td>58</td><td>0.01</td><td><goto stmt> |</td></tr>
<tr><td>80</td><td>2260</td><td>0.57</td><td><empty stmt></td></tr>
<tr><td>81</td><td>8165</td><td>2.07</td><td><assignment stmt >::=<variable>:=<expression> |</td></tr>
<tr><td>82</td><td>340</td><td>0.09</td><td><function id>:=<expression></td></tr>
</table>

Table 4.12-Syntax Rules Usage (cont.)

83	24304	6.17	<variable>::=<entire variable>|
84	7614	1.93	<component variable>|
85	2247	0.57	<referenced variable>
86	3476	0.88	<component variable>::=<indexed variable>|
87	3842	0.98	<field designator>|
88	296	0.08	<file buffer>
89	3476	0.88	<indexed variable>::=<array var >[<expr>{,<expr>}]
90	3842	0.98	<field designator>::=<record var > . <field id>
91	296	0.08	<file buffer>::=<file var > ↑
92	2247	0.57	<referenced var >::=<pointer var > ↑
93	30175	7.66	<expression>::=<simple expression>|
94	3753	0.95	<simple expr.><relational operator><simple expr>
95	1718	0.44	<relational operator>::= = |
96	990	0.25	<> |
97	259	0.07	< |
98	149	0.04	<= |
99	102	0.03	>= |
100	259	0.07	> |
101	276	0.07	in

Table 4.12-Syntax Rules Usage (cont.)

102	35947	9.13	<simple expr.>::=<term> |
103	51	0.01	<sign><term>|
104	1683	0.43	<simple exp.><add op.><term>
105	1054	0.27	<add op> ::= + |
106	434	0.11	- |
107	338	0.09	or
108	38605	9.80	<term>::=<factor> |
109	902	0.23	<term><mult. op.><factor>
110	165	0.04	<mult. op>::= * |
111	50	0.01	/ |
112	137	0.03	div|
113	57	0.01	mod|
114	587	0.15	and
115	19900	5.05	<factor>::=<variable>|
116	5683	1.44	<unsigned constant>|
117	1636	0.42	(<expression>) |
118	1684	0.43	<function designator>|
119	384	0.10	<set>|
120	744	0.19	not<factor>

Table 4.12-Syntax Rules Usage (cont.)

121	5683	1.44	<unsigned constant>::=<unsigned number> |
122	5365	1.36	<string>|
123	4783	1.21	<constant id>|
124	1068	0.27	nil
125	59	0.01	<function designator>::=<function id>|
126	789	0.20	<function id> (<actual par.>{,<actual par>}|
127	836	0.21	<standard function>
128	384	0.10	<set>::=[<element list >]
129	340	0.09	<element list>::=<element>{,<element>}|
130	44	0.01	<empty>
131	668	0.17	<element>::=<expression>|
132	129	0.03	<expression> . . <expression>
133	992	0.25	<procedure stmt>::=<procedure id>|
134	3395	0.86	<procedure id>(<actual par>{,<actual par>})|
135	3722	0.95	<standard procedure call>
136	7611	1.93	<actual par >::=<expression> |
137	3015	0.77	<variable>|
138	17	0.00	<procedure id> |
139	0	0.00	<function id>
140	58	0.01	<go to stmt>::= goto <label>
141	2260	0.57	<empty stmt>::=

Table 4.12-Syntax Rules Usage (cont.)

142	3381	0.86	<struct.stmt>::=<compound stmt> |
143	3632	0.92	<conditional stmt> |
144	1070	0.27	<repetitive stmt > |
145	793	0.20	<with stmt>
146	3381	0.86	<compound stmt.>::=begin <stmt>{;<stmt>} end
147	3419	0.87	<conditional stmt>::=<if stmt> |
148	213	0.05	<case stmt>
149	1507	0.38	<if stmt>::=if <expr> then <stmt> |
150	1912	0.49	if <expr> then <stmt> else <stmt>
151	213	0.05	<case stmt>::=case <expr> of <case list el.>{,<case list el. >} end
152	1236	0.31	<case list el.>::=<case label list> : <stmt> |
153	0	0.00	<empty>
154	1554	0.39	<case label list>::=<case label>{,<case label>}
155	556	0.14	<repetitive stmt>::=<while stmt> |
156	183	0.05	<repeat stmt> |
157	331	0.08	<for stmt>
158	556	0.14	<while stmt>::=while <expr> do <stmt>
159	183	0.05	<repeat stmt>::= repeat <stmt>{;<stmt>}until<expr>
160	331	0.08	<for stmt>::=for <control var> :=<for list> do <stmt>
161	793	0.20	<with stmt>::=with<record variable list> do <stmt>
162	962	0.24	<record variable list>::=<record var.>{,<record var>}

Total number of rules applied 393779

Table 4.13-Pascal fragments usage

Order	Static Count	Dynamic Count	Fragment Description
0	38	23	Main program
1	122	73	External files binding overhead
2	1577	260514	Procedure entry overhead
3	7	10	Open and close of local files
			Assignments - right hand side composition
			Constants by type
4	429	2523	Integer
5	9	5	Real
6	254	53641	Char
7	981	15647	Boolean
8	729	9599	Scalar
9	577	14166	Subrange
10	47	18	Set
11	745	12836	Array
12	0	0	Record
13	415	12237	Pointer
			Variables by type
15	244	2506	Integer
16	15	2	Real
17	253	176353	Char
18	49	1574	Boolean
19	47	10864	Scalar
20	319	15580	Subrange
21	5	0	Set
22	242	8501	Array
23	149	5061	Record
24	1175	49739	Pointer
			Expressions
26	454	7994	Integer
27	65	824	Real
28	61	13369	Char
29	256	20993	Boolean
30	3	14	Scalar
31	757	224344	Subrange
32	57	628	Set
33	0	0	Array
34	0	0	Record
35	165	28059	Pointer

Table 4.13-Pascal fragments usage (cont.)

			Value parameters -
			Constants by type
37	334	377	Integer
38	1	0	Real
39	85	4207	Char
40	93	10056	Boolean
41	418	1674	Scalar
42	503	1284	Subrange
43	31	2199	Set
44	305	2	Array
45	0	0	Record
46	21	1	Pointer
			Variables by type
48	209	148	Integer
49	15	0	Real
50	38	10119	Char
51	87	522	Boolean
52	49	3252	Scalar
53	393	2383	Subrange
54	56	3505	Set
55	646	1918	Array
56	183	1987	Record
57	793	69138	Pointer
			VAR parameters
70	211	9250	Integer
71	3	0	Real
72	15	653	Char
73	250	504	Boolean
74	110	1	Scalar
75	180	6604	Subrange
76	8	0	Set
77	472	30106	Array
78	843	38823	Record
79	373	78833	Pointer
80	550	8830	File
81	992	85416	Procedure calls with no parameters
82	3395	110650	Procedure call with parameters
83	121	107891	Standard procedure GET
84	20	32531	PUT
85	170	51	RESET
86	90	34	REWRITE
87	198	15135	READ
88	2239	55548	WRITE
89	211	6004	NEW
90	0	0	DISPOSE

Table 4.13-Pascal fragments usage (cont.)

91	0	0	PAGE
92	516	1399	CMS procedure
93	48	2247	Goto
94	10	92	Interlevel jump
95	1507	158953	If-then
96	1507	53760	If-then =true
97	1912	343425	If-then-else
98	1912	85661	If-then-else = true
99	213	10847	Case statement
100	556	26164	While statement
101	556	145958	While test
102	183	7332	Repeat statement
103	183	59497	Repeat test
104	331	6838	For statement
105	331	70119	For incrementing -testing and return
106	793	49844	With statement
			Relational operators - operand type
107	983	163992	Integer
108	19	0	Real
109	346	201811	Char
110	12	12	Boolean
111	479	80854	Scalar
112	256	17786	Subrange
113	301	14669	Set
114	635	127111	Array
115	1	100	Record
116	721	119260	Pointer
			Relational operators - operand class
118	2852	548656	Constant - Variable
119	70	1192	Constant - Expression
120	727	171797	Variable - Variable
121	92	3994	Variable - Expression
122	12	46	Expression - Expression
			Add operators - operand class
123	695	118792	Constant - Variable
124	47	52	Constant - Expression
125	194	4003	Variable - Variable
126	78	2692	Variable - Expression
127	40	370	Expression - Expression
128	138	7061	Set Union

Table 4.13-Pascal fragments usage (cont.)

			Subtract operator -operand class
129	270	9022	Constant - Variable
130	27	186	Constant - Expression
131	67	689	Variable - Variable
132	41	511	Variable - Expression
133	29	1903	Expression - Expression
134	20	617	Set Difference
			Cr operator - operand class
135	0	0	Constant - Variable
136	0	0	Constant - Expression
137	29	96	Variable - Variable
138	48	1219	Variable - Expression
139	261	75212	Expression - Expression
			Multiply operators - operand class
140	68	2135	Constant - Variable
141	59	350	Constant - Expression
142	22	360	Variable - Variable
143	13	13	Variable - Expression
144	3	0	Expression - Expression
145	3	0	Set product
			Divide operators - operand class
146	111	1168	Constant - Variable
147	50	343	Constant - Expression
148	19	269	Variable - Variable
149	6	0	Variable - Expression
150	1	0	Expression - Expression
			Modulo operator - operand class
151	30	177	Constant - Variable
152	14	20	Constant - Expression
153	5	0	Variable - Variable
154	2	0	Variable - Expression
155	6	0	Expression - Expression
			And operator - operator class
156	0	0	Constant - Variable
157	0	0	Constant - Expression
158	24	8623	Variable - Variable
159	81	10250	Variable - Expression
160	482	105772	Expression - Expression
161	384	12644	Set expressions
162	406	1473	Not operator - variable
163	338	95750	Not operator - expression

Table 4.13-Pascal fragments usage (cont.)

			Constants usage by type
164	6586	383247	Integer
165	23	6	Real
166	1820	272492	Char
167	1102	29097	Boolean
168	2263	100089	Scalar
169	0	0	Subrange
170	0	0	Set
171	4037	35955	Array
172	0	0	Record
173	1068	128469	Pointer
			User functions by type
175	109	921	Integer
176	15	0	Real
177	24	20782	Char
178	343	11816	Boolean
179	3	7	Scalar
180	142	8521	Subrange
181	0	0	Set
182	0	0	Array
183	0	0	Record
184	212	29079	Pointer
186	4	0	Standard function ABS
187	5	4	SQR
188	12	7	ODD
189	85	100215	EOL
190	141	73452	EOF
191	6	0	TRUNC
192	0	0	ROUND
193	158	5719	ORD
194	90	635	CHR
195	208	109293	SUCC
196	47	261	PRED
			Variables within expresions -by type
197	2481	38688	Integer
198	149	833	Real
199	1074	490037	Char
200	1467	57950	Boolean
201	999	119927	Scalar
202	4387	747966	Subrange
203	397	19977	Set
204	2908	303148	Array
205	1187	46724	Record
206	3146	319800	Pointer
207	1705	19964	File

Table 4.13-Pascal fragments usage (cont.)

			Record item access - by type
208	335	5131	Integer
209	65	0	Real
210	52	3245	Char
211	167	16595	Boolean
212	285	34406	Scalar
213	534	75160	Subrange
214	44	46	Set
215	1088	322732	Array
216	298	115251	Record
217	974	102176	Pointer
			Pointed element access - by type
219	17	0	Integer
220	0	0	Real
221	0	0	Char
222	0	0	Boolean
223	0	0	Scalar
224	4	0	Subrange
225	10	0	Set
226	156	0	Array
227	2060	304924	Record
228	0	0	Pointer
			Array element access - by type
230	1946	70657	Integer
231	0	0	Real
232	247	67951	Char
233	17	11313	Boolean
234	250	139	Scalar
235	1024	272025	Subrange
236	0	0	Set
237	0	0	Array
238	0	0	Record
239	0	0	Pointer
241	182	228649	Access to a file buffer-scalar element
242	3	0	Subrange
243	0	0	Set
244	37	36712	Array
245	74	9009	Record
246	0	0	Pointer

5 - Evaluation of the P4-machine

5.1 Introduction

After a computer has been specified on paper, its design must be evaluated to see whether it meets the required standards and to detect the areas of the design which require improvement. This chapter is an exercise in language oriented computer evaluation using the technique of language fragments (Wor72a). The goal of this chapter is to investigate the behaviour of the Pascal P4 Intermediate code machine (Nor74a) as a Pascal engine. Although the P4 Intermediate Language was designed to meet portability constraints, the P4-machine has several other advantages as a starting point towards a Pascal machine:

-it is a well structured design

-it is Pascal oriented

-has been implemented in hardware and software.

A proposed machine can be evaluated on the basis of its resource consumption which defines a "cost measure" for running a workload in this machine. If the machine is oriented towards a high-level language, then the evaluation problem can be restated as: "evaluate the cost measure of running a set of programs representing the workload in the proposed machine".

Since it is not always possible to use the real workload to which the machine is going to be applied, we must use an "anticipated workload" for evaluation. The anticipated workload used in this study is the set of Pascal programs studied in the last chapter.

In order to be independent of low-level implementation considerations we will define a cost measure which is function only of the amount of information used to store and run the anticipated workload

(Wor72a). This cost measure has three components:

- i-the static size of programs
- ii-the number of memory references at run time.
- iii-the amount of information - transferred to and from store at run time.

More specifically, the cost measure is a function of six cost parameters defined as:

- a1-code size in bits.
- a2-static data size in bits.
- a3-the number of memory fetches to instructions during execution.
- a4-number of memory references to data " "
- a5-number of instruction bits fetched
- a6-number of data bits accessed " "

The total cost measure is a weighted sum of these parameters:

$$CM=\sum_{i=1}^{6} w_i.a_i$$

where the w_i are constants used to convert the cost parameters to a common measuring unit.

The cost of parameters of the workload can be estimated by calculating the parameters for each code fragment, and then accumulating these using knowledge of the static and dynamic usage of code fragments in the workload.

Let us associate with each code fragment f_i a cost vector $v_i[i..6]$. We can thus generate the attribute matrix, A_{ij}, where $i \varepsilon [1..6]$ and $j \varepsilon [1..n]$ n being the number of fragments. The attribute matrix thus contains the cost parameters of all fragments.

If the static and dynamic counts of the usage of code fragment f_i in the workload are S_i and D_i, then the total cost parameters a_i, can be expressed as:

$$a_i = \sum_{k=1}^{n} A_{ik} . S_k \qquad (i=1,2)$$

Eq. 1

$$= \sum_{k=i}^{n} A_{ik} D_k \qquad (i=3..6)$$

Since the weighting factors w_i are technology dependent, we will simplify the evaluation procedure to the study of the six total cost parameters - a_i. Thus instead of dealing with a single scalar cost measure, we will analyse a cost measure which is a vector:

$$CM' = (a1, a2, a3, a4, a5, a6).$$

Based on the data presented in (Nor74a, and Jen73a) we describe an initial implementation for the P4 machine. With this description we can evaluate the cost measure for each one of the P4 instructions. Since fragments are sequences of instructions, we can then evaluate the cost parameters for each of the fragments, and hence the cost measure CM' to run the anticipated workload.

When we know the total cost measure CM' we can now evaluate how possible modifications in the P4 machine would alter the total cost measure, and by some iterations arrive at an improved Pascal machine.

5.2 The P4_machine

5.2.1 Introduction

The P4-machine is the abstract machine defined by the intermediate language used in the P4 Pascal compiler (Nor74a). It is a stack machine with zero address instructions. The basic operations of the P4-machine are derived out of logical requirements due to Pascal with extra operations introduced for matching the needs of data structure access.

The P4-machine has 6 registers and one memory. The registers are:

-PC	the program counter
-SP	the stack pointer
-MP	the mark pointer
-NP	the new pointer
-EP	the extreme stack pointer
-IR	the instruction register.

The memory is divided in two parts: one for code and one for data. IR contains the instruction currently in execution and PC is a pointer to the next instruction to be executed. The meaning of the other registers will become clear in the course of the description of the P4-machine.

5.2.2 Data memory structure

The data memory has three parts: the stack, the constants area and the heap. The stack grows from address 0 upwards and contain all directly accessible data, the register SP pointing to the first free position above the stack. The constants area occupies the other extreme of the memory and contains strings, reals, sets, small integers and boundary-pairs (for range check). Other constants are stored in instruction fields. The heap area, contains all dynamically created data, grows downwards from the constants area and its growth is directed by the use of the standard procedures new and dispose. The register NP, the new pointer, points to the beginning of the heap area. The register EP, points to the maximum position the stack may grow when a given procedure is active (fixed at compile time) such that a condition of data memory overflow can be detected when EP and NP meet.

The stack has a further level of structuring. It consists of a series of activation records, each one generated by the call of a user procedure. Each activation record, in turn, has four separated areas: the mark stack, parameters area, local variables and temporary storage areas.

The mark stack contains 5 fields: function return value, static link, dynamic link, extreme stack pointer and return address of the calling procedure.

The parameter section has three parts: pointers to implement var parameters and addresses of structured value parameters (of type array or record) constitute the first part. The second part contains the value parameters which are not of array or record type; and the third section contains the value of the parameters of type array or record.

5.2.3 Procedure call and variable access

Call to both procedures and functions is executed the same way. It is realized in four phases:

1- a "mark stack" instruction is executed to fill the links.
2- parameter passing
3- proper "call" instruction transfers control to the called procedure
4- enter phase, which allocates space for local variables and copies the value parameters of type array or record.

The return phase resets the stack to its state before the call and does the necessary adjusting if a function value is being returned.

Directly accessible variables are defined by a pair: (level-difference, offset), where level-difference is the difference between the static level of the procedure actually in execution and the lexicographical level of the accessed variable (a level-difference of 0 means a local variable to the procedure in execution). A level-difference of n implies then n indirections in the static chain to obtain the address of the data area where this variable is located. Global variables can be accessed directly, only by their offset. Indirectly accessible variable like reference parameters and pointed variables are accessed via the absolute stack address.

5.2.4 Data and instruction sizes

The set of instructions of the P4 machine defines a P4 processor. We will assume that the P4 processor uses the following data formats for Pascal data types:

8 bits for characters, booleans and user defined scalars
32 bits for integers
64 bits for reals and sets
24 bits for pointers.

The P4-machine instructions can be short or long. A short instruction has only one field OP whereas a long instruction has three fields: OP, P and Q. The meaning of these fields will be clarified in the description of the instruction set which follows.

The instruction container sizes we defined for the P4 machine are based on the ones for the P4 machine given in Nor74a.

Since we would like to extend the instruction set incorporating new instructions, a value of 8 bits for the OP field, (instead of the 6 bits allowing a maximum of 64 instructions) is reasonable. The P field is used for storing lexical level data, so we have followed Nor74a in assigning 4 bits to this field. Since we would like a long instruction size to be a multiple of the short, a choice of 20 bits for the Q field (used for address) was made.

5.2.5 The instruction set description

The evaluation of the cost measure of running the anticipated workload in the P4-machine involves the evaluation of the cost parameters for each P4-machine instruction.

To evaluate the cost parameters of instructions, we need a detailed description of the actions taken by the P4-processor when executing each instruction. For this description we need to postulate a set of properties and elements in the P4-processor which are not actually seen at the intermediate language level. These properties are:

- it has 3 registers ac1, ac2, ac3 in which operations of the form ac= ac1 op ac2 can be realised; with op being a arithmetic or logical operation.

- it has a counter aux and a flip-flop flag.

- the processor has a primitive function findbase(p), which follows the static chain for p nodes and returns the address of the data area at lexical level (n-p) where n is the lexical level of the active procedure.

- the registers in the processor can be incremented or decremented with the primitives inc and dec.

The description of the instruction set semantics will be made using a dialect of Pascal with the following alterations :

- data movements are indicated by "=".

- if a register x contains an address of a memory position then x↑ denotes the contents of that location.

- the register SP has the special property that it is automatically pre-incremented when in the left-hand side of an assignment and post-decremented when in the right-hand side.

- the register IR, used to hold instructions, has the three fields, denoted as IR.op, IR.p and IR.q.

The P4-machine as defined in (Nor74a) and in (Jen73a) has a set of 64 basic instructions, some of them can have variants according to the type of the data being dealt with. See appendix 2 for an informal description of the instruction set.

The instruction set can be divided in 9 groups according to the type of operation performed. The groups are:

1- polish binary operators
2- " unary "
3- relational operators
4- procedure call instructions
5- branches
6- address manipulation
7- loads
8- stores
9- others

We proceed to the description of the instruction set:

Group 1

Contains the following instructions:

- on integers	:ADI, SBI,DVI, MPI, MOD
- on boolean	:AND, IOR
- on sets	:DIF, INT, UNI
-on reals	:ADR, SBR, MPR, DVR

The instruction format is simply : opcode, where opcode occupies the OP field. The instruction execution can be described as :

```
opcode-group 1: begin
                    ac1 = SP↑; (*read first operand*)
                    ac2 = SP↑  (*second*)
                    ac3 = ac1 op ac2; (*executes operation*)
                    SP  = ac3; (*result back to back*)
                end;
```

Group 2 - Unary operators

The instruction format is opcode. It contains the instructions:

- on integers	: ABI, INC, DEC, NGI, SQI
- on boolean	: NOT
- on reals	: ABR, FLO, FLT, SQR
- on sets	: SGS
- transfer	: CHR, ORD, TRC, ODD.

```
<opcode-group2>: begin
                    ac1 = SP ; (*read top of stack*)
                    ac3 = op ac1 ; (*do operation*)
                    SP  = ac3; (*store back*)
                 end;
```

Group 3 Relational operators.

There are two formats of instructions in this group:

i-opcode	a simple opcode of 8 bits for relational operators between simple types, i.e. - integers, reals, characters, scalars, sets and pointers.
ii-opcode, n	a 32 bit instruction for arrays and records. The parameter n occupies the address field of the instruction and specifies the size of the elements being compared.

Their operation is:

```
<opcode-group3-i> : begin
                        ac1 = SP↑; (*read first operand*)
                        ac2 = SP↑; (*read second       *)
                        ac3 = ac1 op ac2 ; (*compare*)
                        SP↑ = ac3 ; (*boolean back to stack*)
                    end;
<opcode-group3-ii>: begin
                        aux = IR.q; (*size of element to counter*)
                        flag= 1   ; (*flip-flop set*)
                        while flag and(aux > 0)
                         do begin
                          if ac1↑= ac2↑
                            then begin
                              dec(aux)
                              inc(ac1);
                              inc(ac2)
                            end
                            else flag = 0
                         end
                         ac1 = flag and (aux = 0);
                         SP↑ = ac1;
                     end;
```

Note : in the group ii there can be some variations according to the type of comparison being performed.

Group 4 - Procedure calls

The machine has a set of 4 instructions for executing procedure calls.

MST, p : mark stack to fill the links

CUP, p, q : proper call

ENT, p, q : enter - updates stack pointer

RET, p : return and adju stack pointer

4.1 - mark stack

```
MST  p  :  begin
             acl = MP + 2 (*link address*)
             aux = IR.p
             while aux> 0
               do
                begin
                  ac3 = acl ;
                  acl = acl↑; (*read link of level-1*)
                  dec(aux) ;
               end; (*aux = 0*)
             SP = SP + 2;
             SP↑= acl ;       (*copy link static*)
             SP↑= MP  ;       (*dynamic link*)
             SP↑= EP  ;       (*pass extreme stack pointer*)
           end;
```

4.2 - call user procedure

```
CUP  p, q  :  begin
                acl = SP - (IR.p+4); (*adjust the base of
                                        activation rec*)
                MP  = acl;
                acl = acl + 4;       (*PC address*)
                ac↑ = PC;            (*save return program counter*)
              end;
```

4.3 - enter

```
ENT  p, q  :  begin
                if IR.p = 1
                   then
                    SP = MP + IR.q   (*local variable size*)
                   else
                    EP = SP + IR.q   (*extreme stack*)
              end;
```

4.4 - return

```
RET  p      :  begin
                 if IR.p = 0
                   then SP = MP -1   (*procedure return*)
                   else SP = MP;
                 PC = (MP+4)↑;   (*return PC*)
                 EP = (MP+3)↑;   (*return EP*)
                 MP = (MP+2)↑;   (*return MP*)

               end
```

Group 5 - Jumps

There are 4 instructions in this group: FJP q, UJP q, XJP and UJC.

```
FJP q    :  begin
               acl = SP↑; (*read top of stack*)
               if acl = 0
                  then
                     PC = IR.q
            end;

UJP q    :  begin
               PC = IR.q
            end;

XJP q    :  begin
               acl = SP↑; (*read index from stack*)
               PC  = acl + IR.q
            end;

UJC      :  begin
               halt       (*error in case statement*)
            end;
```

Group 6 - Address manipulation instructions

LAO, LCA, LDA, IXA are in this group.

```
LAO q; LCA, q :  begin
                    SP↑ = IR.q
                 end;

LDA p, q      :  begin
                    acl = findbase(p); (*locate address p levels
                                                   down*)
                    ac3 = acl + IR.q ; (*index*)
                    SP↑ = ac3          (*address to stack*)
                 end;
```

```
IXA  q :  begin
             ac1 = SP↑ ;      (*index*)
             ac1 = ac1 * IR.q (*scale by size *)
             ac2 = SP↑ ;      (*array base*)
             ac3 = ac1 + ac2 ;(*element address*)
             SP↑ = ac3 ;      (*address is stacked*)
          end;
```

Group 7 - Loads

LOD, LDO, LDC, LCI, IND are in this group.

```
       LOD  p, q : begin
                      ac1 = findbase(p) ; (*get address of activation rec*)
                      ac1 = ac1 + IR.q  ; (*offset inside*)
                      ac1 = ac1 ↑         (*write on stack top*)
                   end;

   LDO  q ; LCI, q : begin
                      ac = IR.q ;         (*data obsolute address*)
                      ac1= ac1↑ ;         (*read*)
                      SP↑= ac1  ;         (*write on stack*)
                   end;

          LDC  q : begin
                      ac1= IR.q;          (*data is immediate*)
                      SP↑= ac1 ;          (*write back on stack*)
                   end;

          IND  q : begin
                      ac1= SP↑;           (*address is on stack*)
                      ac1= ac1 + IR.q     (*offset if necessary)
                      ac1= ac1 ↑ ;        (*read data*)
                      SP↑= ac1   ;        (*write on top of stack*)
                   end;
```

Group 8 - Store

STR, STO, SRO, MOV are in this group.

```
STR  P, q : begin
                acl = findbase(p)  (*get address of activation rec*)
                acl = acl + IR.q   (*offset*)
                ac2 = SP↑ ;        (*data to be stored*)
                acl↑= ac2 ;        (*store at address*)
              end;

  SRO  q    : begin
                acl = IR.q;        (*absolute address*)
                ac2 = SP↑          (*data to be moved*)
                acl↑= ac2;         (*store*)
              end;

  STO       : begin
                acl = SP↑;          (*address is in stack*)
                ac2 = SP↑;          (*data also*)
                ac2 = acl;          (*move*)
              end;

  MOV  q    : begin
                acl = SP↑ ;         (*source address*)
                ac2 = SP↑ ;         (*destination address*)
                aux = IR.q;         (*size of data*)
                while aux> 0
                  do
                   begin
                     MR = acl↑      (*read to memory register*)
                     ac2↑ = MR;     (*move*)
                     dec(aux); inc(acl); inc(ac2);
                  end;
               end;
```

Group 9 - Bounds check

```
CHK  q :  begin
              acl = IR.q;      (*address of bound pair*)
              ac2 = SP↑ ;      (*bound to be checked*)
              acl = acl↑;      (*read bound*)
              if acl > ac2
                then error;
              acl = IR.q + 1;
              acl = acl↑
              if acl < ac2
                then error
          end;
```

5.3 Cost parameters of instructions

Using the instruction definition above we can determine the cost parameters for the components of the P4 instruction set. These are shown in table 5.1.

The cost parameters of some instructions may vary according to the level of addressing or the size of the data being operated upon. So, in order to proceed with the evaluation experiment we will make the following assumptions:

1- all data accesses are to either the local or global activation records. The effect here is to ignore the form by which data in intermediate levels is accessed, e.g. using a display like the Burroughs B-6500 or a chain as in the IBM 370 (P4 implementation).

2- we have assumed both EOL and EOF to be standard functions implemented through calls.

3- String sizes are assumed to be 10 bytes long. This apply also for record sizes.

5.4 Evaluating the attribute matrix and the total fragment cost

The cost parameters of a fragment can be derived by a suitable addition of the cost parameters of the P4 instructions of which the fragment is composed.

Some assumptions had to be made for the evaluation of the atrribute matrix since we have not collected all the necessary information. The reason for this is that a complete data collection for fragment analysis would require a very large number of fragments and the overhead in terms of monitoring instructions would be too large. If the number of code fragments is large, and in consequence, the code overhead is large, the procedures will tend to be larger than the maximum limit of 12Kbytes of code which is a restriction in our version of the P4 compiler. To run, then would imply to break the larger procedures in smaller ones, which is not only difficult but could mask the results of the experiment.

The assumptions are:

1- the proportion of variables which is directly accessible is equal to the proportion which is indirectly accessible.

2- we have not taken into account the total cost for standard procedures and functions. The only cost we accounted was the linkage cost, i.e. call (a simple branch and link) and return. For the standard functions: CHR, ORD, SUCC, and PRED we assumed that a simple instruction was inserted in their place.

3- we have ignored record item access cost since its exact evaluation would require much more information than it is available. See section 6.5 for a discussion of this case.

We have also simplified the total fragment cost by ignoring some fragments, whose utilization is very low, or whose accounting is controversial such as:

- program linkage and external files
- operations with reals and in some sets

We have introduced a new fragment which is not in the original set: range check (no. 255), whose count can be derived form the counts of assignments, array accesses and value parameter passing.

The process of attribute matrix generation for the P4 machine is presented in appendix 3. The resulting matrix is shown in table 5.2.

Using the attribute matrix and the static and dynamic distribution of fragments we can now derive the total cost according to eq. 1, i.e. $CM' = (a_1, a_2, a_3, a_4, a_5, a_6)$

$$\text{where} \quad a_i = \begin{cases} \sum_{k=1}^{n} A_{i,k}.S_k & \text{for } i=1,2 \\ \sum_{k=1}^{n} A_{i,k}.D_k & \text{for } i=3,\ 4,\ 5,\ 6. \end{cases}$$

5.5 Conclusion

There are two possible routes to follow when we have found the total cost measure for the anticipated workload:

1- evaluate the attribute matrix for a different machine, e.g. the IBM 370 and make a comparative performance evaluation analysis.

2- we can study the fragments which are more expensive to implement and suggest alternative constructs. They will give rise to a new attribute matrix and a new cost measure, which shows the effect of this particular change on the total cost. In this case, we are using the technique of fragments as a tool for design improvement.

Since our interest is to determine the more important primitives of the language and find optimum implementations for them we will take the second route. To proceed in a systematic way, we look at the fragment cost matrix, which is presented in table 5.3. From this table we take for each of the cost parameters the 10 most expensive fragments and arrange them in order of expense. This is shown in table 5.4.

From this table, we know which are the areas of the machine that need attention since they are using most of the resources. But, of the fragments displayed above, not all of them are capable of further optimization (for example, procedure calls are implemented in an optimized way in the P4 machine). However, there are some fragments whose cost can be decreased, and we shall concentrate our first iteration step in the following areas, each of which may involve one or more fragments:

1- array and pointer access

2- range checks

3- arithmetic and relational operators

4- assignments

5- for statement.

Table 5.1 - Cost parameters of P4 instruction set.

Instruction	a1	a2	a3	a4	a5	a6
ADI	8	0	1	3	8	96
SBI	8	0	1	3	8	96
MPI	8	0	1	3	8	96
DVI	8	0	1	3	8	96
MOD	8	0	1	3	8	96
AND	8	0	1	3	8	24
IOR	8	0	1	3	8	24
DIF	8	0	1	3	8	192
INT	8	0	1	3	8	192
UNI	8	0	1	3	8	192
ADR	8	0	1	3	8	192
SBR	8	0	1	3	8	192
MPR	8	0	1	3	8	192
DVR	8	0	1	3	8	192
INC	32	0	1	2	32	64
DEC	32	0	1	2	32	64
ABI	8	0	1	2	8	64
NGI	8	0	1	2	8	64
SQI	8	0	1	2	8	64
NOT	8	0	1	2	8	16
ABR	8	0	1	2	8	128
FLO	-	-	-	-	-	-
FLT	-	-	-	-	-	-
SQR	8	0	1	2	8	128
CHR	8	0	1	2	8	40
ORD	8	0	1	2	8	40
ODD	8	0	1	2	8	40
TRC	8	0	1	2	8	96
SGS	8	0	1	2	8	96
Relational int.	8	0	1	3	8	72
" chr. bool	8	0	1	3	8	24
" scalar	8	0	1	3	8	24
real	8	0	1	3	8	136
sets	8	0	1	3	8	104
pointer	8	0	1	3	8	56
array/rec	32	0	1			
MST	32	0	1	5	32	120
CUP	32	0	1	1	32	24
ENT	32	0	1	0	32	0
RET	32	0	1	3	32	72
FJP	32	0	1	1	32	8
UJP	32	0	1	0	32	0
XJP	32	0	1	1	32	32
LCA	32	0	1	1	32	24
LAO	32	0	1	1	32	24
LDA	32	0	1	1	32	24
IXA	32	0	1	3	32	96
LOD	32	0	1	2	32	dd.
LDO	32	0	1	2	32	dd.
LDC	32	0	1	1	32	dd.

Con't of Table 5.1

Instruction	a1	a2	a3	a4	a5	a6
IND	32	0	1	3	32	dd.
LCI	32	dd	1	2	32	dd.
STR	32	0	1	2	32	dd.
SRO	32	0	1	2	32	dd.
STO	8	0	1	3	8	dd.
MOV	32	dd	1	8	32	240
CHK	32	64	1	3	32	96

NOTE: dd means data type dependent.

Table 5.2- Attribute matrix for the P4 machine

Fragment	a1	a2	a3	a4	a5	a6
4	56	0	2	2.3	56	104
5	0	0	0	0	0	0
6	56	0	2	2.3	56	32
7	56	0	2	2.3	56	32
8	56	0	2	2.3	56	32
9	56	0	2	2.3	56	104
10	56	64	2	2.3	56	200
11	64	80	2	9	64	264
12	64	80	2	9	64	264
13	56	0	2	2.3	56	80
15	52	0	2	5	52	152
16	0	0	0	0	0	0
17	52	0	2	5	52	56
18	52	0	2	5	52	56
19	52	0	2	5	52	56
20	52	0	2	5	52	152
21	52	0	2	5	52	264
22	64	0	2	9	64	264
23	64	0	2	9	64	264
24	52	0	2	5	52	104
26	24	0	1	2.3	24	136
27	0	0	0	0	0	0
28	24	0	1	2.3	24	40
29	24	0	1	2.3	24	40
30	24	0	1	2.3	24	40
31	24	0	1	2.3	24	136
32	24	0	1	2.3	24	264
33	0	0	0	0	0	0
34	0	0	0	0	0	0
35	24	0	1	2.3	24	104
37	32	0	1	1	32	32
38	0	0	0	0	0	0
39	32	0	1	1	32	8
40	32	0	1	1	32	8
41	32	0	1	1	32	8
42	32	0	1	1	32	32
43	32	64	1	2	32	64
44	128	80	4	11	128	312
45	0	0	0	0	0	0
46	32	0	1	1	32	24
48	32	0	1	2.5	32	60
49	0	0	0	0	0	0
50	32	0	1	2.5	32	24
51	32	0	1	2.5	32	24
52	32	0	1	2.5	32	24
53	32	0	1	2.5	32	60
54	32	0	1	2.5	32	108
55	128	0	4	11	128	312
56	128	0	4	11	128	312
57	32	0	1	2.5	32	48
59	0	0	0	0	0	0
60	0	0	0	0	0	0
61	0	0	0	0	0	0
62	0	0	0	0	0	0
63	0	0	0	0	0	0
64	0	0	0	0	0	0
65	0	0	0	0	0	0
66	0	0	0	0	0	0
67	0	0	0	0	0	0
68	0	0	0	0	0	0
70	32	0	1	1	32	24
71	32	0	1	1	32	24
72	32	0	1	1	32	24
73	32	0	1	1	32	24
74	32	0	1	1	32	24
75	32	0	1	1	32	24
76	32	0	1	1	32	24
77	32	0	1	1	32	24
78	32	0	1	1	32	24
79	32	0	1	1	32	24
80	32	0	1	1	32	24

Table 5.2-Attribute matrix for the P4 machine (cont.)

Fragment	a1	a2	a3	a4	a5	a6
81	64	0	3	9	96	216
82	64	0	3	9	96	216
83	32	0	2	2	64	48
84	32	0	2	2	64	48
85	32	0	2	2	64	48
86	32	0	2	2	64	48
87	32	0	2	2	64	48
88	32	0	2	2	64	48
89	32	0	2	2	64	48
90	32	0	2	2	64	48
91	32	0	2	2	64	48
92	32	0	2	2	64	48
93	32	0	1	0	32	0
94	32	0	1	0	32	0
95	32	0	1	1	32	8
96	0	0	0	0	0	0
97	64	0	1	1	32	8
98	0	0	1	0	32	8
99	664	64	7.5	9	216	280
100	32	0	1	1	32	8
101	64	0	2	1	64	8
102	0	0	0	0	0	0
103	32	0	1	1	32	8
104	128	0	4	6	128	176
105	232	0	8	14	232	400
106	64	0	2	2	64	48
107	72	0	3	7	72	191
108	0	0	0	0	0	0
109	72	0	3	7	72	65
110	72	0	3	7	72	65
111	72	0	3	7	72	65
112	72	0	3	7	72	191
113	72	0	3	6	72	264
114	96	80	3	10	96	264
115	96	80	3	10	96	264
116	72	0	3	7	72	149
123	72	0	3	6.5	72	204
124	40	0	2	4	40	128
125	72	0	3	8	72	248
126	40	0	2	5.5	40	172
127	8	0	1	3	8	96
135	0	0	0	0	0	0
136	0	0	0	0	0	0
137	72	0	3	8	72	80
138	40	0	2	5.5	40	52
139	8	0	1	3	8	24
162	40	0	2	4.5	40	44
163	8	0	1	2	8	16
175	96	0	3	9	96	216
176	96	0	3	9	96	216
177	96	0	3	9	96	216
178	96	0	3	9	96	216
179	96	0	3	9	96	216
180	96	0	3	9	96	216
181	96	0	3	9	96	216
182	96	0	3	9	96	216
183	96	0	3	9	96	216
184	96	0	3	9	96	216
186	32	0	2	2	64	48
187	32	0	2	2	64	48
188	32	0	2	2	64	48
189	32	0	2	2	64	48
190	32	0	2	2	64	48
191	32	0	2	2	64	48
192	32	0	2	2	64	48
193	8	0	1	2	8	40
194	8	0	1	2	8	40
195	32	0	1	2	32	64
196	32	0	1	2	32	64
227	32	0	1	2.5	32	60
230	118	0	3.7	7	118	198
255	32	64	1	3	32	96

Table 5.3-Cost measure for the P4 machine

Fragment	a1	a2	a3	a4	a5	a6
4	24024	0	5046	5803	141288	262392
5	0	0	0	0	0	0
6	14224	0	107282	123374	3003896	1716512
7	54936	0	31294	35988	876232	500704
8	40824	0	19198	22078	537544	307168
9	32312	0	28332	32582	793296	1473264
10	2632	3008	36	41	1008	3600
11	47680	59600	25672	115524	821504	3388704
13	23240	0	24474	28145	685272	978960
15	12688	0	5012	12530	130312	380912
17	13156	0	352706	881765	9170356	9875768
18	2548	0	3148	7870	81848	88144
19	2444	0	21728	54320	564928	608384
20	16588	0	31160	77900	810160	2368160
21	260	0	0	0	0	0
22	15488	0	17002	76509	544064	2244264
23	9536	0	10122	45549	323904	1336104
24	61100	0	99478	248695	2586428	5172856
26	10896	0	7994	18386	191856	1087184
28	1464	0	13369	30749	320856	534760
29	6144	0	20993	48284	503832	839720
30	72	0	14	32	336	560
31	18168	0	224344	515991	5384256	30510784
32	1368	0	628	1444	15072	165792
33	0	0	0	0	0	0
34	0	0	0	0	0	0
35	3960	0	28050	64515	673200	2917200
37	10688	0	377	377	12064	12064
39	2720	0	4207	4207	134624	33656
40	2976	0	10056	10056	321792	80448
41	13376	0	1674	1674	53568	13392
42	16096	0	1284	1284	41088	41088
43	992	1984	2199	4398	70368	140736
44	39040	24400	8	22	256	624
46	672	0	1	1	32	24
48	6688	0	148	370	4736	8880
50	1216	0	10119	25298	323808	242856
51	2784	0	522	1305	16704	12528
52	1568	0	3252	8130	104064	78048
53	12576	0	2383	5958	76256	142980
54	1792	0	3505	8763	112160	378540
55	82688	0	7672	21098	245504	598416
56	23424	0	7948	21857	254336	619944
57	25376	0	69138	172845	2212416	3318624

Table 5.3-Cost measure of P4 machine (cont.)

Fragment	a1	a2	a3	a4	a5	a6
70	6752	0	9250	9250	296000	222000
71	96	0	0	0	0	0
72	480	0	653	653	20896	15672
73	8000	0	504	504	16128	12096
74	3520	0	1	1	32	24
75	5760	0	6604	6604	211328	158496
76	288	0	0	0	0	0
77	15104	0	30106	30106	963392	722544
78	26976	0	38823	38823	1242336	931752
79	11936	0	78833	78833	2522656	1891992
80	17600	0	8830	8830	282560	211920
81	63488	0	256248	768744	8199936	18449856
82	217280	0	331950	995850	10622400	23900400
83	3872	0	215602	215602	6899264	5174448
84	640	0	65062	65062	2081984	1561488
85	5440	0	102	102	3264	2448
86	2880	0	68	68	2176	1632
87	6336	0	30270	30270	968640	726480
88	71648	0	111096	111096	3555072	2666304
89	6752	0	12008	12008	384256	288192
90	0	0	0	0	0	0
91	0	0	0	0	0	0
92	16512	0	2798	2798	89536	67152
93	1536	0	2247	0	71904	0
94	320	0	92	0	2944	0
95	48224	0	158953	158953	5086496	1271624
96	0	0	0	0	0	0
97	122368	0	343425	343425	10989600	2747400
98	0	0	85661	0	2741152	685288
99	141432	13632	81353	97623	2342952	3037160
100	17792	0	26164	26164	837248	209312
101	35584	0	291916	145958	9341312	1167664
102	0	0	0	0	0	0
103	5856	0	59497	59497	1903904	475976
104	42368	0	27352	41028	875264	1203488
105	76792	0	560952	981666	16267608	28047600
106	50752	0	99688	99688	3190016	2392512
107	70776	0	491976	1147944	11807424	31322472
108	0	0	0	0	0	0
109	24912	0	605433	1412677	14530392	13117715
110	864	0	36	84	864	780
111	34488	0	242562	565978	5821488	5255510
112	18432	0	53358	124502	1280592	3397126
113	21672	0	44007	88014	1056168	3872616

Table 5.3-Cost measure of P4 machine (cont.)

Fragment	a1	a2	a3	a4	a5	a6
114	60960	50800	381333	1271110	12202656	33557304
115	96	80	300	1000	9600	26400
116	51912	0	357780	834820	8586720	17769740
123	84528	0	393882	853411	9453168	26783976
124	7880	0	1902	3804	38040	121728
125	22104	0	15954	42544	382896	1318864
126	5600	0	6432	17688	128640	553152
127	632	0	2276	6828	18208	218496
135	0	0	0	0	0	0
136	0	0	0	0	0	0
137	3816	0	26157	69752	627768	697520
138	5160	0	22938	63080	458760	596388
139	5944	0	180984	542952	1447872	4343616
162	16240	0	2946	6629	58920	64812
163	2704	0	95750	191500	766000	1532000
175	10464	0	2763	8289	88416	198936
176	1440	0	0	0	0	0
177	2304	0	62346	187038	1995072	4488912
178	32928	0	35448	106344	1134336	2552256
179	288	0	21	63	672	1512
180	13632	0	25563	76689	818016	1840536
181	0	0	0	0	0	0
182	0	0	0	0	0	0
183	0	0	0	0	0	0
184	20352	0	87237	261711	2791584	6281064
186	128	0	0	0	0	0
187	160	0	8	8	256	192
188	384	0	14	14	448	336
189	2720	0	200430	200430	6413760	4810320
190	4512	0	146904	146904	4700928	3525696
191	192	0	0	0	0	0
192	0	0	0	0	0	0
193	1264	0	5719	11438	45752	228760
194	720	0	635	1270	5080	25400
195	6656	0	109293	218586	3497376	6994752
196	1504	0	261	522	8352	16704
227	71904	0	304924	762310	9757568	18295440
230	411112	0	1561714	2954595	49806030	83572830
255	161280	322560	664831	1994493	21274592	63823776
TOTAL	2877440	476064	10271800	21239942	295171894	511935301

Table 5.4 The 10 most expensive fragments arranged by cost parameter

a1	a3	a4	a5	a6
array access	array access	array access	array access	array access
procedure call	range check	range check	range check	range check
range check	relop char[1]	relop char	for-body	relop array
case statement	relop int	relop int.	relop array	assign subrange
if-then	aritop CV[2]	procedure call	relop int.	for-body
procedure call	relop array	for-body	if-then-else	aritop CV
aritop CV	pointer acc.	assign char	procedure call	procedure call
array param.	assign char	aritop CV	pointer acc.	procedure call
for-body	if-then-else	relop pointer	aritop CV	pointer access

Notes :
1-relop stands for relational operation between elements of the given type.
2-aritop CV stands for arithmetic operation between a constant and a variable.

Chapter 6 - Improving the P4-machine

6.1 Introduction

Given the information about usage patterns of a high-level language and a proposed language oriented machine, there are two possible types of optimization which can be made on the latter:

a- agglutination of primitives - a sequence of instructions which appears very often in the object code is replaced by a single one. (McKe67a)

b- container optimization - the most common forms of data and instructions are coded with fewer bits, in different variants of Huffman coding. The total effect is to decrease "information redundancy" (Wil72a) although the meaning of the primitives is not changed. A good example of this technique is given by Wilner (reference above) in the design of the Burroughs B-1700 S-languages, which are forms of defining language oriented machines which are to be interpreted by the micro-programming system of the B-1700. A full study of possible optimizations using this technique would require a different set of data about programming language usage patterns e.g. patterns of data size and address usage and it will not be dealt with here. These two types of optimization constitute only one of the design steps towards a language-oriented machine. Following this step a second step must be made attempting to adapt the hardware low level mechanisms to the high level requirements of the machine (e.g. the use of the fast registers as top of the stack as in the B-5500).

The main objective of this chapter is to generate an improved version of the P4 machine described in chapter 5. Two main alterations will be introduced:

a- sequences of code in very costly fragments will be coded as single instructions.

b- the descriptor mechanism (described in chapter 3) will be incorporated into the machine.

After making these changes, we use the information about fragment usage to calculate the variation in the cost measure. The real decision whether this change should be introduced or not, can only be made by the implementor when considering the trade-off between: the gain that the change will introduce to the cost measure and the cost of implementing these changes (Wor72a). If the reduction in the cost measure offsets the cost of implementing the proposed changes then they should be implemented.

We presented as a conclusion of chapter 5, the constructs which make the highest demand on the resources. We propose alternative constructs for these fragments or fragment groups in such a way as to decrease the cost measure associated with that fragment.

The final result of this chapter is a demonstration of the use of the methodology described earlier in designing a machine, and a proposal for a Pascal machine based on the P4 machine with a descriptor mechanism and some new instructions added to it to fit the patterns of Pascal programming usage.

6.2 Expression evaluation

The most commonly used form of Pascal expressions (as measured in the analysis of Pascal programs in chapter 4) is very simple - i.e. the average number of operators per expression is 0.21 (statically).

We can thus optimize expression evaluation as a whole because, the expressions being simple, they will usually be mapped into a single code fragment so that optimization of code fragments cost is tantamount to optimization of expressions in general.

We deal with two classes of expression fragments: the first involving relational operators and the second, arithmetic operators.

6.2.1 Relational Expressions

Relational expressions have a different pattern than arithmetic expressions. A relational expression has 1.15 operators on average, i.e. has usually two operands. In 75% of the cases these operands will be a single variable and a constant. The reverse polish generated by the P4 compiler for a construct like: x = c is:

```
LOD   x   {load variable}
LDC   c   {load constant immediate}
EQU       {test if equal and replace operands by boolean result}
```

The three instructions above can be reduced to a single one, with three fields like:

```
OPr   p,q,r   where (p,q) specify the variable address and the
              field r contains the constant. OPr is any of the
              relational operators.
```

We also need a 'compare indirect' for the cases in which the variable address is not known at compile time, i.e.:

```
OPIr      r   as above with the difference that the variable
              address is in the top of the stack.
```

A considerable part of the cost of this fragment in the P4 machine arises because the result of a relational operation must be deposited back to the stack for the posterior use of (possibly) a FJP instruction which will branch according to the contents of the top of stack. It seems reasonable to suppose that an additional gain could be obtained if the result, instead of being stacked, could be used to set a condition code. This case would give a gain, not only in the relational fragment but also in the jump instruction, which would not need to read the top of stack for the branching decision. But the introduction of condition codes would necessitate an additional register which has to be saved and restored at procedure entry/exit. This would add an additional cost overhead of around 2 million bytes of condition codes being moved to and fro at procedure calls. On the other hand, the old stack solution can be considerably improved in cost by the introduction of fast registers at top of stack whilst the cost of saving condition codes in the activation records of procedure calls cannot (probably) be decreased by further refinement steps. Hence, we shall retain the stack solution for condition codes.

The new instruction description is :

```
    OPr  p  q  r : begin
                        ac1 = IR.q;    (*address, suppose p=0*)
                        ac2 = IR.r;    (*constant immediate*)
                        ac1↑= ac1 ;    (*variable value*)
                        ac3 = ac1 op ac2; (*op depends on instruction*)
                        SP↑ = ac3;     (*result is stacked*)

                    end;
```

```
OPIr     r :  begin
                   ac1  =  SP↑ ;    (*read address*)
                   ac2  =  IR.q;    (*constant*)
                   ac3  =  ac1 op ac2  (*op depends on instruction*)
                   SP↑  =  ac3      (*stack result*)
```

A further optimization is possible in the case of the very common construct ptr OP nil, where ptr is a variable of type pointer. Since the pointer constant nil is uniquely identified, the instructions above do not need the field r for the constant value, and they are reduced to

OPNIL p,q   -compare variable at (p,q) with nil

OPNIL   -same as above but indirect

The final result of this set of modifications in the cost measure associated with expressions (as a whole) is seen in the table below:

Cost parameter	Reduction %
a1	15
a3	23
a4	27
a5	21
a6	24

6.2.2 Arithmetic Expressions

The same form of modification can be extended, in an orthogonal manner, to arithmetic operators to handle arithmetic fragments between a variable and a constant. Their form is (operations on integers only):

OPa p q r - execute the arithmetic operation between variable at (p, q) and constant in field r.

OPIa r - as above with address in top of stack.

The combined effect of the relational and arithmetic operators between constant and variable is shown in the table below:

Cost parameter	Reduction % (for fragments associated with expressions)
a1	21
a3	30
a4	34
a5	25
a6	36

Other agglutinations can be tried but since the above are the most frequent constructs with the simpler implementations, we will limit the consideration to these at this level of refinement. Other cases like variable-variable operators can be optimized in the next step of refinement through the use of fast registers.

6.3 Assignments

A very common construct in Pascal texts is the assignment of a single constant to a variable, for initialization purposes. This construct has a major influence in the static cost where about 30% of all assignments are of this form. P4 code for this construct is:

LDC c - load constant to top of stack

STR p, q - store at address (p, q)

This sequence can be merged in a single instruction SET whose format is:

SET p q r - set the content of address (p,q) to the value in r

SETI r - as above but address in top of stack.

These instructions can be defined as:

```
SET  p  q  r :  begin
                    ac1  = IR.q ; (address, suppose p=0*)
                    ac2  = IR.r ; (*constant value*)
                    ac1↑ = ac2  ; (*store*)
                end;
SETI       r :  begin
                    ac1  = SP↑  ; (*address in top of stack*)
                    ac2  = IR.r   (*constant*)
                    ac1↑ = ac2  ; (*store back*)
                end;
```

The introduction of these instructions will affect mainly the instruction static occupancy (because of the reduction in code size); its effect is less prominent in the dynamic cost parameters since assignment of a constant has a smaller share of resource consumption at run time than it has statically. The effect of these instructions is to reduce instruction occupancy by 11%; thus reducing cost parameter a1 by 11% in all fragments associated with assignments. We can ignore the effect on the other cost parameters.

6.4 For instruction

The for instruction is compiled by the P4 in two parts: one to evaluate and set the initial and final value of the control variable and a second (the for body) which has in turn two parts - one to test if the control variable is less than the limit and a second to increment it. A substantial reduction in the cost measure can be achieved by introducing primitives to perform these tasks. So, we create two new instructions:

DOE r s : compare local variables at addresses r and s (both must be local variables) return result to top of stack.

DOR p q : decrement/increment control variable at offset q.

With these modifications a for loop can be coded easily as:

1- evaluate initial value for control variable.

2- evaluate final index of loop and store in temporary location.

3- insert DOE r, s.

4- insert FJP to out of loop.

5- insert code for loop statements.

6- insert DOR p,q.

7- insert UJP to step 3.

A further compression could be achieved by merging the DOE with FJP and DOR with UJP, but we reject this choice in view of the reduced gains it would introduce if a fast top of stack is used.

The instruction can be defined as:

```
DOE  r  s : begin
                ac1 = IR.r ; (*first address*)
                ac2 = IR.s ; (*second address*)
                ac1 = ac1↑ ; (*read control variable*)
                ac2 = ac2↑ ; (*read limit*)
                ac3 = ac1 less ac2; (*for up counting*)
                SP↑ = ac3  ;
            end;
```

```
DOR  p   q :  begin
                ac2   = IR.q ; (*address of control variable*)
                ac1   = ac2↑ ; (*fetch control variable*)
                ac3   = ac1 + 1 ; (*minus if down to*)
                ac2↑ = ac3    (*store back*)
              end;
```

The effect of the introduction of these instructions on the cost measure associated with statements (fragments 93 to 106) is seen in the table below:

Cost parameter	Reduction %
a1	6
a3	16
a4	28
a5	13
a6	43

6.5 Data structure access

6.5.1 Introduction

The aim of this section is to define a more efficient data structure access method for the P4 machine using the descriptor mechanism presented in chapter 3. It is worth noting that this mechanism includes range check on arrays and subranges, so we are optimizing not only array access but also all checks on subrange variables.

The scheme presented in chapter 3 should be seen as the first step of refinement in a process of deriving a descriptor mechanism for a Pascal machine. In this section, an additional refinement step will be made with new constraints which will have the effect of changing the descriptor operator forms as defined in chapter 3.

The new constraint to be imposed in the design is that the implementation of the descriptor mechanism should present a lower cost measure for data structure access than the P4-machine scheme. To obtain a lower cost measure than the P4 method the new implementation should reduce the traffic of redundant information which is, in turn, caused by the constraint imposed on its design that the translation process should be as simple as possible. This simplicity constraint implies that every time a selected element appears, its full semantic specification is loaded to the stack, even if the next descriptor operator will only use a part of it.

6.5.2 Descriptors for the P4-machine.

To decrease the cost measure for data access, we have to modify the mechanism presented in chapter 3 - with new formats for the descriptor operators and also allow more work to be made at compile time. The modifications introduced are:

1- descriptors are presented (conceptually but not physically) in two different formats: short and long. The long format descriptor is the same as defined in chapter 3, whilst the short format consists simply of a tag and an address. The descriptor operators, in consequence, have two variants to enable them to work with the two different descriptor formats.

2- the descriptor operators defined in chapter 3 are zero-address polish operators. In this new implementation, two new forms of specifying the descriptor operand (which was implicit in the old form) are provided:

i- by an address field with the absolute address of the descriptor, i.e. the operator is changed from a zero address to a one-address operator.

ii- immediate - in this case the descriptor follows the operators. Immediate operands are an advantage when they are small enough to be packed in the same container size as the address, saving thus one extra reference at run time.

The causes for these changes are:

a) there is the need for reducing the instruction static size cost by merging the load descriptor operation with its descriptor operator successor into a single operation code.

b) in the case of record item access there is the need for a low cost instruction for generating record item descriptors, to compete with the P4 machine which uses simply an "increment address" instruction in this case. The P4 compiler also takes advantage of the fact that the record item offset is known at compile time and, in some cases, does all the address evaluation with no instruction being generated at run time. This is achieved by merging the information about record item offset and type in the current instruction successor. If the record item is directly accessible, the item offset is added to the offset field of the next instruction, or else is added to the offset field of the following "indirect fetch" (IND) instruction.

An instruction is generated only when there is a need for an absolute address - i.e. after the evaluation of the left-hand, or preceding an array access.

3- a new operator Arrowdot is created. This operator is a combination of the operators arrow and dot in sequence. Its appearance is due to the necessity of optimizing the very frequent programming construct a↑.x where a is an arbitrary name.

(See also table 13 for pointed element fragments, where almost all of the pointers point to records). In the case of the P4-machine the code generated is:

i- evaluate the address of a.

ii- insert IND q -to fetch the value of a.

iii- insert INC x -to increment address in stack by the offset of x.

Using the operator Arrowdot, the above construct can be translated as:

i- evaluate the descritor of a,

ii- insert Arrowdot <par> - where <par > specifies the record item descriptor.

4- In the case of the same construct as above, but when the descriptor of a↑ is known at compile time, a different sequence can be generated:

i- load descriptor of a↑

ii- insert Dot <par > - where <par > specifies the record item descriptor.

5- A new descriptor format is introduced in addition to the ones described in chapter 3 to define subrange bounds of type integer. Since most of the array and subrange bounds can be coded with a few bits, there are two new tags:

i- one for subranges of integer which require a full integer format, e.g. the very common type: Positive_Integer = 0..MAXINT.

ii- one for subranges of integer such that the lower bound can be coded with 8 bits each (smallest unit for arithmetic purposes). With this coding we can describe 100% of the lower bounds of arrays

(99% of subranges and 97% of the upper bound of arrays (50% for subranges). The total space for bounds is now 16 bits or 1/4 of the previous need , which will give a big saving in the static constant area and in information traffic for range check.

7- we need also primitives for load, store and move data through descriptors in the top of the stack:

i- Lodd - unary operator for leading a piece of data using descriptor in the top of the stack.

ii- Stod - store data in the top of the stack using descriptor immediately below, and do the range check if necessary.

iii- Movd - move data described by the descriptor below the top of the stack to the position described in the top of the stack. Do range check if necessary.

8- We also need one primitive for loading descriptors to the top of the stack:

i- Ldesc <par> : where <par> specifies the descriptor form and address.

6.5.3 Descriptor operator formats

The final result of the primitives available for data structure access is presented in tables 6.1 and 6.2 below. A full description of the execution of the descriptor operation is given in Table 6.3.

Table 6.1 - Descriptor operator formats

Each descriptor operator has four fields named as: opcode, form or f, spec or s, and q (used as an address or data field). The opcode field can specify one of the five possible descriptor operators:

Ldesc, Dot, Arrow, Bracket and Arrowdot.

- The form field defines the descriptor to be operated on to be in long or short format.
- The spec field specifies if the descriptor operand is defined by its address or follows the operator as a literal.

Descriptor operators mnemonics

	Mnemonic		Description
1-	Ldesc	f s q	load descriptor specified by the f, s and q fields to the stack.
2-	Bracket	f s q	do range check and indexing with descriptor specified in (f s q) fields with index and array base address in top of stack.
3-	Dot	f s q	record item descriptor generation with item descriptor defined in (f s q) and record descriptor in stack.
4-	Arrow	f s q	generate descriptor of pointed variable with (f s q) specifying descriptor and top of stack containing address of pointer.
5-	Arrowdot	f s q	generate the descriptor of a record item whose descriptor is defined by (f s q) fields. The top of the stack contains the descriptor of the pointer which is pointing to the record.

Table 6.2 - Primitives for load, store and move data via descriptors.

	Mnemonic	Description
1	Lodd	Load data specified by descriptor in top of stack to top of stack, replacing descriptor.

	Mnemonic	Description
2-	Stod	Store data in top of the stack to address specified by descriptor in position immediately below the top of the stack.
3-	Movd	Move data specified by descriptor in top of the stack to position specified by descriptor below the top of the stack.

Table 6.3 - Descriptor operators

In the description below, suppose f=short, s=address and q contains the address of the descriptor. The descriptor formats are as specified in chapter 3.

```
1-Ldesc  f  s  q : begin
                       ac1 = IR.q;          (*address of descriptor*)
                       ac2.tag = ac1↑.tag;  (*read tag*)
                       ac2.address = ac1↑.address;  (*read address*)
                       SP↑ = ac2;    (*move result descriptor to stack*)
                   end;

2- Dot   f  s  q :  begin
                       ac1 = IR.q ; (*address of descriptor*)
                       ac2.tag = ac1↑.tag; (*read tag*)
                       ac2.offset =ac1↑.offset ; (*record item offset*)
                       ac1 = SP↑ ; (*record descriptor*)
                       ac1.tag = ac2.tag ;
                       ac1.address = ac1.address + ac2.offset;
                       SP↑ = ac1 ; (*move result descriptor to stack*)
                    end;
```

```
3-Arrow   f   s   q :   begin

                          acl = SP↑;   (*read pointer address*)

                          acl.address = acl↑.address ; (*pointer value*)

                          ac2 = IR.q ; (*pointed element descriptor address*)

                          acl.tag = ac2 .tag;

                          SP↑  = acl

                        end;

4- Bracket f   s   q : begin

                          acl = SP↑;   (*array index*)

                          ac2 = IR.q; (*array element descriptor address*)

                          ac2 = ac2↑; (*read descriptor*)

                          if (acl<ac2.lower) or (acl>ac2.upper)

                            then error; (*bounds check*)

                          ac3 = acl*length (ac2.tag);(*indexing*)

                          acl = SP↑;   (*array descriptor*)

                          ac3.tag = ac2.tag;

                          ac3.address=ac3.address+acl.address

                          SP↑= ac3;

                        end;



 5- Arrowdot   f   s   q : begin

                              acl = SP↑ ;   (*pointer address*)

                              ac2 = IR.q;

                              ac2 = ac2↑;   (*record item descriptor*)

                              acl = acl↑;   (*record address*)

                              ac3.tag = ac2.tag;

                              ac3.address = acl.address+ac2.address;

                              SP↑  = ac3 ;

                            end;
```

6.5.4 Examples of Use

We show in the examples below, various sequences of code generated by the P4 compiler using the algorithms in chapter 3 and the new descriptor implementation described above.

Suppose, variables Sigma and Phi are both of type epsilon as in Chapter 3.

i- Sigma [j] := Phi [j] ;

P-4 sequence	Old descriptor	New descriptor
Lda Sigma	Ldesc Sigma	Ldesc Sigma
Lod j	Ldesc Sigma-e	Load j
Check bounds	Lod j	Bracket Sigma-e (long)
Dec 1	Bracket	
Ixa Size		
Lda Phi	Ldesc Phi	Ldesc Phi
Lod j	Ldesc Phi-e	Lod j
Check bounds	Lod j	Bracket Phi-e (short)
Dec 1	Bracket	
Ixa Size		
Mov Size	Mov	Movd
11 instructions	9 instructions	7 instructions

ii- Sigma [j].z := Phil [j].z

P-4	Old descriptor	New descriptor
Lda Sigma	Ldesc Sigma	Ldesc Sigma
Lod j	Ldesc Sigma-e	Lod j
Check bounds	Lod j	Bracket Sigma-e (short)
Dec 1	Bracket	
Ixa Size		
Inc offset of z	Ldesc z	Dot z (long)
	Dot	
Lda Phi	Ldesc Phi	Ldesc Phi (short)
Lod j	Ldesc Phi-e	Lod j
Check bounds	Lod j	Bracket Phi-e (short
Dec 1	Bracket	
Ixa Size		
Inc z	Ldesc z	Dot j (short)
Mov size	Dot	Mov.d
	Mov	
13 instructions	13 instructions	9 instructions

iii- Sigma [j].u↑.z :=Phil[j].u↑.z

P4 code	Old descriptor	New descriptor
Lda Sigma	Ldesc Sigma	Ldesc Sigma (short)
Lod j	Ldes Sigma-e	Lod j
Check bounds	Lod j	Bracket Sigma-e (short)
Dec 1	Bracket	
Ixa Size		
Ind u	Ldesc u	Arrowdot u (short)
Inc z	Dot	Dot z (long)
	Arrow u-p	
	Ldexc z	
	Dot	
(as above with Phi)	(as above with Phi)	(as above with Phi)
Mov size	Mov	Movd
15 instructions	19 instructions	11 instructions

6.5.5 - Evaluation of the descriptor mechanism

The combined effect of using the new format for descriptors and descriptor operators is that record and pointer access will have approximately the same cost measure. An additional reduction in the cost measure is obtained in the case of the bounds checking required in the assignment to a subrange variable. The instruction 'store via descriptor' will perform automatic range checking without the need for an explicit 'check bounds' instruction.

We will evaluate, in this section, only the reduction in the cost measure associated with array accesses. The combined effect of array accesses and subrange checking with the new descriptor mechanism in the total cost measure is presented in table 6.4. The reduction of the cost measure for array accesses is shown in the table below:

Cost parameter	Reduction %
a1	42
a3	41
a4	27
a5	42
a6	36

6.6 - Standard procedures and functions.

The cost measure we have associated with standard functions and procedures in the P4 machine is the estimated cost of a simple 'branch and link' instruction, i.e. a jump to the procedure code saving only the return address without any actions on the scope.

We have evaluated the improvement in the cost measure which comes from the implementation of standard procedures as one-byte, zero address polish operators. The effect of this change in the total cost measure is shown in table 6.4.

6.7 - Comments on line tracking and Post-mortem-dump.

The Post-mortem-dump (PMD for short) is undoubtedly one of the more costly elements in the execution of Pascal programs. Although it is possible to estimate the cost measure associated with PMD, using the data about language usage, we have deliberately refrained from discussing it because it is a complex and controversial subject beyond the scope of this work. We shall mention briefly the simpler case of line-tracking, i.e. the possibility of knowing, at every instruction, the physical line number of the source code sequence which originated this instruction. In the P4 compiler there are two forms of line tracking:

a-minimal: line numbers are introduced only at procedure calls. The activation record lay-out is modified to include one entry for line number, which is passed as an implicit parameter to every user procedure call.

full: in this case, one instruction of the intermediate language specifying the line number is generated for each physical line. The line number at procedure call is still passed as a parameter.

The problem of an efficient implementation for line tracking was investigated by Wortman (Wor72a). He proposes two lines of solution :

a-a software solution involving tables and searching.

b-a hardware solution, using another memory, with the same size as the code memory, in which every instruction is paired with a line number.

We shall not discuss the cost measure associated with line tracking, although it can be derived from the data about fragment usage.

6.8 - Conclusions.

The discussion of the improvements in the P4 machine have been concentrated on the effects of alternative constructs in the cost measure associated with particular groups of fragments, e.g. fragments associated with expressions, assignments etc. Table 6.4 shows the effect of each one of the proposed alterations on the total cost measure. The entry for 'data access' covers the case of array accesses and checking for variables of type subrange.

Table 6.4 - Modifications on the total cost measure (%)

Fragment group	Expr.	Assign.	For	Data access	Std. Proc.	Total(%)
a1	3.3	1.6	1.2	9.5	3.1	19.8
a3	8.6	0.5	2.7	10.7	3.8	30.2
a4	11.8	0.4	2.6	9.0	3.7	27.6
a5	6.0	0.6	2.5	11.8	8.5	29.3
a6	10.2	0.4	3.5	13.2	3.7	31.1

From the table above we conclude that most of the gains can be associated with two factors:

a - the use of instructions for executing operations between a constant and a variable with the result being stacked.

b - the gains in data access for array elements and subrange variables checks.

In spite of its simplicity, the P4 machine is a very well designed machine with its primitives being well adequate for Pascal requirements. This fact means that the improvements achieved, in the region of 20% to 30% in the cost parameters are satisfactory. Further refinements in the P4 machine are possible but the gains are likely to be insignificant when compared with the actual implementation details of the P4 machine, e.g. the nature of the microcode machine or the translation to machine code.

7-Conclusions

This thesis presented a study of language directed computer design, more specifically a study of Pascal-orientated intermediate language machines.The problem was approached from the point of view of intermediate forms of compilation, which in turn, define an abstract machine for their execution.

First, we studied the case of an intermediate language machine derived to meet a specifically designed hardware configuration. The difficulties encountered in the mapping of language data structures in the case of full Pascal structuring methods lead us to the study of a descriptor mechanism to meet Pascal requirements, using only the normal random access memory as a hardware base. The scheme presented is derived from a study of the ICL 2900 descriptor mechanism which posed some problems to Pascal implementors. This solution is general, supports the needs of Pascal data structures and simplifies the code generation for Pascal names.

The efficiency of a given intermediate language machine can be substantially improved with the knowledge of its usage patterns. In chapter 4 we presented a detailed analysis of form and behaviour of a given set of Pascal programs. A set of tables detailing textual, syntactic and language fragments usage was given. This data can be used both by the compiler writer and the machine implementor to evaluate and improve their designs.

This data has been used to evaluate and improve the Pascal P4 intermediate language machine. With the aid of the patterns of language usage the most expensive source language constructs were detected and alternative constructs resulting in a more efficient P4-machine were suggested and the improvement measured.

There are several avenues of research open, using some of the results presented here.

a-Our study was deliberately kept on an abstract, implementation independent, level. The next logical step is the study of the implementation of intermediate language constructs suggested, including the descriptor implementation. This study would also include the effects of the suggested constructs if the machine is going to be interpreted by microprogram or suffer further translation.

b-A study could be made of the operating system interface, including the file interface and its primitives.

c-A study could be made of a multiprogrammed Pascal intermediate language machine.

d-Our study of Pascal programs should be extended to include more user programs - possibly student programs as opposed to the system orientated workload studied here.

e-The results of the study of Pascal programs could be used to obtain synthetic programs to simulate the whole workload, and possibly to develop a form of "Gibson's mix" based on Pascal program composition.

References

Ale75a-Alexander, W.G. and Wortman, D.B.
Static and dynamic characteristics of XPL programs
Computer 8, 1975, pp.41-46

Bou73a-Bourne, S.R.
Zcode-a simple machine. Technical Report
Univ. of Cambridge Computer Laboratory.

Buc78a-Buckle, J.K.
The ICL 2900 series
MacMillan Press, 1978

Chev78a-Chevance, R.J. and Heidet, T.
Static profile and dynamic behaviour of Cobol programs
Sigplan, v13-4, April 1978, pp.44-57

Col78a-Coleman, D.
A structured programming approach to data
MacMillan Computer Science Series, 1978

Hau68a-Hauc, E.A. and Dent, B.A.
Burroughs 6500 stack mechanism
Proc. 1968 SJCC, pp.245-251

Hoa73a-Hoare, C.A.R. and Wirth, N.
An axiomatic definition of the programming language Pascal
Acta Informatica, 2, 1973, pp.335-355

How76a-Howarth, D.J.
Lecture notes on computer architecture (unpublished)
Imperial College, 1976

ICL76a-ICL 2900 primitive level interface
P.S.D. 2.5.1
ICL Ltd., Bracknell, 1976

Ili68a-Iliffe, J.K.
Basic machine principles
Macdonald Computer Monographs, 1968

Iza79a-Izatt, W. and Schmitz, E.A.
Data structures and descriptors in the series 2900
and beyond
CCD report 79/23
Imperial College, London

Jen73a-Jensen, K. and Wirth, N.
An assembler/interpreter for a hypothetical stack computer
Program listing

Jen75a-Jensen, K. and Wirth, N.
Pascal user manual and report.
Springer-Verlag, N.Y. 1975

Knu68a-Knuth, D.E.
The art of computer programming, v.1
Addison-Wesley, Reading, Mass. 1968

Knu71a-Knuth, D.E
An empirical study of Fortran programs
Software-practice and experience
v1, 1971, pp.105-153

McKe67a-McKeeman, W.
Language directed computer design
Proc. FJCC 1967
AFIPS Press

McKe75a-McKeeman, W.
Stack machines
Introduction to Computer architecture , S.R.A., 1975
H.S. Stone, Editor

Pug79a-Pugh, C.G.
Pascal P4 for VM/370 CMS
Computing and Control Dept.
Imperial College

Ree77a-Rees, M.J. et al.
Pascal on an advanced architecture
Third Annual Computer Studies Symposium
Univ. Southampton, March 1977

Ric71a-Richards, M.
The portability of the BCPL compiler
Software-practice and experience
v1, pp.135-146

Sch79a-Schmitz, E.A.
A study of static data type usage in Pascal
Computing and control Dept. Technical report 79/17
Imperial College, 1979

Str58a-Strong, J. et al.
The problem of communicating with changing machines
CACM v1-8, 1958

Tan78a-Tanenbaum, A.S.
Implications of structured programming for machine architecture
CACM, v21-3, March 1978, pp,237-246

Wic69a-Wichman, B.A.
A comparison of Algol 60 execution speeds
NPL report CCU-3, January 1969

Wic70a-Wichman, B.A.
Some statistics from Algol programs
NPL Report, CCU-11, August 1970

71a-Wichman, B.A.
The performance of some Algol systems
Proc. IFIP Congress, 1971

Wil72a-Wilner, W.T.
Burroughs B1700 memory utilization
Proc. FJCC, AFIPS v41-1972, pp.579-586

Wil72b-Wilner, W.T.
The design of the Burroughs B1700
Proc. FJCC, AFIPS v41-1972

Wir71a-Wirth, N.
The programming language Pascal
Acta Informatica, v1, 1971, pp.35-63

Wir71b-Wirth, N.
The design of a Pascal compiler
Software-practice and experience
v1, 1971, pp.309-333

Wir73a-Wirth, N.
Systematic Programming, an introduction
Prentice-Hall, 1973

Wir76a-Wirth, N.
'Algorithms + Data Structures = Programs'
Prentice-Hall, 1976

Wor72a-Wortman, D.B.
A study of language directed computer design
Ph.D. thesis, Stanford Univ., 1972

Appendix 1-Cardinality of subrange ,arrays,scalar and records.

Appendix 1.1-In type declarations.

```
Subrange Bounds
***************
Lower Bound

Size(bits)              Count          Percent       Cumulative
         1               111            93.28          93.28
         2                 1             0.84          94.12
         7                 2             1.68          95.80
         8                 4             3.36          99.16
        16                 1             0.84         100.00
Mean =        1.46      Variance =     520.00
Upper Bound

Size(bits)              Count          Percent       Cumulative
         1                 3             2.52           2.52
         2                10             8.40          10.92
         3                13            10.92          21.85
         4                12            10.08          31.93
         5                 8             6.72          38.66
         6                 4             3.36          42.02
         7                10             8.40          50.42
         8                21            17.65          68.07
         9                 2             1.68          69.75
        10                 3             2.52          72.27
        11                 5             4.20          76.47
        12                 1             0.84          77.31
        14                 1             0.84          78.15
        15                 2             1.68          79.83
        16                 5             4.20          84.03
        17                 2             1.68          85.71
        18                 1             0.84          86.55
        24                 2             1.68          88.24
        25                 1             0.84          89.08
        31                13            10.92         100.00
Mean =        9.91      Variance =   19660.00

Scalar type cardinality
***********************

    Size                Count          Percent       Cumulative
         2                27            32.14          32.14
         3                11            13.10          45.24
         4                 7             8.33          53.57
         5                 9            10.71          64.29
         6                 5             5.95          70.24
         7                 5             5.95          76.19
         8                 3             3.57          79.76
         9                 3             3.57          83.33
        10                 1             1.19          84.52
        12                 2             2.38          86.90
        13                 1             1.19          88.10
        14                 1             1.19          89.29
        15                 1             1.19          90.48
        16                 3             3.57          94.05
        26                 1             1.19          95.24
        32                 4             4.76         100.00
Mean =        6.60      Variance =    7368.00

Record Sizes
************

    Size                Count          Percent       Cumulative
         1                 2             1.14           1.14
         2                38            21.59          22.73
         3                26            14.77          37.50
         4                27            15.34          52.84
         5                17             9.66          62.50
         6                13             7.39          69.89
         7                14             7.95          77.84
         8                 3             1.70          79.55
         9                 5             2.84          82.39
        10                 3             1.70          84.09
        11                 5             2.84          86.93
        12                 8             4.55          91.48
        13                 5             2.84          94.32
        15                 1             0.57          94.89
        17                 2             1.14          96.02
        18                 1             0.57          96.59
        19                 2             1.14          97.73
        21                 1             0.57          98.30
        27                 3             1.70         100.00
Mean =        5.95      Variance =    9328.00

Array index type
****************

Type Distribution
Types                   Count          Percent
Char                       4             2.86
Scalar                     1             0.71
Subrange                 135            96.43

Total of Types =          644
```

```
Array Bounds
************
Lower Bound

Size(bits)              Count           Percent       Cumulative
         1                635             98.76          98.76
         2                  3              0.47          99.22
         8                  5              0.78         100.00
Mean =        1.06       Variance =      298.00
Upper Bound

Size(bits)              Count           Percent       Cumulative
         2                 56              8.71           8.71
         3                190             29.55          38.26
         4                285             44.32          82.58
         5                 47              7.31          89.89
         6                  8              1.24          91.14
         7                 15              2.33          93.47
         8                 28              4.35          97.82
        10                 10              1.56          99.38
        11                  1              0.16          99.53
        12                  1              0.16          99.69
        14                  1              0.16          99.84
        15                  1              0.16         100.00
Mean =        4.02       Variance =     9584.00

Subrange Bounds
***************
Lower Bound

Size(bits)              Count           Percent       Cumulative
         1                756             96.43          96.43
         2                  7              0.89          97.32
         3                  2              0.26          97.58
         5                  1              0.13          97.70
         7                  3              0.38          98.09
         8                 13              1.66          99.74
        11                  1              0.13          99.87
        15                  1              0.13         100.00
Mean =        1.19       Variance =     1220.00
Upper Bound

Size(bits)              Count           Percent       Cumulative
         1                  7              0.89           0.89
         2                 41              5.23           6.12
         3                 55              7.02          13.14
         4                 67              8.55          21.68
         5                 39              4.97          26.66
         6                 14              1.79          28.44
         7                 74              9.44          37.88
         8                 98             12.50          50.38
         9                 11              1.40          51.79
        10                 61              7.78          59.57
        11                 13              1.66          61.22
        12                  4              0.51          61.73
        13                 15              1.91          63.65
        15                 25              3.19          66.84
        16                 26              3.32          70.15
        17                 11              1.40          71.56
        24                 20              2.55          74.11
        25                  1              0.13          74.23
        31                202             25.77         100.00
Mean =       13.96       Variance =   235574.00

Scalar type cardinality
***********************

Size                    Count           Percent       Cumulative
         2                447             78.42          78.42
         3                  8              1.40          79.82
         4                 28              4.91          84.74
         5                  7              1.23          85.96
         6                 15              2.63          88.60
         7                  5              0.88          89.47
         8                  1              0.18          89.65
         9                  7              1.23          90.88
        10                  1              0.18          91.05
        13                 10              1.75          92.81
        14                  1              0.18          92.98
        16                 15              2.63          95.61
        26                  5              0.88          96.49
        32                 20              3.51         100.00
Mean =        4.25       Variance =    31160.00

Record Sizes
************

Size                    Count           Percent       Cumulative
         1                 25              9.16           9.16
         2                 75             27.47          36.63
         3                 23              8.42          45.05
         4                 19              6.96          52.01
         5                  9              3.30          55.31
         6                 41             15.02          70.33
         7                  5              1.83          72.16
         8                 15              5.49          77.66
         9                  3              1.10          78.75
        10                  3              1.10          79.85
        11                 10              3.66          83.52
        12                 24              8.79          92.31
        17                  1              0.37          92.67
        21                 18              6.59          99.27
        27                  2              0.73         100.00
Mean =        6.12       Variance =    16966.00

Array index type
****************

Type Distribution
Types                   Count           Percent
Char                        6              0.90
Scalar                     15              2.26
Subrange                  643             96.84

Total of Variables =          3952
```

Appendix 1.3-In value parameters declarations.

```
Array Bounds
************
Lower Bound

Size(bits)            Count        Percent      Cumulative
         1             264         100.00         100.00
Mean =        1.00     Variance =       0.00
Upper Bound

Size(bits)            Count        Percent      Cumulative
         2              25           9.47           9.47
         3             108          40.91          50.38
         4             105          39.77          90.15
         5              20           7.58          97.73
         6               2           0.76          98.48
         7               2           0.76          99.24
        10               2           0.76         100.00
Mean =        3.56     Variance =    2682.00

Subrange Bounds
***************
Lower Bound

Size(bits)            Count        Percent      Cumulative
         1             179          90.40          90.40
         7               9           4.55          94.95
         8              10           5.05         100.00
Mean =        1.63     Variance =     938.00
Upper Bound

Size(bits)            Count        Percent      Cumulative
         1               3           1.52           1.52
         2               7           3.54           5.05
         3              10           5.05          10.10
         4               7           3.54          13.64
         5               7           3.54          17.17
         6               2           1.01          18.18
         7              16           8.08          26.26
         8              35          17.68          43.94
         9               3           1.52          45.45
        10              13           6.57          52.02
        11               5           2.53          54.55
        16              14           7.07          61.62
        17               2           1.01          62.63
        24               7           3.54          66.16
        25               1           0.51          66.67
        31              66          33.33         100.00
Mean =       16.25     Variance =   74674.00

Scalar type cardinality
***********************

    Size              Count        Percent      Cumulative
         2              70          55.12          55.12
         3               3           2.36          57.48
         4              13          10.24          67.72
         5               3           2.36          70.08
         6               3           2.36          72.44
         7               2           1.57          74.02
         9               5           3.94          77.95
        10               1           0.79          78.74
        13               5           3.94          82.68
        16               9           7.09          89.76
        26               1           0.79          90.55
        32              12           9.45         100.00
Mean =        7.26     Variance =   16492.00

Record Sizes
************

    Size              Count        Percent      Cumulative
         1              15          18.99          18.99
         2              14          17.72          36.71
         3              10          12.66          49.37
         4               1           1.27          50.63
         5               2           2.53          53.16
         6               8          10.13          63.29
         7               1           1.27          64.56
         8               3           3.80          68.35
        10               1           1.27          69.62
        11               1           1.27          70.89
        12              15          18.99          89.87
        21               8          10.13         100.00
Mean =        6.77     Variance =    6130.00

Array index type
****************

Type Distribution
Types                 Count        Percent
Subrange               264         100.00

Total of value parameters =        1095
```

Appendix 1.4-In reference parameter declarations.

```
Array Bounds
************
Lower Bound

Size(bits)              Count          Percent      Cumulative
        1                115            94.26         94.26
        5                  4             3.28         97.54
        6                  2             1.64         99.18
        8                  1             0.82        100.00
Mean =          1.27     Variance =     196.00
Upper Bound

Size(bits)              Count          Percent      Cumulative
        2                 13            10.66         10.66
        3                 42            34.43         45.08
        4                 26            21.31         66.39
        5                 12             9.84         76.23
        6                  7             5.74         81.97
        7                  2             1.64         83.61
        8                 14            11.48         95.08
       10                  6             4.92        100.00
Mean =          4.46     Variance =    2448.00

Subrange Bounds
***************
Lower Bound

Size(bits)              Count          Percent      Cumulative
        1                 44           100.00        100.00
Mean =          1.00     Variance =       0.00
Upper Bound

Size(bits)              Count          Percent      Cumulative
        1                  6            13.64         13.64
        3                 10            22.73         36.36
        4                  1             2.27         38.64
        5                  1             2.27         40.91
        7                  2             4.55         45.45
        8                 12            27.27         72.73
       10                  3             6.82         79.55
       24                  1             2.27         81.82
       31                  8            18.18        100.00
Mean =         10.39     Variance =    9110.00

Scalar type cardinality
***********************

    Size                Count          Percent      Cumulative
        2                117            86.03         86.03
        4                 10             7.35         93.38
        6                  4             2.94         96.32
        9                  1             0.74         97.06
       14                  1             0.74         97.79
       26                  2             1.47         99.26
       32                  1             0.74        100.00
Mean =          2.98     Variance =    3020.00

Record Sizes
************

    Size                Count          Percent      Cumulative
        2                 44            16.99         16.99
        3                 13             5.02         22.01
        4                 26            10.04         32.05
        5                 46            17.76         49.81
        6                 30            11.58         61.39
        7                  9             3.47         64.86
        8                  6             2.32         67.18
        9                  8             3.09         70.27
       10                  1             0.39         70.66
       11                  1             0.39         71.04
       12                  5             1.93         72.97
       13                  3             1.16         74.13
       15                 14             5.41         79.54
       17                 41            15.83         95.37
       21                  6             2.32         97.68
       27                  6             2.32        100.00
Mean =          8.26     Variance =   25740.00

Array index type
****************

Type Distribution
Types                   Count          Percent
Char                       2             1.61
Subrange                 122            98.39

Total of Var Parameters =              833
```

Appendix 2 - P4 machine code mnemonics

In the instructions below the C parameter field is used to indicate the instruction variants according to the type of data being operated upon i.e. character, address, integer or string

Mnemonic	Parameter	description
ABI		absolute value of integer
ABR		absolute value of real number
ADI		integer addition
ADR		real addition
AND		Boolean "and"
CHK	C P Q	check against upper and lower bounds
CHR		convert integer to character
CSP	Q	call standard procedure
CUP	P Q	call user procedure
DEC	C Q	decrement
DIF		set difference
DVI		integer division
DVR		real division
ENT	P Q	enter block
EOF		test on end of file
EQU	C (Q)	compare on equal
FJP	Q	false jump
FLO		float next to the top
FLT		float top of the stack
GEQ	C (Q)	greater or equal
GRT	C (Q)	greater than
INC	C Q	increment

Con't of Appendix 2

Mnemonic	Parameter	Description
INC	C Q	indexed fetch
INN		test set membership (in)
INT		set interconnection
IOR		boolean "inclusive or"
IXA	Q	compute indexed address
LAD	Q	load base-level address
LDA	Q	load address of constant
LDA	P Q	Load address with level P
LDC	C Q	load constant
LDO	C Q	load contents of base-level address
LEQ	C (Q)	less than or equal
LES	C (Q)	less than
LOD	C P Q	load contents of address
MOD		modulus
MOV	Q	move
MPI		integer multiplication
MPR		real multiplication
MST	P	mark stack
NEQ	C (Q)	not equal
NGI		integer sign inversion
NGR		real sign inversion
NOT		Boolean "not"
ODD		test on odd
ORD	C	convert to integer

Mnemonic	Parameters			Description
RET	C			return from block
SBI				integer subtraction
SBR				real subtraction
SGS				generate singleton set
SQI				square integer
SQR				square real
SRD	C		Q	store at base level address
STO	C			store indirect
STP				stop
STR	C	P	Q	store at level P
TRC				truncation
UJP			Q	unconditional jump
UNI				set union
XJP			Q	indexed jump

Appendix 3 - Evaluation of fragment cost for the P4 machine

We show below, for each group of fragments, the possible code sequence for evaluation of the fragment cost.

1- Assignments

There are 3 cases to consider :

i- rhs is a constant

ii- rhs is a variable

iii- rhs is an expression

Case i - the code patterns for single variables is:

```
a-Ldc   q          b-Ldc   q          c-Ldc  q
  Str p, q           Sro   q            Sto
```

the code for arrays and records is

```
d-Lca  q
  Mov  3
```

Case ii - four possible cases for simple variables

```
a-Lod  p, q       b-Lod  p, q     c-Ind q        d-Ind  q
  Str  p, q         Sto             Str p,q        Sto
```

for arrays and records

```
e-Lda  q
  Mov  3
```

Case iii - three possible sequences

```
a-Str  p, q       b-Sro q     c-Sto
```

2- Parameter passing by value

There are three cases to consider:

i-actual parameter is a constant

ii-actual parameter is a variable

iii-actual parameter is an expression

Case i- only one case for simple type formal parameters

```
a- Ldc  q
if an string then
b- Lca  q
   Lod  p,q
   Lda  q
   Mov  3
```

Case ii- if simple type then

```
a- Lod p,q     b-Ind q
if string or record
c- Lda p,q
   Lda p,q
   Lod p,q
   MOv
```

Case iii- if expression then the fragment cost is evaluated in expressions.

3- Parameter passing by reference

only one sequence is possible:

```
a-Lda p,q
```

4-Procedure call (user procedures and functions)

For static cost parameters and sequence is:

```
a-Mst 1
  Cup p, q
```

For dynamic cost parameters the return must be accounted

```
b-Mst 1
  Cup p, q
  Ret
```

5- Standard procedure calls other than ORD, CHR, SUCC and PRED

For static cost we estimated the cost of a branch-and-link instruction, while for dynamic cost the cost parameters are accounted for a branch-and-link plus a return.

6-Gotos-We assume all gotos being to the same level
(i.e. neglect interlevel jumps)

```
a-Ujp  q
```

7-If-then

```
a-Fjp  q
```

8-If-then-else

the sequence to be accounted statically is:

```
a-Fjp  q
  Ujp  q-
```

but at run time the second instruction is executed only if the condition is true, and is accounted in fragment 98.

9-Case statement

we assume a case statement with 7.5 case elements.

The static code sequence is:

```
a-Lod  p, q
  Ujp  q
  Ujp  q (repeated n times, where n is the number of elements)
  Chk  q
  Ldc  q
  Sbi
  Xjp
  Ujp  q (n times for jum table)
  Ujc    (error)
```

at run time only the following sequence is executed:

```
b-Lod  p, q
  Ujp     q
  Chk     q
  Ldc
  Sbi
  Xjp
  Ujp     q(to statement)
  Ujp     q(out of statement)
```

10-While statement

Assuming that almost all conditions are expressions then:

```
a-expr
  Fjp  q
  statement
  Ujp  q (to head)
```

So, for while head overhead we use only a FJP q adn for while body (fragment 101) we use the above sequence: Fjp q, Ujp q.

11-Repeat statement

The code pattern is:

```
a-statement
  expression
  Fjp
```

12-For statement

It is composed of two parts: the head and the body.

Case i - for head, assume both limits as constants.

```
a-Ldc q
  Str p, q
  Ldc q
  Str p, q
```

Case ii - for body

```
b-Lod p, q
  Lod o, q
  Leq
  Fjp    q (out of loop)
  Lod p, q
  Inc
  Str p, q
  Ujp q
```

13-With statement

```
a-Lda
  Str p, q (in temporary location)
```

14-Expressions

We have considered three cases:

i-relational operators

ii-arithmetic operations on integers

iii-logical operators

Case i-Relational operators.

The possible sequences are, according to operand class:

```
a-Ldc q       b-Ldc q   (for operations between constant and
                                   variables)
  Lod p, q      Ind q

c-Lod p, q    d-Ind q   (between variables)
  INd q         Lod p, q

e-Lda p,q               (for arrays and records)
  Lda p,q
  Equ
```

Case ii - Arithmetic operations on integers

There are cases according to operands classes:

```
a-Ldc    q        b-Ldc   q    (constant-variable)
  Lod p,q           Ind   q
  Opr               Opr

c-Ldc    q                     (constant-expression)
  OP

d-Lod p,q         e-Ind    q    (variable-variable)
  Ind    q          Lod p,q
  Opr               Opr

f-Lod p,q         g-Ind    q    (variable-expression)
  INd   q           Lod   p,q
  Opr               Opr
h-OPr                           (expression-expression)
```

Case iii-Boolean operators

Same as above.

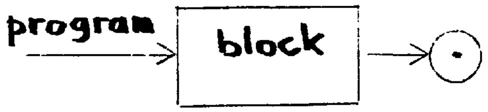
15-Array access:

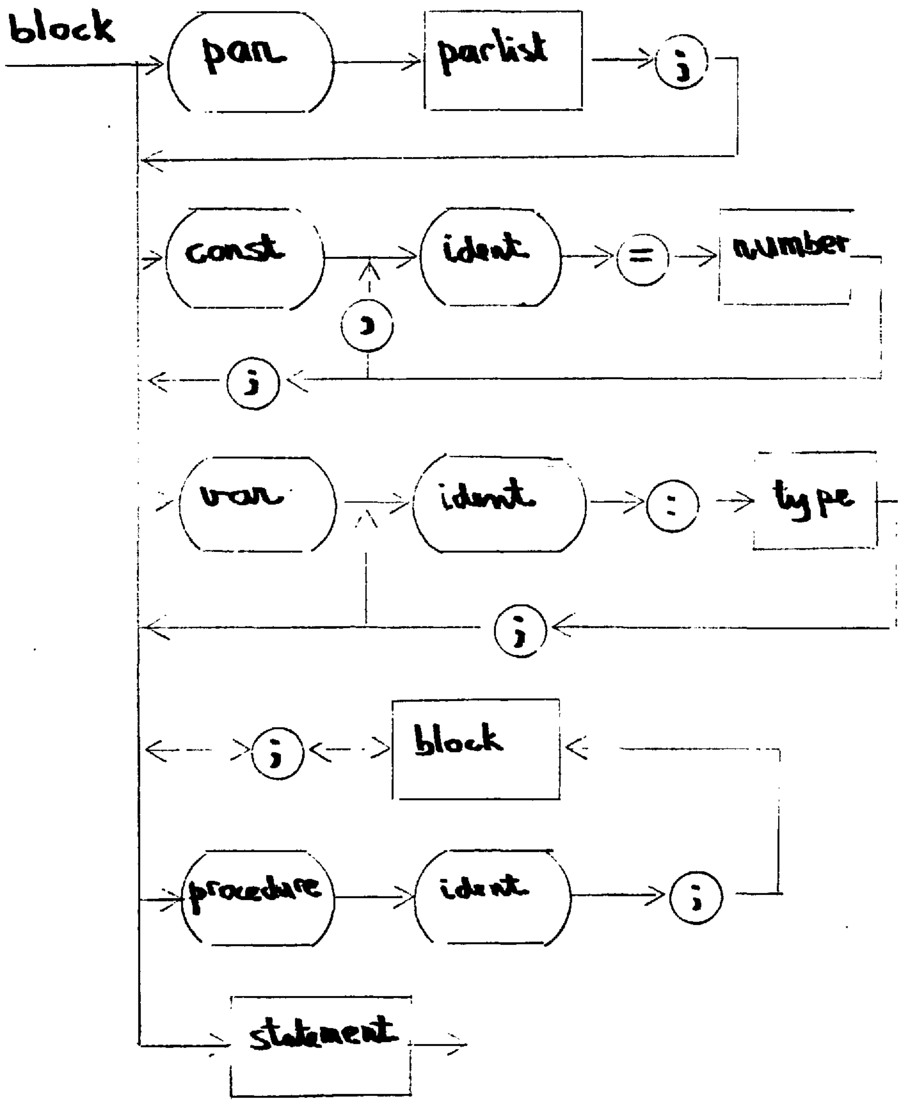
```
a-Lda p, q
  expression
  Chk   q   (check bounds)
  Dec   q
  Ixa   q   (index)
```

16-Pointer access

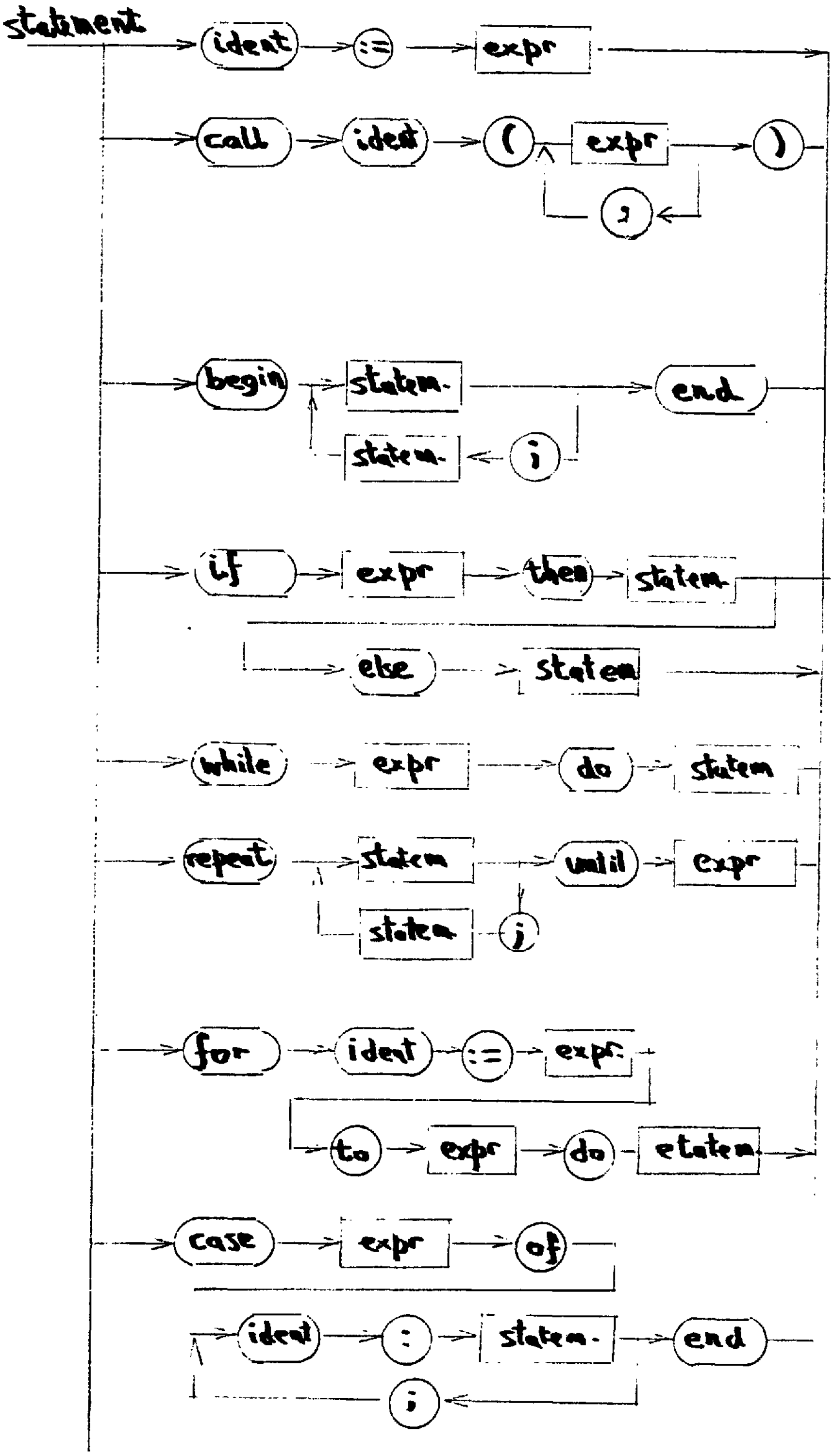
```
a-Lod p, q        b-Ind q
```

Appendix 4- EPL/0 syntax flowgraph

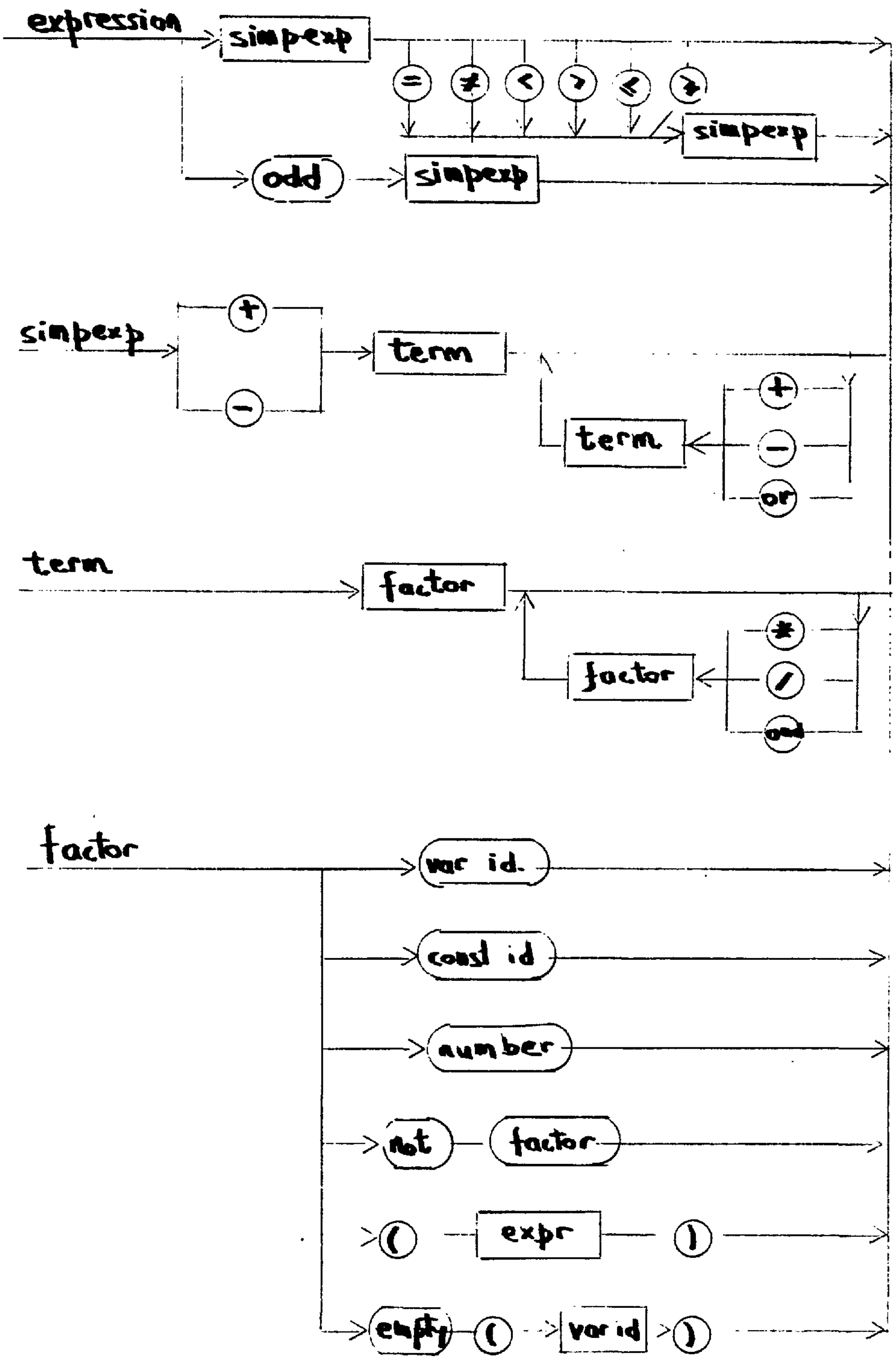




Appendix 4- EPL/O syntax flowgraph (continued)



Appendix 4- EPL/0 syntax flowgraph (continued)



Appendix 4- EPL/0 syntax flowgraph (continued)