STATIC ANALYSIS OF PROGRAMS
FOR CONTEXT-SENSITIVE
COMPUTER ARCHITECTURE

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
OF
THE AUSTRALIAN NATIONAL UNIVERSITY

CHRISTOPHER JOHN LOKAN

AUGUST 1984
STATEMENT

I hereby state that this thesis contains only my own original work except where explicit reference has been made to the work of others.

C.J. Lokan

C.J. Lokan
ACKNOWLEDGEMENTS

I would like to thank the staff members and my fellow research students in the Department of Computer Science, for providing a stimulating research environment and for many interesting discussions. I particularly wish to thank my supervisor, John Hurst. Many of the ideas presented here originated in discussions with John. His encouragement and guidance have been invaluable; so too were his comments on drafts of this thesis.

Special thanks to Morag, for reading and commenting on drafts of this thesis, for helping to type it, and for giving me so much moral support. Her contribution to this thesis is enormous.

Thanks too to my family; to my housemates, Mark and Peter; and to the many other friends who have coped so cheerfully with life with a Ph.D. student.
ABSTRACT

A context-sensitive computer architecture is one in which the virtual architecture changes dynamically, to take advantage of knowledge of the current needs of the executing program. This thesis develops a method for automatically identifying the points during program execution when the virtual architecture (or "environment") should be changed. Those points are recognized at compile time, from a static analysis of the program text. This knowledge is used to guide the encoding and subsequent interpretation of the program, so that both static code space and dynamic execution time are minimized. When the relationships between successive dynamic environments are considered, the effort of changing environments is also minimized.

Two items of information about each program unit are required: a prediction of its dynamic execution profile, and a tree that represents its structure. A model of program execution is developed, which enables the dynamic execution profile of a program unit to be predicted from a static examination of its text. Construction of the "structure tree" is described, based on the syntax of the program unit, the use of operands and operators within it, and the flow of data through it. The algorithm for finding points where the environment should change is then presented in detail. Shortcomings of the method are discussed; other applications of the structure tree and program execution model, and avenues for further research, are suggested.
# Table of Contents

1. Introduction
   1.1. Context-sensitive architecture 2
   1.2. Static program analysis 3
   1.3. Organization of the thesis 4

2. Review of related literature
   2.1. High-level language computer architecture 6
   2.2. Organization of computer architectures
      2.2.1. Addressing schemes 9
      2.2.2. Instruction formats and encoding schemes 11
      2.2.3. Instruction sets 12
   2.3. Measurement of program characteristics
      2.3.1. Use of language features 15
      2.3.2. Other statistical properties of program behaviour 17
      2.3.3. Dynamic software metrics 18
   2.4. Modelling program behaviour 19
   2.5. Automatic program analysis
      2.5.1. Program verification 21
      2.5.2. Measuring program quality 22
      2.5.3. Other applications 23

3. Model of Host Architecture
   3.1. Canonical Measures of Interpretation
      3.1.1. Definitions 24
      3.1.2. Use of the canonical measures 27
   3.2. Architectural features of the image machine and host machine 29
   3.3. Representation of the image architecture on the host machine
      3.3.1. Representing instructions 31
      3.3.2. Representing environments
         3.3.2.1. Defining the entries in environment tables: basic approach 39
         3.3.2.2. Defining the entries in environment tables: implementation 40
   3.4. Run-time system operation 41
   3.5. Binary representation of the image program 44
   3.6. Summary 45
4. Context changes during program execution

4.1. Different levels for grouping statements
   4.1.1. Single source statement
   4.1.2. Entire program
   4.1.3. Individual program unit
   4.1.4. Parts of program units

4.2. Environment change due to changing locality

4.3. Environment change for local optimization

4.4. Code segmentation strategy

4.5. Summary

5. Models of dynamic program execution

5.1. Basic framework for modelling program profiles
   5.1.1. Sequence
   5.1.2. Alternative
   5.1.3. Iteration
   5.1.4. The Goto-statement
      5.1.4.1. Basic control structuring
      5.1.4.2. *Premature* loop exit
      5.1.4.3. Error trapping
      5.1.4.4. Miscellaneous
      5.1.4.5. Skipped code
   5.1.5. Summary

5.2. Individual heuristic rules
   5.2.1. Case-statements
   5.2.2. If-statements
      5.2.2.1. If-Then statements
      5.2.2.2. If-Then-Else statements
      5.2.2.3. If-Then and If-Then-Else statements combined
      5.2.2.4. Summary
   5.2.3. Iteration statements
      5.2.3.1. Repeat-loops
      5.2.3.2. While-loops
      5.2.3.3. For-loops
      5.2.3.4. Iteration statements considered together
      5.2.3.5. Summary
   5.2.4. Goto-statements

5.3. Comparison between different models
   5.3.1. Simple model
   5.3.2. Complex model
   5.3.3. Comparisons

5.4. Summary

6. Segmentation of program units

6.1. Introduction
6.2. Construction of a tree to represent program structure
6.2.1. Syntax analysis
6.2.2. Detecting segments with constant locality
6.2.3. Pruning the tree
6.2.4. Summary: tree construction

6.3. Finding a good segmentation
6.3.1. Tree traversal
6.3.2. Decision at each node
   6.3.2.1. Example
6.3.3. Parameter settings
6.3.4. Summary: segmenting program units

6.4. Producing the binary form of the program unit

6.5. Results
6.5.1. Testing strategy
   6.5.1.1. Implementation of algorithms
6.5.1.2. The test data
6.5.1.3. Testing method
   6.5.2. Predicted and observed results
5.6.2.1. ICL assembly code
6.5.2.2. "Ideal" intermediate language
6.5.2.3. Summary of results

6.6. Summary

7. Exploiting similarities between segments
7.1. Introduction
7.2. No change to tables
7.3. Changing windows on the environment tables
   7.3.1. Introduction
   7.3.2. Classification of nodes
   7.3.3. Grouping nodes
   7.3.4. Ordering the entries in environment tables
   7.3.4.1. Two nodes
   7.3.4.2. Three nodes
   7.3.4.3. Ranking the orderings
   7.3.4.4. Four nodes
5.3.4.5. Summary
   7.3.5. Binary encoding
   7.3.6. Summary

7.4. Results
7.4.1. Mechanics of the algorithm
   7.4.1.1. Discussion of results
7.4.2. Changes in program size
   7.4.2.1. Discussion of results

7.5. Summary

8. Conclusions
8.1. Context-sensitive architecture
8.2. Optimal ordering of environment table entries  
8.3. Modelling program execution  
8.4. Segmenting program units  

References  

Appendix A. *Ideal* intermediate language for Pascal
List of Figures

| Figure 3-1: | Example Source Program | 32 |
| Figure 3-2: | Example Image Program | 33 |
| Figure 3-3: | Image architecture for example program | 34 |
| Figure 3-4: | Encoding of image program for example image machine | 36 |
| Figure 3-5: | Binary form of image program | 45 |
| Figure 4-1: | Procedure encoding to reflect changing locality | 51 |
| Figure 4-2: | Procedure encoding with local optimization | 54 |
| Figure 5-1: | Distribution of $p_{\text{then}}$ for If-Then statements | 71 |
| Figure 5-2: | Distribution of $p_{\text{then}}$ for If-Then-Else statements | 73 |
| Figure 5-3: | Distribution of $p_{\text{then}}$ for If-statements | 75 |
| Figure 5-4: | Distribution of $n_{\text{iter}}$ for Repeat-loops | 77 |
| Figure 5-5: | Distribution of $n_{\text{iter}}$ for While-loops | 78 |
| Figure 5-6: | Distribution of $n_{\text{iter}}$ for For-loops | 80 |
| Figure 5-7: | Distribution of $n_{\text{iter}}$ for Repeat-loops and While-loops | 83 |
| Figure 6-1: | Representation of compound statement | 89 |
| Figure 6-2: | Representation of iteration statements | 90 |
| Figure 6-3: | Representation of alternative statements | 91 |
| Figure 6-4: | Syntax tree for example source program | 92 |
| Figure 6-5: | Adding nodes to reflect locality | 95 |
| Figure 6-6: | Modified tree for example source program | 96 |
| Figure 6-7: | Result of pruning the tree for the example source program | 98 |
| Figure 6-8: | Example of decision algorithm | 112 |
| Figure 7-1: | Sample segmentation tree | 138 |
| Figure 7-2: | Window change, in optimal table for two nodes | 151 |
List of Tables

Table 3-1: Format codes and their meanings 35
Table 7-1: Potentially optimal table orderings for three nodes 153
Table 7-2: Sequence for trying potentially optimal orderings for three nodes 156
Table 7-3: Potentially optimal table orderings for four nodes 159
Table 7-4: List of orderings to check, for four nodes 161
Table 7-5: Sequence for trying potentially optimal orderings for four nodes 163
Chapter 1
Introduction

A general goal of computer architects is to provide a sympathetic architecture on which to run computer programs. In some specialized problem domains, this can be achieved by providing an architecture whose physical characteristics are geared towards solving problems from that domain. This "special purpose" architecture provides an "ideal" environment for solving those problems.

More often, a general-purpose computer is designed. A set of primitive operations is provided; programs to solve a wide variety of problems must be translated into a form that can be executed by the general-purpose machine. The simplicity of this mapping, and by implication the suitability of the architecture as an environment in which to solve the problem, varies depending on the nature of the problem. These architectures generally provide a low-level facility for solving problems, with a large "semantic gap" [Myers 82] between the logical actions expressed in programs and the primitive actions executed by the computer.

A common alternative to translating a source program directly into the machine language of the general-purpose computer is to translate it into an intermediate language, which is closer to the source language in semantics. The intermediate language program is interpreted by a program running on the underlying host. In a sense the intermediate language is the machine language of a virtual computer, which provides a higher-level facility for solving problems.

This thesis investigates the relationship between computer programs written in high-level languages, and a particular type of virtual computer architecture.
1.1. Context-sensitive architecture

A model is proposed for a context-sensitive architecture [Hurst 82]: a virtual architecture that can adapt itself dynamically, during the course of execution of a program, in an effort to provide an ideal environment to suit the program at all times. As the program’s actions and requirements vary in phases during its execution, the virtual architecture changes to match it.

The virtual architecture constitutes the environment, or context, in which the intermediate-language program is interpreted. Interpretation of the binary form of the program requires knowledge of the context. A given bit string has different meanings in different configurations of the virtual architecture - hence the name context-sensitive architecture.

The proposed architecture is based on one defined by Flynn and Hoevel [Flynn 80, Flynn 83] for research on directly executed languages. In essence, it is comprised of tables which include only the operators, operands, and labels in current use. From the viewpoint of an executing program, the virtual architecture provides no unnecessary operators or addressing capability, and is thus ideally suited to the program. The representation of the context-sensitive architecture on the underlying host, and the interpretation of the binary form of the program on the context-sensitive architecture, largely follow methods suggested by Flynn [Flynn 80].

There are two stages in the creation of the binary form for a source program:

1. The source program is compiled into an intermediate language;

2. The binary form of the intermediate program is generated. In addition to the encoded instructions, it includes encoded information on when and how to set up the different configurations of the context-sensitive architecture.

Except for a discussion of desirable characteristics for the intermediate language,
compilation is not considered here. This thesis addresses the second stage: analysing a program to determine how the context-sensitive architecture should be configured to suit it best, and generating appropriate binary code.

1.2. Static program analysis

To find the best way to tailor the context-sensitive architecture to suit the intermediate-language program, a static analysis of the program is performed. The analysis must determine the points in the program at which the virtual architecture should be changed during execution, and how the new virtual architecture should be set up after each change. This information is encoded in the final binary form of the program, along with the encoded instructions.

A good encoding leads to a good match at all times between what the virtual architecture provides and what the program currently needs. It should also minimize the static size of the binary encoding, the dynamic effort of decoding the binary code, and the work involved in changing the virtual architecture. The first three of these goals usually conflict with the last: the best balance is sought.

The virtual architecture is embodied in three tables, each with an associated window: access to table entries that lie outside the windows is precluded. Setting up the virtual architecture requires the identification of three things: what entries should appear in the tables, the order they should appear in, and the size and placement of each window. The entries that must appear in the tables follow as a consequence of the decisions on when to change the environment. Order of table entries, window placement, and timing of environment changes are all varied in an attempt to find the optimal combination.

As a starting point, it is assumed that the virtual architecture is changed on entry to each program unit; each window always covers the entire table; and the order of the table entries is irrelevant.
The analysis is presented in two stages, in which the initial assumptions are progressively modified:

1. Subdivisions are sought within program units, splitting them into segments; the virtual architecture is changed at each transition between segments. The match between program and architecture is improved, and the static size and dynamic decoding effort can also be improved. The cost of changing environments can become high, however.

When segmenting a program unit, regard is paid to its syntax; data and control flow; patterns of use of operators and operands; and dynamic execution profile. A model of program execution is developed so that the latter can be predicted, using information derived from a static analysis of the program unit’s source text.

In this first stage, each segment is considered independently. The order of table entries is still irrelevant, and windows still cover entire tables.

2. Program units are segmented as before, and relationships between the segments are considered. Table entries are carefully ordered, and windows vary in size and position, as the repeated use of some operators and operands in different segments is exploited to reduce the cost of changing the virtual architecture. The architecture is even better tailored to the program, and static size and dynamic decoding effort are further reduced.

1.3. Organization of the thesis

This thesis is presented in the following format:

• Chapter two reviews the literature on some of the topics related to this work.

• Chapter three describes the context-sensitive architecture: the physical
requirements of the underlying host, the representation of the virtual architecture on the underlying host, and the encoding of source programs on the virtual architecture.

- There are two different circumstances in which a separate segment is identified within a program unit. Chapter four describes the characteristics of the two types of segment.

- Chapter five presents a model of program execution, developed to enable a prediction of the dynamic execution profile of a program unit to be made from a static examination of its source text. The predicted profile is a major item of information used in splitting a program unit into segments.

- Chapter six describes the algorithm used to find the best segmentation of a program unit, and to produce the binary code to be executed on the context-sensitive architecture. Experimental results are reported.

- Chapter seven explains how the relationships between segments can be exploited to reduce the cost of changing the virtual architecture. The algorithm for producing optimal binary code is presented, and experimental results are reported.

- In chapter eight the work presented in this thesis is summarized, and improvements that could be made are pointed out. Topics for further research are mentioned, and other applications for the methods developed here are discussed.
Chapter 2
Review of related literature

This thesis is related to several topics in current computing research, including computer architecture, measurement and modelling of program behaviour, and automatic program analysis. These topics are discussed in the following sections, with references to the literature.

2.1. High-level language computer architecture

A conventional computer architecture provides a machine language containing a low-level set of instructions that it can execute directly. Programs written in high-level languages must be translated by compilers into that machine language. The primitive instructions of the computer are usually far removed from the semantic actions expressed in the high-level source language: this "semantic gap" [Myers 82] makes compiler-writing complex and compilation expensive.

Recent years have seen considerable research on computer architectures that are better suited to implementing high-level languages. With a better match between languages and machines, the semantic gap is reduced, compilers are simpler, and the size and speed of the compiled programs is improved. Several different approaches have been taken.

At the opposite extreme from the conventional Von Neumann architecture is the "direct execution" architecture [Chu 75, Chu 77, Chu 81a, Chu 81b]. High-level language programs are directly executed by the hardware, with no need for compilation, assembly, linkage, and loading. In effect, the high-level language itself
is the image architecture. One of the motivations for this architecture is the removal into hardware of the many layers of software, reducing the mounting software cost [Chu 77] and taking advantage of the decreasing cost of hardware. Another important advantage is the ability for a programmer to follow the execution of a program interactively [Chu 81a].

In the SYMBOL project [Rice 71, Laliotis 75], programs written in a high-level language (also called SYMBOL, developed as part of the project) are accepted directly for execution by the SYMBOL architecture. A hardware-implemented compiler translates a program into an intermediate form, which is then executed directly by the processor. The use of an intermediate form leads Chu to call SYMBOL an *indirect execution* architecture [Chu 75].

Chu’s next classification is the *syntax-oriented* architecture [Chu 75]. High-level language source programs are converted by a software translator into an intermediate language, which is oriented towards the syntax of the high-level language. The intermediate form is executed directly by the hardware. The implementation of Algol on the Burroughs B5500 computer [Organick 73] exemplifies this classification.

The direct execution, indirect execution, and syntax oriented architectures all use specially designed hardware. The hardware is responsible for many system functions traditionally performed by software, and is designed specifically to support particular high-level languages. Several authors have criticized this approach, commenting on the low flexibility and high development cost of implementing compilers and other system functions in hardware (made even less cost-effective by an increasing understanding of how to produce compilers in software) [Ditzel 80, Wulf 81, Kavi 82], and the poor suitability of single-language machines in the multilanguage environments that prevail today [Ditzel 80, El-Halabi 82].
Another approach to the implementation of a high-level language is to use an interpreter. Source programs are translated by a compiler into an intermediate form, which is interpreted by a program running on a general-purpose computer. The intermediate form defines a virtual architecture; the interpreter implements the virtual architecture on the *host* general-purpose computer. The interpreter could be a conventional machine language program, as in the SNOBOL4 system [Griswold 72, Griswold 71] and in the implementation of Pascal [Jensen 78] via P-code [Nori 81]. Alternatively, microprograms in the control store could interpret the instructions of the virtual architecture directly [Agrawala 76].

Some computers have been designed specifically to support the use of interpreters. They provide no standard instruction set; instead, their microprograms are optimized for interpretation. Different architectures are emulated by loading different microprograms into the control store. The Burroughs B1700 [Burroughs 72, Wilner 72a] is perhaps the best known such machine; the Nanodata QM-1 [Nanodata 74, Salisbury 76] is another.

Implementing architectures in microcode is an extremely flexible approach. With dynamic microprogramming and writable control stores it is possible to change architectures easily, and to tune their instruction sets to suit particular applications [Agrawala 76].

### 2.2. Organization of computer architectures

The microprograms resident in control store determine the addressing scheme, instruction formats, and instruction set used in an architecture. Each of these topics has been the subject of research, in an attempt to develop architectures that have low overall system cost, and on which programs can be encoded concisely and executed quickly.
2.2.1. Addressing schemes

Several different addressing schemes have been investigated, including:

- Zero-address ("stack") architectures (e.g. B5500, Pascal P4 machine);

- One-address ("accumulator") architectures (e.g. PDP-8, IBM 1800, Whirlwind 1 [Bell 71]);

- Two-address architectures: these can be characterised as either storage-to-storage (e.g. the implementation of COBOL on the B1700) or register-to-register ("register file" [Cragon 79]) architectures (e.g. IBM 360/370);

- Three-address storage-to-storage architectures (e.g. CDC6600);

and some combinations of these schemes.

The relative merit of the schemes is determined by the distribution of the instruction types, and complexity of expressions, used in practice. Different authors have compared these addressing schemes and reached different conclusions:

- Cragon [Cragon 79] uses empirical data from [Alexander 75] and [Tanenbaum 78] on the frequency of use of different types of instructions. A theoretical analysis leads to the conclusion that no significant differences exist in execution speed or code space.

- Malik [Malik 78] compares 3-operand, 2-operand, and 1-operand addressing with program tree and Polish string representations of programs. Comparisons are made concerning the number and size of instructions, stack and register requirements, complexity of interpretation, and the software science "level" [Halstead 77]. Three-operand instructions provide the smallest execution time estimates, the highest "level", and the greatest potential for microcode optimization [Ma 81].
Keedy recommends the use of a combined accumulator/stack technique [Keedy 78a, Keedy 78b, Keedy 79], as in the ICL2900 [Keedy 77]. In evaluating all but the simplest expressions this technique is said to be the best [Keedy 79]; for simple expressions two-address storage-to-storage is superior. Three-address instructions are considered briefly [Keedy 78b]: marginal improvements in bits per instruction, and elements decoded, are not seen as sufficient to justify extra implementation costs.

Myers [Myers 77a, Myers 78, Myers 82] believes that two-address storage-to-storage is best, since simple expressions prevail [Knuth 71a, Tanenbaum 78], and that stack- or register-oriented instruction sets should not be used in future architectures [Myers 78].

Another addressing scheme has been proposed [Flynn 80, Hurst 82], which combines the features of a stack and three-address storage-to-storage instructions. Three-operand instructions are used, but temporary values are stored on an expression-evaluation stack instead of in named temporary variables. Some format bits are needed in each instruction, to indicate whether each operand refers to the stack or to a named literal or variable. This scheme is used in the DELtran architecture [Flynn 83], and in the context-sensitive architecture considered in this thesis.

The three-address/stack, or "context sensitive" [Hurst 82], addressing scheme has been compared with zero-address, one-address, two-address storage-to-storage, and three-address storage-to-storage techniques [LeeMK 81]. Based on reported distributions of instruction frequencies [Knuth 71a, Tanenbaum 78], the context sensitive scheme is shown to reduce code space requirements for assignment statements by some 20% over a typical mix of assignments. Similar improvements are reported for other language constructs, such as loops and if-statements.
2.2.2. Instruction formats and encoding schemes

Traditionally, instructions are encoded using a fixed number of fixed-width fields, which are encoded independently in a fixed order. It has been found that other methods of encoding instructions can reduce the code space requirements of programs dramatically, leading to performance improvements as well.

Alternative encoding schemes are produced by using:

- variable-width fields: frequency-based encodings, or fields whose size depends on the number of different objects to be distinguished; and/or

- variable numbers of fields in instructions, due either to the addressing scheme or the encoding together of some combinations of operands and operators.

In a frequency-based encoding, common instructions are given short bit patterns and instructions that occur infrequently are given longer bit patterns. The Burroughs B1700 uses this scheme in its implementation of S-machines [Wilner 72a, Wilner 72b]. Frequency-based encodings using variable-length fields, whose length can be any number of bits without regard to byte boundaries, reduce memory requirements by between 25% and 75% compared to byte-oriented systems [Wilner 72b].

Tanenbaum’s EM-1 machine [Tanenbaum 78, Stevenson 79] uses frequency-based encodings of instructions, in which each instruction is encoded in one or more 8-bit bytes. The most frequent instructions consist of a single one-byte field.

The format of EM-1 instructions differs from the traditional format in which each field is encoded independently [Tanenbaum 78]. Some common combinations of opcodes and operands are represented by single instructions in which the operand is built in; other instructions indicate only the opcode, with explicit operands following. The method for determining which instructions should be encoded with
built-in operands, and which should have their fields encoded independently, is presented in [Stevenson 79]. By building operands into some instructions, and using a frequency-based encoding of the resulting instruction set, program code size can be reduced by a factor of 3, compared to conventional systems [Tanenbaum 78].

The DELtran architecture [Flynn 80, Flynn 83] uses the three-address/stack addressing scheme, so the number of fields encoded in each instruction is variable. Each instruction includes some format bits, an opcode, and from zero to three named operands; each field is encoded independently. Field widths can be an arbitrary number of bits, with no need to consider byte or word boundaries. Minimal width fields are used: a field that identifies one of \( n \) distinct values is encoded using \( \lceil \log_2 n \rceil \) bits. Code space requirements for programs on the DELtran architecture is half that for conventional systems [Flynn 80].

The context-sensitive architecture described in chapter 3 of this thesis is based on the DELtran architecture, and uses the same addressing scheme and instruction format.

### 2.2.3. Instruction sets

Instruction sets can be designed with different goals in mind. In single-language machines, or machines where different interpreters provide a different virtual architecture for each high-level source language, an instruction set can be designed to mirror closely the semantic actions of the source language. A simple, transparent translation from source programs to image programs is important in research on directly executed languages [Flynn 80], so DELtran instruction sets are designed to suit particular high-level languages.

More general instruction sets are needed for computers that must handle several high-level source languages. There is a wide range of complexity among such instruction sets. Some systems provide large numbers of instructions, to deal with many different types of operations and operands; others attempt to provide small,
canonical instruction sets. An example of the latter is the Reduced Instruction Set Computer (RISC) [Patterson 80, Patterson 81]. Performance tests indicate that the RISC competes well with computers that have complex instruction sets [Patterson 82].

The process of designing an instruction set is still largely intuitive, depending on the skill and experience of the designers [Stevenson 79]. In an attempt to make it less intuitive, a systematic methodology has recently been proposed [Marovac 83]. The applications for which a computer is to be used are considered, and ultimately an instruction set that is optimal for that application environment is produced. The RISC instruction set was developed using a similar approach [Marovac 83].

A quantitative measure of how well a given instruction set is suited to the application environment in which it is used would be a useful tool for designers of instruction sets. The use of software science metrics [Halstead 77] has been proposed [Kavipurapu 79] in an attempt to provide such a tool.

Once an instruction set is designed, and implemented in microcode, it can still be adapted to suit a given application environment. If the microprograms that define the instruction set are held in writeable control store, they can be changed - this makes it possible to tune the instruction set to optimize the performance of the computer for its typical workload. For example, the instruction set of the Mesa computer [Johnsson 82] was optimized by examining the machine code generated for many programs, and defining new single instructions to replace commonly-occurring sequences of instructions. The new instructions are smaller to encode than the original sequences, and they execute faster because there is less traffic of operands between registers and memory [Sweet 82].

Several projects have aimed to automate the tuning of an instruction set, using
data obtained from dynamic traces of program executions. The heuristic synthesis approach [Abd-Alla 74] recognizes program segments (in particular, short loops with high execution frequencies) that are candidates for new instructions. Microcode to implement the new instructions is synthesised automatically, and added to the control store. When this process is applied several times, the instruction set defined by the microprograms in control store becomes increasingly well suited to the execution of a typical mix of programs.

This approach is extended by collecting the program trace data and adapting the instruction set dynamically during program execution [Abd-Alla 76, El-Ayat 77], producing an instruction set tailored to the particular program execution. The same approach is extended further by recording the changes in a database [Sakamura 79], adding a "learning" capability to the system.

Another proposal along similar lines [Rauscher 78] shares some goals with the work described in this thesis. Candidates for new instructions are identified from compile-time predictions of execution frequencies of blocks in a program. The compiler generates microcode for these instructions, which are added to the normal instruction set to produce an optimal instruction set for executing the particular program; the source program is then translated into that optimal intermediate form.

2.3. Measurement of program characteristics

This thesis is concerned in part with defining an architecture, and also with modelling the expected execution behaviour of programs. Both of these topics require knowledge of the characteristics of programs. This section describes the sorts of measurements made of program characteristics, and why they are useful.
2.3.1. Use of language features

The importance of empirical research on the use of language features in actual sample programs has been recognized since the appearance of Knuth's classic paper [Knuth 71a]. Knowledge of how languages are used is clearly useful to language designers; it provides compiler writers with code improvement information; and it gives architecture designers a foundation for selecting instruction sets and addressing schemes [CookR 82]. In addition, Berry suggests that this knowledge can be used to select manageably small, yet sufficient, sets of test programs for use in the debugging of new compilers [Berry 83].

Statistics can be collected statically, by examining the source text of programs, or dynamically, by monitoring the dynamic execution profiles of programs [Ingalls 72]. Static figures are cheaper and easier to collect, and experience has shown that the same language features dominate static and dynamic statistics [Berry 83]. Consequently, many studies have only measured static figures.

Languages for which usage statistics have been gathered include ALGOL 60 [Wichmann 73], PL/I [Elshoff 76, Elshoff 77], FORTRAN [Knuth 71a, Robinson 75], SAL [Tanenbaum 78], XPL [Alexander 75], COBOL [Salvadori 75, Chevance 78, Al-Jarrah 79, Torsun 81], SIMULA 67 [Misherghi 80], APL [Saal 77], and Pascal [Shimasaki 80, Brookes 82, CookR 82]. Some of these studies report static figures, some report dynamic figures, and some report both.

Statistics are gathered on the use of different types of statements; complexity of expressions; use of different operators in expressions; use of different types of operands in expressions; use of particular constants; number of dimensions in arrays; etc. Variations have been observed in the use of specific features in different languages (e.g. procedure calls are much less frequent in FORTRAN than in SAL [CookR 82]); in the use of particular features of a language in different applications (e.g. FOR-statements range from 0 - 5% of static statements in non-scientific Pascal programs, and from 6 - 14% of static statements in scientific programs [Brookes 82]);
and in the use of a language by different types of programmers (e.g. Knuth found differences in the use of If-, Goto-, and assignment statements between programmers from Lockheed and Stanford [Knuth 71a]). However, most variations are minor [Berry 83], and the universal conclusion is that most programs are very simple: control statements are used in their simplest forms, arrays rarely have more than one dimension, expressions rarely involve more than one operator (and most have none), loop control variables are almost always incremented by one, the constants 0 and 1 are common, etc. This is true even in languages that have special features, such as SIMULA with its classes and coroutines [Berry 83]. The simple use of basic algorithmic constructs appears dominant.

From an architectural viewpoint, these results indicate (among other things) that a memory-to-memory assignment instruction is highly desirable [CookR 82]; immediate addressing is valuable, as many constant operands are used [CookR 82]; and in general that expensive general-purpose features are largely unnecessary. The frequencies from which to derive frequency-based encodings, and from which to decide on the best addressing schemes, are available; the instructions that need to be most highly optimized are evident. Dynamic usage figures are useful for tuning instruction sets (as discussed in section 2.2.3).

Figures on the misuse of language features have also been published; these too have implications for language and system designers. Reports on syntax errors [Gannon 77, Gannon 78] demonstrate the usefulness of type checking and the superiority of using semicolons as statement terminators instead of separators [Leblanc 82]. Reports on run-time errors [Zelkowitz 76, Zelkowitz 77, Leblanc 82] emphasize the importance of run-time checking mechanisms.
2.3.2. Other statistical properties of program behaviour

Other aspects of program behaviour also have implications for computer architects. For example:

- The effectiveness of pipelining [Ramamoorthy 77] is influenced by the average number of instructions executed between branch instructions. As this is usually only about four instructions [Lenfant 75, Wiecek 82], multistream architectures [Flynn 72] such as the IBM 360/91 [Anderson 67], which prefetches instructions down both branches of a conditional jump, are of interest. Algorithms for predicting the outcome of conditional branches [SmithJE 81, LeeJK 84] are helpful.

- Average branch displacements, together with the desire to avoid having too many jumps to target instructions on different virtual memory pages, affect the choice of page size. Wiecek reports that 25% of branches, in compilers running on a VAX-11, are to destinations on different 512-byte pages [Wiecek 82].

- Loop buffers [LeeJK 84] can hold small loops in high-speed memory, eliminating the cost of many instruction fetches. Buffer sizes are affected by average loop lengths, which Kobayashi shows to be about ten machine instructions or less [Kobayashi 84] (at the source language level, Knuth found that most FORTRAN DO-loops involve only one or two statements [Knuth 71a]). The Cray-1 [Russell 78] is one computer that features loop buffers.

The advantage gained from loop buffers is proportional to the average number of iterations per loop occurrence. Figures reported for this average are 4 [Lenfant 75], 5 [Lokan 83a], and a range from 2.6 to 13.3 [Kobayashi 84].

- The effectiveness of different memory management schemes is
determined by the sequence of pages referred to (i.e. the *reference string*) during program execution. A great deal of research has been done on this topic: see [Denning 80] for a survey.

Branch prediction strategies, and knowledge of the average number of iterations per loop execution, are of interest in chapter 5 of this thesis, where a model for predicting profiles is developed.

2.3.3. Dynamic software metrics

Another reason for collecting statistics on programs is the interest in software metrics [Gilb 76]. Various characteristics of programs are measured, such as the number of operands and operators they use. This information is used to develop, validate, or refute different measures of program properties.

Measures based on a static analysis of the text of a program are generally concerned with evaluating the complexity, and in some way the quality, of the program. This is discussed further in section 2.5.2.

Operators and operands can also be counted dynamically. One potential application for these counts was mentioned in section 2.2.3: the suitability of an architecture for its application environment might be quantified [Kavipurapu 79] using software science metrics [Halstead 77]. A similar suggestion is made in [Oldehoeft 79]: the notion of *work* done in executing a program is defined using dynamic counts of software science primitives, and the possibility of comparing work for the same program on different architectures is mentioned. It is also suggested that different programs that solve the same problem can be compared, to see which is the most efficient (i.e. requires the least work to execute).
2.4. Modelling program behaviour

Models of program behaviour are developed in order to enable some aspect of the execution of a program to be predicted.

Memory reference models are a familiar example. Several models have been developed to describe the patterns of page references in program reference strings [Spirn 77, Graham 77, Denning 80]. Memory management policies, such as page replacement policies and the choice of partition sizes when page tables are shared between programs in a multiprogramming environment, are based on these models.

Ribeyre and Saintoyant investigate the improvement in paging performance that results if relocatable segments are rearranged to improve locality of reference [Ribeyre 77]. (As an aside: they comment that it would be useful to be able to split segments into smaller subsegments, possibly with help from programmers - the algorithms developed in this thesis may enable it to be done automatically.) It has been shown that reductions of 15-20% in working set size can be achieved, and that page faults are reduced by a factor of between two and ten [Hatfield 71]. Ribeyre and Saintoyant aim to develop a method for evaluating the advantage gained by each potential rearrangement, without having to change the actual programs. As part of the method, they use a Markov model of program behaviour, to predict the execution profile of a program.

The use of Markov models to estimate program execution time has a long history, going back at least to [Ramamoorthy 65]. Others to use them are [Beizer 70, Wegbreit 75, Rauscher 78]. Knowledge of the probability of selection of each path from a decision point is needed; this depends on the distribution of data values. If this distribution is unknown, it is either estimated in some manner (e.g. by assuming all values to be equally likely) or determined by gathering statistics. When path selection probabilities and the means and variances of execution times for each statement are known, the estimates of mean and variance of total execution time are computed. Remarkably accurate predictions can be made using this approach;
Rauscher reports that a correlation coefficient in excess of 0.995 has been measured between predicted and observed execution times [Thall 77].

A different approach to the estimation of execution time is described in [Booth 79]. A flow graph is constructed for the given program, and a set of flow balance equations [Knuth 73] is formed and solved. Execution times assigned to each node in the graph are then used to produce an expression for program execution time. The actual time estimates rely on measurements of the program profile; thus Booth's work explains why a program takes as long as it does to execute, instead of predicting execution time beforehand.

Similarly, [Cohen 74] and [Oldehoeft 83] describe systems in which execution times can be calculated, once a user has provided estimates of the likelihood of selecting each branch of an alternative statement, and of the number of iterations of each loop.

Another method for computing execution time [Haase 81] is more concerned with verifying the fulfilment of constraints. The concept of *guarded commands* [Dijkstra 76] is used with formal means of predicate transformers, and estimates of execution time overheads at various stages, to check that execution time does not exceed real-time constraints.

In chapter 5 of this thesis, a model of program execution is developed in an effort to enable one to predict dynamic profiles from syntactic information. Knowledge of branch selection probabilities and loop iteration counts is required, as it is in all of the models described above. This work differs in the method of obtaining this knowledge. Markov models and the models of [Cohen 74] and [Oldehoeft 83] require specific values to be supplied, either interactively or in the form of detailed information on distribution of data values. By contrast, in this work general rules are applied to estimate the branch selection probabilities and loop iteration counts. The rules are applied automatically, and are based on program syntax. One of the aims of the thesis is to see how satisfactorily such a model performs.
2.5. Automatic program analysis

2.5.1. Program verification

One of the most common reasons for analysing programs automatically is the desire to prove them correct. The analysis can take many forms, both static and dynamic.

Data flow analysis [Hecht 77] is a useful static tool, with applications ranging far beyond verification. Unreachable code, unused parameters and variables, and variables that are used before any value is assigned to them are all identified by flow analysis. (In addition, it is useful as a basis for program optimizations such as common-subexpression elimination.)

Another useful static technique is symbolic evaluation of program text [Clarke 81]. Symbolic execution can test programs on classes of data values, instead of just single values. Systems have been designed that can just test programs [Boyer 75, Howden 78]; others can also generate test data automatically [Clarke 76].

Dynamic verification techniques implant *sensing* statements into source programs, to determine their behaviour during execution. This method is called program instrumentation [Fairley 78, Huang 78]. An enormous amount of raw data is typically accumulated about the execution of a program, from which it is possible to verify that the program execution was correct (or otherwise!). Arthur and Ramanathan [Arthur 81] comment that static techniques such as flow analysis and symbolic execution complement dynamic program instrumentation, and can be used to reduce the amount of dynamic tracing required. They propose a system that includes both static and dynamic analysis, in an attempt to combine the advantages of each.

In other work, the Programmer's Apprentice [Rich 78, Waters 79, Waters 82], under development at MIT, attempts to deduce the logical structure of programs
and to use its understanding of a program to help the programmer develop it correctly. The deduced logical structure, or "plan", is monitored during the development of the program, to ensure that the original plan is adhered to. The deduction of logical structure is built up from the recognition of small parts of the program which can be "understood".

2.5.2. Measuring program quality

The last decade has seen the proposal of many measures designed to evaluate some aspects of program quality. This is an active research area, with many experiments performed to develop new measures and to compare existing ones [BakerA80, Henry81a, Basili83].

One of the most influential and most studied proposals is that of "software science" [Halstead77]. Lexical counts of the number of operands and operators used in programs are the basis from which measures such as language level and software programming effort are computed. Although some of the proposed metrics have performed poorly (e.g. language level [Johnston80]), on the whole the software science metrics have proven remarkably effective.

The other measure to have received extensive consideration is the cyclomatic complexity of a program [McCabe76, Myers77b]. This measure is based on the number of branch points in a program, and indicates the number of control paths in the program. Several studies have compared its effectiveness with the effort measure of software science [Henry81b, Basili83], concluding that neither performs convincingly better than the other.

Other measures proposed have been based on the information theoretic notion of entropy [Chanon74]; nesting of loops and alternative statements [Tamine83]; and information flow [Henry81b]. The latter attempts to measure the structure of large-scale systems, using "structured design" notions [Yourdon79] to assess procedure complexity, module complexity, and module strength.
2.5.3. Other applications

Other topics in the area of automatic program analysis include prediction of program performance and execution time (discussed in section 2.4); extraction of control structure from program text, and restructuring programs [Baker 77]; and designing optimal architectures on which to execute programs [Rauscher 78, Flynn 80].
Chapter 3
Model of Host Architecture

A "context-sensitive" computer architecture has been defined to be one in which the virtual architecture changes dynamically, to take advantage of knowledge of the current needs of the executing program. Such an architecture is presented in this chapter. The physical characteristics that are required of the host machine, on which the virtual architecture is implemented, are described. The representation of the virtual architecture on the host is explained.

This architecture is based largely on that considered by Flynn and Hoevel in their research on directly executed languages [Flynn 74, Flynn 80, Flynn 83, Hoevel 74]. Most of the material presented in sections 3.1.1, 3.2, 3.3.1 is drawn directly from [Flynn 80].

3.1. Canonical Measures of Interpretation

3.1.1. Definitions

Consider a source program written in a high-level language. That program is assumed to be a good representation of the original problem, and is optimized to the degree desired. (This optimization occurs only at the source level.)

The source program is translated into some intermediate form, called the image program. This image program may be interpreted directly by an image machine, which is represented as a virtual architecture on an underlying host machine. An "ideal" image machine is sought: one for which the translation from the source
program to a representation of the image program is simple, and for which the interpretation of the image program on the host machine is also simple.

In [Flynn 80] a number of measures of source program behaviour are defined. These *canonical measures of interpretation* give an indication of how ideal an image machine is on which to execute a given program. It is assumed that these measures are independent of a machine's technology i.e. they focus on logical representations and architectures. The measures relate to the space required to represent the image program, and the time taken to interpret that representation during execution.

The measures are Correspondence, Size, Activity, Stability, and Distance. The first two determine static program size; the remaining three affect the time taken to interpret the program.

1. **Correspondence**: ideally there should be exactly one image instruction for each semantic action (ADD, MULTIPLY, etc.) in the source program, and the actions should appear in the same order in each form of the program. Temporary storage locations should not be named, and there should be no need to name any operand more than once in any instruction even if it is repeated as an argument. Good correspondence implies a simple transparent translation from source code to image code, and that nothing unnecessary appears in the generated code.

2. **Size**: the form of the image program interpreted by the image machine is a binary-encoded representation of the instructions of the image program. Ideally the size measure requires that coding be as concise as possible. If a sequence of instructions that includes N distinct operands is encoded, the operands can be uniquely identified using \( \lceil \log_2 N \rceil \) bits. Similarly, if the sequence of instructions contains M distinct operations then the operation field (opcode) of each instruction can be encoded using \( \lceil \log_2 M \rceil \) bits. Identification of labels (used as branch destinations) within the sequence of instructions is done in the same way.
The sequence of instructions is encoded within the context of an environment. This environment encompasses all of the operations, operands, and labels used within the sequence of instructions. Decoding and interpreting the instructions requires the same environment to be recreated, so a coded representation of each environment must be part of the representation of the program that is interpreted by the image machine.

3. **Activity**: Activity measures the dynamic number of objects interpreted. Ideally the total number of instructions interpreted should be equal to the number of semantic actions specified dynamically in the source program. The total number of data references required is ideally equal to the number of dynamic operands encountered.

Activity is essentially a dynamic measure that corresponds to the static correspondence measure. If there is good correspondence between the source program and the image program, the activity measure will perforce be good.

4. **Stability**: Stability measures the dynamic number of disruptions to normal linear interpretation of instructions. Each branch instruction, change of environment, or interpreted operand definition (e.g. array element computation, or pointer access) reduces stability. Stability may be thought of as a measure of the extent to which pipelining may improve execution time: a program that scores well on the stability measure will achieve close to the maximum possible speedup if instruction decoding is pipelined.

5. **Distance**: this measures the *initialization* required in a program. Distance counts the number of unique environments, unique interpreted operand definitions, and unique branches encountered dynamically. It represents the distance the program has traversed i.e. the number of
localities it has visited. As distance measures the limit on stability, a program's score on distance should be linked to its score on stability.

3.1.2. Use of the canonical measures

The hardware and firmware of the host machine determine the features that can be supported in an intermediate language. The nature of the intermediate language in turn determines the correspondence that will be observed between source programs and their respective image programs.

In most systems the intermediate language is simply the assembly language for the underlying host machine, making the interpretation of the image program a trivial task. The correspondence between the source program and the image program is generally very poor: this is the well-known "semantic gap" [Myers 82] between the semantic actions of the source language and the primitive actions of the host computer.

Ideally the instruction set of the host machine would contain one instruction for every semantic action in the source language, including all functions such as SQRT, SIN and EOF. Such an instruction set would obviously bias the machine towards a particular source language, or perhaps a class of related source languages. If the instruction set is defined in microcode, then the contents of the control store could perhaps be changed to provide different instruction sets for use with different source languages. An ideal image language could then always be provided, giving optimal correspondence and activity measures as source programs are processed.

The adoption of a particular host machine and image machine architecture, and a particular compiler from a given source language to the (image) intermediate language, fixes the correspondence (and hence activity) that will be measured for programs written in that source language for use on the chosen system.

Two of the properties measured by stability are fixed for any given program:
the number of objects requiring interpreted definition, and the number of branch
instructions. The number of environments is open for manipulation. The sizes of
instruction encodings alter as the environments are manipulated.

Two new notions of size are introduced. These notions represent a distillation
of the five canonical measures:

1. **Total Static Size**: the total number of bits required for the static
   representation of the program. It includes all instructions, and all
   overheads for defining data and for setting up environments.

2. **Total Dynamic Size**: the total number of bits decoded at run-time
during interpretation of the image program. It includes all instructions
as they are encountered dynamically, and all overheads for defining data
and for changing environments dynamically.

Total static size is a measure of the space required to represent the image
program. It can be computed exactly at compile time. It is a function of:

- the number of static instructions generated in the image program
  (reflected in the correspondence measure)

- the number of components in each instruction (correspondence)

- the size of the encoding of each component in each instruction (size)

- the static number of environments encoded and their respective sizes (as
  explained in section 3.3.2, the size of the coded representation of an
  environment is proportional to the number of objects included in the
  environment).

Total dynamic size is a measure of the work needed to decode the image
program during execution. If the total dynamic size of a program can be altered, the
decoding time for the program should be altered. The time change may not be
directly proportional to the change in total dynamic size, because there is more to decoding an instruction than just reading bits, but one should expect to observe that a change in total dynamic size leads to a change in program execution time. Total dynamic size can only be estimated at compile time, since the number of times each instruction will be executed at run time is not always known. Total dynamic size is a function of:

- the dynamic number of instructions executed (activity)
- the number of components in each instruction (activity)
- the sizes of the encodings of the instructions (size)
- the dynamic number of environments encountered and their respective sizes (stability).

For a given program, translated by a given compiler to run on a given machine, most of the canonical measures (namely correspondence, activity, and most of what is measured by stability and distance) are determined. Size may still be manipulated, as may the environment component of stability, in an attempt to produce an encoding with optimal total static and dynamic sizes. One of the main objectives of this thesis is to investigate how this manipulation may be performed.

3.2. Architectural features of the image machine and host machine

The desire to develop a system that performs as well as possible on the canonical measures imposes a number of demands on the features and capabilities of the image machine and the underlying host machine.

It has already been noted that a changeable control store is desirable in the host, to contain the microcoded instruction set(s).
If the image machine includes an expression-evaluation stack, there is no need to name any temporary storage locations. The stack can be used implicitly to hold intermediate values during the evaluation of an expression. This requires that the host must provide efficient mechanisms to support the operation of a stack.

A four-bit format code [Flynn 80] is added to each image instruction. This indicates the number of distinct named arguments in the instruction, the order in which the operands are to be used, the sources, and the destination. Since the format code indicates when any argument is repeated, naming any operand more than once is avoided. When used in conjunction with an expression-evaluation stack, this ensures that nothing unnecessary appears in the instructions in the image program.

It is desirable to encode operations, operands and labels in fields of minimal width. To do this the image machine must contain tables, which relate the minimal-width binary values to the operations, etc. that they actually represent. (This is illustrated in an example in section 3.3.1, where the operation of this system is demonstrated). Registers must be provided to tell how many bits are used for each opcode, operand and label, and where each of the tables is located in memory. The tables should be held in a high-speed memory such as a cache, and indexed addressing must be efficient to enable rapid access to the table entries. Addressing to the level of individual bits must be possible.

The entries in the opcode table can be pointers into control store to the start of the microcode routines that implement the operations. The label table entries will be the bit addresses of the branch destinations. The operand table itself can be used as the memory for simple constants and variables. The values of these simple variables need to be copied to main memory only during environment changes. For more complex data structures, the operand table entries can be descriptors. This implies the use of tagged data, which must be supported by the host machine. Generic instructions may then be used, bringing several advantages [Myers 82].
In summary, the requirements of the underlying host machine are:

- registers

- a high-speed memory (presumably small and part of a two-level memory system)

- rapid indexed addressing

- an efficient stack mechanism

- the ability to address memory to the level of individual bits

- tagged data

- a control store containing the microcoded instruction set.

3.3. Representation of the image architecture on the host machine

3.3.1. Representing instructions

The operation of this architecture will be demonstrated by tracing the processing of a simple example. In particular, the configuration of the image machine will be shown, and the procedure for interpreting image instructions within that configuration will be explained.

A simple Pascal program is listed in Figure 3-1. It adds a series of positive integers, terminated by a non-positive value, and prints out their sum. Figure 3-2 lists an equivalent image program, expressed in an *ideal* intermediate language.

The details of this intermediate language are given in Appendix A. It was developed by the author as a good language into which to translate Pascal programs. Each image instruction (with one exception: see the appendix) is a quadruple of the
form (Operation, Operand-1, Operand-2, Destination). For operations that do not require one or more of the operands and/or do not produce a result, the appropriate fields are null. In branch instructions the Destination field refers to the number of the quadruple to which control should be transferred; otherwise it refers to the location where the result of the operation should be placed. "T" refers to the top of the expression-evaluation stack.

Note that exactly one image instruction is generated for each semantic action in the source program. A stack is used, so there is no need to name a temporary variable (quadruples 3 and 4). In quadruple 5, "sum" is used as both an operand and the destination: when this quadruple is encoded the format field will indicate that "sum" is to be used twice, and it will only need to be named once in the encoded quadruple. This image program has optimal correspondence.

Figure 3-1: Example Source Program

Program Addpositives (input, output);
var sum, val : integer;
begin
  sum := 0;
  read (input, val);
  while val > 0 do
    begin
      sum := sum + val;
      read (input, val)
    end;
  writeln (output, "sum is ", sum:6)
end.

The four-bit format provides sixteen different format codes. For binary operations there are thirty valid combinations of named operands and stack locations that can represent instruction formats. This can be reduced to twelve distinct formats by eliminating some permutations and by avoiding sub-optimal formats [Flynn 74]. The same twelve formats can be used to implement operations of one or zero operands. The interpreter recognizes these operations by the opcode.
The four remaining formats (bringing the total to sixteen) can be used to mark instructions that use more than three arguments, or that have a label in the Destination field. Three of them are used here for different formats of branching instructions. The final (rarely used) format is used as an "escape" to a more complicated decoding routine, for either a rare form of branching instruction, a rare form of input instruction, or for the sole operation that uses four arguments.

The sixteen format codes and their meanings are listed in Table 3-1. Several of them are overloaded, and have different meanings in conjunction with different operators, although the number or different named operands implied by each format code is always the same (F15 is an exception, but its use should be rare). If the complexity of instruction decoding would be sufficiently reduced by removing the overloading to warrant the extra cost, five format bits could be used instead of four. Each format can then have a unique meaning.

The virtual architecture of the image machine is defined by three tables: one each for operations, operands and labels. Associated with each table are three registers: an "environment pointer", a "width register", and a "window pointer".

The environment pointer points to the beginning of the table in memory.

Each table is accessed through a window. The size of the window is determined
Figure 3-3: Image architecture for example program

<table>
<thead>
<tr>
<th>Opcode Environment Pointer</th>
<th>Opcode Window Pointer</th>
<th>Opcode Width Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 assign</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>001 read</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>010 &gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 jump-if-false</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>100 add</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>101 goto</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110 write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 halt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode Environment Pointer</th>
<th>Opcode Window Pointer</th>
<th>Opcode Width Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>0001 sum</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>0010 input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 val</td>
<td>&quot;sum is &quot;</td>
<td>*</td>
</tr>
<tr>
<td>0100 7</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>0110 output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 6</td>
<td>eol</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Label Environment Pointer</th>
<th>Label Window Pointer</th>
<th>Label Width Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 8</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>1 3</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Table 3-1: Format codes and their meanings

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>A op B =&gt; C</td>
</tr>
<tr>
<td>F1</td>
<td>A op B =&gt; B</td>
</tr>
<tr>
<td>F2</td>
<td>A op B =&gt; A</td>
</tr>
<tr>
<td>F3</td>
<td>A op B =&gt; T</td>
</tr>
<tr>
<td>F4</td>
<td>A op T =&gt; B</td>
</tr>
<tr>
<td></td>
<td>op A =&gt; A</td>
</tr>
<tr>
<td>F5</td>
<td>T op A =&gt; B</td>
</tr>
<tr>
<td>F6</td>
<td>T op A =&gt; A</td>
</tr>
<tr>
<td>F7</td>
<td>A op T =&gt; A</td>
</tr>
<tr>
<td></td>
<td>op A =&gt; A</td>
</tr>
<tr>
<td>F8</td>
<td>A op T =&gt; T</td>
</tr>
<tr>
<td></td>
<td>op A =&gt; T</td>
</tr>
<tr>
<td>F9</td>
<td>T op A =&gt; T</td>
</tr>
<tr>
<td>F10</td>
<td>T op T' =&gt; A</td>
</tr>
<tr>
<td></td>
<td>op T =&gt; A</td>
</tr>
<tr>
<td></td>
<td>op =&gt; A</td>
</tr>
<tr>
<td>F11</td>
<td>T op T' =&gt; T</td>
</tr>
<tr>
<td></td>
<td>op T =&gt; T</td>
</tr>
<tr>
<td></td>
<td>op =&gt; T</td>
</tr>
<tr>
<td>F12</td>
<td>conditional branch to B, depending on A</td>
</tr>
<tr>
<td>F13</td>
<td>conditional branch to T, depending on A</td>
</tr>
<tr>
<td>F14</td>
<td>conditional branch to B, depending on T</td>
</tr>
<tr>
<td></td>
<td>unconditional branch to B</td>
</tr>
<tr>
<td>F15</td>
<td>conditional branch to T, depending on T'</td>
</tr>
<tr>
<td></td>
<td>unconditional branch to T</td>
</tr>
<tr>
<td></td>
<td>higher-order operation</td>
</tr>
<tr>
<td>Never generated</td>
<td>A op A =&gt; B</td>
</tr>
<tr>
<td></td>
<td>A op A =&gt; A</td>
</tr>
<tr>
<td></td>
<td>A op A =&gt; T</td>
</tr>
<tr>
<td></td>
<td>T op T =&gt; A</td>
</tr>
<tr>
<td></td>
<td>T op T =&gt; T</td>
</tr>
</tbody>
</table>

T refers to the location on top of the stack
T' refers to the location below the top of the stack
A, B, C refer to named operands
by the *width register*, whose value is \[\log_2(\text{window size})\]. Its placement over the table is determined by the *window pointer*, which points to the first table entry that falls within the window. Frequently the window will cover the whole table, but sometimes only a portion of the table is of interest.

Figure 3-3 illustrates the tables and registers that make up the image machine for the example program. In this example the windows cover the entire tables.

There are eight distinct operations in this image program: three bits are required to identify each operation, and *3* is stored in the opcode width register. This indicates that three bits are used for the opcode field of instructions coded within this environment, and that a window of \(2^3\) locations is in use. All of the window locations are used in this environment.

Since there are nine distinct operands in the image program four bits are needed to identify each operand, so *4* is stored in the operand width register. The value stored in the label width register is *1*, since one bit will distinguish between the two different labels.
The encoding of the instructions of the image program, targeted to this particular configuration of the image machine, is presented in Figure 3-4.

When the program is executed, each instruction will be interpreted as follows. Initially, four bits are read: this is the format. The opcode width register is then consulted, indicating the number of bits to be read for the opcode (in the example it is 3). That number of bits is read, and the value is used as an offset from the opcode window pointer to select an entry in the opcode environment table (for instance, in the example the fifth image instruction has the opcode field value *100*: this offset of 4 from the opcode window pointer indicates that it is an *add* instruction). Once the format and the operation are known, the interpreter knows how many arguments are to be named, which arguments (if any) refer implicitly to the stack, which are operands, which is the destination, and whether the destination is an operand or a label. The operand width register tells how many bits are to be read for each named operand, and the values are used as offsets from the operand window pointer to determine the actual named operands; labels are handled similarly. Note that some named operands refer to literal values, while others refer to variables from which values must be read. These cases are distinguished by the tag bits associated with each data object.

The procedure differs slightly in the rare case that the format code is F15. The opcode is read as before, identifying either a branch instruction (in which case no operands or labels need to be read - see Table 3-1), a *read* operation (where again no operands need to be read, as the top two stack entries define the source and destination for the data), or a *write* operation. In the latter case, the first argument will be a real value; writing a real number requires four arguments. Another four bits are read, to indicate the disposition of the four operands. Any needed operands are then read as normal. Since this operation represents a very small percentage of total operations (certainly well under 1% of dynamic operations, and doubtless also static operations [CookR 82, Knuth 71a]) the extra four format bits are a minimal added cost.
Once all the components of the instruction have been decoded, the operation can be applied to its arguments and any result disposed of as indicated by the format.

The tables, stack, and registers constitute a virtual architecture on which the image program executes. From the point of view of the program, it appears to run on a machine that provides precisely the operations and addressing capability needed. An ideal virtual architecture has been provided to suit the program.

An architecture very similar to this was emulated on the EMMY computer at Stanford University [Flynn 80]. No facilities were available to this author to make similar performance measurements for this system; the best that can be done is to quote the performance figures for Flynn's system and to assume that the performance of this system would be similar.

Flynn found the execution speed of his system to be only slightly below that of conventional systems. The decoding of instructions was slower because it was more complex, but the actual execution of instructions took the same time. Execution time dominated over decoding time, so overall execution speed was little different.

The concise encoding of instructions in Flynn's system meant that conventional systems required about twice as much space to represent programs, once the cost of defining data and environments was accounted for. A similar observation was made in this system.

3.3.2. Representing environments

The image machine described above is termed a "context-sensitive" architecture because the instructions only have meaning within the context of a particular environment. Identical bit strings can represent completely different instructions if the environments differ; the interpretation of an encoded instruction depends on the context within which it is decoded.
The procedure for decoding instructions has been demonstrated with a simple example. Given a particular configuration of the environment tables and registers, the encoding and subsequent decoding of instructions is a simple matter; the mechanism by which the appropriate values are placed in the tables and registers is now discussed.

### 3.3.2.1. Defining the entries in environment tables: basic approach

The binary version of the image program (i.e. the form of the program that is actually executed) must contain some information to enable correct entries to be placed in the tables and registers. The approach used is enumeration, in an established order, of the operations, operands, and labels included in the environment.

When the system recognizes that a new environment is to be set up, it invokes a routine that sets up the operation environment table and its three associated registers. When that has been completed, the same approach is used to set up the operand environment table and its associated registers. Finally, labels are handled in the same way. The new environment is then complete; interpretation of instructions continues in the context of this new environment.

The first action in setting up a table is the reading of two control bits. These indicate whether the most general case of environment set-up is to be performed or if one of three simpler forms can be used. The general form is described here - other forms are described in section 3.4.

Initially a fixed number $m$ of bits is read. This is treated as a binary number $n$ in the range $0 \leq n \leq 2^m-1$, whose value is the number of different operations to be placed in the table. The width value $wr = \lfloor \log_2 n \rfloor$ is computed and placed in the opcode width register. The environment pointer and window pointer are both set to point to the starting memory location for the new operation environment table.
The \( n \) opcodes are then enumerated. \( m \) bits are read for each of them; the value represented by those bits indicates the actual opcode. The \( i^{th} \) lot of \( m \) bits defines the \( i^{th} \) opcode, which is placed in the table at an offset of \((i-1)\) locations from the environment pointer.

The two control bits for the operand table follow immediately after the last lot of \( m \) bits that identified an opcode. Similarly, labels follow after operands and instructions resume directly after labels.

Apart from an overhead of \((2+m)\) bits, the number of bits required to specify the entries for a table is directly proportional to the number of entries in the table. Thus the cost of specifying an environment is proportional to the number of objects in the environment, plus an overhead that is usually very small by comparison.

\( m \) can represent a different number of bits for each of operations, operands, and labels.

### 3.3.2.2. Defining the entries in environment tables: implementation

It is desirable for the bit cost of specifying environments to be minimized. Environments and instructions each contribute to total static size and total dynamic size. Concise instruction encodings are of little use if the cost of defining environments is very high.

To provide adequate addressing and branching capability on a real machine, \( m \) would need to be quite large (about 24 bits at least). To avoid specifying this many bits every time, another level of indirection is introduced, and the system is restricted to cope with programs that use no more than 128 different operations, 512 different operands and 256 different labels.

At the very beginning of the program the enumeration process is used, with \( m = 24 \) bits, to construct global tables that identify all operations, operands, and labels used anywhere within the program. The global tables are of known size and
can be stored at predetermined locations. Thereafter, whenever opcodes are enumerated for an environment only \( m = 7 \) bits are needed, using the value as an index into the global opcode table to identify the opcode. Operands are enumerated similarly using \( m = 9 \) bits, and labels are enumerated using \( m = 8 \) bits. In practice the global opcode table could be "hardwired" in the control store; global operand and label tables need to be set up for each program. This scheme reduces the cost of environment specifications by a factor of about three, and the added cost of the initial enumeration is recovered for each operation, operand, or label in only its second definition within an environment in the program.

The limit of 128 operations is no restriction on the system because the intermediate language contains less than that number of different operations. The limits on operands and labels may prevent the system from handling some large programs. If necessary they can be extended by powers of two without much extra bit cost. The tables should, however, be kept to a size that does not usually waste a lot of space. The current limits strike a reasonable balance.

### 3.4. Run-time system operation

Decoding and executing instructions at run-time was explained in section 3.3.1. The remaining details of run-time system operation are concerned with providing the appropriate environments. A run-time stack is used for this purpose.

The "environment" is the particular state of the image machine i.e. the particular configuration of entries in the three windows and nine registers. Setting up an environment involves creating the requisite new state, either by redefining all values from scratch or by altering the values in one or more of the registers.

A new environment needs to be set up:

- at the beginning of the main program body, to provide the initial environment within which to begin interpretation.
• on entry to a new procedure or function. There is a new scope of
definition for variables, and there are typically some new variables to
incorporate into the environment (for some languages, such as Algol, any
block can define a scope for variables: entry to a block, rather than just
a program unit, may require a new environment).

• whenever it is deemed to be advantageous during the execution of a
program unit. A special "change-environment" instruction
(Change-env) in the program will signal the change.

A run-time stack, which holds the sequence of environments encountered during
execution, is maintained. Initially the stack is empty, but when the program starts
executing an entry representing the initial environment for the main program body
is placed on the stack. Whenever a change to a new environment occurs, the new
environment is pushed onto the stack. The top stack entry is always the current
operating environment. When an environment is no longer required (eg. on return
from a program unit after its body has been executed) it is popped from the stack.
The newly-exposed top stack entry then becomes the current operating
environment.

When execution reverts to a previous environment, the stack provides a record
of the entries that must be in the tables and registers. No bits need to be specified in
the program. It is up to the system implementation to provide an efficient scheme
for restoring this previous state.

Conceptually, each environment definition requires the identification of the
entries to go in each table and register, and each environment is independent of all
others. If successive environments are related (i.e. they share some common entries),
the change of environment can sometimes be effected by simply changing the sizes
and/or locations of the windows. On such occasions a lot of work is avoided - and a
lot of bits in the program are saved - in changing the environment.
The parameters that enable the new environment to be set up appear at the beginning of the code for each program unit, and immediately after each Change-environment instruction. For each of operations, operands, and labels the change to the new configuration is achieved as follows:

1. Read two control bits.

2. Use their value to select one of four options:

   "00"  (perform the full enumeration) An entirely new table is defined, and both the environment pointer and the window pointer are set to the start of the table. The value for the width register is computed and placed in the register.

   "01"  (change the window, but do not change the table entries) Read m bits that tell at what offset from the environment pointer to set the window pointer. Read \[\log_2 m\] bits to set the width register.

   "10"  (change the window size, but not its location) Read \[\log_2 m\] bits to set the width register.

   "11"  (no change at all) No further action needed.

Since each of operations, operands and labels is dealt with independently, it may be possible to change one of them by just altering its window even while another requires full enumeration. The "no change" option allows selective change to parts of the environment, costing only two bits each to indicate that other parts of the environment are to be left unchanged.

In general, whenever a new program unit is entered a full enumeration of the new environment is needed. Often the table entries can be arranged in such a way that any changes of environment within the program unit can be effected by only
changing the windows - this is the subject of chapter 7. Such environment changes are cheap in both time and space.

Establishment of new environments generally requires the initialization of some variable values in the operand tables, taking the values from further down in the stack. Reversion to previous environments generally involves copying back some updated variable values.

Changing an environment table may cost nothing (in bits) when the stack is used to revert to a previous environment; a small constant (in bits and time) when only a window change is needed; or a cost (in bits and time) proportional to the number of entries in the table, when a full table enumeration is needed.

3.5. Binary representation of the image program

The binary form of an image program is illustrated in Figure 3-5. The encoded program units appear in sequence, as shown in Figure 3-5(a). The sequence is that in which the bodies of the program units are parsed by a one-pass compiler (i.e. in their textual order of appearance in the source program). This sequence is adopted for convenience - it is not mandatory.

Each encoded program unit begins with some bits that define the initial environment for the program unit, followed by the instructions (see Figure 3-5(b)). All of the instructions might be encoded within that initial environment, or there might be an internal change of environment somewhere within the program unit. Figure 3-5(c) shows the form of an internal environment change. There can be any number of internal environment changes, and they can be nested to any depth.
In this chapter a computer architecture is defined. It is represented as a virtual architecture on an underlying host machine. The characteristics required of the host are discussed, as is the method of representing the virtual architecture on the host.

Performance monitoring of a similar architecture suggests that although this architecture runs at very nearly the same speed as conventional systems, only about half as much space is required to represent programs.
One of the most important notions developed is that of the "environment" within which instructions are encoded and decoded. The ability to change environments during program execution leads to the term "context-sensitive architecture"; just when and how to change environments is one of the main problems addressed in this thesis.

Based on five "canonical measures of interpretation" defined in [Flynn 80], two new measures of a program's size on this architecture are defined:

- "total static size" (a measure of the space needed to represent the program)
- "total dynamic size" (a measure of the time needed to execute the program).

This architecture is used hereafter as a "testbed", with the two measures of program size as performance yardsticks, in the investigation of some types of program analysis that can be used to help provide an optimal image machine for each program.
Chapter 4

Context changes during program execution

In chapter 3 the encoding and decoding of image instructions within the context of a given environment was described, as was the procedure for changing from one environment to another. The problem still to be resolved is when changes of context should be made.

The environment changes are built into the encoded representation of the image program. The problem is to determine how the image program should be segmented into blocks of instructions for encoding within their own local environments.

Large blocks of instructions tend to include many different operations, operands, and labels. Consequently the environment tables and the instruction encodings are large, but there are few changes of context. Small blocks of instructions lead to small environment tables, concise instruction encodings, and frequent changes of context.

Each change of context introduces overheads for the new environment definition, but generally allows smaller instruction encodings since the environment tables are made smaller. A balance in this trade-off is sought.
4.1. Different levels for grouping statements

4.1.1. Single source statement

At one extreme, a local environment could be defined for every statement in the source program. Each image instruction could then be given the most concise encoding possible. This would, however, make the canonical stability and distance measures very poor, and the total static and dynamic sizes very bad. The savings due to concise instruction encodings would be far outweighed by the overheads in continually changing environments. Individual source statements are generally too small to be units for encoding in their own local environments.

4.1.2. Entire program

At the other extreme, one environment might be used for the entire program. Global tables are already being used to help make environment definitions cheap; those tables could be used directly, with no need ever to define new environments. Width registers and environment pointers (but not window pointers) would have to be used instead of forcing the tables to have fixed sizes and locations. If recursive procedure calls are allowed, care would need to be taken to save copies of the operand table in a dynamic stack.

The main problem with this extreme environment is that the tables always contain all entries ever needed, whether or not they are needed for the current task. Instruction encoding lengths are therefore as great as could be. Ideally the environment tables should function analogously to a hardware cache: they should contain only the required "working sets" of operations, operands, and labels. This gives the most concise instruction encodings feasible.

In general, a complete program is too large a unit for encoding in a single environment.
4.1.3. Individual program unit

For two reasons it seems natural to change the environment whenever a new procedure is entered:

- as noted in section 3.4, entry into a scope of variable definitions [Johnson 71] in which a new variable is declared should imply entry into a new environment. Objects with the same scope of definition should be encompassed by an environment with that scope [Flynn 80]. This implies that each program unit will have its own environment, in languages like Pascal (or each block, in languages like Algol).

- Labels in different program units certainly refer to different locations; different program units can be expected to do different things and so favour different sets of operations and operands. Changing the environment on procedure entry helps to provide only the required working sets of operations, operands, and labels.

The addition of environment definition to normal subroutine linkage would be straightforward, and would provide naturally a dynamic stack of environments during program execution. Context information, including values of entries in the operand table, would be part of the normal run time activation record [Pratt 75] for a procedure.

4.1.4. Parts of program units

It is quite possible that the environment for the whole procedure may be too broad at times during the execution of a procedure. As execution progresses through the procedure the favoured sets of operands and operations may change; changing the context to reflect this could be advantageous.

Such context changes aid the conception of the environment tables as caches that hold only the working sets of required objects. More concretely, context
changes within a procedure can reduce the total static size and/or total dynamic size of the procedure. These will be the criteria used to decide when internal changes of context are appropriate.

This is one end of the spectrum of computer architectures. At the other extreme, conventional systems provide a single fixed architecture, on which to execute any program written in any language. The Burroughs B1700 [Burroughs 72, Wilner 72a] is more flexible than that, providing different architectures for different programming languages. The next level of flexibility is to construct a different architecture for each program; then for each program unit, or scope of definition (as done by Flynn and Hoevel [Flynn 83]); and now for each part of a program unit. The extreme case is the provision of a different architecture for each individual statement; as explained in section 4.1.1, this is too inefficient to contemplate.

The circumstances under which a change of environment is appropriate within a procedure may be classified under two headings: *changing locality* and *local optimization*.

4.2. Environment change due to changing locality

While a program works on a particular task it tends to favour a particular set of operands and operations, which varies from task to task. These favoured sets are exactly what the environment tables should model. A change in the favoured sets of operands and operations during program execution will be termed a "change of locality". (This is analogous to the idea of intrinsic program locality in memory reference models [Denning 80].)

Figure 4-1 illustrates the sort of encoding that is obtained when the environment is changed to reflect a change of locality. Consider a procedure that is composed of two distinct phases, each with its own favoured set of operations, operands, and labels. If the procedure is encoded as a single unit, its environment
Tables must include all entries that are needed anywhere in the procedure - thus they hold the union of the favoured sets from each of the two parts of the procedure. The number of bits required to encode each operand (for example) is determined by the total number of different operands used within the procedure. If the procedure is split into two parts, the number of bits required to encode each operand in each part is determined only by the number of different operands in that part. This is often bounded by a smaller power of two, so less bits are required to encode each operand; this can happen in either or both parts of the procedure. The same applies to labels and to opcodes. By splitting the procedure into parts for encoding, substantially smaller instruction encodings can be achieved.

**Figure 4-1:** Procedure encoding to reflect changing locality

<table>
<thead>
<tr>
<th>Total environment</th>
<th>Part A environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Part A code</td>
</tr>
<tr>
<td>(A)</td>
<td>Part B environment</td>
</tr>
<tr>
<td></td>
<td>Part B code</td>
</tr>
</tbody>
</table>

The smaller instruction encodings are not achieved without some cost. There are now two places at which environment tables must be defined. If full table specification is needed for each environment definition, then an extra bit cost has been introduced which is proportional to the amount of overlap between the two environments: any entries used in both parts need to be specified in each environment, where before they were only specified once. There are also the added overheads in the second environment definition. Even when the environment
change is achieved by just changing windows and there is no extra cost from specifying table entries, the overhead costs are still incurred.

Two extra instructions must be inserted into the code when the environment is changed: a "Change-environment" instruction must be inserted to signal the change, and a "Revert-environment" instruction must be added to indicate when to revert from the new environment back to the original one. These operators must appear in the tables, and so must be taken into account when computing the sizes of the operator encodings.

As procedures are analysed, notable changes of locality (i.e. sudden changes in the favoured sets of opcodes and/or operands and/or labels) will be sought. If procedures are split at these points for encoding, substantial reductions in instruction sizes will often be possible, with little extra work in specifying environments. Such changes usually produce reductions in both the static size and the dynamic size of the procedures.

Labels are usually inserted by the compiler that translates the source program into the image program, and most of them are only referred to in one or two branch instructions. Although they convey information on possible paths for control flow, their use tends to be so restricted that labels are of no use when points of changing locality are sought. Operations and operands provide almost all of the information.

It has been observed that transitions between tasks in the code of a procedure tend to be accompanied by changes of locality. If the converse is assumed, that a notable change of locality implies a transition between tasks, it may be possible to deduce roughly the logical structure of a procedure by simply looking at the patterns of usage of operands and opcodes and labels within its code. If the change of locality is very distinct, it may indicate that the procedure could or should have been written as two separate procedures in the first place.
4.3. Environment change for local optimization

Sometimes a sequence of instructions is found within a larger sequence of instructions, which uses a subset of the operands, etc., of the larger sequence. The enclosed sequence may be a good candidate for encoding within its own local environment, despite its needed environment being a subset of that of the enclosing sequence, instead of a (mostly) disjoint set.

The environment tables for the enclosed sequence of instructions contain fewer entries, so it is likely that fewer bits are required to encode operators, operands, and labels. The instructions in the enclosed sequence can usually be encoded using fewer bits each.

If the enclosed sequence is encoded separately, it may be considered a "black box" from the point of view of the enclosing instruction sequence. This means that any opcodes, operands, or labels that appear in the enclosed sequence, but not the enclosing sequence, are no longer needed in the environment for the enclosing instruction sequence. Sometimes the instructions in the enclosing sequence can be encoded using less bits.

Figure 4-2 shows the change of encoding of a procedure in this situation. The environment for Part A is usually little different from that for the whole procedure, because normally there are few things that only appear in Part B. The code for Part A is usually no smaller than when the procedure is encoded as a single unit. The environment for Part B is generally a re-specification of some of the entries for Part A, so it is all "extra" cost. (It is often possible to change the environment by just changing the windows, which makes the "extra" cost small.) The instructions for Part B are generally given much smaller encodings than if they are encoded in the context of the whole procedure.

If it is necessary to respecify tables in order to change the environment, it is generally found that overall there is a small increase in the total static size of the
Figure 4-2: Procedure encoding with local optimization

<table>
<thead>
<tr>
<th>Total environment</th>
<th>Part A environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code (A)</td>
<td>Part A code</td>
</tr>
<tr>
<td></td>
<td>Part B code</td>
</tr>
<tr>
<td></td>
<td>Part A code</td>
</tr>
</tbody>
</table>

program unit. The extra cost of defining the environment for Part B exceeds the static size reduction in the instructions of Part B, giving an overall size increase. If it is only necessary to change windows, the total static size normally decreases slightly.

If Part B is (or contains) a loop, a dynamic size reduction can often be obtained. The environment cost is incurred once, on entry to the new environment, but the saving due to smaller instruction encodings accrues with every iteration of the loop.

In principle the code in Part B could be any sort of instructions, and not necessarily a loop. However, when environment changes of this form are found it is usually the case that Part B is a loop with enough iterations to overcome the dynamic cost of the change of environment and to produce a substantial dynamic gain. In a sense this provides a way of optimizing the execution of a loop.
4.4. Code segmentation strategy

In the introduction to this chapter it was explained that the major problem still to be solved is how to segment the code of a program into blocks of instructions to be encoded in separate environments. Clearly it is necessary to develop an algorithm to solve this problem.

The programmer's division of the program into named subprograms is used as the first stage of the solution. Every program unit is then treated individually in exactly the same way:

1. A new environment is defined on entry to every program unit, to set up the initial environment for interpretation of its instructions. This requires a complete static specification of the table entries for the new environment. (With recursive procedure calls there will be no need to actually respecify the table entries at run time, although a new dynamic stack entry must be created and the old operand values saved).

2. A scan is made over the code of the program unit, seeking points in the code at which an environment change would be worthwhile. The information used includes the patterns of usage of operands, opcodes, and labels in the program unit. The metrics used to decide on the value of a potential environment change are the resulting static and dynamic sizes of the program unit.

3. When changes of environment are found that significantly reduce one or both of the size measures, those changes are made. When no such changes are found, the whole program unit is encoded in a single environment.

The circumstances in which appropriate points for changing the environment are found can always be described as either changing locality or local optimization. Changing locality can be recognised by a relatively small overlap between successive
environments: it could signal a transition between logical tasks in the program unit. Local optimization is characterised by a narrowing of the environment to a subset, to optimize a local piece of code: it may represent a functional submodule within a larger logical task, and it often helps to optimize execution of a loop.

4.5. Summary

The two circumstances in which it may be advantageous to change the environment during the execution of a program unit have been described. The changes may be made to reflect a change of locality within the program unit, or to optimize the execution of part of the program unit (usually a loop).

The following algorithm can be used to decide how a program should be segmented for encoding:

• every program unit is analysed and encoded in turn. Each unit is dealt with completely before analysis commences on the next program unit.

• when each program unit is "analysed", internal points are sought at which to change the environment: these points are either boundaries of a local optimization or points of changing locality.

• if internal changes of environment are found to be advantageous, the program unit is encoded with environment definitions at the identified points, and at entry to the program unit.

• if no appropriate internal changes of environment are found, the only environment definition appears at the start of the program unit and the whole thing is encoded within the one environment.

The procedure for identifying potential points for changing the environment, and deciding which changes (if any) should actually be made, is the subject of the remainder of this thesis.
Chapter 5
Models of dynamic program execution

Static program size and dynamic program size are used as measures for deciding whether a given change of environment would be worthwhile. Static program size can always be determined exactly, but dynamic program size can usually only be estimated.

Dynamic program size is the sum over all instructions and changes of environment of:

\[(\text{size of encoded instruction}) \times (\text{frequency of execution of instruction})\]

and

\[(\text{bit cost of environment change}) \times (\text{frequency of environment change}).\]

The instruction sizes and bit costs of environment changes are known exactly. Precise knowledge of how often each instruction will be executed, and how often each environment change will occur, is usually lacking. Sometimes dynamic trace data may be available from previous executions of the program; if so, it can certainly be used to estimate the frequency of execution of each instruction and environment change (i.e., to predict the dynamic profile of the program that will be observed at run time). In general only static information is available; a model is required that enables a "sufficiently accurate" prediction to be made of the dynamic profile of a program, using only data that is available from a static examination of the program text.

In this chapter the basic principles used to develop such a model are described, and details of the different versions of the model that were investigated are given. The performance of the model when it was applied in practice is described, and its
successes and shortcomings are discussed. Some conclusions are drawn about the
general feasibility of using static information to make predictions about the dynamic
behaviour of a program.

5.1. Basic framework for modelling program profiles

The model must enable a prediction to be made of how often each statement in
a piece of code will be executed (on average) each time that piece of code is invoked.
It is applied in turn to each program unit in the source program, giving a prediction
of its dynamic execution profile. It is a simple matter then to map the profile of the
source program to that of the image program.

The three basic control-structuring components of a program are the sequence,
the alternative, and the iteration. All of these components must be handled by any
model of program execution.

5.1.1. Sequence

This is the simplest control structure. Execution begins with the first statement
in a sequence; control then flows linearly through the statements of the sequence,
with each statement being executed once in turn. The statements are executed in
the order in which they appear in the source text.

The Goto-statement will be excluded from this discussion: it is considered
separately in section 5.1.4.

The Pascal construction that corresponds to a sequence is the compound
statement. Each Pascal statement in the sequence is an assignment statement,
input/output statement, procedure call, alternative statement, or iteration
statement.

The latter two statement types are "structured statements". They include
some control-structuring details that govern the flow of control through internal compound statements. The internal compound statements are themselves sequences.

Sequences are modelled with the following rule:

\[
\text{SEQ} : \quad \text{assume that each statement in a program unit will be executed exactly once each time the program unit is invoked.}
\]

This initial assumption will be modified by other rules, according to the enclosure of each statement within iteration statements and alternative statements. The presence of Goto-statements in a program unit may also affect the predicted execution frequencies of some statements in the program unit.

5.1.2. Alternative

An alternative statement provides a choice of actions to take, depending on some condition. Only one of the different paths is selected each time control reaches the alternative statement.

To model the execution of an alternative statement, the probability \( p_i \) must be estimated that branch \( i \) will be selected for execution when control reaches the alternative statement. If control reaches the alternative statement \( n \) times, the expected number of times branch \( i \) will be executed is \( n \cdot p_i \).

Pascal provides two alternative-statement constructs:

- The *Case-statement* may be used to choose between any number of alternatives;
- the *If-statement* is used to select one of two alternatives. There are two forms of the If-statement: the *If-Then-Else* statement, and the *If-Then* statement. The If-Then statement can be considered a special case of the If-Then-Else statement (where the Else-clause is null). Initially the two forms of If-statement will be modelled separately; later they will be considered together.
The following rules are used for modelling alternative statements:

• for a Case-statement with \( k \) different branches, a general rule (designated \( ^*\text{CASE}* \)) is used to predict the average probability \( p_i \) that branch \( i \) will be selected when the Case-statement as a whole is executed \((1 \leq i \leq k)\).

• for an If-Then statement, a general rule (designated \( ^*\text{IF-1}* \)) is used to predict the average probability \( p_{then} \) that the Then-clause will be executed when the If-statement as a whole is executed.

• for an If-Then-Else statement, a general rule (designated \( ^*\text{IF-2}* \)) is used to predict the average probability \( p_{then} \) that the Then-clause will be selected when the If-statement as a whole is executed. Clearly this estimates \( p_{else} = 1 - p_{then} \).

When an alternative statement is encountered during the static examination of a program, the appropriate rule is applied to estimate the probability of selecting each branch. The basic execution frequency of each statement in each branch (initially predicted by rule SEQ to be 1) is multiplied by the probability of choosing that branch. The statements within each branch may themselves be alternative statements: this same procedure is applied recursively to them.

5.1.3. Iteration

An iteration statement, or loop, consists of a boolean test and a sequence of instructions called the *body* of the loop. Control iterates repeatedly through the body of the loop, stopping when evaluation of the boolean test produces a desired result; execution of the iteration statement as a whole is then finished. The body of the loop is thus executed \( n \) times before the iteration statement is considered to have completed its one execution; the value of \( n \) may differ each time the iteration statement is executed.
If an iteration statement is executed \( k \) times, and on the \( i^{th} \) execution there are \( n_i \) iterations through the loop body, then the average number of iterations through the body of the loop is

\[
 n_{avg} = \frac{N}{k} \quad \text{where} \quad N = \sum_{i=1}^{k} n_i
\]

Some value \( m \), that is expected to be the average number of iterations through its body each time the iteration statement is executed, is predicted for each loop. If a good predictive rule is used, \( m \) should be close to the observed value \( n_{avg} \) that is measured in actual program execution.

In Pascal there are three loop constructs:

- Loops where the number of iterations (\( n \)) is not predictable with certainty:
  - A "Repeat-loop" is used when it is natural to test for satisfaction of a condition after an action has been taken, and to keep performing the action until the desired condition is satisfied.
  - A "While-loop" is used when it is natural to test for satisfaction of a condition before taking an action, and to keep performing the action for as long as the conditions are right to do so.

- A "For-loop" is used when \( n \) is known to be a precise function of the values of some of the constants and variables used in the program. If the function involves only constants then \( n \) is fixed and can be determined at compile time; if not, \( n \) is only predictable with certainty just before the loop is executed at run time.

Iteration statements are modelled as follows:

- for a For-loop with constant bounds, the actual number of iterations that will be made is computed. The expected number of iterations (designated \( m \)) is then assumed to be that computed value.
for a loop of unknown number of iterations, an appropriate general rule is used to estimate the expected number of iterations \( m \). (The appropriate rule is designated \( \text{REPEAT} \), \( \text{WHILE} \), or \( \text{FOR} \), depending on the loop type.)

When an iteration statement is encountered during the static examination of a program, the appropriate rule is applied to estimate the expected average number of iterations through the loop body. The basic execution frequency of each statement in the loop body (initially predicted by SEQ to be 1, but possibly modified by the alternative-statement predictions) is multiplied by the expected average number of iterations. The statements within the loop body may themselves be iteration statements: this same procedure is applied recursively to them.

5.1.4. The Goto-statement

Although the controversy in recent years over the need for the Goto-statement (\cite{Dijkstra68,Knuth74}, etc.) has seen its use diminish, it appears in many programs. In some lower-level languages it is still the major control-structuring feature; it has been proven that some computations can be expressed with Goto-statements that cannot be represented equivalently without them\cite{Knuth71b,Knuth74}; its use may sometimes be desirable for efficiency\cite{Knuth74}; it may be needed to simulate desirable features that are lacking from the source language; or it may just be present because of sloppy programming. The model framework must therefore be able to accommodate its use.

In this research, a Goto-statement may only branch to a label in the same program unit (this is necessary because the environment is changed on entry to each program unit, and a Goto-statement and its target must be encoded within the same environment). If this restriction is removed, the ideas presented in this section must be extended to cover non-local jumps.

If a program unit contains a Goto-statement, it should be subjected to a control
flow analysis [Hecht 77]. (This may be done anyway: flow analysis can be a useful part of the examination of every program unit, as explained later.) The result of the flow analysis is a directed graph representing the possible paths for control flow within the program unit.

It is assumed that each program whose execution profile is to be predicted was written carefully i.e. a reasonable attempt was made to produce a well-structured program, making appropriate use of the control-structuring facilities of the source language. Any Goto-statement is therefore present for a good reason.

Outlined below are some categories into which many Goto-statements can be classified. This is not a definitive list: it simply identifes some circumstances in which Goto-statements are used, how those circumstances may be recognized, and how they can be dealt with. A *catch-all* category will take care of the remainder (no doubt including most Goto-statements in programs for which the assumption of careful writing is not completely valid!)

The recognition of program structure from flow graphs is discussed in [BakerB 77].

5.1.4.1. Basic control structuring

In programs written in languages with limited control-structuring facilities, it will often be found that Goto-statements are used to simulate basic alternative statements or iteration statements. Especially with loops, this generally reflects the lack of a desirable feature in the source language.

When a conditional backward jump is found, with the code from the target statement to the jump forming a sequence of unstructured and structured statements, a simulated Repeat-loop has been identified.

An unconditional backward jump generally represents the end of a simulated While-loop, or a loop that exits from somewhere in the middle [Knuth 74]. In this
case the statements from the target through to the jump must form a sequence, from which there is one conditional Goto-statement leading forwards out of the loop.

Simulated alternative statements are recognizable from a combination of conditional and unconditional forward jumps that direct execution from one point in the program to a later point via one of two or more distinct sequences.

When Goto-statements are found to simulate alternative statements or iteration statements, some rules should be applied to modify the predicted execution frequencies of the enclosed statements. The rules described in sections 5.1.2 and 5.1.3 could be used, or (very similar) rules could be introduced to handle these specific cases.

5.1.4.2. *Premature* loop exit

In many programs a loop is used to provide access to each element of a data structure in turn; normal termination of the loop occurs when every element has been accessed once. Sometimes it is desirable to perform some particular computation on each element, stopping as soon as some condition is satisfied and processing no further elements of the data structure. A Goto-statement is often used then to exit *prematurely* from the loop.

A program that uses a Goto-statement like this can be rewritten to perform an almost equivalent computation without using a Goto-statement, although it may be impossible to attain exact equivalence [Knuth 71b]. However, the programmer may feel that the rewritten program is less clear or less efficient [Knuth 74], and therefore opt to retain the Goto-statement.

These Goto-statements may be recognizable from the flow graphs, as forward branches from within a loop to a point that is reached sequentially if the loop terminates normally.

The presence of such a Goto-statement may affect the predicted number of
iterations to be made through the loop, so it should be taken into account by the
loop prediction rules. It may further affect the predicted frequency of execution of
any statements in the loop body that come after the Goto-statement.

5.1.4.3. Error trapping

Frequently a programmer needs to program a computation that may be aborted
at any of several points along the way. The computation proceeds linearly in stages;
before each stage it is necessary to ensure that the required conditions are satisfied
for the computation to continue (the data to be used next must be within a given
domain; it may be necessary to check that no arithmetic overflow or other errors
occurred in previous stages; etc.). If the conditions are not satisfied, control is
transferred to a separate piece of error-handling code.

Such a computation can be expressed using boolean variables and repeated
nesting of If-Then-Else statements. If there are more than one or two points at
which the computation may be aborted, however, this tends to obscure the
essentially linear nature of the computation.

A different technique is to write the program linearly, inserting Goto-statements
to abort the computation where necessary. In some cases this can produce a more
comprehensible program than is possible without Goto-statements, so this technique
might be preferred. (If the language provides higher-level exception-handling
facilities, they should be used instead.)

These Goto-statements may be recognized from the flow graph as jumps from
several points within a sequence to a common destination.

The chance that each of these jumps may be taken lessens the predicted
execution frequencies of following statements, and may increase the predicted
execution frequency of the common destination. A rule is needed to explain how
these frequency predictions are to be modified: designate it rule "GOTO-1".
5.1.4.4. Miscellaneous

Many Goto-statements can be classified into the three categories described above. Sometimes, however, it may not be possible to recognize any pattern in the flow graph that would permit the classification of a particular Goto-statement. The list of categories could perhaps be extended, but there will always be some Goto-statements that defy easy classification. (This might occur frequently: it has long been appreciated that it is difficult to recover logical intent and algorithm structure from flow charts.)

In these cases a general "catch-all" rule must be applied, to account for the effect of the Goto-statement(s) on the execution frequencies predicted for other statements. Designate it rule "GOTO-2".

5.1.4.5. Skipped code

There is one respect in which an unconditional Goto-statement to any destination can affect the predicted execution frequencies of some other statements. It is a situation that should never arise in practice, but programming errors and changes introduced in program maintenance may occasionally give rise to it. The following rule is then used:

**SKIP**: If there are any statements intervening textually between an unconditional Goto-statement and the next labelled statement, those intervening statements can never be executed: their predicted execution frequencies are set to zero. If the Goto-statement occurs within a branch of an alternative statement, and there is no succeeding labelled statement within that branch, the predictions of zero executions only apply to the rest of the statements within that branch.
5.1.5. Summary

The principles for modelling sequences, alternative statements, and iteration statements have been described. An indication has also been given of how some Goto-statements may influence the predictions. Those ideas are brought together now, and the basic framework of static models for predicting dynamic program profiles is described:

1. Applying the rule SEQ, assign every statement a basic predicted execution frequency of 1.

2. If a Goto-statement occurs within the code, perform a control flow analysis. From the resulting flow graph, classify each Goto-statement and identify the rule to be applied to account for each one. Apply those rules to modify the predicted execution frequencies of all statements in the code: some are unchanged; some may be set to zero by rule SKIP; some may be modified according to rule GOTO-1 or GOTO-2; some are marked as representing alternative statements or iteration statements, to be modified in the next stage of application of this model.

3. Modify the predicted execution frequencies according to the enclosure of each statement within alternative statements and iteration statements:

   a. For each alternative statement in the code, use the appropriate rule (CASE, IF-1, or IF-2) to estimate the probability of selecting each branch. Multiply the predicted execution frequency for each statement in each branch by the probability of selecting that branch. Apply this process recursively to nested alternative statements.

   b. For each iteration statement in the code, use the appropriate rule (REPEAT, WHILE, or FOR) to estimate the average expected number of iterations through the loop body. Multiply the predicted execution frequency of each statement within the loop
body by the predicted average iteration count. Apply this process recursively to nested iteration statements.

Note that steps 3(a) and 3(b) can be combined into a single recursive-descent scan of the code.

This model can be applied to any program, to predict the execution profile of each program unit in that program. If dynamic trace data is available it can be used in preference to this static model, but in general it will be necessary to use the static model.

A general framework for predictive models has been defined. SEQ and SKIP are basic definitions at the foundation of all models; selection of specific heuristic rules for each of CASE, IF-1, IF-2, REPEAT, WHILE, FOR, GOTO-1, and GOTO-2 completes the definition of a particular model for Pascal programs. The accuracy of the predictions made by each individual rule affects the overall accuracy of that particular model. Several heuristic rules were investigated; they are discussed in the remainder of this chapter, and the accuracy attained is reported.

5.2. Individual heuristic rules

Some empirical observations on the dynamic use of If-statements and loops in Pascal programs, gleaned from dynamic traces of 29 programs, are presented in a recent technical report [Lokan 83a]. Figures 5-1 to 5-6 are reproduced here from that report.

5.2.1. Case-statements

Consider a Case-statement that has \( k \) distinct branches, the last of which may be an "otherwise" branch. Branch \( i \) is selected for execution (\( 1 \leq i \leq k \)) if the selector takes one of a set of \( s_i \) different values; the sets must all be disjoint. The total number of different selector values is \( S = \sum s_i \quad (1 \leq i \leq k) \). To predict
the probability $p_i$ of selecting each branch one would need to know the expected distribution of the $S$ selector values.

The simplest assumption is that each of the $S$ values is equally likely. This is expressed by the rule

**CASE (a):** assume $p_i = s_i / S$.

This rule fails badly if an "otherwise" branch is present and the domain of selector values is large, as the predicted probability of selecting the "otherwise" branch is often disproportionately high.

Another simple assumption is that selection of each branch is equally likely:

**CASE (b):** assume $p_i = 1 / k$.

These rules are very simple, and frequently inaccurate. From the limited empirical evidence available to the author in dynamic program traces, it appears that neither performs well, and the second rule is even worse than the first. A hybrid rule, which has the marginally greater accuracy of CASE(a) but avoids the problem with "otherwise" branches, is:

**CASE (c):** if an "otherwise" branch is present, predict its probability of selection to be

$$p_i = 1 / k.$$  

The prediction for all other branches is weighted according to the number of selector values for each branch, as in CASE(a).

There was not enough empirical evidence from which to derive a more accurate static model, so CASE(c) was adopted. Much more empirical evidence is needed if a reasonable general static model of Case-statements is to be developed.
5.2.2. If-statements

The likelihood of a particular conditional branch being taken is known to be highly dependent on the way the same condition was decided previously [Shustek 78, SmithJE 81]. (This is the basis of a dynamic branch prediction strategy presented in [SmithJE 81]: predict that any branch will be decided the same way as on its last execution; if it has not been previously executed, predict that it will be taken. This is asserted to be an accurate strategy that performs about as well as possible.) If decisions always tend to be resolved in the same way, it would be expected that \( p_{\text{then}} \approx 1 \) or \( p_{\text{then}} \approx 0 \) for any particular If-statement. This is confirmed by Figures 5-1 to 5-3 (taken from [Lokan 83a]). The difficulty in making static predictions of \( p_{\text{then}} \) is that although it is known that one of two values should be predicted it is not known which one will be correct.

5.2.2.1. If-Then statements

Figure 5-1 shows the distribution of \( p_{\text{then}} \) values observed for 376 If-Then statements. Three quarters of the values are either greater than 0.95 or less than 0.05, and the proportions of the two extreme cases are very similar. The remaining values are evenly distributed between 0.05 and 0.95. Overall the mean value of \( p_{\text{then}} \) is approximately 0.5. (The mean varies between programs, from 0.47 to 0.52, but the distribution always has the same shape: it seems reasonable to assume that in practice the mean value of \( p_{\text{then}} \) is 0.5.)

Consider the rule

\[
\text{IF-1 (a): } \quad \text{assume } p_{\text{then}} = 0.5 \text{ for every If-Then statement.}
\]

It has been noted that 50 percent of all Then-clauses are selected dynamically; this rule assumes that each Then-clause individually is selected dynamically 50 percent of the time. For predicting profiles this assumption will certainly be wrong in almost every case. However, it should predict the total number of instructions executed quite accurately (provided there is no correlation between the number and
nature of statements in a Then-clause and the selection or non-selection of the Then-clause). Total dynamic program size is closely related to the total number of instructions executed. Consequently, rule IF-1(a) can be expected to perform well at predicting dynamic program size (which is what is required in the application under investigation here), but it is inadequate for predicting program profiles.

**Figure 5-1:** Distribution of $p_{then}$ for If-Then statements

```
<table>
<thead>
<tr>
<th>N</th>
<th>376</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>0.474</td>
</tr>
<tr>
<td>S.D.</td>
<td>0.452</td>
</tr>
<tr>
<td>Median</td>
<td>0.375</td>
</tr>
</tbody>
</table>
```

This basic rule can be "tuned", to recognize certain boolean tests in which the likelihood of evaluating to 'true' (and thus selecting the Then-clause) is much greater or less than 0.5. Using empirical data from program traces, the following rule for If-Then statements was developed. To apply the rule, examine the boolean test and make the prediction that corresponds to the form of the test, as follows ($a$ and $b$ represent constants, variables, or expressions):

**IF-1 (b):**

1. If $a <> b$ then ... : assume $p_{then} = 0.25$
2. If $a <> nil$ then ... : assume $p_{then} = 0.95$
3. If $a = b$ then ... : assume $p_{then} = 0.40$
4. If not (a in b) then ... : assume $p_{then} = 0.10$

5. If $[a < b | a <= b]$ then ... : assume $p_{then} = 0.80$

6. otherwise assume $p_{then} = 0.50$

Note that $p_{then}$ for $(a = b)$ is not equal to $p_{else}$ for $(a <> b)$, and that the prediction for $(a <> b)$ is different from that for not$(a = b)$. The predictions are based on observed usage, indicating that different constructions are favoured in different circumstances - they are not chosen randomly.

This rule is tuned to the particular set of 29 programs investigated, so it may not be universally applicable. It can easily be modified: the form of the rule is the important idea, not the numerical details.

Like rule IF-1(a), rule IF-1(b) is based on averages, so it can be expected to do as well as IF-1(a) as a basis for predicting total dynamic program size; any improvement over IF-1(a) will be small. Better predictions of program profiles should be made, particularly when predictions close to 0 or 1 are made for $p_{then}$.

It is doubtful that a static rule can be developed for If-Then statements, which performs better than IF-1(b) but whose form is different. Examination of the statements within Then-clauses may enable some more cases to be identified where $p_{then}$ is close to 0 or 1, but the extra work entailed is unlikely to be justified by the improvement in profile prediction.

5.2.2.2. If-Then-Else statements

Figure 5-2 shows the distribution of $p_{then}$ values observed for 372 If-Then-Else statements. Again, three quarters of the values are greater than 0.95 or less than 0.05, and the intermediate values are evenly distributed. The two peaks are not similar though: $p_{then} = 1$ occurs more than twice as often as $p_{then} = 0$. 
The mean value of $p_{then}$ varies between programs from 0.47 to 0.62. The true mean lies in the range 0.5 - 0.6. Analogously to rule IF-1(a), the following rule may be used:

**IF-2 (a):** assume $p_{then} = 0.6$ for all If-Then-Else statements.

The accuracy of the predictions made by this rule will be the same as for IF-1(a): poor predictions of program profiles, but reasonable predictions of total dynamic program size.

Analogously to rule IF-1(b), the following empirically-derived rule was developed for If-Then-Else statements:

**IF-2 (b)**

1. If $a <> b$ then ... : assume $p_{then} = 0.25$
2. If \( a \neq \text{nil} \) then ... : assume \( p_{\text{then}} = 0.95 \)

3. If \( a = b \) then ... : assume \( p_{\text{then}} = 0.60 \)

4. If \( \text{not} (a \in b) \) then ... : assume \( p_{\text{then}} = 0.10 \)

5. If \( a < b \mid a \leq b \) then ... : assume \( p_{\text{then}} = 0.80 \)

6. otherwise assume \( p_{\text{then}} = 0.50 \)

5.2.2.3. If-Then and If-Then-Else statements combined

In most respects the behaviour of If-Then statements and If-Then-Else statements is very similar. The empirical rules IF-1(b) and IF-2(b) differ in only one detail, and then not by much. Rules IF-1(a) and IF-2(a) also give similar predictions of values close to the middle of the range even though three quarters of the data values lie at the extreme ends of the range. These statements can be considered together, using a single rule for all If-statements.

Figure 5-3 shows the distribution of \( p_{\text{then}} \) values observed when all 748 If-statements are considered together. The mean value of \( p_{\text{then}} \) is 0.54, varying between programs from 0.48 to 0.55, and it is safe to assume that the true mean is slightly greater than 0.5 . For simplicity, represent it by the following rule:

\[
\text{IF (a)}: \quad \text{assume } p_{\text{then}} = 0.5 \text{ for all If-statements.}
\]

\[
\text{IF (b)}: \quad \text{follows from IF-1(b) and IF-2(b), with the average prediction of } p_{\text{then}} = 0.5 \text{ made in the one case where those rules differ.}
\]

5.2.2.4. Summary

Some rules have been defined for predicting the likelihood of selecting each branch of an If-statement. Depending on the level of accuracy desired, the individual rules IF-1(b) and IF-2(b) or the simple composite rule IF(a) can be used. Predictions of program profiles will vary significantly when rules with differing
Figure 5-3: Distribution of $p_{then}$ for If-statements

- **N** = 748
- **Mean** = 0.542
- **S.D.** = 0.449
- **Median** = 0.625

The mean was found to be almost 20% less than the upper limit of the confidence interval. This is consistent with the observation by earlier researchers that the mean $p_{then}$ is lower in the case of the if loop compared to other types of loops. The confidence interval was used to determine the reliability of the mean $p_{then}$ for the if loop.

**5.3.1. Repeat-loops**

Figure 5-4 shows the distribution of $p_{then}$ for 52 of the 55 Repeat-loops in the sample (the other values extended to 0). These distributions are generally observed in all programs, so the mean value of $p_{then}$ should be a good overall estimate of the loop average. This observation is the basis of the following study.
accuracy are used, but predictions of total dynamic program size will vary little. The application determines what is needed.

5.2.3. Iteration statements

To predict the number of iterations ($n_{iter}$) that will be made through the body of a loop, the average distribution of loop iteration counts must be known for each different type of loop. Empirical data for these average distributions are presented now.

In the discussions that follow, attention is restricted to those loops with $n_{iter}$ less than 30. It was found that this excludes only 3% of loops for which $n_{iter}$ could not be determined at compile time.

The mean number of iterations per loop was computed for those loops with $n_{iter} < 30$; every loop was given equal weight in calculating the mean. The computed mean is a good estimate of the loop iteration counts that will be observed in practice for almost all loops. If the few loops with very large iteration counts are included, the resulting mean is too large to be a reasonable reflection of the loop iteration counts that are usually observed.

The overall mean is $n_{iter} \approx 5$. This may appear surprisingly small (eg. in [Kuck 74] work was reported on loops with $n_{iter}$ usually less than 10, but it was suggested that larger values would be "more realistic"), but it supports a similar observation elsewhere [Lenfant 75]. Although this result is based on a small data sample of 156 loops, it was observed consistently for each program in the sample.

5.2.3.1. Repeat-loops

Figure 5-4 shows the distribution of $n_{iter}$ for 52 of the 53 Repeat-loops in the sample (the other value exceeded 30). The same distribution was consistently observed in all programs, so the mean value of $n_{iter} \approx 3$ should be a fairly accurate estimate of the true average. This observation is the basis of the following rule:
REPEAT (a): assume $n_{iter} = 3$ for every Repeat-loop.

**Figure 5-4:** Distribution of $n_{iter}$ for Repeat-loops

This simple rule can be tuned to give more accurate predictions in some circumstances. The following rule was developed empirically from program traces; a full explanation of the reasoning behind each prediction appears in [Lokan 83b]. To apply the rule, the termination condition of the loop is examined, and the prediction indicated by the first of the following conditions to be satisfied is made:

REPEAT (b):

1. condition includes a boolean variable and a separate logical expression: assume $n_{iter} = 3$.

2. condition includes a pointer variable: assume $n_{iter} = 5$.

3. condition includes a character variable: assume $n_{iter} = 4$.

4. condition includes "Eoln" or "Eof": assume $n_{iter} = 10$.

5. otherwise assume $n_{iter} = 3$.

The predictions are occasionally major underestimates, and sometimes they are slight overestimates. Generally they are quite accurate, with REPEAT(b)
performing slightly better. Both rules form sound bases for predicting either
dynamic program profiles or total dynamic program sizes.

5.2.3.2. While-loops

While-loops behave similarly to Repeat-loops: they were found to exhibit a very
consistent distribution for \( n_{iter} \), and very few loops had iteration counts exceeding
30. Figure 5-5 shows the distribution of \( n_{iter} \) for 78 of the 81 While-loops in the
sample (the other three values exceeded 30). The mean value is \( n_{iter} \approx 4.5 \), which
should be a good estimate of the true mean. The following rules are based on the
observed mean:

**WHILE** (a):

\[
\text{assume } n_{iter} = 4.5 \text{ for all While-loops.}
\]

**WHILE** (b):

\[
\text{the same as REPEAT(b), except that the "otherwise" clause}
\]
\[
\text{assumes } n_{iter} = 4.5 \text{ (instead of } n_{iter} = 3).}
\]

**Figure 5-5:** Distribution of \( n_{iter} \) for While-loops

![Distribution of n_iter for While-loops](image.png)

- \( N = 78 \)
- Mean = 4.47
- S.D. = 5.44
- Median = 2

In addition to the four separate cases identified in **WHILE**(b), another category
can be identified. While-loops are often used to scan through an array seeking a
particular condition; often only part of the array is in use, and often a Goto-
statement is used for premature exit from the loop. Detection of either of these cases requires examination of the loop body. If that cost is acceptable, WHILE(b) can be augmented as follows (assume \( m \) is the cardinality of the relevant array dimension):

\[
\text{WHILE (c):} \quad 1-4. \text{as before.}
\]

5. termination condition involves a variable used in the loop to access array elements, or a Goto-statement jumps out of a loop that processes an array:

\[
\begin{align*}
&\text{if } m > 50 \text{ assume } n_{\text{iter}} = 10 \text{ else} \\
&\text{if } m > 15 \text{ assume } n_{\text{iter}} = \text{mean of } (5,m/3) \text{ else} \\
&\text{if } m \geq 5 \text{ assume } n_{\text{iter}} = 5 \text{ else} \\
&\text{if } m < 5 \text{ assume } n_{\text{iter}} = m/2.
\end{align*}
\]

6. otherwise assume \( n_{\text{iter}} = 4.5 \)

5.2.3.3. For-loops

Of the 71 For-loops in the sample, 49 had constant bounds and only 22 had variable bounds that could not be determined at compile time. Ten of the 49 loops with constant bounds had more than 30 iterations each, compared with only one of the 22 loops with variable bounds.

The distribution of \( n_{\text{iter}} \) for each type of For-loop is shown in Figure 5-6. For neither type of For-loop was there any consistency between programs. A regular distribution might emerge from more empirical data; all that can be said here is that on average For-loops, especially those with constant bounds, tend to have more iterations than either Repeat-loops or While-loops.

For For-loops with variable bounds the mean number of iterations is approximately 10. Less confidence can be placed in this value as an estimate of the true mean than is possible for other loop types, because the data is less consistent and there is less of it. More empirical data is needed, to increase the confidence that can be placed in the rules for modelling these For-loops.
Figure 5-6: Distribution of $n_{iter}$ for For-loops

For-loops with variable bounds can be modelled as follows:

1. Compute the exact number of iterations ($n$).

2. If a Goto-statement leaves out of the loop body, use the same algorithm as in WHILE(a) part 3 to predict $n_{iter}$.

3. Otherwise, if $n_{iter} = 4$.

4. Combine Repeat-loops and While-loops (see Figure 5-7) gives the following rules:

   \[ n_{iter} \geq 4 \text{ for all Repeat-loops and While-loops.} \]
For For-loops with constant bounds, too many values exceeded 30 for it to be appropriate to compute a mean from the remaining values. The resulting mean would not be a true reflection of the loop iteration counts that would be observed in practice.

The following rules can be used for modelling For-loops with variable bounds:

\[ \text{FOR (a)} : \quad \text{assume } n_{iter} = 10 \text{ for all For-loops with variable bounds.} \]

\[ \text{FOR (b)} : \quad 1. \text{ if the index variable is used to scan through an array in the loop, make the same predictions as in WHILE(c) part 5.} \]

\[ \quad 2. \text{ otherwise assume } n_{iter} = 10. \]

For-loops with constant bounds can be modelled as follows:

\[ \text{FOR (c)} : \quad 1. \text{ compute the exact number of iterations } (m). \]

\[ \quad 2. \text{ if a Goto-statement leads out from the loop body, use the same algorithm as in WHILE(c) part 5 to predict } n_{iter}; \]

\[ \quad 3. \text{ otherwise predict } n_{iter} = m. \]

5.2.3.4. Iteration statements considered together

The predictions made by the individual rules for different loop types are quite similar, so general rules covering all loops together should be satisfactory.

Since For-loops are not modelled reliably, they will initially be excluded. Combining Repeat-loops and While-loops (see Figure 5-7) gives the following rules:

\[ \text{LOOP (a)} : \quad \text{assume } n_{iter} = 4 \text{ for all Repeat-loops and While-loops.} \]

\[ \text{LOOP (b)} : \quad \text{the same as WHILE(b), except that the "otherwise" clause predicts } n_{iter} = 4 \text{ for all Repeat-loops and While-loops (instead of } n_{iter} = 4.5). \]
LOOP (c): the same as WHILE(c), except that the *otherwise* clause predicts \( n_{iter} = 4 \) for all Repeat-loops and While-loops (instead of \( n_{iter} = 4.5 \)).

If For-loops with variable bounds are included, the rules become:

LOOP (d): assume \( n_{iter} = 5 \) for all loops

and obvious changes are made to LOOP(b) and LOOP(c).

5.2.3.5. Summary

Some rules have been developed for modelling loops, ranging from a simple global rule to finely detailed rules for each individual loop type. Although they are all reasonably sound as bases for predicting profiles and total dynamic program sizes, some combinations of rules are more accurate at the cost of more work. The accuracy required by the application determines the rules used.

5.2.4. Goto-statements

Very little empirical evidence was available from which to derive rules for dealing with *unclassifiable* Goto-statements or Goto-statements that transfer control to error-handling code. (Pascal programs were examined, and few Goto-statements appeared in them.) The rule that was adopted here is:

GOTO: assume that the probability of taking a conditional Goto-statement, if it is not otherwise estimated by a rule for alternative statements or iteration statements, is zero. The probability of taking an unconditional Goto-statement is one.

This rule gave good predictions in the few cases encountered. Beyond a belief that the probability of taking these conditional Goto-statements is generally low, no claim is made of the universal applicability of this rule. A better rule would need to be developed if Goto-statements appeared frequently in the programs to be modelled; the empirical evidence would then be available to do so.
Figure 5-7: Distribution of $n_{iter}$ for Repeat-loops and While-loops

<table>
<thead>
<tr>
<th>N</th>
<th>Mean</th>
<th>S.D.</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>3.86</td>
<td>4.56</td>
<td>2</td>
</tr>
</tbody>
</table>

5.3. Comparison between different models

Several rules have been described for making individual predictions about the execution of each type of statement that can appear in a program. Some rules are very simple; others are more complex and make more accurate predictions. If one rule for each statement type is selected, the combination of rules constitutes a model for predicting the execution profile and total dynamic size of a program.

For this research into context-sensitive architecture, the important prediction is the total dynamic size of a program. Provided total dynamic size is predicted accurately for a block of code, accurate prediction of the execution profile of that block of code is not required (although to a large extent the two can be expected to go together).
Two different models were investigated. One is the simplest that could be built from the rules described above; the other is the more complex.

5.3.1. Simple model

The simple model was constructed by selecting rules SEQ, SKIP, CASE(c), IF(a), LOOP(d), and GOTO. Thus it is assumed (unless it can be determined otherwise at compile time) that all loops have 5 iterations; branches of a Case statement are selected with probability according to the respective proportions of selector values; the chance of selecting the Then-clause of any If-statement is 0.5; and no conditional Goto-statement is taken (if it has not already been accounted for by a different rule).

Prediction of iteration statements by this model is good for both program profiles and total dynamic sizes. Profiles are predicted poorly for alternative statements, but total dynamic program size is still predicted well. Overall, the parts of the program that will account for most execution time are correctly identified, but in other respects the profile prediction is poor.

Prediction of total dynamic program size is good. The mean ratio of predicted dynamic size to observed dynamic size is 0.97, and the coefficient of correlation between predicted size and actual size is 0.83. The prediction of total dynamic size is a major underestimate in only 10% of program units.

5.3.2. Complex model

The complex model was constructed by selecting the detailed rules: SEQ, SKIP, CASE(c), IF-1(b), IF-2(b), REPEAT(b), WHILE(c), FOR(b), FOR(c), GOTO. As many special cases as possible are recognized, and only if none of them can be found is the overall average prediction made.

A noticeable qualitative improvement is made in the prediction of program
profiles. Chiefly this is due to an improvement in the prediction of If-statements: on several occasions it is possible to predict accurately a low or high value for $p_{\text{then}}$, and in those cases the profile is modelled well accordingly. The overall prediction of profiles is still poor, however, because in Case-statements and many If-statements an average value is still predicted that is usually inaccurate in each specific case.

The modelling of loops is slightly improved. Numerous slight overestimates have been corrected, and some correction has been made to underestimates. Major underestimates still occur in 10% of program units, since changing a prediction from 5 iterations to 10 has little effect when the true number is 100 or so!

The mean ratio of predicted dynamic size to actual dynamic size is 0.99, and the coefficient of correlation between predicted size and actual size is 0.87.

5.3.3. Comparisons

These two models represent the two extremes of complexity among static models for predicting program profiles. A significant difference is observed in the prediction of profiles: the more complex model is often found to give a better prediction. Both models successfully identify the parts of a program that will consume the most execution time. However, neither model does a really satisfactory job of modelling the profile accurately in detail.

The performance of the two models in predicting total dynamic program size is similar: both do very well. In the application considered here, the performance of the simple model is good enough for it to be used. The improved accuracy of the complex model is not sufficient to justify the extra cost, when the simple model already performs well.

In summary, the simple model is sufficient to give good predictions of total dynamic program size, but it gives poor predictions of execution profiles. The prediction of profiles can be improved by increasing the complexity of the model, but even the best prediction is generally not accurate in detail.
5.4. Summary

A framework for static models that predict the execution profile and total dynamic size of a program has been presented. Various detailed models of each type of program statement, each based on empirical evidence on the usage of different statement types, have been discussed. Two models, of widely differing complexity, have been investigated.

Loops are modelled well by these static models. It has been found that nearly all loops have only a small number of iterations, and the suggested rules give accurate predictions in 90% of program units.

Alternative statements are not modelled well. Static information alone seems insufficient to enable accurate predictions to be made of the probability of selecting each branch of an alternative statement.

Although the predictions regarding alternative statements are usually wrong individually, collectively they are correct on average. Consequently the predictions of total dynamic size are usually correct. The limit on accuracy is imposed by loop predictions, which are good in 90% of program units and bad underestimates in the remaining 10%.

Profile predictions are generally unsatisfactory. The parts of each program that will account for most execution time are identified correctly, but otherwise the predicted profiles are rarely accurate in detail. The failure lies in the prediction of alternative statements.

Accurate prediction of total dynamic program size is more important in this research than accurate prediction of detailed execution profiles. To underestimate total dynamic size is to err on the safe side. Thus the models of program behaviour developed here are adequate for the purposes of this research.
Chapter 6
Segmentation of program units

6.1. Introduction

The subdivisions sought within program units, and one of the main tools used in
that search, have been described in the last two chapters. Those ideas are brought
together in this chapter, with further information that can be observed or deduced
about program units. The process that uses all of this information to find a suitable
segmentation of a program unit is explained.

Not every possible segmentation of a program unit is allowable. For a
segmentation to be valid, the resulting dynamic sequence of environments must
conform to a simple stack model. There can be only one point of entry and one
point of exit for every segment encoded within the context of its own local
environment. No change of environment can occur except at these entry and exit
points; no transfer of control is possible from code in one context to code in another
context except through the appropriate exit and entry points.

In order to segment a program unit, the following information is required. It is
all available from a static examination of the program text:

- a parse tree, representing the syntax of the program unit;
- patterns of usage of operands, operators, and labels in the program unit;
- results of control flow and data flow analysis of the program unit: a flow
graph, a depth-first ordering of its nodes, and a list of "lifetimes" of
definitions in the program unit;
predicted profile of the program unit (n.b. this may sometimes be derived from past dynamic information, but in general it must be generated statically).

There are two stages involved in finding the best segmentation of a program unit. The first is the construction of a tree that represents the structure of the program unit. Every node in the tree represents an instruction, or group of instructions, that might be worth encoding within the context of its own environment. The tree is based initially on the syntax of the program unit; it may subsequently be modified as other knowledge about the program unit is incorporated; finally it may be pruned, to eliminate nodes that can confidently be assumed to be not worth encoding in their own environments. Initial tree construction is described in section 6.2.1; modification in section 6.2.2; and pruning in section 6.2.3.

The second stage is the traversal of the tree. As each node is reached in turn, a decision is made as to whether or not it represents a segment that would be worth encoding in its own environment. The predicted dynamic profile and the usage of operators, operands, and labels constitute the data from which the total static and dynamic sizes are computed; the decisions are based on those sizes. The process of traversing the tree and making a decision at each node is described in section 6.3.

The result of the tree traversal is a list of nodes that are to be encoded as separate segments. From this list, the points at which the environment should be changed (either to enter a new environment or to revert to a previous one) are apparent. The program unit is then encoded into binary form; the details of this are given in section 6.4.

An implementation of this scheme was tested in practice; the results are presented in section 6.5. The chapter concludes with a discussion of how successful these ideas have proved, and how they might be improved.
6.2. Construction of a tree to represent program structure

6.2.1. Syntax analysis

The possible paths for flow of control through a program unit are determined by its syntax. The control paths in turn limit the potential ways in which the program unit can be segmented (recall that a potential segmentation is only valid if every segment has only one entry point and one exit point, and every branch instruction is in the same segment as its target). When subdivisions are sought within a program unit, its syntactic structure must be considered at the very least.

A tree that represents the syntactic structure of the program unit is constructed. It contains one node for each statement, be it a compound statement, structured statement, or simple unstructured statement. Each node for a compound statement or structured statement has children that represent the substructure of the statement; the children in turn may represent compound statements or structured statements with further substructure. Nodes representing unstructured statements have no children, as there is no substructure to represent.

The representation of a compound statement is shown in Figure 6-1.

**Figure 6-1:** Representation of compound statement

```
begin
  S_1; S_2; ...; S_n;
end;
```

A structured statement is an iteration statement or alternative statement (or a With-statement in Pascal). The representation of iteration statements is shown in Figure 6-2; that of alternative statements in Figure 6-3. Each loop body, or branch
of an alternative statement, is a compound statement. A With-statement is treated as a compound statement.

**Figure 6-2:** Representation of iteration statements

```
loop-statement
```

An unstructured statement is an assignment statement, input/output statement, procedure call, or Goto-statement. Each of these is represented by a node with no children. When a Goto-statement is encountered, the tree must be adjusted to ensure that the Goto-statement and its target are in the same node.

Initially the tree consists of only one node, representing the compound statement that constitutes the body of the program unit. That node is *expanded* by replacing it with the appropriate construction, shown in Figure 6-1. Every unexpanded node that represents a compound statement or structured statement is expanded in turn, until no further expansion is possible. Every leaf node then either represents a single unstructured statement, or contains a Goto-statement and its target in a configuration that precludes further expansion (because an invalid segment would be the result).

An example, given in Figure 6-4, shows the tree that is constructed for the sample program listed in Figure 3-1 (see page 32).

A tree based on syntax alone can be used successfully to identify *local optimization* environment changes: such changes are usually made for loops, which are represented by nodes in the tree. However, few *changing locality* environment changes can be identified, as the tree contains few nodes that represent segments of *constant locality* (i.e. segments in which the favoured sets of operators, operands,
Figure 6-3: Representation of alternative statements

(a) Case-statement

(b) If-Then statement

(c) If-Then-Else statement

and labels do not alter). Any such nodes represent segments that happen to coincide with syntactic units in the source program.

The syntax tree should be augmented, to include nodes that represent segments of constant locality: this is the subject of the next section.
6.2.2. Detecting segments with constant locality

One method for finding segments of constant locality is to consider the text of the program as a string of operators, operands, and labels, and to analyse the string directly. This is analogous to a "reference string" of page references, on which much memory-management research has been done [Denning 80].

A different approach is suggested here. It makes more use of information that can be observed and deduced about the program unit.

First, the sets of operators, operands, and labels that appear in the image instructions represented by each node are enumerated. As explained later in this section, some relationships may be recognized between some nodes by comparing these sets, and some boundaries may be identified between phases of constant locality. These boundaries represent points at which the environment should be changed; the segments that they delimit should be represented by new nodes added to the tree.

Data flow analysis provides extra information about the flow of control and
data through the program unit. This may enable more segments of constant locality to be found. It might also help to make decisions when the enumerated sets of operators, operands, and labels do not provide sufficiently conclusive evidence.

The first step in the flow analysis is the construction of a graph to represent the different paths for flow of control through the program unit. From this graph it is possible to identify loops and alternative statements that have been implemented with Goto-statements: nodes can be added to the syntax tree to represent such statements.

The flow graph is the basis for analysing the flow of data through the program unit. A depth-first ordering of its nodes is found, and the "reaching definitions" and "live definitions" problems are solved using well-known algorithms [Aho 77, Hecht 77]. From the information obtained, the set of "lifetime histories" is deduced for all definitions of all variables in the program unit. (A "definition" is the assignment of a value to a variable in an assignment statement or input statement; a definition is "used" when its value is used as an operand in an expression or output statement; it is "killed" by another assignment to the same variable.) Each lifetime history has the form

(defined at $p$; used at $p_1$, $p_2$, ..., $p_n$; killed at $p_k$).

Once the sets of operators, operands, and labels have been enumerated for each node, and the lifetime histories have been determined, the search for locality changes begins.

A top-down (breadth-first) traversal is made over the syntax tree. When a non-terminal node is visited, its children are considered together to see what connections appear to exist. No action occurs when a terminal node is visited, since by definition it has no children.

When some children of a node appear to be closely connected, a new node is added to the tree to represent the connection (see Figure 6-5). These new nodes represent phases of constant locality.
Principles such as the following can be used to decide whether or not adjacent nodes are connected:

1. using only the sets of operators and operands in each node:
   
   a. if the sets of operators and operands share a certain proportion of common entries (n.b. sets of labels will generally be disjoint), the nodes are considered to be part of the same segment: they are connected.
   
   b. if the proportion of common entries is below another threshold, the nodes are considered to be in different segments: they are not connected.
   
   c. for intermediate cases, whether weakly-related nodes are connected or not depends on the strength of relationships between the node and each of its neighbours, and between its neighbours. For example, in Figure 6-5, suppose \((s_1; s_2; s_3)\) and \((s_5; s_6)\) are definitely separate, and \(s_4\) is weakly related to both. They certainly cannot all be connected, and there is no reason to connect \(s_4\) to one group rather than the other, so the weak relationships are broken: \(s_4\) is connected to neither group.

2. where several definition lifetime histories begin and end at about the same places, the union of those lifetimes may constitute a phase of constant locality. This can be especially useful in finding structure within straight line code.

For example, consider the tree shown in Figure 6-4. All operands and one of the two operators in the initial statements also appear in the following loop, so the loop and the initial statements can all be connected. The succeeding output statement shares only one variable with the loop, out of sets of 6 and 4 operators and 4 and 5 operands, so it remains separate from the loop. Within the loop there is already a
Figure 6-5: Adding nodes to reflect locality

Procedure W;
begin
  \( S_1; \)  (\( S_1; S_2; S_3 \)) appear closely related
  \( S_2; \)
  \( S_3; \)  (\( S_5; S_6 \)) appear closely related internally, but
clearly separate from the first group
  \( S_4; \)
  \( S_5; \)
  \( S_6; \)  \( S_4 \) appears weakly related to both
end;

This program unit is initially represented by the following tree:

```
  W
     /\     /\     /\     /\     /\  
    S_1 S_2 S_3 S_4 S_5 S_6
```

The initial tree is transformed into the tree shown below:

```
  W
     /\     /\     /\     /\  
    new S_4 new
        /\     /\     /\  
       S_1 S_2 S_3 S_5 S_6
```
node connecting its two constituent statements, so there is no scope there for augmenting the tree. A node is added to the tree to represent the connection between the initialisation statements and the loop. The modified tree is shown in Figure 6-6.

**Figure 6-6:** Modified tree for example source program

```
sum := 0
read(input, val)
While-loop
  sum := sum + val
  read(input, val)

writeln(output, "sum is ", sum:6)
```

The nature of the tree is changed when new nodes are added to reflect changing locality. Although it is still based primarily on syntax, the tree now conveys information about the operators and operands used in the program unit as well. In this form, the nodes near the root may give a rough outline of the logical structure of the program unit. Trying to strengthen these deductions about program logic, using more data flow knowledge and some heuristic rules, could be a promising avenue for future research.
6.2.3. Pruning the tree

In its present form, the tree is too finely detailed. Most (if not all) leaf nodes represent single unstructured statements: it is unlikely that any of them merit encoding in separate environments, so time spent considering them would be wasted. Thus it is desirable to eliminate some nodes that can confidently be predicted to be of no use: this makes the tree more manageable, without altering its basic nature.

Some policy should be adopted for pruning the tree, ranging from one extreme of doing nothing to the other extreme of deleting all terminal nodes.

The policy adopted here was to group terminal nodes into basic blocks. All adjacent terminal nodes, that have the same parent and that each represent an unstructured statement, are collapsed into a single node. Terminal nodes that represent groups of instructions including Goto-statements and their targets are not included. Whenever this policy results in a new terminal node that represents the same instructions as its parent, it is deleted.

Figure 6-7(a) shows the result of pruning the tree for the example program, and Figure 6-7(b) shows the image instructions represented by each node.

6.2.4. Summary: tree construction

A tree that represents the structure of a program unit has been constructed. Its initial construction is based on the syntax of the program unit. It may then be modified to reflect changing use of operators and operands. Finally it is pruned, to delete some nodes that are unlikely to be useful in the given application.

Each node represents a group of image instructions that form a valid segment, and that may warrant encoding in their own local environment. Attached to each node is a list of the operators, operands, and labels used in the instructions it represents.
**Figure 6-7:** Result of pruning the tree for the example source program

(a) Resulting tree

(b) Image instructions represented by each node
This tree is one of the main items of information about a program unit, to be used in the search for a suitable segmentation.

6.3. Finding a good segmentation

With the construction of a tree representing the valid potential segments in a program unit, and the prediction of the dynamic execution profile for the program unit, all of the required information has been gathered. Each node in the tree is now considered in turn, to decide which of them represent segments for separate encoding within their own local environments. The method for making those decisions is the subject of this section.

Section 6.3.1 describes the mechanisms for visiting the nodes in correct sequence; for ensuring that the lists of operators, operands, and labels used in a node are up to date when it is visited; and for building a list of nodes that represent segments to be encoded separately.

Section 6.3.2 describes the actions performed to decide whether or not a given node represents a segment for separate encoding. The decision involves comparisons with several thresholds, whose values may be varied as parameters; this is discussed in section 6.3.3.

The results of applying this algorithm are a list of segments to be given separate encodings; and the instructions of the image program, which include newly-inserted instructions to change the environment when necessary. This provides the information needed for the actual encoding of the program unit into binary form, discussed in section 6.4.
6.3.1. Tree traversal

The algorithm presented here is based on a post-order traversal of the tree: each node is visited once, after all of its children have been visited; the last node visited is the root.

The details given below are concerned with visiting the nodes in correct sequence, and with maintaining accurate lists of operators, operands, and labels needed in the environment. A list of segments that will be encoded within their own environments is built; at the end of the traversal this list, designated *Segments*, represents the chosen segmentation for the program unit.

As well as the tree, use is made of the image program itself. Associated with every instruction is a boolean flag *Already encoded*: initially false, it indicates whether or not the instruction is included in any segment identified so far for separate encoding. In addition, new instructions are added to the image program to signal each transition between environments.

Each node contains the following information:

*Parent*  
a pointer to its parent node

*Eldest*  
a pointer to its first child

*Sibling*  
a pointer to the next node that has the same parent

*First_instruction, Last_instruction*  
pointers to the first and last instructions represented by the node

*Operand_list*  
a list of operands used within this sequence of instructions. Each entry is a record with two fields:

*Operand* : the operand itself

*Use_count* : the total number of times the operand
appears in the whole segment (computed before the tree traversal begins).

**Operator_list, Label_list**

analogous to **Operand_list**

**Contains_segment** a boolean flag, initially 'false', which indicates if any descendants of the node have been identified as segments for separate encoding.

The following algorithm is used to traverse the tree:

\[
\begin{align*}
\text{begin} \\
\quad &\text{Segments} := \text{empty list;} \\
\quad &(* \text{Select the first node to be visited} \*) \\
\quad &\text{Current_node} := \text{Root;} \\
\quad &\text{while} \text{Current_node.Eldest exists do} \\
\quad &\quad \text{Current_node} := \text{Current_node.Eldest;} \\
\quad &\text{while} \text{Current_node} <> \text{Root do} \\
\quad &\quad \text{begin} \\
\quad &\quad \quad (* \text{Visit one node in the traversal} *) \\
\quad &\quad \quad \text{if} \text{Current_node.Contains_segment then} \\
\quad &\quad \quad \quad \text{begin} \\
\quad &\quad \quad \quad \quad \text{Add *Change-env* to the Current_node.Operator_list;} \\
\quad &\quad \quad \quad \text{end;} \\
\quad &\quad \text{Use the algorithm described in section 6.3.2 to decide if the Current_node should be encoded in its own local environment: set the global boolean flag Worthwhile accordingly;} \\
\end{align*}
\]
if Worthwhile then
begin

(* Make arrangements for separate encoding *)

Add *Change-env* and *Revert-env* instructions to the code at the appropriate places;

Add *Revert-env* to Current_node.Operator_list;

For each instruction represented by Current_node, set the flag Already_encoded to 'true';

Trace the path from Current_node.Parent to Root, inclusive. At each node, find the entry in its Operand_list that matches each entry in Current_node.Operand_list, and decrement its Use_count by the amount stored in the entry in Current_node.Operand_list. If the resulting value is zero that operand is no longer needed in the environment for that node, so delete that record from the Operand_list. Repeat for labels and operators (excluding *Revert-env*, which does not appear in any Operator_lists for ancestors of Current_node.

Construct a record listing start and end instructions of this node, and all objects that must be present in its environment. Append the record to the list of Segments;

end;

or Current_node.Contains_segment
or Worthwhile;
(* Select the next node to visit *)

if Current_nodeSibling exists then
begin
  Current_node := Current_nodeSibling;
  while Current_node.Eldest exists do
    Current_node := Current_node.Eldest;
  end
else Current_node := Current_node.Parent;
end; (* of while-loop : end of visiting one node *)

(* The traversal has reached the root. Add a record to Segments that represents all instructions not already part of a segment; *)

if Root.Contains segment then
begin
  Add "Change-env" to Root.Operator_list;
end;

Construct a record listing the start and end instructions of this node (i.e. the whole procedure), and all remaining objects that must be present in its environment. Append the record to the list of Segments;

end; (* of the tree traversal *)

6.3.2. Decision at each node

If a node represents a new scope of definition (which can be recognized from syntax) it definitely must be encoded as a separate segment. If it does not, there is a decision to be made.

In deciding whether or not a given node (designated N) is worth encoding as a separate segment, reference is made to that node and its parent (designated P). The instructions encompassed by each of them, and the respective sets of operators, operands, and labels, are considered. The predicted execution profile is also used.
The approach is to suppose that $N$ will be encoded as a separate segment, and to compute the total static and dynamic sizes of the instructions represented by $N$ and $P$. These are compared with the total static and dynamic sizes of the same instructions if $N$ is encoded in a more global environment. The "more global environment" assumed is the local environment for $P$, as this is the most conservative assumption and it guarantees that the actual savings are at least as great as predicted. If the estimated savings in total sizes are big enough, $N$ is be encoded as a separate segment.

Some instructions in $N$ or $P$ may lie within previously-identified separate segments; they are recognizable from their *Already encoded* flags, which have the value 'true'. They are disregarded, and the remaining instructions are treated as though they occur in a single contiguous block.

Consider the following sets, with indicated cardinalities:

- $R_N = \{\text{operands used in } N, \text{ but nowhere else in } P\}$: the cardinality of this set is $r_N$.
- $R_P = \{\text{operands used in } P, \text{ but not in } N\}$: the cardinality of this set is $r_P$.
- $R_B = \{\text{operands used in both } N \text{ and } P\}$: the cardinality of this set is $r_B$.
- $L_N = \{\text{labels used in } N, \text{ but nowhere else in } P\}$: the cardinality of this set is $l_N$.
- $L_P = \{\text{labels used in } P, \text{ but not in } N\}$: the cardinality of this set is $l_P$.
- $L_B = \{\text{labels used in both } N \text{ and } P\}$: the cardinality of this set is $l_B$.
- $R^N_N = \{\text{operators used in } N, \text{ but nowhere else in } P, \text{ if } N \text{ is encoded in } N's \text{ environment}\}$: the cardinality of this set is $i^N_N$.
- $R^P_N = \{\text{operators used in } N, \text{ but nowhere else in } P, \text{ if } N \text{ is encoded in } P's \text{ environment}\}$: the cardinality of this set is $i^P_N$. 

\[ P_N = \{ \text{operators used in } P, \text{ but not in } N, \text{ if } N \text{ is encoded in } N's \text{ environment} \} : \text{the cardinality of this set is } i_N^P. \]

\[ P_P = \{ \text{operators used in } P, \text{ but not in } N, \text{ if } N \text{ is encoded in } P's \text{ environment} \} : \text{the cardinality of this set is } i_P^P. \]

\[ N_B = \{ \text{operators used in both } N \text{ and } P, \text{ if } N \text{ is encoded in } N's \text{ environment} \} : \text{the cardinality of this set is } i_N^N. \]

\[ P_B = \{ \text{operators used in both } N \text{ and } P, \text{ if } N \text{ is encoded in } P's \text{ environment} \} : \text{the cardinality of this set is } i_P^P. \]

Note that \( F_N = \sum_i + \{ * \text{Revert-env} \} \). The presence of *Change-env* in the operator-lists of \( N \) and \( P \) depends on decisions at previous nodes; *Change-env* can be a member of any of \( F_N, P_P, P_B, P_B \).

The savings (or *gain*, designated \( g \)) in bits per operand, label, and operator in the instructions of \( N \), if \( N \) is encoded as a separate segment, are respectively

\[ g_{NR} = \lceil \log_2(r_B + r_N + r_P) \rceil - \lceil \log_2(r_B + r_N) \rceil \]

\[ g_{NL} = \lceil \log_2(l_B + l_N + l_P) \rceil - \lceil \log_2(l_B + l_N) \rceil \]

\[ g_{NI} = \lceil \log_2(i_B^P + i_N^N + i_P^P) \rceil - \lceil \log_2(i_B^N + i_N^N) \rceil \]

Let \( Q_j \) represent the \( j \)th instruction.

Let \( a_j \) be the number of named operands in instruction \( Q_j \).

Let \( b_j \) be the number of named labels in instruction \( Q_j \).

Let \( c_j \) be the predicted number of times instruction \( Q_j \) will be executed, given by the predicted program profile.

Suppose that \( N \) represents \( k \) instructions, which are a subset of the \( m \).
instructions represented by $P$. Assume that instructions $Q_1$ to $Q_i$ are in $P$; instructions $Q_{i+1}$ to $Q_{i+k}$ are in $N$; instructions $Q_{i+k+1}$ to $Q_m$ are in $P$. Note that $i = 0$ is possible, as is $i+k = m$.

The static saving in bits in the encoding of each instruction $Q_j$ in $N$, if $N$ is encoded as a separate segment, is

$$s_j = g_{N_1} + a_j * g_{N_2} + b_j * g_{N_3} \quad (i+1 \leq j \leq i+k)$$

The dynamic saving in bits in the encoding of each instruction $Q_j$ in $N$, if $N$ is encoded as a separate segment, is

$$d_j = c_j * s_j \quad \quad (i+1 \leq j \leq i+k)$$

The total static saving in the instructions of node $N$ is

$$S_N = \sum_{i+1}^{i+k} s_j$$

The total dynamic saving in the instructions of node $N$ is

$$D_N = \sum_{i+1}^{i+k} d_j$$

Balanced against the savings in instruction encodings are the costs of two new instructions and a change of environment. Seven more sets are used to determine the costs; these sets are constructed by comparing $N$ with the $Root$ (recall that the operators, operands, and labels used in previously-identified separate segments have been removed from consideration):

$$F^N_{Root} = \{ \text{all entries in } Root.\text{Operator list} \} + \{ \text{Change-env} \} : \text{the cardinality of this set is } r^N_{Root}.$$  

$$F^P_{Root} = \{ \text{all entries in } Root.\text{Operator list} \} : \text{the cardinality of this set is } r^P_{Root}.$$
\[ L_{\text{ROOT}} = \{ \text{all entries in } \text{Root.Label\_list} \} : \text{the cardinality of this set is } l_{\text{ROOT}}. \]

\[ R_{\text{ROOT}} = \{ \text{all entries in } \text{Root.Operand\_list} \} : \text{the cardinality of this set is } r_{\text{ROOT}}. \]

\[ I_u = \{ \text{all operators unique to } N \text{ within the whole remaining program unit} \} : \text{the cardinality of this set is } i_u. \]

\[ L_u = \{ \text{all labels unique to } N \text{ within the whole remaining program unit} \} : \text{the cardinality of this set is } l_u. \]

\[ R_u = \{ \text{all operands unique to } N \text{ within the whole remaining program unit} \} : \text{the cardinality of this set is } r_u. \]

Note that if \( P = \text{Root} \), the following equivalences are observed:

\[ l_u = \text{~} P^N ; \quad r_u = \text{~} P^N + P_p + P_b ; \quad l_u = \text{~} P^N + P_p + P_b ; \quad L_u = L_N ; \quad L_{\text{ROOT}} = L_N + L_b + L_p ; \quad R_u = R_N ; \quad R_{\text{ROOT}} = R_N + R_b + R_p. \]

The number of operators left in the more global environment, after \( N \) is encoded separately, determines the size of the "Change-env" instruction that is added to \( P \) to signal the change to \( N \)'s environment. That number of operators is at least \((i_B^N + i_U^N)\), if \( P \) is also encoded as a separate segment; and at most \((i_{\text{ROOT}}^N - i_u)\) if no further segments are identified and the rest of the program unit is encoded in the environment of the root. The maximum size of the added "Change-env" instruction is therefore \((4 + \log_2(i_{\text{ROOT}}^N - i_u))\) bits (the "4" is for the format field). The size of the added "Revert-env" instruction is \((4 + \log_2(i_N^N + i_b^N))\) bits.

The number of operands duplicated in environment specifications is at least \( r_b \), and at most \((r_b + r_N - r_u)\); similar observations apply to labels and operators. Suppose

\[ w_r = \text{number of bits to specify one operand} \]
\[ w_l = \text{number of bits to specify one label} \]
\[ w_i = \text{number of bits to specify one operator} \]
\[ w_o = \text{overhead: number of bits to specify widths} \]
Assume that, with one exception, every change of environment requires complete re-specification of all tables (this assumption is removed in chapter 7). The only exception occurs when the new environment includes no labels (or possibly no operands): instead of \(2 + m\) bits being generated to define an empty table, two bits are generated to indicate that no change needs to be made to that table.

Normally all tables need re-specification, in which case the maximum cost due to extra specification of table entries is

\[
w = w_o + w_r^r(b + r - i) + w_i^r(b + i - i) + w_i^r(i + i - i)
\]

and the maximum total static cost for the change of environment is

\[
Cost_s = w + (4 + \lceil \log_2(i_i + i_i) \rceil) + (4 + \lceil \log_2(i_i + i_i) \rceil).
\]

The number of times this cost will be incurred (designated \(c\)) is evident from the predicted profile of the surrounding instructions: the total dynamic cost is

\[
Cost_d = c \ast Cost_s.
\]

If \(N\) is encoded separately, the overall reduction in static size of \(N\) itself is \((S_N - Cost_s)\); the overall reduction in the dynamic size of \(N\) is \((D_N - Cost_d)\). These reductions are compared with thresholds to see if they are sufficient to justify the separate encoding. If either

\[
(S_N - Cost_s) \geq T_1 \text{ or } (D_N - Cost_d) \geq T_2
\]

then \(N\) is worth encoding as a separate segment: the boolean flag \(Worthwhile\) is set accordingly.

If both thresholds are passed it is clearly desirable to encode \(N\) as a separate segment; if neither threshold is passed it is undesirable to do so. If only one threshold is passed it may be risky to encode \(N\) as a separate segment, as no check is made on how close the other threshold is to being passed. In principle, while one size measure is improved the other could be worsened by any amount.

In practice, no potential change of environment was found that decreases static
size while increasing dynamic size. It is therefore safe to encode any node as a separate segment if it leads to a static size reduction, as a dynamic size reduction is likely to occur as well.

Frequently a reduction in dynamic size is made at the cost of an increase in static size. Some assurance is needed that static program size cannot worsen arbitrarily while dynamic size is improved. One way to do this is to reject any changes of environment that cause the static size to increase by more than a given amount, regardless of the dynamic change. Alternatively, the dynamic threshold can be made high, so that even if the static size does increase the reduction in total dynamic size would make it worthwhile. The latter policy was adopted here; this is elaborated upon in section 6.3.3, where parameter settings are discussed.

The decision with respect to $N$ alone has been described; the additional consideration of $P$ is described now.

If $N$ is encoded as a separate segment, some things are no longer needed in the environment for the rest of the program unit. Sometimes enough things are no longer needed for savings to be made within $P$ as well as $N$. The maximum saving is made if $P$ is also encoded separately: the saving in $P$ due to the separate encoding of $N$, in bits per operand (for example), is then

$$[\log_2(r_B + r_N + r_P)] - [\log_2(r_B + r_P)].$$

The minimum saving is made if no further segments are identified, and $P$ is encoded as part of the $Root$: the saving in bits per operand in $P$ is then

$$[\log_2(r_{\text{Root}})] - [\log_2(r_{\text{Root}} - r_U)].$$

The minimum must be assumed: hence the savings in bits per operand, label, and operator in $P$, due to the separate encoding of $N$, are estimated to be:

$$g_{PR} = [\log_2(r_{\text{Root}})] - [\log_2(r_{\text{Root}} - r_U)]$$

$$g_{PL} = [\log_2(l_{\text{Root}})] - [\log_2(l_{\text{Root}} - l_U)].$$

$$g_{PI} = [\log_2(i_{\text{Root}}^P)] - [\log_2(i_{\text{Root}}^N - i_U)].$$
These savings may not be made in every instruction of $P$. It must be assumed that they will not be made in those parts of $P$ which are represented by unvisited siblings of $N$, because those siblings might be encoded separately. Define $P'$ to represent those instructions of $P$ that are not contained in $N$ or any of $N$'s unvisited siblings. All of the instructions \{\(Q_j \leq j \leq i\)\} must be in $P'$, because of the order in which nodes are visited in the traversal; most of the instructions \{\(Q_j : i+k+1 \leq j \leq m\)\} are not in $P'$ because they are in unvisited siblings of $N$.

The static saving in each instruction of $P'$ is
\[
s_j = g_{pi} + a_j * g_{pr} + b_j * g_{pl}.
\]
The estimated dynamic saving in each instruction of $P'$ is
\[
d_j = c_j * s_j.
\]
The total savings in the instructions of $P$ are at least as great as those from the instructions of $P'$. The total static saving is estimated to be:
\[
S_P = \sum s_j \quad \{j : Q_j \in P'\}.
\]
The total dynamic saving in the instructions of $P$ is estimated to be:
\[
D_P = \sum d_j \quad \{j : Q_j \in P'\}.
\]
The resulting values are compared with thresholds. If either
\[
(S_N + S_P - Cost_S) \geq T_9 \quad \text{or} \quad (D_N + D_P - Cost_D) \geq T_4
\]
then $N$ is worth encoding as a separate segment, and the boolean flag \text{Worthwhile} is set accordingly.

Further gains may be made in the program unit beyond $P$ if $N$ is encoded as a separate segment, but only if $P$ is not also encoded separately. This cannot be assumed, so the effect of $N$ on the program unit beyond $P$ cannot be predicted.

If a node fails to pass any of the thresholds $T_1$ to $T_4$, it is rejected as a candidate for separate encoding.
The decision process described is simple to implement: one pass is needed over the instructions of the given node, and perhaps also its parent. The minimum total static and dynamic savings can be estimated easily, using the predicted execution profile and the intersections and differences between the sets of operands, operators, and labels used in each node. The maximum cost is also easily computed, so it is simple to compute the minimum estimated profit if the given node is encoded separately. Trivial threshold comparisons are then all that remain.

6.3.2.1. Example

The workings of this algorithm will be demonstrated by considering the sample program from page 33, which is reproduced with its structure tree in Figure 6-8.

The first three nodes visited in the traversal represent the initialization statements, the loop body, and the while loop. At all of them it is found that encoding them separately would increase both total static size and total dynamic size. The next two nodes to be visited are designated \( N_I \) and \( N_e \). If either of them is encoded as a separate segment, the program unit is split into two phases: the division occurs after the while loop.

The fourth node visited is \( N_I \), which represents the combination of initialization statements and the loop. Its parent, designated \( P \), represents the whole program. The decision at \( N_I \) is now explained in detail.

The following sets are constructed:

\[ F_N = \{ \text{assign, read, >, jump-if-false, add, goto, Revert-env} \} \]

\[ F_P = \{ \text{write, halt, Change-env} \} \]

\[ F_P = \{ \text{write, halt} \} \]
Figure 6-8: Example of decision algorithm

1: (assign, 0, sum)  
2: (read, input, val)  
3: (>, val, 0, T)  
4: (jump-if-false, T, 8)  
5: (add, sum, val, sum)  
6: (read, input, val)  
7: (goto, 3)  
8: (write, "sum is ", 7, output)  
9: (write, sum, 6, output)  
10: (write, eof, output)  
11: (halt,...)

<table>
<thead>
<tr>
<th></th>
<th>a_j</th>
<th>b_j</th>
<th>c_j</th>
<th>s_j</th>
<th>d_j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>5.5</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>5.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>4.5</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
<td>4.5</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>4.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
\[ R_N = \{0, \text{input}, \text{val}\} \]

\[ R_B = \{\text{sum}\} \]

\[ R_p = \{\text{sum is } 6, 7, \text{output}, \text{eol}\} \]

\[ L_N = \{3, 8\} \]

\[ T_B = T_p = L_B = L_p = \{\} \]

\[ T_{\text{ROOT}} = T_p + T_n \]

\[ T_{\text{ROOT}} = T_p + T_n \]

\[ L_{\text{ROOT}} = L_N \]

\[ R_{\text{ROOT}} = R_N + R_B + R_p \]

\[ I_U = T_n \]

\[ L_U = L_N \]

\[ R_U = R_N \]

The gains in bits per operator, operand, and label if \( N_I \) is encoded separately are found to be

\[ g_{NR} = \lceil \log_2 9 \rceil - \lceil \log_2 4 \rceil = 2 \]

\[ g_{NI} = \lceil \log_2 8 \rceil - \lceil \log_2 7 \rceil = 0 \]

\[ g_{NL} = \lceil \log_2 2 \rceil - \lceil \log_2 2 \rceil = 0 \]

\[ g_{PR} = \lceil \log_2 9 \rceil - \lceil \log_2 6 \rceil = 1 \]
\[ g_{p1} = [\log_2{8}] - [\log_2{3}] = 1 \]

\[ g_{pL} = 0, \text{ since } (l_p + l_D) = 0 \]

\( P' \) contains no instructions, as the only instructions in \( P \) that are not in \( N \) are included in an unvisited sibling of \( N \).

Two bits are saved in every operand in the initialisation statements and the loop. One bit is saved in every opcode and operand in the rest of the program.

Within the loop, 49 bits are decoded per iteration instead of 61. If the number of iterations is large enough for total dynamic size to be dominated by the loop instructions, with context specification and other instructions contributing little, the saving in total dynamic program size is about 20%.

The column labelled \( c_j \) in Figure 6-8 shows the predicted number of iterations for each instruction; the columns labelled \( s_j \) and \( d_j \) show the static and dynamic savings per instruction if \( N_j \) is encoded separately. The total gains are

\[ S_N = 20 \quad ; \quad D_N = 66 \quad ; \quad S_p = 0 \quad ; \quad D_p = 0 \]

The net cost of the environment change is 31 bits; the \( \text{\textasteriskcentered} \text{Change-env} \) instruction costs 6 bits; and the \( \text{\textasteriskcentered} \text{Revert-env} \) instruction costs 7 bits. The overall costs of the environment change would be

\[ \text{Cost}_S = \text{Cost}_D = 44 \text{ bits.} \]

The four threshold tests are:

\[ (S_N - \text{Cost}_S) \quad = -24 \geq T_1 \]

\[ (D_N - \text{Cost}_D) \quad = 22 \geq T_2 \]

\[ (S_p + S_N - \text{Cost}_S) \quad = -24 \geq T_3 \]

\[ (D_p + D_N - \text{Cost}_D) \quad = 22 \geq T_4 \]
It is unlikely that any of the thresholds would be so low that this change of environment would be acceptable. However, if a past dynamic trace indicated a high number of loop iterations, the decision might be different.

Assuming that $N_I$ is rejected, the traversal passes on to node $N_e$. This time it is found that $g_{NR} = g_{NI} = 1, g_{PR} = 2, P' = N_I, g_{PI} = g_{NR} = g_{NL} = 0$. The static and dynamic gains are then

\[ S_N = 12 \quad ; \quad D_N = 12 \quad ; \quad S_p = 20 \quad ; \quad D_p = 66 \]

and the static and dynamic costs are each 44 bits. Thus the threshold tests are:

\[ (S_N - Cost_s) = -32 \geq T_1 \]
\[ (D_N - Cost_d) = -32 \geq T_2 \]
\[ (S_p + S_N - Cost_s) = -12 \geq T_3 \]
\[ (D_p + D_N - Cost_d) = 34 \geq T_4 \]

Although it is still unlikely that any threshold is passed, the last two comparisons may succeed where the corresponding comparisons for $N_I$ failed.

6.3.3. Parameter settings

Most of the information used, in deciding whether or not a node should be encoded as a separate segment, is extracted from the image program instructions. There are two exceptions:

1. The values of $w_o, w_i, w_l, \text{and } w_r$ depend on the scheme used to specify environments;

2. The threshold values $T_1, T_5, T_8, \text{and } T_4$ must be supplied as parameters.

The values of $w_o, w_i, w_l, \text{and } w_r$ are fixed in any particular scheme for specifying environments. If the scheme is changed, those values may also change. They should be supplied as parameters to the decision process. If full environment
specifications are made using the scheme outlined in section 3.3.2.2, the parameter values are \( w_z = 8, w_i = 7, w_r = 9, w_o = 30 \).

The threshold values \( T_1, T_2, T_3, \) and \( T_4 \) can be manipulated to "tune" the system. If they are high, changes of environment are restricted to those that make a substantial difference to total static or dynamic program size. Low threshold values lead to frequent environment changes, most of which only make a minor difference to total static and dynamic program size.

Experiments suggest that every change of environment that reduces the total static size of a program unit also reduces its total dynamic size. To admit all environment changes that improve both size measures, the static thresholds \( (T_1 \) and \( T_3) \) are therefore set at zero. The extent to which further changes are made to the program unit depends on the values of the dynamic thresholds.

If the dynamic thresholds are high, they only admit changes of environment that substantially reduce the total dynamic size of the program unit. Each change admitted by the dynamic threshold causes the total static size to increase (changes that decrease total static size have already been accepted by the static threshold), but the dynamic gain is sufficient to compensate for the static loss. More dynamic gains are accepted as the dynamic thresholds are lowered, which increases the overall reduction in total dynamic size; at the same time, more static losses are incurred. If the dynamic thresholds are too low, the extra dynamic gains accepted are too small to compensate for the extra static losses. A balance must be struck, whereby useful dynamic reductions are accepted without the total static size deteriorating too much. The optimal conditions are influenced by the relative importance of static code space and dynamic code space; and by the overheads in changing environments, since dynamic gains must compensate for the time spent changing environments as well as for static size increases.

Experiments indicate that optimal values for the dynamic thresholds are of the order of 2% of the average estimated total dynamic program unit size. For ICL2900
assembly code (see section 6.5.1.3), the optimal values are $T_2 = 800$ and $T_4 = 1350$ bits. At those settings, the mean ratio of

(dynamic bits gained) : (static bits lost),

which was calculated for each change admitted by the dynamic thresholds, is maximized at 128:1.

High dynamic thresholds provide some margin for error. The dynamic number of executions of each statement can only be estimated. If it is underestimated, the dynamic gain will be larger than predicted, and the change of environment is especially worthwhile (this is why underestimates of total dynamic size are *safe* errors). If it is overestimated, the dynamic gain will be less than expected and may even be too small to recover the cost of the environment change. The thresholds used ($T_2 = 800$, $T_4 = 1350$) are 3 to 5 times the average total cost of environment specification, making it unlikely that a predicted dynamic gain will turn out to be a real dynamic loss.

6.3.4. Summary: segmenting program units

A detailed algorithm for deciding when to change the environment during the execution of a program unit has been presented. A tree that represents possible segments within the program unit is needed, along with a prediction of the execution profile for the program unit. The tree is used to govern the order in which potential segments are considered; the profile is used, with reference to the code, during the consideration of each individual segment.

Changes of environment are made, on entry to and exit from a given segment of a program unit, if it appears that the result is a significant reduction in either or both of the total static size and total dynamic size of the program unit. *Significant* reductions are those that exceed certain thresholds; principles for assigning numerical values to those thresholds have been discussed in section 6.3.3.

The result, after each potential segments are considered, is a list of segments
that are to be encoded in their own local environments. The last entry represents the parts of the program unit that are not included in other segments; if no other segments are found, the list contains a single entry which indicates that the program unit is to be encoded as a whole.

6.4. Producing the binary form of the program unit

By the end of the tree traversal, a list of segments is constructed, and "Change-env" and "Revert-env" instructions are inserted at appropriate places in the image code. The list is used to generate absolute binary code for the image instructions of the program unit. The first step is initialisation of the Already_encoded flag of each instruction to 'false'; thereafter, the encoding algorithm is based on a traversal of the list.

Each entry in the list gives the following information about the segment it represents:

- the first and last instructions to be encoded in the local environment

- sets of operators, operands, and labels that must appear in the environment tables.

Each entry is considered in turn, beginning with the first. The procedure to encode the given segment is:

1. The elements in the set of operators are enumerated, in any order. Operands and labels are enumerated similarly.

2. Bits are generated to specify the environment (see pages 39-41). For each of operators, operands, and labels in turn, the actions performed are:

   1. Two bits are generated: *00* indicates full enumeration; *11* indicates no further action for this table.

   2. (Assuming full enumeration) m bits tell how many table entries are to be specified.
3. For each table entry, \( m \) bits are generated. The value is the index in the global table of the particular operator, operand, or label.

3. Each instruction in the segment is encoded:
   1. If its `Already_encoded` flag is set, the instruction is skipped.
   2. Four bits are generated to represent the format.
   3. The minimum possible number of bits is used to encode the operator. The value encoded is that assigned to the operator during the enumeration of step 1.
   4. Any named operands are encoded in the same way.
   5. The destination, if it is another named operand or label, is encoded in the same way.
   6. The resulting bit pattern is stored in a field of the instruction record.
   7. The `Already_encoded` flag is set to 'true'.

4. The bits for the environment definition are stored with those for the first instruction encoded within the segment.

When the list is completely traversed, binary code for each instruction and environment has been generated. The bit patterns are picked out from the instructions, producing the absolute binary code for the program unit.

Any labels within the program unit are now partially resolved in the global table. Previously the relevant entries in the global table held instruction numbers; they now hold bit offsets from the start of the binary code of the program unit.

If the program unit is the main program body, all of the encoded program units are gathered together now to produce the absolute binary code for the whole program. The actions are:

1. Labels are resolved to absolute bit addresses, now that the sizes of the global tables and all program units are known.
2. The start address of the main program body (actually the beginning of its environment specification) is encoded.

3. The global tables are encoded. The entries are descriptors for operands, control store addresses for operators, and bit addresses for labels. They are enumerated like any environment, except that the two control bits per table are not needed and a fixed number of bits is used for each entry.

4. The encoded program units are appended in turn to the binary code file that is being built.

The resulting binary file has the logical structure illustrated in Figure 3-5 (page 45).

6.5. Results

6.5.1. Testing strategy

6.5.1.1. Implementation of algorithms

The techniques described in this thesis are applied to a source program after it has been compiled and its image program produced. In the later stages of processing, the image program alone is used; in the first two stages (i.e. predicting the execution profile and building a syntax tree), information is also needed about the syntax of the source program. Direct reference could be made to the source code, or information about the source program could be embedded in the image program. The latter approach is adopted here: the compiler was modified to generate image code that includes dummy instructions, from which the syntax of the source program can be reconstructed. After the initial compilation, all analysis is done on the image program.

The procedure described in chapter 5 and sections 6.2 to 6.4 was implemented,
with one omission. Programs were written to perform the following analysis on each program unit in the image program:

1. Predict the dynamic execution profile (section 5.3.1). The simple model of program execution is used, as it is sufficiently accurate for this application.

2. Construct a tree representing the potential segments in the program unit:
   1. build a tree based on syntax (section 6.2.1)
   2. prune the tree (section 6.2.3).

3. Traverse the tree (section 6.3.1), visiting each node and deciding if it represents a segment worthy of separate encoding (section 6.3.2).

4. Encode the image instructions into binary form (section 6.4), using the results from the tree traversal.

5. Compute the total static size and estimated total dynamic size of the encoded program unit.

The process of modifying the tree (described in section 6.2.2) is not implemented; the traversed tree is based only on syntax. Environment changes that optimize the execution of local segments are found directly, but changes of locality are only found by coincidence.

*Local optimization* environment changes have a large effect on total dynamic size, mainly by optimizing loops, and a small effect on total static size. *Changing locality* environment changes would have less effect on total dynamic size; the effect on total static size, while more likely to be beneficial, would still be small. Local optimization changes are the most significant in determining the performance of the system. The current implementation, which finds the local optimization changes, should give a good indication of the magnitude of the size changes that can
be achieved with internal environment changes. A full implementation should show a slight further improvement; this is worthy of further research.

6.5.1.2. The test data

Pascal programs were collected from several sources to provide data on which to test the algorithms described. The resulting set comprised programs from textbooks on Pascal; programs from textbooks on data structures and algorithms; programs written by staff and students (including the author) of Computer Science at the Australian National University; and parts of a commercial Pascal compiler that was written in Pascal. Some programs were so large that the analysing programs ran out of memory while processing them, so they had to be deleted from the set of test programs. Also deleted were trivial programs, programs that duplicated others, and programs that included Goto-statements out of procedures. Twenty-nine programs remain in the data set, covering a broad range of applications.

The twenty-nine test programs contain a total of 83 program units. The smallest program units comprise 2 lines of source code (for a simple function or a main program body consisting only of procedure calls); the largest contains 72 lines. The mean length is 17 lines, the median 11, and the standard deviation 14.

6.5.1.3. Testing method

Provided the image instructions can be expressed as quadruples of the form

(operation; zero to two operands; destination),

and provided they include dummy quadruples that provide information about the source program's syntax, the details of the intermediate language are irrelevant. It does not matter what the actual operations and operands are - all that matters is that different operations and operands can be distinguished from each other. The analysis is independent of any particular intermediate language.

Two different intermediate languages were investigated. The test programs
were compiled by hand into the "ideal" intermediate language defined in Appendix A; and by a commercial compiler (modified to generate the necessary dummy quadruples) into assembly code for the ICL2000 computer. The two sets of image programs were then analysed. This enabled the effect of changing the intermediate code to be assessed. Incidentally, it enabled some conclusions to be drawn about the suitability of different types of intermediate languages for representing Pascal source programs.

The following paragraphs describe the testing performed on each set of image programs:

1. The total static and dynamic sizes that result if all program units are encoded as whole segments were found. This provided the base measure, for ascertaining the amount of advantage that can be gained by making internal changes of environment in program units.

2. Each test program was executed several times on a real machine, and an average dynamic profile was found for each program. The real profiles were used to compute the real total dynamic size of each program unit.

3. The set of image programs was then analysed several times, with different values used each time for the four thresholds mentioned in section 6.3.2. The optimal threshold settings, which led to the best average dynamic saving per static loss, were found. Those threshold settings were adopted, and the consequent encoding was found for each test program.

4. For those program units where some change was made, the total static sizes and estimated total dynamic sizes of the new encodings were computed. These sizes were compared with the original predicted sizes of those program units, and average changes were computed. In the remaining cases no worthwhile internal change of environment was found; the total static and dynamic sizes were then unchanged from the base measures.
5. The actual total dynamic size was also found for each program unit in which an internal change of environment was made. It was compared with the actual total dynamic size when no internal change was made, and again average changes were computed.

The information obtained is:

- how often it is best to encode program units as whole segments, and how often that can be improved upon;
- how much improvement is expected when internal changes of environment are made; and
- how much improvement is actually observed when internal changes of environment are made.

6.5.2. Predicted and observed results

The program units were translated into two intermediate languages - ICL2900 assembly language and an ideal intermediate language - and segmented according to the algorithms of section 6.3. The results obtained with ICL2900 assembly code are presented in section 6.5.2.1. Those obtained with *ideal* intermediate code are presented in section 6.5.2.2. The conclusions are summarized in section 6.5.2.3.

6.5.2.1. ICL assembly code

As an intermediate language in which to represent Pascal programs, ICL2900 assembly code is by no means ideal. Nevertheless, it is useful to apply these algorithms to ICL assembly language programs. It demonstrates that they can be applied competitively even to intermediate languages that were specifically designed for very different architectures; and it provides data for considering the effect of different intermediate languages.

When the test program units are compiled into ICL assembly code, and
analysed according to the procedure of sections 6.2 and 6.3, the following results are obtained:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of program units</td>
<td>= 83</td>
</tr>
<tr>
<td>Smallest program unit</td>
<td>= 2 instructions</td>
</tr>
<tr>
<td>Largest program unit</td>
<td>= 298 instructions</td>
</tr>
<tr>
<td>Mean length</td>
<td>= 60 instructions</td>
</tr>
<tr>
<td>Standard deviation of length</td>
<td>= 53 instructions</td>
</tr>
<tr>
<td>Median length</td>
<td>= 44 instructions</td>
</tr>
</tbody>
</table>

Internal environment changes are made in 13 of 29 programs, to 16 of 83 program units. A total of 30 internal environment changes are identified.

Static changes in program units (compared with sizes if no internal changes of environment are made):

- 8 increases in total static size
- 8 decreases in total static size
- Mean change = 2.9% increase in total static size
  (standard deviation = 5.6%)
- Largest increase in total static size = 14.0%
- Largest decrease in total static size = 3.9%

Predicted dynamic changes in program units (compared to sizes if no internal changes of environment are made):

- 0 increases in total dynamic size
- 16 decreases in total dynamic size
- Mean change = 10.1% decrease in total dynamic size
  (standard deviation = 7.6%)
- Largest decrease in total dynamic size = 29.9%
- Smallest decrease in total dynamic size = 0.9%
Observed dynamic changes in program units:
14 decreases in total dynamic size
2 increases in total dynamic size
Mean change = 10.0% decrease in total dynamic size
(standard deviation = 9.3%)
Largest decrease in total dynamic size = 28.8%
Largest increase in total dynamic size = 4.5%

The coefficient of correlation between predicted and observed total dynamic sizes is 0.83.

The predicted and observed dynamic behaviours are very similar. The mean changes are almost identical, and the correlation between predicted and observed sizes is high. This gives confidence in the model used to predict dynamic sizes.

Changes are only made in about one fifth of the program units, because most of them are small. Almost all of the larger program units are segmented: changes of environment are made in 11 of the 13 program units with more than 25 lines of source text, and 14 of the 23 program units with more than 20 lines of source text. These are quite small, which indicates that program units do not have to be very big or complex to benefit from internal segmentation for execution.

Lowering the dynamic thresholds enables more changes of context to be made. The maximum number of changes are made when the thresholds are set at zero: in that case, changes are made in about half of the program units. The average static loss increases from 3% to 15-25%, and the average dynamic gain increases from 10% to 15-20%.

When the thresholds are set at the optimal values, a small static size increase, of about 3% on average, can be expected in some of the program units to which these methods are applied. An average reduction of 10% of total dynamic size can be expected. Generally the larger program units are changed; since they tend to
dominate the total static and dynamic sizes of the programs in which they appear, an average static size increase of nearly 3% and an average dynamic size decrease of nearly 10% can be expected in those programs. The observed average values were a 2.7% static increase and an 8.2% dynamic decrease.

These figures are based on a small sample, and so may not be strictly accurate. However, they show that non-trivial savings in total dynamic size can be made at small static cost. Reducing the total dynamic size by 10% effectively reduces the memory bandwidth needed for decoding instructions by 10%. This is worth doing, as it reduces instruction decoding time and consequently execution time.

Full respecification of every table is made at every change of environment, so environment changes are as expensive as possible. Any more sophisticated scheme, which recognizes some common entries in the old and new tables and avoids respecifying them, would lessen the cost of environment changes. This could substantially reduce total static sizes, perhaps even making the average static change a reduction in size. Total dynamic sizes would also be reduced. Cheaper means of changing environments are investigated in chapter 7.

The best dynamic saving possible is about 35%. This occurs when a small loop with many iterations dominates the execution of a large program unit which contains numerous operators and operands. The number of bits per operator and operand in the loop drops from 5 or 6 to 3 or 4, and the average instruction length drops by 35% from 17 bits to 11. Such a situation arose in the tests: a reduction of 40% in the size of the instructions in a small loop was translated into a dynamic size reduction of 29% for the whole program unit.

Each of the two dynamic losses observed in the tests arises from an overestimate of loop iteration frequencies: only one iteration is made through a loop, when five are predicted. A small (in percentage terms) dynamic gain is predicted, but a small loss is actually made. In one of the two programs, another program unit is also segmented; the overall effect is to reduce the total dynamic program size. In the
other program, segmenting a program unit internally causes the total dynamic program size to increase. This is avoided if the complex model of program behaviour is used, as a better estimate of loop iteration frequency is made. Whether the extra work inherent in the complex model is worth while depends on the cost of making this sort of mistake. Raising the dynamic thresholds also prevents these mistakes, but some useful environment changes are then not made.

Assembly instructions are encoded into 24 bits each, on the ICL2900 computer. This is 1.5 times the average instruction size when the same assembly instructions are encoded for the context-sensitive architecture. Some of the advantage of the context-sensitive architecture is offset by the need to specify environments, which is unnecessary on the ICL2900. The total static size of each program unit encoded for the context-sensitive architecture, including internal changes of environment, is 3% greater on average than for the ICL2900; total dynamic size averages 18% less.

The context-sensitive approach produces encodings whose static and dynamic sizes are comparable to, and perhaps better than, the corresponding sizes on the architecture for which the intermediate language was designed. This happens even though that architecture is very different, and the intermediate language makes no use of some features of the context-sensitive architecture. If a more suitable intermediate language is used, the total static and dynamic sizes improve substantially.

6.5.2.2. *Ideal* intermediate language

When the program units are compiled into the ideal intermediate language defined in Appendix A, and analysed according to the procedure of sections 6.2 and 6.3, the following results are obtained:
Number of program units  =  83
Smallest program unit  =  2 instructions
Largest program unit  =  136 instructions
Mean length  =  28 instructions
Standard deviation of length  =  25 instructions
Median length  =  19 instructions

The intermediate program units are smaller in this intermediate language than in ICL assembly code. Potential savings are therefore smaller. Each saving must still compensate for time spent changing the environment, which is altered only slightly by the different intermediate language. The dynamic thresholds must remain high enough to ensure sufficient savings: although they can be reduced from their ICL levels, the proportional reduction is not as great as that in the number of instructions.

Consequently, some of the lesser gains that made it worth changing the environment in the ICL code are no longer sufficient in the ideal intermediate language. Thus less environment changes are made.

Internal environment changes are made in 10 of 29 programs, to 11 of 83 program units. A total of 18 internal environment changes were identified. This compares with 30 internal changes in 16 program units, in 13 programs, when ICL assembly code is used.

Larger program units are still segmented. The 11 program units in which internal environment changes are made are all among the 17 largest program units, with 25 or more lines of source text.

Among the environment changes that are not made in this intermediate language, but which are made in the ICL code, are the two changes that lead to increased total dynamic size. Every change of environment made in the test programs leads to a reduction in observed total dynamic size.
Static changes in program units (compared with sizes if no internal changes of environment are made):

- 9 increases in total static size
- 1 decrease in total static size
- 1 instance of no change in total static size

Mean change = 4.1% increase in total static size
(standard deviation = 3.5%)
Largest increase in total static size = 9.3%
Largest decrease in total static size = 0.1%

Largest increase in total static size = 3.3%

Predicted dynamic changes in program units (compared with sizes if no internal changes of environment are made):

- 11 decreases in total dynamic size

Mean change = 10.3% decrease in total dynamic size
(standard deviation = 7.6%)
Largest decrease in total dynamic size = 33.0%
Smallest decrease in total dynamic size = 4.9%

Observed dynamic changes in program units:

- 11 decreases in total dynamic size

Mean change = 10.4% decrease in total dynamic size
(standard deviation = 9.9%)
Largest decrease in total dynamic size = 33.0%
Smallest decrease in total dynamic size = 3.1%

The coefficient of correlation between predicted and observed total dynamic sizes is 0.77 (for ICL assembly code it is 0.83).
Comparing ICL intermediate language with ideal intermediate language:

- Mean number of static instructions in ideal language = 47%
- Mean number of dynamic ideal instructions predicted = 46%
- Mean static size of ideal program unit = 50%
- Mean predicted dynamic size of ideal program unit = 46%

Only half the space is needed to represent the image program in the ideal intermediate language instead of ICL assembly code (a similar observation was made by Flynn [Flynn 80]). There are half as many instructions, and the instructions contain fewer different operators and so are usually given smaller encodings. Environment changes are proportionally more expensive when the number and size of instructions is reduced, which is why the average reduction in total static size is not quite as great as the average reduction in the static number of instructions. This also explains the slightly greater average static loss, compared with ICL code, when the environment is changed.

Total dynamic size is determined mostly by instruction sizes, so the smaller instructions are more significant than the cost of specifying environments. This explains the greater average improvement in total dynamic size, compared with ICL code, when the environment is changed.

Overall, the changes predicted in static and dynamic size, when the environment changes, are remarkably similar to those predicted with ICL assembly code. This is true of both the average changes and the maximum possible changes. The predicted results are very similar to the observed results.

6.5.2.3. Summary of results

Segmenting program units and changing the environment during their execution can be seen to be beneficial to most program units with 25 or more lines of source text. Although total static size increases by an average of 3%, the total dynamic size is reduced by about an average of 10%. A reduction of up to about one third in total dynamic size is possible.
The nature of the intermediate language affects total program size. A program expressed in an ideal image language is half the size of the same program expressed in a conventional image language. The savings produced by changing environments are similar when different image languages are used.

The accurate prediction of dynamic savings indicates that the model of program execution performs well.

6.6. Summary

A method for analysing programs, to find the best way to encode them for execution on the context-sensitive architecture, has been presented in this chapter. The results of applying this method in practice have also been described.

If dynamic information is available from past execution traces of the program, it can be used to predict the execution profile. In general, all of the information used in analysing the program, including its predicted execution profile, must be derived from a static analysis of its source text.

The analysis of a program unit proceeds in two stages:

1. A tree is built, that represents the potential segments in the program unit:
   a. The program unit is parsed, and a tree that represents its syntax is built.
   b. The tree can be augmented, to include nodes that represent phases of constant locality. Suggestions on how to do this are given in section 6.2.2. These suggestions are not implemented here: further research is needed to ascertain the effect of adding this stage to the process.
   c. The tree is pruned, to reduce its size while retaining its useful information.
2. The tree is traversed, to see which nodes represent segments of the program unit that should be encoded in their own environments. When a node is visited, the effect of encoding it as a separate segment is assessed; if a significant reduction in either or both of total static size and total dynamic size is predicted, the node is encoded as a separate segment. The model of program execution developed in chapter 5 is used here.

The environment is set up on entry to every program unit; often this analysis identifies further points at which the environment should be altered.

This scheme has been tested in practice. Results have been presented for programs expressed in two intermediate languages. The following conclusions are drawn:

- For most program units with 25 or more lines of source text, it is useful to change the environment at least once during their execution.

- Total dynamic size is reduced by an average of 10%, at the expense of an average 3% increase in total static size.

- Total dynamic size can be reduced by up to one third.

- The intermediate language has a major effect on the number of instructions, and so on total program size. It has little effect on the proportional changes in static and dynamic size that result from changing the environment.

- Programs expressed in an ideal intermediate language, for execution on the context-sensitive architecture, are half the size (or less) of the same programs on conventional architectures. This is true of both total static size and total dynamic size.

- The accuracy of the predictions of average size changes indicates that the model of program execution produces accurate predictions for this application.
All of the results described so far have been achieved despite the use of a simple but costly method for changing environments. Better results, and wider applicability of internal environment changes, follow if a better method for specifying environments is used; that is the subject of the next chapter.
Chapter 7
Exploiting similarities between segments

7.1. Introduction

In this chapter, a second refinement of the original notion of encoding each program unit as a whole is explored.

The first refinement is to look for internal segments within program units. Each segment is treated independently: a full environment specification appears first, followed by the encoded instructions. Instruction sizes are reduced, but more work is done in specifying environments. Although the total static size of a program unit often increases slightly, the total dynamic size is reduced.

The second refinement is to consider the relationships between the internal segments. Sets of operators, operands, and labels that appear in two or more segments are constructed; it is often possible to avoid repeated specification of environment table entries for those sets. Full environment specifications are no longer needed in all segments. Instruction sizes are unchanged, and less work needs to be done in specifying environments. The total static size of a program unit is now generally reduced, and total dynamic size is reduced further.

Throughout this chapter, discussion focuses on specifying environment tables for operands. Operators or labels could equally well be used: they are each handled independently using the same algorithm.

The key observation is that the operands used in a segment need not necessarily
appear in a contiguous block in the environment table. Suppose a segment contains \( m \) operands; the minimum number of bits needed to identify each operand uniquely is \( n = \lceil \log_2 m \rceil \). Provided the \( m \) operands all appear within a contiguous block (or "window") of \( 2^n \) entries, minimal operand encodings can be used. There are \( v = 2^n - m \) "free entries" in the window, which can be distributed anywhere and can contain any operands.

In particular, the free entries, and other entries beyond the window boundaries, can contain operands used in another segment. If the table entries are suitably arranged, setting up the new environment on entry to the other segment only involves changing the size and/or position of the window. More work is done in specifying the table initially, because more entries are included, but the other environment change costs almost nothing. A saving, equal to the cost of respecifying the operands used in both segments, is made overall. If the cheap environment change occurs within a loop, the dynamic saving accumulates with every iteration.

In a few cases, in which the overlap between segments is almost complete, not even a window change is necessary. The table for the first segment can be used directly for the second segment. (A window change at least must be made to one or both of the other two tables, as otherwise there is no reason for a separate segment to be defined.) Section 7.2 describes an algorithm for finding these cases.

An algorithm for finding the best segmentation of a program unit, making maximum use of window changes in carefully ordered tables, is presented in section 7.3. This algorithm has been implemented, and tested in practice. The results are given in section 7.4, and the advantage gained from reducing the cost of environment changes (over and above that from segmenting program units) is discussed.

The chapter concludes with a summary in section 7.5.
7.2. No change to tables

Many worthwhile environment changes allow smaller encodings for each of operators, operands, and labels. In such cases it is advantageous to change all three tables.

Other environment changes can be worthwhile, even though no savings are made in one or more of operators, operands, and labels. For example, changing the environment might lead to smaller encodings of operands and labels, but not operators. It is then beneficial to change the operand and label tables, but no advantage is gained by changing the operator table. The effort to change the operator table would be wasted.

It is desirable to change only the tables from which advantage can be gained (compare this with chapter 6, where it is assumed that every table is fully specified in every environment change). This section explains when tables should be left unchanged, and how the binary encoding is altered from chapter 6.

The program unit is subjected to the analysis described in chapter 6. A list of segments to be encoded separately is produced; associated with each segment is a list of the required contents of the environment for that segment. The actions described below are performed, and the occasions when a table can be left unchanged are identified. The binary code is then generated.

1. Construct a tree that represents the chosen segmentation of the program unit. Each node represents one segment; children of a node represent subsegments.

2. Make a bottom-up traversal of the tree. When considering node $i$ find $R_i$, the set of operands used in the instructions in that segment. Ignore instructions contained in subsegments of node $i$.

For example, given the tree shown in Figure 7-1, the sets (in the order in which they are found) are:
Figure 7-1: Sample segmentation tree

\[
\begin{array}{c}
1 - 40 \\
6 - 10 \quad 11 - 20 \\
14 - 18
\end{array}
\]

\[
R_2 = \{ \text{operands in instructions 6-10} \}
\]
\[
R_4 = \{ \text{operands in instructions 14-18} \}
\]
\[
R_3 = \{ \text{operands in instructions 11-13, 19-20} \}
\]
\[
R_1 = \{ \text{operands in instructions 1-5, 21-40} \}
\]

3. Traverse the tree once more, again from the bottom up. Let node \(i\) be the node under consideration; designate its parent node \(j\). The actions performed are:

\[
\begin{align*}
&\text{begin} \\
&\quad \text{Node}_i.\text{full_spec_needed} := \text{true}; \\
&\quad \text{If } \text{Node}_i \neq \text{Root} \text{ then} \\
&\quad \quad \text{begin} \\
&\quad \quad \quad R_p := R_i \cup R_j; \\
&\quad \quad \quad \text{If } [\log_2 r_p] = [\log_2 r_i] = [\log_2 r_j] \text{ then} \\
&\quad \quad \quad \quad \text{begin} \\
&\quad \quad \quad \quad \quad \text{Node}_i.\text{full_spec_needed} := \text{false}; \\
&\quad \quad \quad \quad \quad R_j := R_p \\
&\quad \quad \quad \quad \end{align*}
\]

\[
\text{end}
\]

If operand encodings are the same size in nodes \(i\) and \(j\) regardless of whether the table is redefined on entry to node \(i\), no redefinition is needed. The table that was set up for \(j\) is also used for \(i\). To ensure that all needed operands are present, the operands needed in node \(i\) are added to the set of operands that must be in the table for node \(j\).
4. When the traversal is complete; a flag in each node indicates whether or not the table must be redefined on entry to that node. In nodes that do require table redefinition, the associated sets \( R_i \) identify the entries to be placed in the table.

A top-down traversal is made over the tree, to define the binary values to be encoded for each operand in each segment. When a node that requires table definition is visited, the entries are enumerated (in any order) and assigned minimal-width binary codes. When a node that needs no table redefinition is visited, the list of operands and associated binary codes is copied from its parent.

When this process is finished, the binary values to be used for each operand in each instruction are defined, and the times when no change to the operand table is needed are identified. The process is repeated for operators and labels; the binary encoding is then generated for the program unit, using the method described in section 6.4.

The binary code thus produced differs from that produced previously in two respects:

- the range of the \( r_i \) binary codes in a given node is not necessarily \( 0 .. (r_i-1) \). Instead, the maximum range is \( 0 .. (2^n-1) \), where \( n = \lfloor \log_2 r_i \rfloor \).

- some environment changes may be encoded with no change to one or more tables.

Whenever no change is made to a table on entry to a segment, the saving is proportional to the overlap between the environments of that segment and the one whose table is used. Things that appear in both environments are only specified once.

By reducing the cost of some environment changes, the total static and dynamic
sizes of the program unit are reduced. However, the percentage improvement in average total size is likely to be small, as most environment changes are likely to still require full redefinition of tables.

Environment changes are still more costly than they need to be, for two reasons:

- they all still have full cost or zero cost: no consideration has yet been given to changing windows on the tables.
- the possibility of cheap environment changes is not considered until after the segmentation has been decided. Deliberately seeking a segmentation that allows cheap environment changes could have a substantial impact on total size.

The next section describes a technique that addresses these problems.

7.3. Changing windows on the environment tables

7.3.1. Introduction

It is rarely possible to leave a table completely unchanged on entry to a new segment. If the cost of environment changes is to be reduced significantly, then, the ability to alter windows on tables must be exploited. Changing a window costs little more than making no change at all, and is applicable far more often.

When a segment is entered and a new environment table defined, the contents of the table are determined by the entries needed for that segment. Consideration should also be given to the entries needed by each of its subsegments: often the initial table can be set up so that to create the new environments on entry to some of the subsegments one only needs to change the window. To achieve this, the table entries must be listed in a carefully chosen order.

The cost of environment changes has significant impact on the total static and
dynamic sizes of the program unit. Whether or not a node is worth encoding as a separate segment can be influenced by the cost of the associated environment change. Thus the possibility of cheap environment changes is considered as part of the process of deciding how to segment the program unit. (This contrasts with section 7.2, where cheap environment changes are only sought after the segmentation has been decided.)

Different combinations of segments and subsegments are considered, to see when the tables can be arranged to make window changes effective. For each successful combination, the predicted total static and dynamic sizes resulting from the implied segmentation of the program unit are computed. The best segmentation is chosen for implementation.

Many nodes in the initial tree represent portions of the program unit that are not worth encoding as separate segments under any circumstances; such nodes are not considered in the combinations. Enough nodes often remain for exhaustive checking of all combinations to be prohibitively expensive. That effort is reduced if the potentially most useful combinations are checked first, so that the best segmentation is found as quickly as possible.

Section 7.3.2 describes the classification of nodes according to their likelihood of being encoded as separate segments. Those with no chance are eliminated, and the rest are classified into three levels. This classification helps in the selection of node combinations to try first; this is the subject of section 7.3.3. The procedure for checking a given combination is described in section 7.3.4. Once the best combinations are found, the implied segmentation is implemented, as explained in section 7.3.5. The whole procedure is summarized in section 7.3.6.
7.3.2. Classification of nodes

To reduce the number of node combinations that need to be checked in the search for the best segmentation, the tree is reduced. Only nodes likely to be encoded as separate segments are retained. So that combinations of the most useful nodes can be considered first, the nodes are classified according to their likelihood of being encoded as separate segments. This section describes a scheme that eliminates useless nodes, and classifies the remaining nodes into three levels.

As in section 6.3, the algorithm is based on a bottom-up traversal of the original tree. When a node is visited, the method of section 6.3.2 is used to compute the static and dynamic savings in instruction encodings that follow if the node is encoded as a separate segment:

\[ \text{Tot}_S = S_N + S_p \]
\[ \text{is the total static saving in the instructions of the node and its parent;} \]

\[ \text{Tot}_D = D_N + D_p \]
\[ \text{is the total dynamic saving predicted for the instructions of the node and its parent;} \]

The net cost of changing the environment on entry to the node is computed. It is assumed that the change is cheap on entry to the node, but that setting up the tables for the node's parent is more expensive, because entries needed in the current node must be included in the earlier set-up.

\[ \text{Cost}_{SC} \]
\[ \text{is the net static cost of the cheap environment change.} \]

\[ \text{Cost}_{DC} \]
\[ \text{is the net dynamic cost of the cheap environment change.} \]

Maximal savings are made if the node is encoded separately and the change of environment on entry to the node is cheap. These optimal static and dynamic savings are computed:

\[ \text{Opt}_S = \text{Tot}_S - \text{Cost}_{SC} \]
\[ \text{is the optimal static gain;} \]
\[ Opt_D = Tot_D - Cost_{DC} \] is the optimal dynamic gain.

The four values \( Tot_S, Tot_D, Opt_S, Opt_D \) are compared with thresholds, to assess the usefulness of the node. An entry is made in one of four lists, to record the assessment.

When changing the environment is cheap, static gains are common. Dynamic thresholds can be lowered from the levels needed when environment changes are always expensive, since it is rarely necessary to compensate for a static loss.

In chapter 6 it was noted that the effect of changing environments within program units was the same for two quite different intermediate languages; it is hereafter assumed to be independent of the intermediate language. Only one intermediate language is considered now.

The intermediate language considered is ICL assembly code. The threshold values here should be compared with the ICL threshold values of section 6.3.3.

The following scheme was developed empirically. Although slightly different threshold values may be appropriate for a different set of test programs, they should not alter much from those quoted below.

1. "Useless" nodes are those for which no change of environment is made under any circumstances; they are certainly not encoded as separate segments. A node is rejected as useless in any of the following circumstances:
   1. no gains to be made:
      \[ Opt_S <= 0 \text{ and } Opt_D <= 0 \]
   2. static loss too great:
      \[ Opt_S <= -300 \]
   3. static loss exceeds dynamic gain:
      \[ Opt_S <= 0 \text{ and } |Opt_S| > |Opt_D| \]
   4. gains are insufficient: none of the following four thresholds are passed:
If a node is not rejected at this stage, it is placed in one of three lists of useful nodes.

2. Nodes with large predicted gains will be encoded as separate segments, even if a complete environment specification is required. A node is entered into the list of large gains if it passes one or more of the following four thresholds:

\[ \text{Tots} \geq 400 \quad \text{or} \quad \text{Opts} \geq 200 \quad \text{or} \]
\[ \text{Tot}_D \geq 800 \quad \text{or} \quad \text{Opt}_D \geq 600 \]

On average about 40% of useful nodes are placed on the list of large gains.

When a node is placed on the list of large gains, it is known that it will be encoded as a separate segment. Its constituent instructions, operators, operands, and labels are no longer considered when assessing savings in the rest of the program unit. By definition, the root node is included in the list of nodes that are certain to have an environment definition.

3. Nodes with moderate gains will be encoded as separate segments, provided the environment change on entry to such a node is not a complete specification (i.e. it shares the environment tables with at least one other node). A node is considered to have moderate gains if it is not on the list of large gains, and passes one or more of the following four thresholds:

\[ \text{Tots} \geq 100 \quad \text{or} \quad \text{Opts} \geq 30 \quad \text{or} \]
\[ \text{Tot}_D \geq 300 \quad \text{or} \quad \text{Opt}_D \geq 250 \]

On average, about 33% of useful nodes are placed on the list of moderate gains.
4. All other nodes have small gains. They will only be encoded as separate segments if they can share their environment tables with least two other nodes.

If a node is predicted to give moderate or small gains, it may or may not be encoded separately. Under the assumption that it will not, its constituent instructions must be considered when savings for other nodes are computed. The same gains can then be counted twice or more; two or more nodes can appear to be useful, when only one of them should be encoded as a separate segment. This situation is recognized when more than a certain amount of the predicted gains for a node are included in the predicted gains of another node, which is already on the list of moderate or small gains.

Sets of *mutually exclusive segments* are constructed. Each node in a set represents a potentially useful segment, but only one node from each set can be encoded as a separate segment. These sets are constructed incrementally:

- If a node's predicted gains have enough in common with the predicted gains of a node that is already in either the list of moderate gains or the list of small gains, it is entered in the same set as the earlier node. Otherwise, a new set is started.

- If the new node is certain to be encoded as a separate segment, and it is added to an existing set of mutually exclusive segments, then all other nodes in that set are removed from the set and the lists of potentially useful segments.

At the end of the tree traversal, a check should be made to see if either of two extreme situations is current:

1. If only one internal node was found to be useful, there is only one possible internal environment change. Although it could certainly be made cheaply, the savings may not make it worthwhile. Thus if the node fails to pass any of these four thresholds:
\[ Tot_s \geq 100 \quad \text{or} \quad Opt_s \geq 30 \quad \text{or} \]
\[ Tot_d \geq 600 \quad \text{or} \quad Opt_d \geq 500 \]

it is rejected; no internal environment changes are made.

2. If there are more than a given number of nodes with high or moderate predicted gains (the threshold used was eight such nodes), the list of nodes with small gains is discarded. This keeps the number of combinations to be checked at a manageable level, while restricting attention to the nodes that are most likely to produce significant savings if they are encoded as separate segments.

The following information results from the processing described here:

- All nodes with a reasonable likelihood of being encoded as separate segments are represented in a tree.

- These nodes are classified into three lists:

  1. Those that will certainly be encoded as separate segments, because large savings will be made.

  2. Those that will give moderate savings if encoded as separate segments. Most of them will be encoded separately. Such a node is not encoded separately if it cannot share tables with at least one other node; this only happens when a better encoding of the program unit is found that does not include that node.

  3. Those that will give small savings if encoded as separate segments. Such a node is only encoded separately if it can share tables with at least two other nodes. If there are enough nodes in the first two lists, this list is ignored.

- The nodes are grouped into sets of mutually exclusive segments: at most one node from each set can be encoded as a separate segment.
This information is used to limit the work required to find the best segmentation of the program unit, as explained in the next section.

7.3.3. Grouping nodes

The processing described in the previous section produces a small tree, which reflects the division of the program unit into potential segments and subsegments. The next goal is to determine which potential segments to encode as separate segments. Where possible, environment changes should be performed by just altering windows instead of respecifying whole tables.

Each table is fully specified on entry to the program unit. Ideally, the table entries should be specified in an order that enables all internal environment changes to be effected by just altering windows. This ideal cannot always be achieved.

Instead, the aim is to set up the initial tables to minimize the number of internal environment changes that require full table respecification. Whenever a table does have to be respecified at the beginning of a segment, its entries are ordered to minimize the number of subsegments that need full table respecifications. As many internal environment changes as possible are thus achieved with window changes.

The following procedure is applied independently to operators, operands, and labels:

1. Some combination of \( n \) nodes is chosen from the tree, subject to the following constraints:

   \( 2 \leq n \leq G_{\text{MAX}} \). \( G_{\text{MAX}} \) is the largest number of nodes for which it is feasible to seek an optimal ordering of table entries.

   One of the nodes must represent a segment of which all the others are subsegments: call it the *local root*.

   no more than one node is included from each set of mutually exclusive nodes.
• If B is any node that will certainly be a separate segment, A is any ancestor of B, and C is any descendant of B, then no combination can include A and C unless B is also included.

2. Between them the n nodes contain N operands. An ordering of the N operands is "satisfactory" to a node that contains m operands if those m operands all appear within a contiguous block of $2^x$ entries, where $x = \lfloor \log_2 m \rfloor$. An "optimal" ordering satisfies all n nodes.

The procedure described in section 7.3.4 is used in an attempt to try to find an optimal table ordering for the chosen n nodes.

3. If no optimal ordering is found, that combination of nodes is rejected. Otherwise, under the assumption that the table is set up appropriately on entry to the local root (so that entry to the other (n-1) nodes only needs window changes), the resulting total static and dynamic program unit sizes are computed.

This procedure is repeated for different combinations of nodes, until it is decided that no untested combination could improve on the best results so far. The best combination is then implemented:

- the optimal table ordering is recorded in the local root;

- for each of the n nodes:
  - the required window placement and size is recorded in the node
  - it is marked as representing a separate segment
  - the set of mutually exclusive nodes from which it was drawn is erased, and all its elements are removed from the lists of useful nodes.

Further environment changes might be made in any of the nodes that remain in the lists of potentially useful nodes. These nodes constitute one or more disjoint
subtrees, of one or more nodes each, from the original tree. This whole procedure is
applied recursively to each subtree of two or more nodes, until all remaining subtrees
are single nodes. Those that definitely represent separate segments are retained, and
the requisite table ordering and window placement is recorded in them; the rest are
discarded.

Exhaustive checking of all combinations of nodes is guaranteed to produce the
best possible encoding, but is prohibitively time-consuming. It is preferable to
consider the combinations in groups, whereby any successful combination from
group $i$ should give better results than any successful combination from group ($i+1$).
The combinations in group 1 are checked first; group ($i+1$) is only considered if all
combinations in group $i$ are unsuccessful.

The best results come from a successful combination of $G_{\text{MAX}}$ nodes, all of which
are predicted to give large or moderate gains. Next best is any other combination of
$G_{\text{MAX}}$ nodes, or a combination of ($G_{\text{MAX}}-1$) nodes that are all predicted to give large
or moderate gains. The third group consists of any other combinations of ($G_{\text{MAX}}-1$)
nodes, and all combinations of ($G_{\text{MAX}}-2$) nodes that are all predicted to give large or
moderate gains; etc. The final group contains all combinations of two nodes
predicted to give large or moderate gains: any combination in this group is
guaranteed to succeed.

It is always possible to find an optimal ordering for two nodes; it is usually
possible for three nodes; it is possible about half the time for four nodes. As the
number of nodes increases, the chance of success drops and the work involved
increases; no attempt is made here to find optimal orderings for more than four
nodes. Thus $G_{\text{MAX}} = 4$. 

7.3.4. Ordering the entries in environment tables

An algorithm for finding optimal orderings should arrive at its decision quickly. If no optimal ordering is possible, this should also be determined quickly. Such an algorithm is presented now, for finding optimal table orderings to satisfy groups of two, three, or four nodes.

7.3.4.1. Two nodes

It is always possible to find an optimal ordering that satisfies two nodes. Let the two nodes be designated \( n_1 \) and \( n_2 \), and let \( n_1 \) be an ancestor of \( n_2 \). Construct the following three sets:

- \( A = \{ \text{all operands used in } n_1, \text{ but not in } n_2 \} \)
- \( B = \{ \text{all operands used in } n_2, \text{ but not elsewhere in } n_1 \} \)
- \( C = \{ \text{all operands used in } n_2 \text{ and elsewhere in } n_1 \} \)

The operands within each of these sets can be listed in any order.

Placing these sets in the operand table in the order \( ACB \) satisfies both nodes: all of the operands used within \( n_1 \) are listed in sets \( A \) and \( C \), which appear as one contiguous block in the table, so this ordering is satisfactory to node \( n_1 \); similarly, sets \( C \) and \( B \) appear as one contiguous block, so this ordering is satisfactory to node \( n_2 \).

On entry to node \( n_1 \), a full environment specification is performed. The window is then set up to cover sets \( A \) and \( C \). On entry to node \( n_2 \), the window is moved so that its first entry coincides with the first element in set \( C \). This is illustrated in Figure 7-2.

The only work involved in finding this optimal ordering is in the construction of the sets \( A, B, \) and \( C \). Once that is done, the optimal ordering can be written down immediately.
Figure 7-2: Window change, in optimal table for two nodes

(a) Initial position of window, on entry to $n_1$

(b) Changed position of window, on entry to $n_2$

7.3.4.2. Three nodes

It is usually possible to find an optimal ordering for the operands used in three nodes.

Let the three nodes be designated $n_1$, $n_2$, and $n_3$. Suppose $n_1$ is the ancestor of the other two: then the full environment specification is made on entry to $n_1$, and window changes only will be needed on entry to $n_2$ and $n_3$ if an optimal ordering is found. Construct the following seven sets:
\[ A = \{ \text{operands used in } n_1 \text{ only} \} \]
\[ B = \{ \text{operands used in } n_2 \text{ only} \} \]
\[ C = \{ \text{operands used in } n_3 \text{ only} \} \]
\[ D = \{ \text{operands used in } n_1 \text{ and } n_2, \text{ but not } n_3 \} \]
\[ E = \{ \text{operands used in } n_1 \text{ and } n_3, \text{ but not } n_2 \} \]
\[ F = \{ \text{operands used in } n_2 \text{ and } n_3, \text{ but not } n_1 \} \]
\[ G = \{ \text{operands used in } n_1 \text{ and } n_2 \text{ and } n_3 \} \]

Let \( a \ldots g \) represent the cardinalities of those sets.

Within each set, the elements can be listed in any order. The seven sets are placed in the environment table; a permutation of the sets that produces an optimal ordering is sought.

There are \( 7! = 5040 \) different permutations in which the sets \( A \ldots G \) could be listed. This can be reduced to 12 that are worth checking, using the following principles:

- since \( G \) is needed in all three windows, place it in the middle.

- the two ends should be occupied by two of \( A, B, C \).

- if \( A \) and \( B \) are the sets on the ends, then \( E \) should be in the same half as \( A \) while \( F \) should be in the same half as \( B \).

- if \( A \) and \( C \) are the sets on the ends, then \( D \) should be in the same half as \( A \) while \( F \) should be in the same half as \( C \).

- if \( B \) and \( C \) are the sets on the ends, then \( D \) should be in the same half as \( B \) while \( E \) should be in the same half as \( C \).

- symmetry can be considered, to keep only one of each group of symmetric orderings.

The twelve orderings are listed in Table 7-1.
Table 7-1: Potentially optimal table orderings for three nodes

<table>
<thead>
<tr>
<th>No.</th>
<th>Ordering</th>
<th>Conditions for Optimality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A E C G F D B</td>
<td>( c+f=v_1 )</td>
</tr>
<tr>
<td>2</td>
<td>A D E G C F B</td>
<td>( c+e&lt;=v_2 )</td>
</tr>
<tr>
<td>3</td>
<td>A E C G D F B</td>
<td>( c&lt;=v_1 ) and ( d&lt;=v_3 )</td>
</tr>
<tr>
<td>4</td>
<td>A E D G C F B</td>
<td>( c&lt;=v_2 ) and ( d&lt;=v_3 )</td>
</tr>
<tr>
<td>5</td>
<td>A D B G F E C</td>
<td>( b+f=v_1 )</td>
</tr>
<tr>
<td>6</td>
<td>A E D G B F C</td>
<td>( b+d&lt;=v_3 )</td>
</tr>
<tr>
<td>7</td>
<td>A B D G E F C</td>
<td>( b&lt;=v_1 ) and ( e&lt;=v_2 )</td>
</tr>
<tr>
<td>8</td>
<td>A D E G B F C</td>
<td>( e&lt;=v_2 ) and ( b&lt;=v_3 )</td>
</tr>
<tr>
<td>9</td>
<td>B F D G E A C</td>
<td>( a+d&lt;=v_3 )</td>
</tr>
<tr>
<td>10</td>
<td>B D A G E F C</td>
<td>( a+e&lt;=v_2 )</td>
</tr>
<tr>
<td>11</td>
<td>B D F G E A C</td>
<td>( f=v_1 ) and ( a&lt;=v_3 )</td>
</tr>
<tr>
<td>12</td>
<td>B D A G F E C</td>
<td>( f=v_1 ) and ( a&lt;=v_2 )</td>
</tr>
</tbody>
</table>

The procedure for finding an optimal ordering is very simple:

1. Define

   \[ r_1 = a + d + e + g \]

   \[ b_1 = \lceil \log_2 r_1 \rceil \]

   \( r_1 \) is the number of operands used in node \( n_1 \). A window of size \( 2^{b_1} \) entries is needed to cover all of those operands. The number of *available* slots in the window is

   \[ v_1 = 2^{b_1} - r_1 \]

   Similarly, define
\[ r_2 = b + d + f + g \]
\[ b_2 = \lceil \log_2 r_2 \rceil \]
\[ v_2 = 2b_2 - r_2 \]
\[ r_3 = c + e + f + g \]
\[ b_3 = \lceil \log_2 r_3 \rceil \]
\[ v_3 = 2b_3 - r_3 \]

\[ v_f, v_e, \text{ and } v_g \text{ constrain the number of extraneous operands that can appear in each window, if the ordering is to be optimal.} \]

2. The right-hand column in Table 7-1 indicates the conditions that must be satisfied for each of the twelve orderings to be optimal. Check them each in turn until one is found whose conditions are satisfied, at which time the search for an optimal ordering ends successfully.

3. If none of the twelve orderings is successful, assume that no optimal ordering can be found.

An optimal ordering is found almost every time, after checking an average of six different orderings. Failure is scarcely more costly to recognize than success, since only another six orderings are checked and checking each ordering is such a simple process. This algorithm meets its goals: it is both efficient and effective.

7.3.4.3. Ranking the orderings

For any three given nodes, the twelve orderings can be ranked in the order in which they are likely to succeed. The first ordering checked is then nearly always successful.

Checking the twelve orderings in turn is so simple that it is not worth the effort to rank them, even though the extra work needed is small. The method for ranking the orderings is explained, however, because it can easily be extended to handle four or more nodes. With more than three nodes, it is more efficient to rank the orderings than to check them in turn.
The method for ranking the orderings is:

1. As before, find the values of $r_I, b_I, v_I, r_D, b_D, v_D, r_S, b_S, v_S$.

2. List the nodes in decreasing order of tightness of constraint. Consider the orderings that add no extraneous operands to the window with the tightest constraint; this restricts attention to six of the twelve orderings.

For example, suppose $v_I$ is the tightest constraint. The orderings to check are those where $A, D, E,$ and $G$ appear in a contiguous block: these are orderings $2, 4, 6, 8, 9, 10$ in Table 7-1. The other six orderings are assumed to fail.

If two nodes share the tightest constraint, test the ten orderings in which at least one of those windows includes no extraneous operands. If all three constraints are equally tight, all twelve orderings may need to be checked.

The next tightest constraint is used to rank the orderings within pairs. For example, suppose that, after $v_I$, $v_2$ is the next tightest constraint. Ordering 4 is more likely to succeed than ordering 2, and so should be tested before ordering 2; ordering 6 should be checked before 8; and 9 should be checked before 10.

3. The sequence in which these pairs of orderings should be checked depends on the sizes of the sets $A, B,$ and $C$. The two largest should be tried on the ends first; then the largest and smallest; finally the two smallest.

For example, suppose $a >= b >= c$; as before, assume $v_I$ is the tightest constraint and $v_S$ the loosest. The first orderings to try have $A$ and $B$ on the ends: thus the first ordering to check is 4, followed by 2. The sequence in which other orderings should be checked if these fail is 6, 8, 9, 10.
Table 7-2: Sequence for trying potentially optimal orderings for three nodes

<table>
<thead>
<tr>
<th>Sets in decreasing size order</th>
<th>Constraints, in decreasing order of <em>tightness</em></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( v_1 \ v_2 \ v_3 )</td>
</tr>
<tr>
<td></td>
<td>( v_1 \ v_3 \ v_2 )</td>
</tr>
<tr>
<td></td>
<td>( v_2 \ v_1 \ v_3 )</td>
</tr>
<tr>
<td>( abc )</td>
<td>( 4,2,6,8,9,10 )</td>
</tr>
<tr>
<td>( acb )</td>
<td>( 6,8,4,2,9,10 )</td>
</tr>
<tr>
<td>( bac )</td>
<td>( 4,2,9,10,6,8 )</td>
</tr>
<tr>
<td>( bca )</td>
<td>( 9,10,4,2,6,8 )</td>
</tr>
<tr>
<td>( cab )</td>
<td>( 6,8,9,10,4,2 )</td>
</tr>
<tr>
<td>( cba )</td>
<td>( 9,10,6,8,4,2 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sets in decreasing size order</th>
<th>Constraints, in decreasing order of <em>tightness</em></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( v_1 \ v_2 \ v_3 )</td>
</tr>
<tr>
<td></td>
<td>( v_1 \ v_3 \ v_2 )</td>
</tr>
<tr>
<td></td>
<td>( v_2 \ v_1 \ v_3 )</td>
</tr>
<tr>
<td>( abc )</td>
<td>( 1,3,5,6,11,9 )</td>
</tr>
<tr>
<td>( acb )</td>
<td>( 5,6,1,3,11,9 )</td>
</tr>
<tr>
<td>( bac )</td>
<td>( 1,3,11,9,5,6 )</td>
</tr>
<tr>
<td>( bca )</td>
<td>( 11,9,1,3,5,6 )</td>
</tr>
<tr>
<td>( cab )</td>
<td>( 5,6,11,9,1,3 )</td>
</tr>
<tr>
<td>( cba )</td>
<td>( 11,9,5,6,1,3 )</td>
</tr>
</tbody>
</table>
There are six permutations in which the sets $A, B, C$ may be listed in decreasing order of size, and six permutations in which the nodes $n_1, n_2, n_3$ may be listed in decreasing order of tightness of constraint. Table 7-2 shows which orderings to check, and in what sequence, for each permutation of sizes and constraints. In practice, the first ordering checked is almost always successful. If none of the indicated orderings is successful, it is assumed that no optimal ordering exists for those three nodes.

### 7.3.4.4. Four nodes

The method for finding an optimal ordering that satisfies four nodes is similar to that for three nodes. There are more sets of operands to place in the operand table, and the number of orderings to check is greater, but the same method is used.

Designate the four nodes $n_1$ to $n_4$. Construct the following sets:

- $A = \{ \text{operands used only in } n_1 \}$
- $B = \{ \text{operands used only in } n_2 \}$
- $C = \{ \text{operands used only in } n_3 \}$
- $D = \{ \text{operands used only in } n_4 \}$
- $E = \{ \text{operands used only in } n_1 \text{ and } n_2 \}$
- $F = \{ \text{operands used only in } n_1 \text{ and } n_3 \}$
- $G = \{ \text{operands used only in } n_1 \text{ and } n_4 \}$
- $H = \{ \text{operands used only in } n_2 \text{ and } n_3 \}$
- $I = \{ \text{operands used only in } n_2 \text{ and } n_4 \}$
- $J = \{ \text{operands used only in } n_3 \text{ and } n_4 \}$
- $K = \{ \text{operands used only in } n_1 \text{ and } n_2 \text{ and } n_3 \}$
- $L = \{ \text{operands used only in } n_1 \text{ and } n_2 \text{ and } n_4 \}$
- $M = \{ \text{operands used only in } n_1 \text{ and } n_3 \text{ and } n_4 \}$
- $N = \{ \text{operands used only in } n_2 \text{ and } n_3 \text{ and } n_4 \}$
- $P = \{ \text{operands used in all of } n_1 \text{ to } n_4 \}$

Let $a..p$ represent the cardinalities of these sets.
The number of operands used in each node is
\[ r_1 = a + e + f + g + k + l + m + p \]
\[ r_2 = b + e + h + i + k + l + n + p \]
\[ r_3 = c + e + h + j + k + m + n + p \]
\[ r_4 = d + g + i + j + l + m + n + p \]

The number of available slots in the window for node \( n_i \) is
\[ v_i = 2^{b_i} - r_i \quad \text{where} \quad b_i = \lceil \log_2 r_i \rceil \]

There are 15! permutations in which the 15 sets may be placed in the table. By considering symmetry, and grouping sets together so that there are always two nodes whose operands appear in contiguous blocks, the number of different orderings to check is reduced to 48. These 48 orderings are listed in Table 7-3; the right hand column indicates the conditions that must be satisfied for each ordering to be optimal.

If the orderings are checked in turn, an average of 24 orderings are checked before a successful ordering is found. Failure is not recognized until all 48 have been checked. Experiments indicate that a successful ordering is found in slightly less than half of the attempts. Thus the average number of orderings checked would be about 36-40. This is too high: the orderings should be ranked in order of likelihood of success. Success or failure will then be recognized more quickly.

The method is similar to that for three nodes:

1. Construct the sets \( A \ldots P \).
2. Compute \( r_1 \) to \( r_4 \), and \( v_1 \) to \( v_4 \).
3. List the four constraints in decreasing order of tightness. If the middle two are equal, build a list of all constraint orderings that may need to be checked.
### Table 7-3: Potentially optimal table orderings for four nodes

<table>
<thead>
<tr>
<th>No.</th>
<th>Ordering</th>
<th>Conditions for Optimality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$AGFTMELPBHINJDC$</td>
<td>$b+d+e+i+l \leq v_8$ and $b+e+f+h+k \leq v_4$</td>
</tr>
<tr>
<td>2</td>
<td>$AGFTMELPBHINCJD$</td>
<td>$b+e+i+l \leq v_8$ and $b+c+e+f+h+k \leq v_4$</td>
</tr>
<tr>
<td>3</td>
<td>$BINNEKLPATGMDJC$</td>
<td>$a+d+e+g+l \leq v_8$ and $a+e+f+h+k \leq v_4$</td>
</tr>
<tr>
<td>4</td>
<td>$BINNEKLPATGMCJD$</td>
<td>$a+e+g+l \leq v_8$ and $a+c+e+f+h+k \leq v_4$</td>
</tr>
<tr>
<td>5</td>
<td>$AGFTMELPBHINJDC$</td>
<td>$b+d+e+g+i+l \leq v_8$ and $b+e+h+k \leq v_4$</td>
</tr>
<tr>
<td>6</td>
<td>$AGFTMELPBHINCJD$</td>
<td>$b+e+g+i+l \leq v_8$ and $b+c+e+h+k \leq v_4$</td>
</tr>
<tr>
<td>7</td>
<td>$BHINEKLPATGMDJC$</td>
<td>$a+d+e+g+i+l \leq v_8$ and $a+e+f+k \leq v_4$</td>
</tr>
<tr>
<td>8</td>
<td>$BHINEKLPATGMCJD$</td>
<td>$a+e+g+i+l \leq v_8$ and $a+c+e+f+k \leq v_4$</td>
</tr>
<tr>
<td>9</td>
<td>$AGELMFKPMNCJIDB$</td>
<td>$c+d+f+j+m \leq v_2$ and $c+e+f+h+k \leq v_4$</td>
</tr>
<tr>
<td>10</td>
<td>$AGELFKMPCJHNBD$</td>
<td>$c+f+j+m \leq v_2$ and $b+c+e+f+h+k \leq v_4$</td>
</tr>
<tr>
<td>11</td>
<td>$BDIAEGLPFKMNJHC$</td>
<td>$a+d+f+g+m \leq v_2$ and $a+e+f+h+k \leq v_4$</td>
</tr>
<tr>
<td>12</td>
<td>$CJHNFKMAPAEGLBD$</td>
<td>$a+f+g+m \leq v_2$ and $a+b+e+f+k \leq v_4$</td>
</tr>
<tr>
<td>13</td>
<td>$AEGLMFKPMNCJIDB$</td>
<td>$c+d+f+g+j+m \leq v_2$ and $c+f+h+k \leq v_4$</td>
</tr>
<tr>
<td>14</td>
<td>$AEGLFKMPCJHNBD$</td>
<td>$c+f+g+j+m \leq v_2$ and $b+c+f+h+k \leq v_4$</td>
</tr>
<tr>
<td>15</td>
<td>$BDIAEGLPFKMNJHC$</td>
<td>$a+d+f+g+j+m \leq v_2$ and $a+e+f+k \leq v_4$</td>
</tr>
<tr>
<td>16</td>
<td>$CJHNFKMAPAEGLBD$</td>
<td>$a+f+g+j+m \leq v_2$ and $a+b+e+f+k \leq v_4$</td>
</tr>
<tr>
<td>17</td>
<td>$AFKMGPLNIDJHC$</td>
<td>$c+d+g+j+m \leq v_6$ and $d+e+g+i+l \leq v_8$</td>
</tr>
<tr>
<td>18</td>
<td>$AFKMLGPDIJSNC$</td>
<td>$d+g+j+m \leq v_2$ and $b+d+e+g+i+l \leq v_8$</td>
</tr>
<tr>
<td>19</td>
<td>$BCHAETFPGMLNJD$</td>
<td>$a+c+g+i+l \leq v_6$ and $a+e+f+h+k \leq v_4$</td>
</tr>
<tr>
<td>20</td>
<td>$CBHAETFPGMLNJD$</td>
<td>$a+f+g+j+m \leq v_2$ and $b+d+g+i+l \leq v_8$</td>
</tr>
<tr>
<td>21</td>
<td>$AEKMGPLNIDJHC$</td>
<td>$d+f+g+j+m \leq v_2$ and $d+g+i+l \leq v_9$</td>
</tr>
<tr>
<td>22</td>
<td>$AEKMLGPDIJSNC$</td>
<td>$a+c+g+j+m \leq v_2$ and $a+e+g+l \leq v_9$</td>
</tr>
<tr>
<td>23</td>
<td>$BCHAETFPGMLNJD$</td>
<td>$a+f+g+j+m \leq v_2$ and $a+b+e+g+l \leq v_9$</td>
</tr>
<tr>
<td>No.</td>
<td>Ordering</td>
<td>Conditions for Optimality</td>
</tr>
<tr>
<td>-----</td>
<td>---------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>25</td>
<td>AGDMFCJKPNKNELIB</td>
<td>c+d+h+j+n &lt;=v₁ and c+e+f+h+k &lt;=v₂</td>
</tr>
<tr>
<td>26</td>
<td>ADGLIEBPKNMFMJC</td>
<td>b+d+h+i+n &lt;=v₁ and b+e+f+h+k &lt;=v₂</td>
</tr>
<tr>
<td>27</td>
<td>BIELHKNPCFJMAGD</td>
<td>c+h+j+n &lt;=v₁ and a+c+e+f+h+k &lt;=v₂</td>
</tr>
<tr>
<td>28</td>
<td>CJFMHKNPBIELAGD</td>
<td>b+h+i+n &lt;=v₁ and a+b+c+f+h+k &lt;=v₂</td>
</tr>
<tr>
<td>29</td>
<td>AGDMFCJKPNLIEB</td>
<td>c+d+h+i+j+n &lt;=v₁ and c+f+h+k &lt;=v₂</td>
</tr>
<tr>
<td>30</td>
<td>ADGLIEBPKNMFMJC</td>
<td>b+d+h+i+j+n &lt;=v₁ and b+e+h+k &lt;=v₂</td>
</tr>
<tr>
<td>31</td>
<td>BIELHKNPCFJMAGD</td>
<td>c+h+i+j+n &lt;=v₁ and a+c+f+h+k &lt;=v₂</td>
</tr>
<tr>
<td>32</td>
<td>CJFMHKNPBIELAGD</td>
<td>b+h+i+j+n &lt;=v₁ and a+b+e+h+k &lt;=v₂</td>
</tr>
<tr>
<td>33</td>
<td>AJCMJDGPNLIKEB</td>
<td>c+d+i+j+n &lt;=v₁ and d+e+g+i+l &lt;=v₂</td>
</tr>
<tr>
<td>34</td>
<td>ACFBENKPILNMGJD</td>
<td>b+c+h+i+n &lt;=v₁ and b+e+g+i+l &lt;=v₂</td>
</tr>
<tr>
<td>35</td>
<td>BHEKILNPDGJMACF</td>
<td>d+i+j+n &lt;=v₁ and a+d+e+g+i+l &lt;=v₂</td>
</tr>
<tr>
<td>36</td>
<td>CATBENKPILNMGJD</td>
<td>b+h+i+n &lt;=v₁ and a+b+e+g+i+l &lt;=v₂</td>
</tr>
<tr>
<td>37</td>
<td>AJCMJDGPNLIKEB</td>
<td>c+d+h+i+j+n &lt;=v₁ and d+g+i+l &lt;=v₂</td>
</tr>
<tr>
<td>38</td>
<td>ACFBENKPILNMGJD</td>
<td>b+c+h+i+j+n &lt;=v₁ and b+e+i+l &lt;=v₂</td>
</tr>
<tr>
<td>39</td>
<td>BHEKILNPDGJMACF</td>
<td>d+h+i+j+n &lt;=v₁ and a+d+g+i+l &lt;=v₂</td>
</tr>
<tr>
<td>40</td>
<td>CATBENKPILNMGJD</td>
<td>b+h+i+j+n &lt;=v₁ and a+b+e+i+l &lt;=v₂</td>
</tr>
<tr>
<td>41</td>
<td>ABEDGILPJMNKFLNC</td>
<td>b+d+i+j+n &lt;=v₁ and d+f+g+j+m &lt;=v₂</td>
</tr>
<tr>
<td>42</td>
<td>ABECFKPJMNGLID</td>
<td>b+c+h+j+n &lt;=v₁ and c+f+g+j+m &lt;=v₂</td>
</tr>
<tr>
<td>43</td>
<td>BAEDGILPJMNKFLNC</td>
<td>d+i+j+n &lt;=v₁ and a+d+f+g+j+m &lt;=v₂</td>
</tr>
<tr>
<td>44</td>
<td>BAECFKPJMNGLID</td>
<td>c+h+j+n &lt;=v₁ and a+c+f+g+j+m &lt;=v₂</td>
</tr>
<tr>
<td>45</td>
<td>ABEDGILPJMNKFLNC</td>
<td>b+d+h+i+j+n &lt;=v₁ and d+g+j+m &lt;=v₂</td>
</tr>
<tr>
<td>46</td>
<td>ABECFKPJMNGLID</td>
<td>b+c+h+i+j+n &lt;=v₁ and c+f+j+m &lt;=v₂</td>
</tr>
<tr>
<td>47</td>
<td>BAEDGILPJMNKFLNC</td>
<td>d+h+i+j+n &lt;=v₁ and a+d+g+j+m &lt;=v₂</td>
</tr>
<tr>
<td>48</td>
<td>BAECFKPJMNGLID</td>
<td>c+h+i+j+n &lt;=v₁ and a+c+f+j+m &lt;=v₂</td>
</tr>
</tbody>
</table>
Table 7-4: List of orderings to check, for four nodes

<table>
<thead>
<tr>
<th>Sets in decreasing size order</th>
<th>Nodes listed in decreasing order of &quot;tightness&quot; of constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1234</td>
</tr>
<tr>
<td>a b c d</td>
<td>1 5</td>
</tr>
<tr>
<td>a b d c</td>
<td>2 6</td>
</tr>
<tr>
<td>b a c d</td>
<td>3 7</td>
</tr>
<tr>
<td>b a d c</td>
<td>4 8</td>
</tr>
<tr>
<td>a c b d</td>
<td>1 5</td>
</tr>
<tr>
<td>a c d b</td>
<td>1 5</td>
</tr>
<tr>
<td>c a b d</td>
<td>1 5</td>
</tr>
<tr>
<td>e a d b</td>
<td>1 5</td>
</tr>
<tr>
<td>a d b c</td>
<td>2 6</td>
</tr>
<tr>
<td>a d c b</td>
<td>2 6</td>
</tr>
<tr>
<td>d a b c</td>
<td>2 6</td>
</tr>
<tr>
<td>d a c b</td>
<td>2 6</td>
</tr>
<tr>
<td>b c a d</td>
<td>3 7</td>
</tr>
<tr>
<td>b c d a</td>
<td>3 7</td>
</tr>
<tr>
<td>c b a d</td>
<td>3 7</td>
</tr>
<tr>
<td>c b d a</td>
<td>3 7</td>
</tr>
<tr>
<td>b d a c</td>
<td>4 8</td>
</tr>
<tr>
<td>b d c a</td>
<td>4 8</td>
</tr>
<tr>
<td>d b a c</td>
<td>4 8</td>
</tr>
<tr>
<td>d b c a</td>
<td>4 8</td>
</tr>
<tr>
<td>e d a b</td>
<td>1 5</td>
</tr>
<tr>
<td>e d b a</td>
<td>3 7</td>
</tr>
<tr>
<td>e d c b</td>
<td>2 6</td>
</tr>
<tr>
<td>e d c b</td>
<td>4 8</td>
</tr>
</tbody>
</table>
Table 7-4, continued

<table>
<thead>
<tr>
<th>Sets in decreasing size order</th>
<th>Nodes listed in decreasing order of &quot;tightness&quot; of constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2314</td>
</tr>
<tr>
<td>\text{abeced}</td>
<td>25</td>
</tr>
<tr>
<td>\text{abdec}</td>
<td>25</td>
</tr>
<tr>
<td>\text{bacd}</td>
<td>25</td>
</tr>
<tr>
<td>\text{badc}</td>
<td>25</td>
</tr>
<tr>
<td>\text{acbd}</td>
<td>26</td>
</tr>
<tr>
<td>\text{acdb}</td>
<td>26</td>
</tr>
<tr>
<td>\text{cabd}</td>
<td>26</td>
</tr>
<tr>
<td>\text{cadb}</td>
<td>26</td>
</tr>
<tr>
<td>\text{adbc}</td>
<td>25</td>
</tr>
<tr>
<td>\text{adcb}</td>
<td>26</td>
</tr>
<tr>
<td>\text{dabc}</td>
<td>26</td>
</tr>
<tr>
<td>\text{dacb}</td>
<td>27</td>
</tr>
<tr>
<td>\text{bcad}</td>
<td>25</td>
</tr>
<tr>
<td>\text{bca}</td>
<td>27</td>
</tr>
<tr>
<td>\text{cbad}</td>
<td>27</td>
</tr>
<tr>
<td>\text{cbda}</td>
<td>28</td>
</tr>
<tr>
<td>\text{bdac}</td>
<td>27</td>
</tr>
<tr>
<td>\text{bdca}</td>
<td>27</td>
</tr>
<tr>
<td>\text{dabc}</td>
<td>27</td>
</tr>
<tr>
<td>\text{dbca}</td>
<td>27</td>
</tr>
<tr>
<td>\text{cdab}</td>
<td>28</td>
</tr>
<tr>
<td>\text{cdba}</td>
<td>28</td>
</tr>
<tr>
<td>\text{dec}</td>
<td>28</td>
</tr>
</tbody>
</table>
Table 7-5: Sequence for trying potentially optimal orderings for four nodes

<table>
<thead>
<tr>
<th>Seq. No.</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 5, 2, 6, 3, 7, 4, 8</td>
</tr>
<tr>
<td>2</td>
<td>2, 6, 1, 5, 4, 8, 3, 7</td>
</tr>
<tr>
<td>3</td>
<td>3, 7, 4, 8, 1, 5, 2, 6</td>
</tr>
<tr>
<td>4</td>
<td>4, 8, 3, 7, 2, 6, 1, 5</td>
</tr>
<tr>
<td>5</td>
<td>5, 1, 6, 2, 7, 3, 8, 4</td>
</tr>
<tr>
<td>6</td>
<td>6, 2, 5, 1, 8, 4, 7, 3</td>
</tr>
<tr>
<td>7</td>
<td>7, 3, 8, 4, 5, 1, 6, 2</td>
</tr>
<tr>
<td>8</td>
<td>8, 4, 7, 3, 6, 2, 5, 1</td>
</tr>
<tr>
<td>9</td>
<td>9, 13, 10, 14, 11, 15, 12, 16</td>
</tr>
<tr>
<td>10</td>
<td>10, 14, 9, 13, 12, 16, 11, 15</td>
</tr>
<tr>
<td>11</td>
<td>11, 15, 12, 16, 9, 13, 10, 14</td>
</tr>
<tr>
<td>12</td>
<td>12, 16, 11, 15, 10, 14, 9, 13</td>
</tr>
<tr>
<td>13</td>
<td>13, 9, 14, 10, 15, 11, 16, 12</td>
</tr>
<tr>
<td>14</td>
<td>14, 10, 13, 9, 16, 12, 15, 11</td>
</tr>
<tr>
<td>15</td>
<td>15, 11, 16, 12, 13, 9, 14, 10</td>
</tr>
<tr>
<td>16</td>
<td>16, 12, 15, 11, 14, 10, 13, 9</td>
</tr>
<tr>
<td>17</td>
<td>17, 21, 18, 22, 19, 23, 20, 24</td>
</tr>
<tr>
<td>18</td>
<td>18, 22, 17, 21, 20, 24, 19, 23</td>
</tr>
<tr>
<td>19</td>
<td>19, 23, 20, 24, 17, 21, 18, 22</td>
</tr>
<tr>
<td>20</td>
<td>20, 24, 19, 23, 18, 22, 17, 21</td>
</tr>
<tr>
<td>21</td>
<td>21, 17, 22, 18, 23, 19, 24, 20</td>
</tr>
<tr>
<td>22</td>
<td>22, 18, 21, 17, 24, 20, 23, 19</td>
</tr>
<tr>
<td>23</td>
<td>23, 19, 24, 20, 21, 17, 22, 18</td>
</tr>
<tr>
<td>24</td>
<td>24, 20, 23, 19, 22, 18, 21, 17</td>
</tr>
</tbody>
</table>
Table 7-5, continued

<table>
<thead>
<tr>
<th>Seq. No.</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25, 29, 28, 30, 27, 31, 28, 32</td>
</tr>
<tr>
<td>26</td>
<td>28, 30, 25, 29, 28, 32, 27, 31</td>
</tr>
<tr>
<td>27</td>
<td>27, 31, 28, 32, 25, 29, 26, 30</td>
</tr>
<tr>
<td>28</td>
<td>28, 32, 27, 31, 26, 30, 25, 29</td>
</tr>
<tr>
<td>29</td>
<td>29, 25, 30, 26, 31, 27, 32, 28</td>
</tr>
<tr>
<td>30</td>
<td>30, 26, 29, 25, 32, 28, 31, 27</td>
</tr>
<tr>
<td>31</td>
<td>31, 27, 32, 28, 29, 25, 30, 26</td>
</tr>
<tr>
<td>32</td>
<td>32, 28, 31, 27, 30, 26, 29, 25</td>
</tr>
<tr>
<td>33</td>
<td>33, 37, 34, 38, 35, 39, 36, 40</td>
</tr>
<tr>
<td>34</td>
<td>34, 38, 33, 37, 36, 40, 35, 39</td>
</tr>
<tr>
<td>35</td>
<td>35, 39, 36, 40, 33, 37, 34, 38</td>
</tr>
<tr>
<td>36</td>
<td>36, 40, 35, 39, 34, 38, 33, 37</td>
</tr>
<tr>
<td>37</td>
<td>37, 33, 38, 34, 39, 35, 40, 36</td>
</tr>
<tr>
<td>38</td>
<td>38, 34, 37, 33, 40, 36, 39, 35</td>
</tr>
<tr>
<td>39</td>
<td>39, 35, 40, 36, 37, 33, 38, 34</td>
</tr>
<tr>
<td>40</td>
<td>40, 36, 39, 35, 38, 34, 37, 33</td>
</tr>
<tr>
<td>41</td>
<td>41, 45, 42, 46, 43, 47, 44, 48</td>
</tr>
<tr>
<td>42</td>
<td>42, 46, 41, 45, 44, 48, 43, 47</td>
</tr>
<tr>
<td>43</td>
<td>43, 47, 44, 48, 41, 45, 42, 46</td>
</tr>
<tr>
<td>44</td>
<td>44, 48, 43, 47, 42, 46, 41, 45</td>
</tr>
<tr>
<td>45</td>
<td>45, 41, 46, 42, 47, 43, 48, 44</td>
</tr>
<tr>
<td>46</td>
<td>46, 42, 45, 41, 48, 44, 47, 43</td>
</tr>
<tr>
<td>47</td>
<td>47, 43, 48, 44, 45, 41, 46, 42</td>
</tr>
<tr>
<td>48</td>
<td>48, 44, 47, 43, 46, 42, 45, 41</td>
</tr>
</tbody>
</table>
4. The aim is to find an ordering that adds no extraneous operands to the two windows with the tightest constraints: thus \( v_{i1}v_{i3}v_{i4} \) and \( v_{i2}v_{i1}v_{i5}v_{i4} \) are treated equivalently. If a constraint ordering found in step 3 is \( v_{i1}v_{i2}v_{i5}v_{i4} \), and \( i_2 > i_1 \), replace the ordering with \( v_{i2}v_{i1}v_{i5}v_{i4} \). (This step is not strictly necessary, but it enables Table 7-4 to have only 12 columns instead of 24.)

5. List the set sizes \( a, b, c, d \) in decreasing order. Equal sizes can be listed in any order.

6. Use the size listing and the constraint listing(s) as indices into Table 7-4. The value found is an index into Table 7-5, which lists the orderings to try and the sequence in which to try them. (This extra indirection is used to save space; the entries in Table 7-4 could themselves hold the lists of orderings to try.)

7. Try the orderings listed in Table 7-3 in the sequence indicated by the entry in Table 7-5. Stop as soon as a successful ordering is found, or when all indicated orderings have been tried. In the latter case, if there are further constraint orderings to consider then check the other orderings indicated; otherwise stop in failure.

7.3.4.5. Summary

A method has been presented for finding optimal orderings of tables, to satisfy groups of two to four nodes. When \( n \) nodes are considered, \( 2^{n-1} \) sets are constructed; a permutation of those sets that satisfies all constraints is sought.

If \( n=2 \), the successful ordering can be written down immediately; if \( n=3 \) up to 12 different orderings may need to be checked; if \( n=4 \) up to 48 different orderings may need to be checked. The possible orderings can be tested in sequence, or an effort can be made to test the most likely orderings first. In the latter case constraints and set sizes need to be considered.
As the number of nodes in the group increases, three effects are noted:

- Constructing the $2^n-1$ sets of operands becomes more expensive, as there are more sets and more comparisons to make;

- It is less likely that an optimal ordering can be found;

- Many more unsuccessful orderings must be tried before failure is assumed.

Thus there is more work for less return. With groups of four nodes the probability of success is still reasonable and the work required to detect failure is usually small; with groups of five or more nodes this is no longer true.

This method works well for up to four nodes. No attempt is made to find optimal orderings for more than four nodes.

7.3.5. Binary encoding

The final result, after all potential environment changes are considered, is a tree in which each node represents a segment that should be encoded in its own environment. The information conveyed in a node is:

- the instructions to be encoded in this environment;

- (for each of operators, operands, and labels independently) how to effect the change of environment: either

  1. full specification: the order of entries to be placed in the table is fully defined. The initial size and placement of the window is specified (it might not be at the start of the table); or

  2. window change: the new size and position of the window is specified; or

  3. no change: the window is unchanged.
"Change-env" and "Revert-env" instructions are inserted at appropriate places in the image program. The binary code is then generated.

If a node needs a full environment specification, the table contents are recorded in that node. Otherwise, the table contents must be copied from the node that was the local root. The window positions and sizes are recorded in every node; the window contents are thus known from the positions of the windows in the tables.

Generating the binary code of the program unit is straightforward. The nodes are encoded in the order in which they are encountered in a post-order traversal of the tree. When encoding a node, environment definitions are encoded according to the principles in sections 3.4 and 3.3.2; the binary values encoded for operators, operands, and labels are the relevant offsets from the start of the windows.

7.3.6. Summary

A method has been described for exploiting the repeated appearance of operators, operands, and labels in different segments of a program unit. The careful placement of entries in environment tables enables many environment changes to be performed by simply shifting windows, instead of having to redefine whole tables. It is then easy to change environments to reflect changing usage of operands, operators, and labels without needing to respecify common entries. The cost of changing environments is reduced, and the total static and dynamic sizes of program units are reduced.

The process consists of two stages:

1. the most likely separate segments are identified;

2. various combinations of these segments are considered to see what use can be made of common entries in the environments. The best combinations are used to produce the binary encoding of the program unit.
The segment combinations that are likely to have the most effect on total size if successful are checked first. As soon as it is unlikely that a better combination can be found, no further combinations are checked; the best so far is used.

The method for checking a given combination of nodes is quick and effective. If it is possible to set up the environment tables to avoid respecifications, that set-up is usually found almost immediately; it is quickly recognized if no optimal set-up is possible.

7.4. Results

The procedure described in section 7.2 was not tested, as it makes few changes to program encodings and has little effect on overall program size.

The technique described in section 7.3 was implemented and tested. Since it appears from chapter 6 that the effect of internal environment changes is independent of the intermediate language, only one intermediate language (namely ICL2000 assembly language) was used in the tests. Otherwise the testing procedure was the same as that used in chapter 6, and the same test programs were used.

Results are reported on the workings of the algorithm itself, and its effect on the size of program units.

7.4.1. Mechanics of the algorithm

The following observations were made when this algorithm was applied to the 83 test program units:

1. Changes were made in 24 program units, in 20 programs, to a total of 64 nodes.
2. Mean number of nodes in each classification of usefulness, averaged over those 24 program units:

- 1.6 nodes with large predicted gains
- 1.8 nodes with moderate predicted gains
- 1.2 nodes with small predicted gains

The root node of each program unit is excluded from these figures. Thus on average there were 5.6 nodes (4.6 internal, plus the root) to be considered in combination; the range was from 2 to 11 nodes.

The largest number of potential segments occurred in a program unit with many short loops. Finding the best segmentation involved checking several combinations, and gave excellent returns. Of the 10 internal nodes, 8 were encoded as separate segments; 6 of the internal environment changes involved only window changes.

3. Average probability of nodes in each classification being encoded as separate segments:

- Large predicted gain: 1.00 (by definition)
- Moderate predicted gain: 0.78
- Small predicted gain: 0.43

Overall, 76% of the nodes identified as potential segments were encoded as separate segments.

4. Checking combinations of 4 nodes:

- 79 checked
- 36 successful
- 11 implemented

Just under half of the 4-node combinations were successful. If a successful combination was not used, it meant that another 4-node combination that produced better overall program size was used instead.

5. Checking combinations of 3 nodes:
27 checked
26 successful
10 implemented
Almost every 3-node combination tried was successful.

6. Checking combinations of 2 nodes:
5 checked successfully, and implemented.
Few were checked, because 4- and 3-node groupings were usually
checked successfully first. These five combinations were checked in
program units where only one internal segment was identified in the first
place; or after a successful 4- or 3-node grouping had been found, and
only a few nodes remained.

7. Proportion of full internal environment changes to cheap internal
environment changes:
64 separate segments identified;
6 of them needed full table respecification;
58 only needed window changes.

8. Mean number of different orderings checked, when seeking optimal table
orderings for a given combination of 3 or 4 nodes:
3 nodes: when successful : 1.3
when unsuccessful : 6
total : 1.5
4 nodes: when successful : 1.3
when unsuccessful : 11
total : 6.6
Overall: total : 5.3
On average, 5.3 different orderings were checked before a decision was
arrived at on whether or not an optimal table ordering was possible.
Success, when it came, was almost immediate; this occurred in 62 of the
106 combinations tried. On average, 11 orderings were checked before
failure was recognized.
7.4.1.1. Discussion of results

From these results it is clear that optimal table orderings can be found for most combinations of 3 or 4 nodes. The effort involved in checking a combination occurs mainly in the construction of the various sets of shared operands (or operators or labels), since the decision is then reached quickly.

The efficiency of the overall procedure is thus dependent on the number of combinations checked. They are checked in groups, which are tested in the best order, so overall efficiency depends on limiting the number of combinations in each group.

The number of combinations depends on the number of potential segments identified, which is closely related to the length of the program unit:

- Small program units (i.e. less than half a page of source text) have little scope for internal segmentation, except to optimize a loop. There are few potential segments, and the best segmentation is found quickly.

- Average program units (i.e. half a page to two pages of source text) contain an average of four to ten potential segments. Exclusion of nodes with small predicted gains is useful here in limiting the number of potential segments to consider. The constraints on valid combinations make their number manageable.

Almost all of the 24 test program units in which internal changes were made fell into this category, and the average number of combinations tested per program unit was 4.6.

- Large program units (i.e. more than two pages of source text) may cause problems. Even after nodes with small predicted gains are excluded, enough nodes may remain to make testing all combinations impracticable. Possible solutions to this problem are:
restrict attention to nodes with large predicted gains, if there are enough of them; or

divide the tree into subtrees of no more than a given size, and consider each subtree independently.

Each of these solutions lessens the chance of finding the optimal segmentation, but the degradation should be small.

Exhaustive checking of all combinations would find the optimal segmentation. This algorithm checks only a few combinations, but in most cases should find a segmentation that is optimal or nearly so. The number of combinations can be limited to manageable levels, regardless of the size of the program unit, and checking a given combination is straightforward and quick. Thus the algorithm performs effectively and efficiently.

7.4.2. Changes in program size

More program units have internal changes of environment than were observed previously. This is attributable to two factors:

- The dynamic thresholds at which a node is deemed to be a separate segment are lower. This accounts for an increase in the number of internal environment changes, from 30 in 16 program units to 45 in 23 program units.

- Several environment changes are made cheaply, rendering them worthwhile when they would otherwise not be. This raises the number of internal environment changes to 64, in 24 program units.

As expected, making environment changes cheaper causes more of them to be performed.

When the total static sizes of those 24 program units are compared with their
sizes if no internal environment changes are made, the following observations can be made:

- 18 decreases in total static size
- 6 increases in total static size

Mean change = 2.8% decrease in total static size
(standard deviation = 3.4%)

Largest increase in total static size = 3.9%
Largest decrease in total static size = 8.2%

The predicted dynamic changes in the sizes of the 24 program units, compared with their sizes if no internal environment changes are made, are:

- 0 increases in total dynamic size
- 24 decreases in total dynamic size

Mean change = 12.9% decrease in total dynamic size
(standard deviation = 5.8%)

Smallest decrease in total dynamic size = 4.1%
Median decrease in total dynamic size = 12.2%
Largest decrease in total dynamic size = 30.0%

The observed dynamic changes are:

- 0 increases in total dynamic size
- 24 decreases in total dynamic size

Mean change = 12.9% decrease in total dynamic size
(standard deviation = 6.3%)

Smallest decrease in total dynamic size = 5.5%
Median decrease in total dynamic size = 12.0%
Largest decrease in total dynamic size = 28.9%

The coefficient of correlation between predicted and observed total dynamic sizes is 0.91.
7.4.2.1. Discussion of results

Predictions of dynamic changes are again accurate. This is expected, since the predicted program profiles that led to accurate predictions of dynamic changes in chapter 6 are used again here.

The results reported in chapter 6 are the best that can be achieved (i.e. the thresholds are set to provide the optimal balance between static losses and dynamic gains) if all environment changes require full table specification, using the particular method described in section 3.4. When the same method is used to specify tables, but few internal environment changes need full table specification, the best results attainable are those just given. The following comparisons show the effect of eliminating most of the cost of most environment changes:

- The number of changes increases from 30 in 16 program units to 64 in 24 program units.

- The average change in total static size becomes a gain of 2.8%, instead of a loss of 2.9%. The standard deviation of the change measurements drops from 5.6% to 3.4%.

- The average reduction in total dynamic size increases from 10.0% to 12.9%, and the standard deviation of the reductions drops from 9.3% to 6.3%.

Total static size is reduced by an average of nearly 6%. This is a valuable improvement, particularly since most environment changes now reduce the total static size of program units. Individual improvements range from 0% to 15%, with clusters at 4% and 13%. Some individual improvements are clearly very useful; the average improvement, although smaller, is still valuable.

The static standard deviation is reduced because all of the large static losses are eliminated. Most of the previous large losses are fixed by cheaper environment
changes; the few that remain are eliminated when those nodes are classified as useless (page 143).

Total dynamic size is dominated by instruction costs, so the cost of changing environments is less significant. For this reason, making environment changes cheaper reduces total dynamic size by an average of only 3%. Individual improvements range from 0% to 15%, but most of them are small. Even so, the average improvement of 3% is still valuable, as it implies a further reduction in instruction decoding time.

Another advantage of cheaper internal environment changes is that instruction sizes are slightly smaller, since more environment changes are made. The effect on static size is minor, compared to the saving in environment changes, so it can be assumed that the 6% improvement is almost entirely due to cheap environment changes. The effect on dynamic size is more significant: not all of the 3% improvement can be attributed directly to cheaper environment changes.

In summary, internal environment changes are more useful and more widely applicable when their cost is reduced.

7.5. Summary

Two methods have been described for reducing the cost of some internal environment changes, by exploiting the use of common operators, operands, and labels in different segments of a program unit.

One method is applied after the segmentation of the program unit is determined. It identifies some occasions when one or more environment tables can be used unchanged across an environment boundary. Few improvements are likely, so this method is not investigated further.

The other method is more useful. It is described in detail, and evaluated in
practice. It is based on the arrangement of environment table entries in suitable orders, so that later environment changes can be achieved by simply altering windows.

An algorithm is presented for finding the best segmentation of a program unit: this is heavily influenced by potential cheap environment changes. The algorithm finds good segmentations quickly for all but very large program units, and suggestions on how to deal with those are made.

When the algorithm is applied in practice, the following conclusions are drawn:

- Reducing the cost of environment changes makes many more of them worthwhile;
- Almost all internal environment changes can be performed by just changing windows;
- Total static and dynamic sizes are both reduced when the cost of changing environment is largely eliminated;
- Where previously environment changes often reduced dynamic size at the expense of a static size increase, most internal environment changes now reduce both static size and dynamic size of a program unit.

Cheap environment changes are well worthwhile, and can usually be arranged easily.
Chapter 8

Conclusions

This thesis investigates the relationship between computer architecture and computer programs written in high-level procedural languages. It develops the notion that, at any moment in the execution of a program, an ideal computer can be defined to suit the current activity. The ideal computer provides everything necessary for the current stage of execution, and nothing unnecessary.

A program generally consists of several phases, in which different \*working sets\* of operators and operands are used. As execution passes through different phases of a program, the details of what the ideal computer should provide vary. When the computer is well matched to the program, with changing program needs reflected by changes to the computer, the dynamic effort of decoding the instructions during execution is minimized. The static size of the binary program representation is also minimized.

A computer architecture that can change dynamically to suit changing program requirements is defined: it is called a \*context-sensitive\* architecture.

The thesis shows how to analyse a program, in order to determine:

1. how to construct an ideal computer on which to execute it (chapter 3). The implementation of the ideal computer as a virtual architecture on a real computer is explained.

2. how to determine when the virtual architecture should change during program execution, in order to keep the architecture well matched to the
program's needs, thus minimizing the static code space and dynamic decoding effort (chapters 4, 5, 6). Programs are split into segments, for each of which an ideal virtual architecture is constructed.

3. how best to make the changes to the virtual architecture (chapter 7).

The context-sensitive architecture is similar to an architecture defined by Flynn and Hoevel for research on directly executed languages [Flynn 80, Flynn 83]. The method of configuring it to suit a given program segment is similar to that described in [Flynn 80]. This work is distinguished from that of Flynn and Hoevel in the way programs are segmented, and in the way the virtual architecture is changed.

The program segmentation used by Flynn and Hoevel is its division into scopes of definition of variables. No analysis of the internal structure of these segments is performed, and no consideration is given to the order in which objects are included in the tables that embody the virtual architecture.

In this work, the internal structure of each program unit is considered. Each scope of definition represents a segment. Analysis of the program often results in some of these segments being split into smaller segments and subsegments. A major feature of this analysis is the use of the predicted execution profile of the program. In order to minimize the cost of subsequent changes to the virtual architecture, whenever a new table is set up its entries are specified in a carefully chosen order.

In the sections that follow, conclusions are drawn on:

- the performance of the context-sensitive architecture;
- ordering the entries in environment tables;
- modelling program execution, to predict dynamic execution profiles; and
- dividing program units into smaller segments.
8.1. Context-sensitive architecture

The context-sensitive architecture is essentially the same as one proposed by Flynn and Hoevel [Flynn 80, Flynn 83] for their DELtran experiment. Windows have been added to the environment tables, to reduce the cost of some environment changes, but otherwise the architecture and its operation is the same as that used by Flynn and Hoevel.

When the DELtran architecture was compared with traditional architectures, the following observations were made:

- the code space needed for programs encoded for the DELtran architecture is about half that for traditional architectures [Flynn 80].

- speed of interpretation of DEL instructions is five times that for conventional machine instructions, on the EMMY computer [Flynn 83].

- execution speed of DEL code on EMMY is nearly that of compiled code on a traditional architecture. The time taken for instruction interpretation is dominated by the time taken by the slower microcode routines to execute the instructions [Flynn 80].

Experiments conducted here confirm that code space needed for programs for the context-sensitive architecture is half that for a traditional architecture. Execution speed of the context-sensitive architecture is assumed to be the same as that of the DELtran architecture.

Comparisons with other types of architectures (i.e. SIMD or MIMD [Flynn 72]) are not appropriate, as the context-sensitive architecture is based on Flynn's five canonical measures of interpretation, which are geared towards SISD architectures. Other measures, such as the number of processors available, might be more appropriate for other types of architecture.

The DEL performance figures are derived under the assumption that the
environment only changes on entry to new scopes of definition (i.e. on procedure entry in Fortran or Pascal). Application of the methods described in this thesis can improve the performance figures, albeit only slightly, by finding suitable points at which to make additional environment changes. Code space can be reduced by an average of 3%, which is nice but not very significant. Dynamic decoding effort can be reduced by an average of 10%, and up to one third. The already narrow advantage in execution speed of traditional architectures over the context-sensitive architecture is reduced.

In short, the context-sensitive architecture is a competitive alternative to traditional architectures.

8.2. Optimal ordering of environment table entries

When environment table entries are specified in carefully-chosen orders, many environment changes can be achieved by just changing the contents of at most six registers; there is no need to alter the tables themselves. This eliminates most of the cost of most internal environment changes within program units.

The algorithm for finding optimal orderings of environment tables works well. A correct decision on whether or not an optimal ordering exists for a given combination of nodes is reached quickly. However, there are two respects in which the algorithm could be improved:

- No more than four nodes are considered in combination, at present. There is a trade-off here: the more nodes that are considered in a group, the greater is the advantage gained when an optimal ordering is found that satisfies them all, but the smaller is the chance of finding such an ordering. It could be useful to extend the algorithm to cope with five or six nodes at once, although optimal orderings for that many nodes will be rare.

- Large program units often contain several potential segments; checking
all combinations of potential segments is then very slow. Two approaches have been suggested for limiting the number of combinations that need to be checked; the success of these approaches, and their effect on the performance of the architecture, should be investigated.

8.3. Modelling program execution

A model of program execution has been developed, with the aim of enabling the dynamic execution profile of a program to be predicted from static information. Iteration statements, alternative statements, and basic sequencing are all modelled, with varying degrees of success.

Modelling the execution of a sequence is trivial, and is merely a matter of definition.

Modelling the execution of an iteration statement (i.e. loop) involves predicting the number of iterations that will be made through its body. This appears to be done well: an accurate prediction is made for almost every loop. Accurate predictions of all constituent loops are made in 90% of program units. This accuracy is obtained because very few loops have large iteration counts, and the overall distribution of iteration counts is regular: accurate predictions are thus made easily.

Modelling the execution of an alternative statement involves predicting the likelihood of selection of each branch. This is done poorly. Any given branching decision is likely to be decided always in the same way: hence one branch is almost always selected, and each other branch is almost never selected. The difficulty lies in predicting the right branch: empirical evidence suggests no reliable method for doing so. Predictions made according to averages are almost always wrong, although the errors tend to cancel each other out.

The poor modelling of alternative statements means that program profiles are
not predicted well in detail. The parts of the program that account for most execution time are correctly identified, because loops are modelled well, but quantitative predictions of the number of times each instruction is executed are unreliable.

The total number of instructions executed dynamically is predicted accurately, however, as the individual errors cancel out. The correlation between the predicted total and observed total is high. In the 10% of program units containing a badly underestimated loop, the total number of instructions executed is also badly underestimated; good predictions are made in the other 90% of program units.

There are three respects in which the model could be improved:

- Goto-statements that are not used to implement iteration statements or alternative statements are currently treated as straight line code: rules are needed to model their effects on control flow.

- Modelling of loops could be improved, to make better predictions in some cases where large underestimates are made now.

- Rules are needed for recognizing which branch of an alternative statement is likely to be selected. This is the most important shortcoming, and probably the hardest to rectify.

Much more empirical data is needed to solve these problems. If possible, the data should be classified according to programming language, problem domain, and programmer. Goto-statements, for which there is very little empirical data so far, are used differently in different languages and by different programmers; the problem domain should be irrelevant. The problem domain influences loop iteration counts to a greater extent than the language or the programmer. Alternative statements are most vulnerable to the idiosyncrasies of programmer style: the outcome of logical conditions depends on their phrasing, which is up to the programmer.
This data could be gathered by tracing the execution of programs and amassing a collection of observations on the use of different statement types and constructions. As more data is gathered, the characteristic style of different programmers with different languages and problems may emerge. More accurate predictions of program profiles should then be possible. This is worthy of further research.

Good profile predictions are useful in this application. They are also of more general use in guiding the effort of program optimization to the most appropriate parts of programs.

8.4. Segmenting program units

Most program units that contain less than 20 lines of source text are best encoded as single units for execution on the context-sensitive architecture. In contrast, it is worth subdividing almost every program unit of 20 or more lines into smaller segments.

Two types of segment have been identified, and methods for recognizing them have been described:

- segments that are candidates for local optimization are found by considering the syntax and predicted profiles of program units;

- segments that represent phases of constant locality are found by considering patterns of use of operators and operands, and data flow analysis.

Only the first of these methods has been implemented here. Some phases of constant locality are found by chance, when they coincide with basic blocks or syntactic units; no others are found.

Better recognition of segments and subsegments within program units would
result if data flow and usage patterns were considered and phases of constant locality recognized. The failure to recognize such phases is the main shortcoming of this work; implementation of the ideas suggested in section 6.2.2 should overcome this problem.

Good results are already obtained here, when usually only one type of segment is found. Further research is needed to determine the effect of finding other segments as well.

The ability to divide program units into smaller segments is useful with the context-sensitive architecture, because it helps to optimize program execution. For the same reason, it could be applied usefully in virtual memory systems. Smaller segments should reduce the problem of fragmentation in segment-based virtual memories, and are easier to allocate to appropriate pages in order to minimize page traffic in paging systems. Also, this work identifies points in programs at which it may be worthwhile to preload pages or cache entries.

In a different sense, this work may have applications in the field of programming methodology. The tree of potential segments carries a great deal of information about a program unit. When data flow analysis, information on operator and operand use, and syntax analysis are all used in its construction, the tree gives a good indication of the logical structure of the program unit. This could be helpful in the documentation and verification of programs [Waters 82], providing useful feedback to programmers. Using the terminology of structured design [Yourdon 79], it may be possible (by considering the data used in, and passed between, different subsegments) to determine automatically whether a program unit has logical, temporal, procedural, communicational, or sequential strength (coincidental and functional strengths would both be characterized by the inability of these algorithms to find substructure in a program unit). Suggestions could be given on how to split large program units with low strength into several higher-strength modules. There is considerable scope for further research along these lines.
References

Heuristic synthesis of microprogrammed computer architecture.

[Abd-Alla 76] Abd-Alla, A.M. and Moffett, L.H.
On-line architecture tuning using microcapture.

[Agrawala 76] Agrawala, A.K. and Rauscher, T.G.
*Foundations of microprogramming: architecture, software, and applications.*

[Aho 77] Aho, A.V. and Ullman, J.D.
*Principles of compiler design.*
Addison-wesley, Reading, Massachusetts, 1977.

An empirical analysis of COBOL programs.

Static and dynamic characteristics of XPL programs.

The System/360 Model 91: machine philosophy and instruction handling.

Design of analyzers for selective program analysis.
A comparison of measures of control flow complexity.

[Baker B 77] Baker, B.S.
An algorithm for structuring flowgraphs.

Metric analysis and data validation across Fortran projects.

[Beizer 70] Beizer, B.
Analytical techniques for the statistical evaluation of program running time.

[Bell 71] Bell, C.G. and Newell, A. (eds.).
*Computer structures : readings and examples.*

[Berry 83] Berry, D.M.
A new methodology for generating test cases for a programming language compiler.

[Booth 79] Booth, T.L.
Use of computation structure models to measure computation performance.

[Boyer 75] Boyer, R.S., Elspas, B., and Levitt, K.N.
SELECT - a formal system for testing and debugging programs by symbolic execution.

[Brookes 82] Brookes, G.R., Wilson, I.R., and Addyman, A.M.
A static analysis of PASCAL program structures.
*Burroughs 7200 Systems Reference Manual*  
Detroit, Michigan, 48232, USA, 1972.

[Chanon 74] Chanon, R.N.  
On a measure of program structure.  

[Chevance 78] Chevance, R.J. and Heidet, T.  
Static profile and dynamic behaviour of COBOL programs.  

[Chu 75] Chu, Y.  
Concepts of high-level language computer architecture.  

[Chu 77] Chu, Y.  
Direct-execution computer architecture.  

[Chu 81a] Chu, Y.  
High-level computer architecture.  
(Guest Editor’s introduction).

[Chu 81b] Chu, Y. and Abrams, M.  
Programming languages and direct-execution computer architecture.  

[Clarke 76] Clarke, L.A.  
A system to generate test data and symbolically execute programs.  

[Clarke 81] Clarke, L.A. and Richardson, D.J.  
Symbolic evaluation methods for program analysis.  
[Cohen 74] Cohen, J. and Zuckerman, C.
Two languages for estimating program efficiency.

[CookR 82] Cook, R.P. and Lee, I.
A contextual analysis of Pascal programs.

[Cragon 79] Cragon, H.G.
An evaluation of code space requirements and performance of various architectures.

[Denning 80] Denning, P.J.
Working sets past and present.

[Dijkstra 68] Dijkstra, E.W.
Goto statement considered harmful.

[Dijkstra 76] Dijkstra, E.W.
A discipline of programming.

[Ditzel 80] Ditzel, D.R. and Patterson, D.A.
Retrospective on high-level language computer architecture.

[El-Ayat 77] El-Ayat, K.A. and Howard, J.A.
Algorithms for a self-tuning microprogrammed computer.

[El-Halabi 82] El-Halabi, H. and Agrawal, D.P.
Some remarks on direct execution computers.

[Elshoff 76] Elshoff, J.L.
A numerical profile of commercial PL/I programs.
[Elshoff 77] Elshoff, J.L.
The influence of structured programming on PL/I program profiles.

[Fairley 78] Fairley, R.E.
Static analysis and dynamic testing of computer software.

[Flynn 72] Flynn, M.J.
Some computer organizations and their effectiveness.

[Flynn 74] Flynn, M.J.
*Directly executed languages.*

[Flynn 80] Flynn, M.J.
Directions and issues in architecture and language.

[Flynn 83] Flynn, M.J. and Hoevel, L.W.
Execution architecture: the DELtran experiment.

[Gannon 77] Gannon, J.D.
An experimental evaluation of data type conventions.

[Gannon 78] Gannon, J.D.
Characteristic errors in programming languages.

[Gilb 76] Gilb, T.
*Software metrics.*

[Graham 77] Graham, G.S.
*A study of program and memory policy behaviour.*
*The SNOBOL4 programming language (2nd edition).*  

[Griswold 72] Griswold, R.E.  
*The macro implementation of SNOBOL4 - a case study of machine-independent software development.*  

[Haase 81] Haase, V.H.  
Real-time behaviour of programs.  

[Halstead 77] Halstead, M.H.  
*Elements of software science.*  

[Hatfield 71] Hatfield, D.J. and Gerald, J.  
Program restructuring for virtual memory.  

[Hecht 77] Hecht, M.S.  
*Flow analysis of computer programs.*  

[Henry 81a] Henry, S., Kafura, D., and Harris, K.  
On the relationships among three software metrics.  

[Henry 81b] Henry, S. and Kafura, D.  
Software structure metrics based on information flow.  

[Hoevel 74] Hoevel, L.W.  
'Ideal' directly executed languages: an analytical argument for emulation.  

[Howden 78] Howden, W.E.  
DISSECT - a symbolic evaluation and program testing system.  
[Huang 78] Huang, J.C.
Program instrumentation and software testing.

[Hurst 82] Hurst, A.J. and Lokan, C.J.
Context sensitive computer architecture.
In *Proceedings of 5th Australian Computer Science Conference*, pages 214-221. Australian Computer Science Communications, Department of Information Science, University of Tasmania, Hobart, Australia, 1982.

[Ingalls 72] Ingalls, D.
The execution time profile as a programming tool.

[Jensen 78] Jensen, K. and Wirth, N.
*PASCAL User Manual and Report (second edition).*

[Johnson 71] Johnson, J.B.
The contour model of block structured processes.

[Johnsson 82] Johnsson, R. and Wick, J.
An overview of the Mesa processor architecture.

[Johnston 80] Johnston, D.B. and Lister, A.M.
An experiment in software science.

HLL architectures: pitfalls and predilections.

[Kaviprapu 79] Kaviprapu, K.M. and Frailey, D.J.
Quantification of architectures using software science.

[Keedy 77] Keedy, J.L.
An outline of the ICL2900 series system architecture.
[Keedy 78a] Keedy, J.L.
On the use of stacks in the evaluation of expressions.

[Keedy 78b] Keedy, J.L.
On the evaluation of expressions using accumulators, stacks and
store-to-store instructions.

[Keedy 79] Keedy, J.L.
More on the use of stacks in the evaluation of expressions.

[Knuth 71a] Knuth, D.E.
An empirical study of FORTRAN programs.

Notes on avoiding 'go to' statements.

[Knuth 73] Knuth, D.E.
*The art of computer programming*. Volume 1: *Fundamental
algorithms (second edition).*

[Knuth 74] Knuth, D.E.
Structured programming with GO TO statements.

[Kobayashi 84] Kobayashi, M.
Dynamic characteristics of loops.
*IEEE Transactions on Computers* C-33(2):125-132, February,
1984.

Measurements of parallelism in ordinary FORTRAN programs.

[Laliotis 75] Laliotis, T.A.
Architecture of the SYMBOL computer system.
In Chu, Y. (editor), *High-level language computer architecture*,
[Leblanc 82] Leblanc, R.J. and Fischer, C.N.  
A case study of run-time errors in PASCAL programs.  
*Software - Practice and Experience* 12(9):825-834, September, 1982.  

[LeeJK 84] Lee, J.K.F. and Smith, A.J.  
Branch prediction strategies and branch target buffer design.  

[LeeMK 81] Lee, M.K.  
Code generation for zero-, one-, two-, and three-address machine architectures.  
1981.  
Honours thesis, Australian National University.  

[Lenfant 75] Lenfant, J. and Burgevin, P.  
Empirical data on program behaviour.  

[Lokan 83a] Lokan, C.J.  
*Dynamic analysis of the use of loops and if-statements in Pascal programs.*  

[Lokan 83b] Lokan, C.J.  
*Static prediction of dynamic program profiles.*  

[Ma 81] Ma, P.-Y. R. and Lewis, T.G.  
On the design of a microcode compiler for a machine-independent high-level language.  

[Malik 78] Malik, K.  
*Optimizing the design of a high level language for microprogramming.*  
Marovac, N.  
A systematic approach to the design and implementation of a computer instruction set.  

McCabe, T.J.  
A complexity measure.  

Misherghi, S.H.  
*An investigation of the architectural requirements of SIMULA 67.*  

Myers, G.J.  
The case against stack-oriented instruction sets.  

Myers, G.J.  
An extension to the cyclomatic measure of program complexity.  

Myers, G.J.  
The evaluation of expressions in a storage-to-storage architecture.  

Myers, G.J.  
*Advances in computer architecture (2nd edition).*  

Nanodata Corporation.  
*QM-1 hardware level user's manual*  

Nori, K.V., Ammann, U., Jensen, K., Nageli, H.H. and Jacobi, Ch.  
Pascal-P implementation notes.  

Oldehoeft, R.R. and Bass, L.J.  
Dynamic software science with applications.  
[Oldehoeft 83] Oldehoeft, R.R.
Program graphs and execution behaviour.

[Organick 73] Organick, E.I.
*Computer system organization - the B5700/6700 series.*

[Patterson 80] Patterson, D.A. and Ditzel, D.R.
The case for the Reduced Instruction Set Computer.

[Patterson 81] Patterson, D.A. and Sequin, C.H.
RISC I: a reduced instruction set VLSI computer.

[Patterson 82] Patterson, D.A. and Piepho, R.S.
RISC assessment: a high-level language experiment.

[Pratt 75] Pratt, T.W.
*Programming languages: design and implementation.*

[Ramamoorthy 65] Ramamoorthy, C.V.
Discrete Markov analysis of computer programs.

[Ramamoorthy 77] Ramamoorthy, C.V. and Li, H.F.
Pipeline architectures.

[Rauscher 78] Rauscher, T.G. and Agrawala, A.K.
Dynamic problem-oriented redefinition of computer architecture via microprogramming.
[Ribeyre 77] Ribeyre, P. and Saintoyant, P.Y.
A predictive tool for the improvement of program behaviour.

[Rice 71] Rice, R. and Smith, W.R.
SYMBOL - a major departure from classic software-dominated Von Neumann computing systems.

[Rich 78] Rich, C. and Shrobe, H.
Initial report on a LISP programmer's apprentice.

[Robinson 75] Robinson, S.K. and Torsun, I.S.
An empirical study of FORTRAN programs.

[Russell 78] Russell, R.M.
The Cray-1 computer system.

[Saal 77] Saal, H.J. and Weiss, Z.
An empirical study of APL programs.

[Sakamura 79] Sakamura, K., Morokuma, T., Aiso, H., and Izuka, H.
Automatic tuning of computer architectures.

[Salisbury 76] Salisbury, A.B.
Nanodata QM-1 architecture.

[Salvadori 75] Salvadori, A., Gordon, J., and Capstick, C.
Static profile of COBOL programs.

An analysis of PASCAL programs in compiler writing.
[Shustek 78] Shustek, L.J.  
*Analysis and performance of computer instruction sets.*  

[SmithJE 81] Smith, J.E.  
A study of branch prediction strategies.  

[Spirn 77] Spirn, J.R.  
*Program behaviour: models and measurements.*  

[Stevenson 79] Stevenson, J.W. and Tanenbaum, A.S.  
Efficient encoding of machine instructions.  

[Sweet 82] Sweet, R.E. and Sandman, J.G.Jr.  
Empirical analysis of the Mesa instruction set.  

[Tamine 83] Tamine, J.  
On the use of tree-like structures to simplify measures of complexity.  

[Tanenbaum 78] Tanenbaum, A.S.  
Implications of structured programming for machine architecture.  

[Thall 77] Thall, R.M.  
*Determining the operational properties of computer programs at compile time.*  

[Torsun 81] Torsun, I.S. and Al-Jarrah, M.M.  
Dynamic analysis of COBOL programs.  

[Waters 79] Waters, R.C.  
A method for analysing loop programs.  
[Waters 82] Waters, R.C.  
The programmer's apprentice: knowledge based program editing.  

[Wegbreit 75] Wegbreit, B.  
Mechanical program analysis.  
*Communications of the ACM* 18(9):528-539, September, 1975.

[Wichmann 73] Wichmann, B.A.  
*ALGOL 60 compilation and assessment.*  

[Wiecek 82] Wiecek, C.A.  
A case study of VAX-11 instruction set usage for compiler execution.  

[Wilner 72a] Wilner, W.T.  
Design of the Burroughs B1700.  

[Wilner 72b] Wilner, W.T.  
Burroughs B1700 memory utilization.  

[Wulf 81] Wulf, W.A.  
Compilers and computer architecture.  

[Yourdon 79] Yourdon, E. and Constantine, L.L.  
*Structured design: fundamentals of a discipline of computer program and systems design.*  

[Zelkowitz 76] Zelkowitz, M.V.  
Automatic program analysis and evaluations.  

[Zelkowitz 77] Zelkowitz, M.V.  
Effects of structured programming on PL/I programmers.  
Appendix A
"Ideal" intermediate language for Pascal

Pascal programs are translated into an intermediate language, for execution on the context-sensitive architecture. An intermediate language is proposed now that is ideal for this purpose.

Every operation and standard function defined in Pascal has a corresponding operation in the instruction set proposed below. Some instructions apply to particular data types or data structures; others can handle arguments of various types. These *generic* instructions can be used because data objects are self-defining: tags and descriptors attached to each data object identify its type, and other significant information. All of the data structures and data types of Pascal are catered for.

Note: although an effort has been made to define a language into which each feature of Pascal can be easily translated, it is quite possible that this proposed language contains inadequacies or errors. Detailed language design is beyond the scope of this thesis.

Format codes and an expression-evaluation stack are used to minimize the number of operands that must be named explicitly. When temporary values are computed during the evaluation of an expression, they are directed implicitly to the stack. Thus there is no need to create named temporary variables for expression evaluation. The format code indicates which arguments are named explicitly, which refer to the stack, and which values (if any) are used for two or more operands in the instruction.
Instruction Format

Each instruction is a tuple of the form

\[
\text{number : (format-code, opcode, [operands] [destination])}
\]

The instructions are numbered for reference. The number of the target instruction is encoded in the destination field of each branch instruction; instruction numbers are not otherwise significant.

Each operation requires a known number of arguments. Some take no arguments and produce no result; the rest use zero to three operands to produce a result, the disposal of which is indicated by the destination field. In general, some arguments are named explicitly and the remainder refer implicitly to an expression-evaluation stack: the format code indicates which is which.

If an operand is named explicitly, its value is used as the operand in the expression. If it refers to the stack, the top entry is removed from the stack and its value is used for the operand.

If the destination is a named variable, that variable is treated as the address to which the result is written. If the destination refers to the stack, in most cases the result of the operation is pushed onto the stack - it is a temporary value, which will be used later in the evaluation of the same expression. However, there are some operations for which, if the destination refers to the stack, the top stack entry is removed and its value is treated as the address to which the result is written.

Instruction Set

Instructions for arithmetic and logical operations, and standard functions

These instructions are either unary or binary operators. For a unary operator there are five different possible formats (formats F4, F7, F8, F10, and F11 in Table
3-1 on page 35); for a binary operator there are twelve (F0 to F11 in Table 3-1). Although binary operations in other formats are possible, they are not generated: in each case an equivalent, but more efficient, operation in one of these twelve formats is generated instead.

The result of any of these operations is either written to a named variable or pushed onto the stack.

1. **Unary operators**

   1. Standard functions:

      - Abs
      - Arctan
      - Chr
      - Cos
      - Eof
      - Eoln
      - Exp
      - Ln
      - Negate
      - Odd
      - Ord
      - Pred
      - Sin
      - Sqr
      - Sqrt
      - Succ
      - Trunc

   2. Arithmetic and logic:

      - Add
      - Multiply
      - Intdivide
      - Subtract
      - Divide
      - Modulo

2. **Binary operators**

   1. Logical operators:

      - And
      - Or

   2. Relational operators:

      - LT
      - LE
      - EQ
      - GE
      - NE
      - GT

   3. Arithmetic operators:

      - Add
      - Multiply
      - Intdivide
      - Subtract
      - Divide
      - Modulo

   4. Set operations:

      - In
      - Set-intersect
      - Set-union
      - Set-difference
Assignment

Assign is a unary operator. If the format is one of F4, F7, or F10, a named variable is the destination for the assignment operation: the operand is written to that variable. If the format is F8 or F11 the stack is listed as the destination, indicating that the address to which the assignment is to be made is held in the top stack entry: the top stack entry is removed, and the operand is written to the indicated address.

Instructions for access to elements of data structures

As each of these operations takes either one or two operands to produce its result, the format codes are used in the same way as before. The result of the operation is either written to a named variable or pushed onto the stack.

Sometimes it is desired to calculate an address; at other times it is desired to calculate an address and to obtain the value stored at that address. Separate instructions are provided for these two cases, to avoid the need for extra flag bits.

With-statements cause a problem: they make it necessary for the compiler to create named temporary variables. Normally the scope of a temporary variable is limited to the evaluation of a single expression, and the use of a stack makes them unnecessary. However, the base address of a record named in a With-statement must be saved for the scope of the whole With-statement; this requires a named temporary variable. A base address used in one of the instructions defined below (either explicitly, or implicitly through the stack) could be either a named record, or an address saved in one of these temporary variables.

1. Pointers

The pointer-manipulation operators each take one operand - a pointer.

_Ptr Result is the address pointed to.

_PtrVal Result is the value stored at that address.
2. Arrays

The two arguments to the operations are the array name and an index.

**Arr**  
Result is the address of the array element.

**ArrVal**  
Result is the value stored in that element.

3. Records

The two arguments to the operations are the record name (or a temporary variable holding a base address) and a field name.

**RF**  
Result is the address of the designated field.

**RFVal**  
Result is the value stored in that field.

4. Pointers to records

These operations are included so that two common sequences of instructions can be replaced by single instructions. They take two arguments - a pointer and a field name - and compute the address of the field in the record pointed to. The microcode must include run-time checks to ensure that a record of the right type is indeed pointed to.

**PtrRF**  
result is the address of the field.

**PtrRFVal**  
result is the value stored at that address.

Note that \((\text{PtrRF}, <\text{pointer}>, <\text{field}>, <\text{result}>)\) is equivalent to \((\text{Ptr}, <\text{pointer}>, <\text{computed base address}>)\)

\((\text{RF}, <\text{computed base address}>, <\text{field}>, <\text{result}>)\)

The instruction sequence equivalent to **PtrRFVal** is obvious.
Other manipulation of data structures

1. Files

These operators each take one argument: the file to be manipulated. The format code is either F10, indicating that the file is named in the destination field of the instruction; or F11, indicating that the top stack entry identifies the file. The operators are:

Get  Page  Readln
Put  Reset  Rewrite

2. Sets

The Newset operator takes one argument, and returns a set whose sole element is that argument. The result is written to a named variable or pushed onto the stack.

3. Pointers: dynamic storage management

Creation and disposal of dynamic storage is accomplished with two unary operators. Each takes one argument (a pointer), which is encoded in the "destination" field of the instruction. If the destination refers to the stack, the top entry is removed from the stack and treated as the address of the pointer variable (n.b. this is the variable itself, not the object to which it currently points).

The operators are New and Dispose.

4. Arrays: packing and unpacking

These operations use two operands: an array name and an index. The result, encoded in the "destination" field, must be another array of suitable type. If the destination field refers to the stack, the top stack entry contains a descriptor for the destination array.
The operators are **Pack** and **Unpack**.

**Input/Output**

1. **Input**

   \[(\text{<format>, Read, <source file>, <destination>})\]

   Input is achieved by a generic, unary instruction. The single operand is the file from which the datum is to be read; the destination is the address into which it is to be placed. If the destination field refers to the stack, the top stack entry contains the relevant address.

   As it is a unary operation, the five format codes F4, F7, F8, F10, and F11 would be expected to cover the different possibilities. Most input instructions will indeed have one of F4, F8, or F10 as its format code. F7 and F11 can never occur, as the input file cannot also be the address to which the information is written.

   There is one further possibility, which needs special handling. The input file and the destination may both have had to be computed: in such cases the top two stack entries define the source and destination. None of the formats F0 to F11 covers this situation. F15 is used in those cases. When F15 is found in conjunction with the **Read** operator, it is immediately known that no named arguments are to be read. F15 is thus further overloaded, but in practice this case rarely arises.

2. **Output**: normal usage

   \[(\text{<format>, Write, <operand>, <field width>, <file>})\]

   Output of most data values is achieved by a generic, binary instruction. The two operands are the data object and the field width; the destination is the output file. If the destination field refers to the stack, the top stack entry identifies the output file.
This instruction is used for output of integers, characters, booleans, strings, and real numbers that are written in scientific notation (as explained below, a different instruction is needed for output of real numbers in decimal notation).

The "writeln" operation is simply treated as a "write" operation, in which the operand is an end-of-line character (e.g. see Figure 3-2, page 33).

As two operands are used, the usual twelve format codes cover most cases. Format codes F0, F3, F4, and F5 are usually used; none of the other eight normal codes can be generated with the Write instruction. However, there are some irregular cases that cannot be handled by the usual formats: they are described in the next section.

3. Output: irregularities

   a. Operand and width are the same explicitly-specified numeric value.

   For example:

   (?, Write, 6, 6, <file>)
   (?, Write, 4, 4, <stack>)

   These two instructions cannot be handled by the usual dozen formats, since those formats assume that no value will be repeated in two operands. By overloading formats F12 and F13 respectively, the problem is solved. Thus F13 in conjunction with Write means that one explicit value is to be read and used for each operand, with the result being written to the file defined in the top stack entry; this is immediately distinguishable from the meaning of F13 in conjunction with any other operator. A similar comment can be made for F12.

   b. Two or three of <operand>, <width>, and <file> must be calculated.
The instructions

(?, Write, a , <stack>, <stack>)
(?, Write, <stack>, a , <stack>)
(?, Write, <stack>, <stack>, <file>)
(?, Write, <stack>, <stack>, <stack>)

cannot be handled by the usual formats. Overloading formats F8, F9, F10, and F11 respectively solves the problem (F10 already covers the third one). Each of those formats has one meaning in conjunction with Write, and a different meaning in conjunction with any other operator.

c. A real number is to be written in decimal notation.

As there are two width fields needed, the instruction needs three operands to be defined, in addition to the destination file. None of the normal formats can handle four arguments. The suggested solution is to use F15 to signal this form of output. Thus F15 is further overloaded: it already means one thing for branch instructions, with a separate meaning if "Read" is the operator. Now a further special meaning is added if "Write" is the operator: there are four arguments to be defined. In practice, almost every appearance of F15 should be with the "Write" operator.

A further four flag bits are read. If the first bit is a '1', the <operand> is named explicitly; otherwise it comes from the stack. The second bit defines the source of the first width value similarly; the third bit defines the source of the second width value; the fourth bit defines the source for the <file>. Each explicit argument is then read as usual.

This has the disadvantage that an argument may need to be named explicitly twice, if the <operand> is an explicit number whose
value is the same as one of the width values. However, this should rarely occur.

Transfer of control

1. Branch instructions

Branch instructions may be conditional or unconditional. Conditional branch instructions (Jump-if-false and Jump-if-true) take one operand, a boolean value which determines whether or not the jump is taken. Unconditional Jump instructions take no operands: the jump to the location specified in the \textit{destination} field is always made. The formats F12 and F14 are normally used.

If the destination refers to the stack, the top stack entry identifies the destination of the branch instruction. (This implies that the destination had to be calculated, which is not normally possible in Pascal.)

2. Procedure calls

Procedure or function calls are identified by the \texttt{Call} and \texttt{CallFn} operators respectively. When a call operator is encountered, all of the following instructions are executed until the EndParm instruction is reached: these instructions compute the parameters to be passed. Each parameter is defined by a Parm instruction, which may be preceded by some instructions to evaluate that parameter. Control is transferred to the called procedure after the parameters have all been evaluated.

The Return instruction is used to return from a procedure to the calling routine. Execution resumes in the calling routine, beginning with the first instruction after the EndParm.

Call and CallFn are unary operators, whose argument is the name of the appropriate program unit. Parm is also a unary operator, whose
argument is the parameter being defined. **Return** takes no arguments. **Endparm** takes no arguments; it is purely a statement separator, whose semantics are those of a *no-op*.

3. Program termination

The **Halt** instruction, which takes no arguments and so uses format F11, terminates the execution of the program.

**Redefinition of virtual architecture**

**Change-env** and **Revert-env** take no arguments. Format F11 is used.
Notes on the presentation of the thesis

This thesis was prepared using the typesetting program "Scribe". Occasionally this led to problems. In particular, the placement of figures and tables in the text, and the form of some subscripts and superscripts, is not always ideal. Although it is acknowledged that these could be improved, they do represent reasonable compromises. Some font difficulties were also encountered; thus (on page 138 in line 13) the letter "U" is used to represent the set union operator.

Changes to the body of the thesis

Abstract. Line 16
Change "within in" to "within it".

Table of Contents. Page 1. Line 22
Change "Canonical Measures of Interpretation" to "Canonical measures of interpretation"

Page 20. Line 16
Change "used with formal means of predicate transformers, and estimates" to "used, with formal predicate transformers and estimates".

Page 24. Line 13
Change "Canonical Measures of Interpretation" to "Canonical measures of interpretation".

Page 25. Line 14
Delete the parenthesized comment.

Page 25. Insert the following paragraph after line 20:
"Semantic actions" are the basic operations expressed or implied in the source program. They include arithmetic operations (including built-in functions) defined in the source language; data-accessing operations (such as accessing an array element, or creating or accessing a dynamic variable) defined in the source language; and branching operations required to implement the flow of control. Hence they include "functional" operations that are implied by the semantics of the source language, but not low-level "overhead" operations (such as LOADs and STOREs that move data around to prepare for functional operations).
The language chosen for study in this thesis is Pascal. Pascal was selected because its well-defined syntax and data structures make analysis of the flow of data and control through programs relatively straightforward; the syntax of Pascal provides a good basis for the methods developed here; and a Pascal compiler was available for study.

Page 31. Insert the following paragraph after line 16:

---

Page 32. Line 9

Insert "e.g." after "(".

Page 32. Line 30

Change "can be used" to "can also be used".

Page 33. Line 10. Replace the parenthesized comment with the following:

---

Page 36. Line 7

Insert "(opcodes)" after "operations".

Page 37. Replace the five sentences in lines 19-26 with the following:

---

Page 38. Line 9

Change "facilities" to "facilities like EMMY".

Page 49. Line 4

Enclose "scope of variable definitions" in quotation marks.

Page 49. Line 4

Change "Johnson" to "Johnston".

Page 49. Line 10

Delete the phrase "Labels in different program units certainly refer to different locations;".
Change "different" to "Different".

They certainly use very different sets of labels (at the image program level), as most labels are generated by the compiler and are unknown beyond the program unit in which they occur.

Change "static and dynamic sizes" to "static size and predicted dynamic size".

The model developed here is geared to the language Pascal. It should be straightforward to adapt it to a wide variety of procedural languages.

Two of the programs did not require data, and the execution profile of a third was independent of its data. One execution was made of each of these three programs, to measure their execution profiles.

For each other program, three or four executions were made. An attempt was made to use "average" data; a different data set was used for each execution. From the three or four resulting execution profiles, an "average" profile was found.

This rule fails badly if an "otherwise" branch is present and the domain of selector values is large. Empirical evidence obtained from the dynamic program traces indicates that this rule often predicts a disproportionately high probability of selecting the "otherwise" branch.

Much more empirical evidence is needed before the development of a reasonable general static model of Case-statements can be attempted.

The vertical axes on the graphs in Figures 5-1 to 5-7 should be labelled "Number of cases".
Page 72. Line 3

Change "otherwise" to "Otherwise".

Page 73. Line 8

Change "If-l(b)" to "IF-l(b)".

Page 74. Line 5

Change "otherwise" to "Otherwise".

Page 76. Line 2

Delete this sentence.

Page 76. Line 8

Change "3% of loops" to "3% of those loops".

Page 79. Replace lines 8-11 with the following:

If \( m \geq 50 \), assume \( n_{\text{iter}} = 10 \).
If \( 50 > m \geq 15 \), assume \( n_{\text{iter}} = \text{mean of } 5 \text{ and } m/3 \).
If \( 15 > m \geq 5 \), assume \( n_{\text{iter}} = 5 \).
If \( m < 5 \), assume \( n_{\text{iter}} = m/2 \).

Page 81. Line 6, and hereafter.

Change "FOR(a)" to "FORV(a)".

Page 81. Line 7, and hereafter.

Change "FOR(b)" to "FORV(b)".

Page 81. Line 11, and hereafter.

Change "FOR(c)" to "FORC".

Page 82. Replace lines 4-6 with the following:

If For-loops with variable bounds are included, the rules are altered to cover all loops with unpredictable iteration counts. LOOP(a) is altered, producing LOOP(d), by assuming \( n_{\text{iter}} = 5 \) instead of \( n_{\text{iter}} = 4 \). LOOP(b) and LOOP(c) are altered, producing LOOP(e) and LOOP(f) respectively, by predicting \( n_{\text{iter}} = 5 \) in the "otherwise" clause instead of \( n_{\text{iter}} = 4 \).
If a program unit contains a Goto-statement that is recognized as simulating a basic iteration or alternative statement, the rule for the simulated statement is used to modify the predicted execution frequencies of the relevant statements. Otherwise, the above rule GOTO is used (thus it is used for both GOTO-1 and GOTO-2). Conditional Goto-statements are assumed by this rule to have no effect. Unconditional Goto-statements will increase the number of times their target statement is executed, by the number of times they themselves are executed.

These results show the effect of applying this scheme. No attempt has been made to analyse the complexity of the algorithms themselves, or to measure their execution performance.

The approach described in this section can be applied to programs written in a wide variety of procedural programming languages - especially those in the "Algol family". Here it is applied particularly to programs written in Pascal.

Note however that this part of the analysis was not actually implemented. It is thus described in outline only.
Delete the phrase "out of sets of 6 and 4 operators and 4 and 5 operands,\)."

Insert ")" at the end of this line.

Change "to the list of" to "to the list".

Change "is be" to "is".

Change "thresholds" to "the thresholds T, and T_4\)."

Change "thresholds" to "the thresholds T, and T_4\)."

Change a from "1" to "2".

Change "log_4\)3" to "log_4\)4".

Change "12" to "15" in each instance.

Change "-32" to "-29" in each instance.

Change "-12" to "-9".

Change "34" to "37".

Change "after each" to "after all".
Change "of step 1" to "in the first step of the whole procedure".

Page 120. Line 20. Insert the following phrase after "code,":

the syntax tree could be constructed during compilation and could persist until needed for this analysis.

Change "The latter approach" to "The last approach".

Enclose "ideal" in quotation marks.

Interchange lines 2 and 3.

Change "worth while" to "worthwhile".

Enclose "ideal" in quotation marks.

That effort is reduced if the potentially most useful combinations are checked first, and the search stops when it seems unlikely that the best case seen so far can be bettered. The best segmentation is then found as quickly as possible.

On average about 80% of nodes are rejected at this stage.

Change "in a set" to "in such a set".

Change "no more" to "No more".
Page 152. Insert the following after line 25:

(The right hand column of Table 7-1 is explained later.)

Page 154. Insert the following after line 8:

They are thus termed "constraints". The lower the value of \( \nu \), the tighter the constraint it represents.

Page 155. Replace line 14 with the following:

The six orderings indicated by the tightest constraint can be grouped into three pairs: the two orderings with the same two sets on the end form a pair. The second tightest constraint is used to rank the orderings within each pair: one of them is more likely than the other to satisfy that constraint, because it includes one less set of extraneous operands.

Page 168. Line 4

Change "quick" to "straightforward".

Page 170. Line 8

Change "were" to "were".

Page 176. Line 1

Change "arrangement" to "arrangement".

Page 185. Line 15

Change "Addison-wesley" to "Addison-Wesley".

Page 191. Line 17

Change "Johnson" to "Johnston" in each instance.

Page 194. Line 10

Change "requirements" to "requirements".

Page 199. Line 2

Insert ", developed by the author, " after "An intermediate language".

Page 200. Line 2

Delete the comma after "opcode".
Page 206. Replace lines 5-7 with the following:

The Pascal "writeln" statement is treated as a series of "write" operations, the last of which has an end-of-line character as its operand (e.g. see Figure 3-2, page 33).