The Production of Optimised Machine-Code for High-Level Languages using Machine-Independent Intermediate Codes.

Peter Salkeld Robertson

D 42234/82

Ph. D.

University of Edinburgh

1981

ABSTRACT

The aim of this work was to investigate the problems associated with using machine-independent intermediate codes in the translation from a high-level language into machine code, with emphasis on minimising code size and providing good run-time diagnostic capabilities.

The main result was a machine-independent intermediate code, I-code, which has been used successfully to develop optimising and diagnostic compilers for the IMP77 language on a large number of different computer systems. In addition, the work has been used to lay the foundations for a project to develop an intermediate code for portable SIMULA compilers.

The major conclusions of the research were that carefully designed machine-independent intermediate codes can be used to generate viable optimising and diagnostic compilers, and that the commonality introduced into different code generators processing the code for different machines simplifies the tasks of creating new compilers and maintaining old ones.

Contents

1. Introduction

Compilers for high-level languages form a significant part of most computer systems, and with an ever increasing number and variety of machine architectures on the market the problems of compiler development, testing, and maintenance consume more and more manpower and computer time. Moreover, as computer technology is improving and changing rapidly it is becoming evident that software costs will increasingly dominate the total cost of a system. Indeed, it may not be long before the lifetime of software regularly exceeds that of the hardware on which it was originally implemented, a state of affairs quite different from that envisaged by Halpern when he concluded that "the importance of the entire question of machine-independence is diminishing .." [Halpern, 1965]. In addition, there is a need to encourage the slowly-developing trend to write the majority of software in high-level languages. Even though the advantages of such an approach are many, a large number of users still have a love of machine-code, usually fostered by thoughts of "machine efficiency". Clearly, techniques must be developed to simplify the production of usable compilers which can "optimise" the match between the executing program and the user's requirements, be they for fast execution, small program size, reasonable execution time but with good run-time diagnostics, or whatever.

One popular method for reducing the complexity of a compiler is to partition it into two major phases: one language-dependent and the other machine-dependent. The idea is that the language-dependent phase inputs the source program and deals with all the syntactic niceties of the language, finally generating a new representation of the program, an intermediate code. This is then input by a second phase which uses it to generate machine-code for the target computer. In this way it should be possible to produce a compiler to generate code for a different machine by taking the existing first phase and writing a new second phase. This ability to move a large portion of the compiler from machine to machine has led to such compilers being referred to as "portable compilers" even though the term is perhaps misleading, as only part of the complete compiler can be moved without change. In practice many existing compilers generate intermediate representations of the program which are passed around within the compiler, for example the "analysis records" produced by the syntactic phase of compilation, but for the purposes of this work it is only when these representations are machine-independent and are made available outwith the compiler that they will be termed intermediate codes.

Much of the emphasis in designing intermediate codes has been on enabling a compiler to be bootstrapped quickly onto a new machine - either by interpreting the intermediate code, or by using a macro generator to expand it into machine-code [Brown, 1977]. Once this has been done the intention is that the quality of the code so produced can be improved at leisure. While this approach has been very successful and relatively error-free, it has been the experience of several implementors that it is difficult to adapt the scheme to produce highly optimised code [Russell, 1974]; apparently considerations of portability and machine-independence have caused the problems of optimisation to be overlooked. The aspect of intermediate-code design which has received most debate concerns the level of the code: low-level with a fairly simple code-generator, or high-level with a more complex code-generator [Brown, 1972].

This thesis attempts to put machine-independence and optimisation on an equal footing, and describes the use of an intermediate code which takes a novel view of the process. Instead of the intermediate code describing the computation to be performed, it describes the operation of a code-generator which will produce a program to perform the required computation. This effectively adds an extra level of indirection into the compilation, weakening any linkage between the form of the intermediate code and the object code required for a particular implementation.
In essence I-code attempts to describe the results required in a way which does not constrain the method of achieving those results.

In particular it should be noted that the code described, I-code, was designed specifically for the language IMP-77, a systems implementation language which contains many of the constructions which pose problems for optimisation [Robertson, 1979]. It in no way attempts to be a "universal" intermediate code. Notwithstanding, the code, with a small number of minor extensions to cover non-IMP features, has been used successfully in an ALGOL 60 compiler and is currently proving viable in projects for writing Pascal and Fortran 77 compilers.

The intermediate code as finally designed is completely machine independent, except inasmuch as the source program it describes is machine dependent, demonstrating that the problems may not be as intractable as thought by Branquart et al. who state that "clearly complete machine independency is never reached" [Branquart, 1973].

In addition to the problems of machine independence there is also the question of operating system independence, as nowadays it is common for machines to have several systems available. For this reason the task of producing a compiler is far from finished when it can generate machine code [Richards, 1977]. To simplify the generation of versions of a compiler for different operating systems, a third phase of compilation was added, although it soon became clear that the extra phase could be used for other purposes as well, as will be shown in section 4.

Throughout the text, examples are given of the code produced by compilers written to demonstrate the power of the intermediate code. The examples of the intermediate code are couched in terms of mnemonics for the various code items, although the production compilers use a compacted representation. The code and its representations are described in Appendix A1 and Appendix A2.

In the examples of code generated for various constructions, it should be appreciated that the exact instructions and machine features used will depend very much on the context in which the code is produced, and so only typical code sequences can be given.

The machines for which code is demonstrated are indicated by the following abbreviations in parentheses:

(Nova)	Data General NOVA
(PDP10)	Digital Equipment Corporation PDP10
(PDP11)	Digital Equipment Corporation PDP11
(VAX)	Digital Equipment Corporation VAX 11/780
(GEC4080)	General Electric Company 4080
(ICL2900)	International Computers Limited 2900
(4/75)	International Computers Limited 4/75
(7/16)	Interdata 7/16
(7/32)	Interdata 7/32
(PE3200)	Perkin Elmer 3200

2 Intermediate codes

This section gives a brief account of the more important intermediate codes which have been discussed and have had an influence on the design of I-code.

2.1 Uncol

UNCOL, UNiversal Computer Orientated Language, [Mock, 1958], was an early attempt to specify a means for solving the M*N problem of producing compilers for M languages to run on N machines. It was proposed that an intermediate language, UNCOL, be defined which would be able to express the constructs from any language, and which could itself be translated into code for any machine, resulting in the need for only M+N compilers. Indeed it was even suggested that programs would be written directly in UNCOL rather than in machine code.
These ideas were very ambitious, but were presented without any concrete examples of what UNCOL might look like. Proposals were made for an UNCOL in [Steel, 1961] but the work was abandoned before anything like a complete specification had been produced.

An UNCOL-like technique which has been used extensively, is to compile for a known type of machine, such as the IBM 360, and then emulate that machine on the target machine. Unfortunately, to give this any chance of being efficient, microcode support will be necessary and this is rarely available to compiler writers.

2.2 Janus

The first attempt at generating an UNCOL which seems to have been at least partially successful was JANUS [Coleman, 1974]. The approach was effectively to enumerate all the mechanisms found in current programming languages and the techniques used to implement them. From this large list was defined a set of primitive data-types and operations upon them. These primitives were then put together to model the objects in the source language. Once JANUS code had been produced the intention was that it would either be interpreted or compiled into machine code by a macro generator.

2.3 OCODE

Of all the languages which claim to be portable, perhaps the most successful has been BCPL [Richards, 1971]. The BCPL compiler generates the intermediate code OCODE which can either be interpreted or translated into machine code for direct execution. As BCPL is a fairly low-level language with only one data type, the word, many of the difficulties in designing intermediate codes do not arise. This means that the code can be pitched at a low level and be "semantically weak" without compromising the efficiency of the compiled code to any great extent.

The OCODE machine works by manipulating single-word objects held on a stack, into which there are several pointers.

e.g. R(1, 2, 3)

```
STACK 3         adjust the top of stack to leave two
                cells free for linkage information.
LN 1            stack the constant 1.
LN 2            stack the constant 2.
LN 3            stack the constant 3.
LL L6           stack the address of label L6 (the entry
                to the routine).
RTAP 5          enter the procedure adjusting the stack
                frame pointer by 5 locations.
.....
.....
.....
ENTRY 1 L6 'R'
                entry point for the routine R.
SAVE 5          set the top of stack pointer to be 5
                locations from the stack frame pointer.
.....
RTRN            return.
```

2.4 P-code

P-code is the intermediate code used by the PASCAL<P> compiler [Nori, 1976; Jensen, 1976] and was designed with the aim of porting PASCAL quickly by means of an interpreter. In this respect it has been very successful, especially on microprocessor-based systems. The code is similar to OCODE but has a greater range of instructions to handle objects of differing types.

```
procedure ERROR(VAL:INTEGER);  begin
                    0:   ENT   4
   TOTAL := TOTAL+1;
                    1:   LDO   138      Stack TOTAL
                    2:   LDCI  1        Stack 1
                    3:   ADDI           Integer add
                    4:   SRO   138      Store into TOTAL
   if INDEX >= 9 then begin
                    5:   LDO   139
                    6:   LDCI  9
                    7:   GEQI           Compare top elements
                    8:   FJP   17       Jump if false
      LIST[10].NUM := 255
                    9:   LAO   140      Stack base of LIST
                   10:   LDCI  10
                   11:   DEC   1        Subtract 1
                   12:   IXA   2        Index*2+base
                   13:   INC   1        Add 1
                   14:   LDCI  255
                   15:   STO
   end else begin
                   16:   UJP   28
      INDEX := INDEX+1;
                   17:   LDO   139
                   18:   LDCI  1
                   19:   ADDI
                   20:   SRO   139
      LIST[INDEX].NUM := VAL
                   21:   LAO   140
                   22:   LDO   139
                   23:   DEC   1
                   24:   IXA   2
                   25:   INC   1
                   26:   LOD   0, 4
                   27:   STO
   end;
end;
                   28:   RETP           Return
```

2.5 Z-code

Z-code [Bourne, 1975] is the intermediate code produced by the ALGOL68C compiler, the main feature of which is the ability for the user to parameterise the first phase to modify the Z-code to suit the target machine, an idea previously investigated in SLANG [Sibley, 1961]. A set of eight registers is assumed by the code and others may be specified explicitly for each transfer. The memory with which the code works is assumed to be "a linear data store that is addressed by consecutive integers", addresses taking the form of base+displacement pairs. Intermingled with the instructions are directives which control the translation of the code into machine orders. Two of these directives are used to divide the code into "basic blocks" or "straight-line segments", and describe the usage of registers on entry to and exit from the blocks, although little use seems to be made of them at present.

As an example here is the Z-code generated by the PDP10 version of the compiler [Gardner, 1977]:

```
int X := 2, Y := 3, Z := 2
                  1:      F000 10 0 +2     load 2
                          F040 10 6 +144   store in X
                          F000 10 0 +3     load 3
                          F040 10 6 +145   store in Y
                  5:      F000 10 0 +2     load 2
                          F040 10 6 +146   store in Z
proc P = (int A, B) int: begin
                  7:      S715 p*Z
                          T246 677 712
   if A > B
                  9:      R0
                 10:      F020 10 5 +4     load A
                 11:      F022 10 5 +5     subtract B
                 12:      F113 10 0 P713   ->L713 if <=
   then A
                 13:      F020 10 5 +4     load A
   else B
                 14:      H116 0 p714      ->L714
                 15:      L713
                 16:      F020 10 5 +5     load B
   fi
                 17:      L714
                 18:      R1 10 1
end
                 19:      R1 10 1
                 20:      T247 667 712     end of P
```

2.6 Summary and conclusions

2.6.1 Error checking and reporting

The UNCOL approach of having one code for all languages and machines may well simplify the generation of some sort of compiler, but has the major disadvantage that the optimisation of error checking and reporting run-time errors cannot be left to the code generator - many errors are language-dependent and the code generator cannot know how to handle all of them. Instead the checks must be programmed into the intermediate representation explicitly. As will be shown later (5.3) this can inhibit several very powerful and effective optimisations. Sadly, this problem can result in the absence of all but the most trivial of run-time checks in the compiled code.

Even when checking is provided in the intermediate code, as in the case of P-code with its CHK instruction for range testing, it is rare for the code to contain enough information to permit the error to be reported in source program terms: line numbers, procedure names, variable names and values etc. As an example, many P-code interpreters locate run-time errors in terms of 'P-code instruction addresses' which are of negligible benefit to most users.

2.6.2 Efficiency

Commonly, little attention is paid to questions of run-time efficiency in the generation of intermediate code. An exception to this is Z-code which is parameterised in order that the match between the code and the target machine can be improved. In particular, the machine-independent phase is intended to perform simple register optimisation, although as the example in 2.5 shows, the insistence on repeatedly using one register will minimise any gains from remembering register contents. However, this is probably just a failure on the part of the current compilers and could be corrected at a later date. Unfortunately, the fact that the compiler purports to optimise the intermediate code inhibits the code generator from attempting any but the most trivial peephole optimisations, as may be seen in the example by considering instructions 10-12. On many machines the subtract operation is not a good choice for value comparison as firstly it may fail with overflow, and secondly it will corrupt a register. A better implementation would be to replace the subtract with a suitable COMPARE, leaving the register untouched and available for later use. This cannot be done by the code generator as it cannot know that the intermediate code does not go on to use the result of the subtraction later.

Similarly, if Z-code had chosen to use a COMPARE instruction in the first place, a machine without a compare would have to work hard to make sure all registers involved in the necessary subtract were restored to their initial values before the intermediate code goes on to use them.

2.6.3 Assumptions

Most machine-independent codes have been designed, at least initially, assuming a linear store with one address increment corresponding to one basic object. In the case of O-code this is a direct result of the language definition, but in languages such as PASCAL it has led to a great loss of information, as the rich information about data types cannot be expressed. The problems associated with putting languages onto machines with different addressing schemes has resulted in some intermediate code generators being updated to accept a limited form of parameterisation to define the gross appearance of the target machine. Typical of the limitations of these codes is P-code where although the basic types of object can have differing sizes of machine representation, objects with enumerated types will always be given a 'fullword' even though the host machine could easily support a smaller item. A typical assumption is that the difference between objects explicitly specified in the original source and those created by the intermediate code generator for its own purposes is insignificant. As will be shown in section 4.6, this is not necessarily the case.

2.6.4 Interpretation

The vast majority of machine-independent intermediate codes in current use have been designed in such a way as to permit execution by interpretation. This immediately imposes constraints on the form of the code, as, for example, it will need to be possible to pre-process the code into some consistent and managable internal form for the benefit of the interpreter. In order to give some sort of efficiency to the interpretation process, the intermediate code of necessity must become like the order code of a 'real' machine. This results in code-generation being seen as fitting the target machine to the intermediate code, rather than fitting the intermediate code to the target machine which is clearly the better strategy for optimisation.

3 Optimisations

The task of any compiler for a high-level language is to fit programs written in that language onto a specific computer system so that the required computations may be performed.

Optimisation may be described as the process by which the fit is improved. Usually the quality of the optimisation is measured in terms of two parameters: the size of the running program, and, more commonly, the speed at which it executes. While it is possible in some cases to make a program smaller and increase its speed of execution, it is well-known that, in general, speed and size are complementary. For example, the following code fragments have the same effect, but the first will probably be smaller than the second, which will execute faster than the first:

```
 -------------------------- -----------------------
!                          !    A(1) = K            !
!                          !    A(2) = K            !
!  for J = 1,1,8 cycle     !    A(3) = K            !
!     A(J) = K             !    A(4) = K            !
!  repeat                  !    A(5) = K            !
!                          !    A(6) = K            !
!                          !    A(7) = K            !
!                          !    A(8) = K            !
!                          !    J = 8               !
 -------------------------- -----------------------
```

3.1 Classification of optimisations

With a subject as complex as optimisation it is difficult to give a useful and definitive classification of the various possibilities for improving programs. In addition, different authors have used many different terms to describe optimisations which have been attempted [Aho, 1974; Lowry, 1969; Wichmann, 1977]. However most optimisations fall into one of the following four groups: Universal, Local, Global and Source.

3.1.1 Universal Optimisations

These are those optimisations which are independent of any particular program, but which depend on the complete environment in which the program is to be compiled or executed. They are the policy decisions taken by the compiler writer during the initial design of the compiler, and include such things as the fixed use of registers (stack pointers, code pointers, link registers etc), the representations of language-defined objects (arrays, records, strings etc), and the standards for communication with external objects.

In addition, universal optimisation must take into account such questions as:

i Compilation speed or execution speed?

If the compiler is to be used in an environment where programs are compiled roughly as often as they are executed, such as in a teaching environment, execution time can be sacrificed for a decrease in compilation time, as the latter will commonly greatly exceed the former.

ii Diagnostics?

If the compiler is to produce code which will provide extensive checking and will give diagnostic information in the event of program failure, allowance must be made for the efficient checking of the program's behaviour and the maintenance of the recovery information used by the diagnostics. If highly optimised code is required these constraints may not apply.

In the current state of the art universal optimisation is done by experience and guesswork; attempts at producing compiler-compilers which can approach the quality of hand-constructed compilers have not met with great success [Brooker, 1967; Feldman, 1966; Trout, 1967]. As will be shown later (4.5), minor changes in the universal optimisation can result in major changes in the form of the generated code, and so rules made at this stage should be as flexible as possible to permit changes to be made in the light of experience.

From the point of view of measurement, universal optimisation provides the base level from which other optimisations are investigated. Roughly, the better the universal optimisation the less effective the other optimisations appear to be.

3.1.2 Local optimisations

Local optimisations may be defined as those optimisations which are performed during a sequential scan of the program, using only knowledge of statements already processed. Not only are these optimisations reasonably simple to perform but they can have a major effect on the generated code. Indeed Wulf et al. state that "In the final analysis the quality of the local code has a greater impact on both the size and speed of the final program than any other optimisation" [Wulf, 1975].

3.1.2.1 Remembering

Remembering optimisations are those optimisations which can be applied to single statements in the light of information gathered during the compilation of previous statements. These optimisations depend on remembering the current state of the machine and applying this knowledge to subsequent statements. Their chief characteristic is that they are applied during a sequential scan of the program, and as such are reasonably cheap to implement and execute.

For example:

```
 ----------------
| X = Y          |
| if X = 0 start |
 ----------------
```

on the PDP11 would generate:

```
 ------------
| MOV  Y,X   |
| BNE  $1    |  remembering that the previous
|            |  line sets the condition code.
 ------------
```

The most powerful of the remembering optimisations is that whereby the correspondence between values in registers and values in store is remembered and references to the store value are replaced by references to the register's value, register operations usually being smaller and faster than their store equivalents. Unfortunately there are several cases where this leads to worse code than the "obvious" version. For example, on the (PE3200) the code on the right is larger and slower than that on the left:

```
        -----------
       | X = 2     |
       | P = P<<2  |
        -----------

 ----------- -------------
| LIS  3,2  | LIS 3,2      | pick up 2
| ST   3,X  | ST  3,X      | store it in X
| L    1,P  | L   1,P      | pick up P
| SLLS 1,2  | SLL 1,0(3)   | shift it by 2
| ST   1,P  | ST  1,P      | store it in P
 ----------- -------------
```

In addition to keeping track of the changes in the state of the machine from the compilation of one statement to another, remembering also includes preserving this state or environment for later use when a label is encountered, either by merging the current environment with the environment saved at the jump to the label, or simply by restoring that latter environment when it is not possible for control to "fall through" from the statements immediately preceding the label.

In all forms of remembering it is vital to be able to keep the information up-to-date, invalidating knowledge when it becomes false, a process which is exacerbated when it is possible for an object to be known by two or more apparently different descriptions as in the following code:

```
integer J, K
integerarray A(1:12)
integername P
P == J
J = 1;  K = 1
```

At this point P and J refer to the same location as do A(J) and A(K).
Except in the most simple of cases all that can be done is to assume the worst and forget anything potentially dangerous after writing to unknown addresses.

3.1.2.2 Delaying

Delaying is the process of generating instructions but not planting them in the code sequence until it is absolutely necessary. This is of advantage if it is discovered that such "pending" instructions are not needed, or can be combined with other instructions.

The two common cases are illustrated below:

```
 ----------------------------------
|  integerfn F(integername X)      |
|     integer T                    |
|     T = X                        |
|     T = 0 if T < 0               |
|     X = 1                        |
|     result = T                   |
|  end                             |
 ----------------------------------
```

The obvious code for the body of this function is (PE3220):

```
 ------------------
|     L   3,X      | address of parameter
|     L   0,0(3)   | value of parameter
|     ST  0,T      |
|     BGE $1       | -> if T >= 0
|     SR  0,0      |
|     ST  0,T      | T = 0
|  $1:LIS 2,1      |
|     ST  2,0(3)   | X = 1
|     LR  1,0      | load result
|     {return}     |
 ------------------
```

By delaying the first store into T until after the conditional statement, and delaying the second store into T until after the label, both instructions can be combined, resulting in the code:

```
-------------------
|     L   3,X      |
|     L   0,0(3)   |
|     BGE $1       |
|     SR  0,0      |
|  $1:ST  0,T      |
|     LIS 2,1      |
|     ST  2,0(3)   |
|     LR  1,0      |
|     {return}     |
-------------------
```

This store itself can now be delayed until the return from the function, at which point, as T is local to the function and will be destroyed, the instruction can be deleted altogether.
Section 3.2 gives a description of one way in which this sort of optimisation has been achieved.

3.1.2.3 Inaccessable code removal

In several cases compilers can generate code which will never be executed.

The common causes of this are either user-specified conditions whose truth is constant and are used to achieve some sort of "conditional compilation", or structural statements following unconditional jumps as below:

```
 ----------------------
|  if X = 0 start      |
|     ERROR(1)         |
|     return           |
|  else                |
|     V = V-X          |
|  finish              |
 ----------------------
```

Here the branch instruction usually generated at the end of the _if_ clause to take control past the _else_ clause, can never be executed.

Such inaccessable code can be eliminated to shorten the program, but without directly effecting its speed of execution.

3.1.2.4 Peephole optimisations

Peephole optimisation [McKeeman, 1965] is the technique of examining small sections of generated code to make fairly obvious, but ad hoc, improvements. Many of the gains from the optimisation come by simplifying code sequences created through the juxtaposition of code areas which were produced separately.

For example (PE3220):

```
          Before              After
      ---------------- -------------------
      ! ST  4,X       !    ST  4,X         !
      ! L   4,X       !                    !
      !-             -!-                  -!
      ! AR  1,2       !                    !
      ! AHI 1,48      !    AHI 1,48(2)     !
      ---------------- -------------------
```

3.1.2.5 <u>Special cases</u>

Special-case optimisations are those which make use of the particular structure and features of the target machine to control the way in which certain statements are implemented.

For example:

```
                  Obvious          Optimised
             ---------------- -----------
(PDP11)     !  MOV  #0,X     !  CLR    X !   X = 0
            !-              -!-         -!
(PDP11)     !  ADD  #1,X     !  INC    X !   X = X+1
            !-              -!-         -!
(PE3220)    !  LHI  1,NULL   !  SR  0,0  !   S = ""
            !  LHI  2,S      !  STB 0,S  !
            !  BAL  15,MOVE  !           !
             ---------------- -----------
```

These optimisations are very similar to peephole optimisations but are distinguished because they actively control the generation of code rather than passively alter code which has already been produced. In particular they avoid one of the drawbacks of peephole optimisation, namely that even though it can reduce fairly complex instruction sequences to a simpler form, the side-effects of generating the long form in the first place often degrade the code. In the example above of setting a string variable to the null string, the optimised form uses only one register, the value of which can be remembered. In the non-optimised version three registers are immediately altered and the knowledge of the contents of all of the registers may need to be forgotten unless the code generator knows how the MOVE routine works and can forget only those registers which it uses.

3.1.2.6 Algebraic manipulations

Algebraic optimisations are improvements brought about by using the algebraic properties of operators and operands, and include:

. Folding, or compile-time evaluation

1+2 is replaced by 3

. Removal of null operations

A+0 is replaced by A

. Using commutative properties

-B+A is replaced by A-B

3.1.3 Global optimisations

Global optimisation may be defined as those improvements to the code which require knowledge of substantial parts of the program. In effect they are performed by examining numbers of statements in parallel, in contrast to the sequential scan required by local optimisation.

3.1.3.1 Restructuring

Restructuring optimisations are those optimisations which may be brought about by changing the order in which the code is laid out in memory without otherwise changing the nature of the code. As will be discussed later (section 4.3), there are many reasons why programs can be improved by changing the order of chunks of the code. A common reason is that many machines have conditional branch instructions with limited range while the unconditional branches have a much larger range.

Hence if {A} represents a large number of statements and {B} represents a small number of statements, the program:

```
 ----------------
 ¦ if X = 0 start ¦
 ¦    {A}         ¦
 ¦ else           ¦
 ¦    {B}         ¦
 ¦ finish         ¦
 ----------------
```

could be improved by reordering as on the right (PDP11):

```
         original               reordered
 ------------------  ----------------
 ¦      MOV  X,R0   ¦      MOV  X,R0  ¦
 ¦      BEQ  $1     ¦      BEQ  $1    ¦
 ¦      JMP  $2     ¦      {B}        ¦
 ¦ $1:  {A}         ¦      JMP  $2    ¦
 ¦      BR   $3     ¦ $1:  {A}        ¦
 ¦ $2:  {B}         ¦ $2:             ¦
 ¦ $3:              ¦                 ¦
 ------------------  ----------------
```

3.1.3.2 Merging

3.1.3.2.1 Forward merging

Forward merging, also somewhat confusingly referred to as "cross jumping" [Wulf, 1975], is the process whereby the point of convergence of two or more code sequences is moved back over common sub-sequences thus removing one of the sub-sequences, as in the case below.

```
 ----------------------
 |  if X > Y start     |
 |     TEST(X, Y)      |
 |  else               |
 |     TEST(Y, X)      |
 |  finish             |
 ----------------------
```

```
            obvious code  (VAX)   after merging
        -------------------- --------------------
        |     CMPL   X,Y    |      CMPL   X,Y    |
        |     BLE    $1     |      BLE    $1     |
        |     PUSHL  X      |      PUSHL  X      |
        |     PUSHL  Y      |      PUSHL  Y      |
        |     CALLS  2,TEST |                    |
P1 ->   |     BRB    $2     |      BRB    $3     |
        |  $1:PUSHL  Y      |   $1:PUSHL  Y      |
        |     PUSHL  X      |      PUSHL  X      |
        |     CALLS  2,TEST |   $3:CALLS  2,TEST |
P2 ->   |  $2:              |   $2:              |
        -------------------- --------------------
```

The simplest way to perform this optimisation is to take the code sequence about the point of a label and a reference to that label, and set two pointers: one, P1, to the unconditional jump and the other, P2, to the label. If the instructions immediately preceding the pointers are identical both pointers are moved back over that instruction.

The label is redefined at the new position of P2 and the instruction passed over by P1 is deleted. The process is repeated until either another label is found or two different instructions are encountered. The redefinition of the label involves creating a completely new label, leaving the old one untouched. This both prevents trouble with multiple references to the label and permits the optimisation to be attempted on those references.

As this optimisation simply causes the sharing of execution paths there is no direct gain in execution speed, but as the code size is reduced an indirect improvement may be achieved if the shorter code moves the label close enough to the reference to it for a shorter and usually faster jump instruction to be used.

The optimisation obviously must be performed while labels and jumps are in a symbolic form, that is before code addresses have been resolved. This permits the merging of instructions which will eventually have program-counter relative operands and consequently be position dependent.

3.1.3.2.2 Backward merging

A second, but much more difficult form of merging involves moving instructions back over the preceding branch code which generates the two paths being considered.

```
                Original  (PE3200)   Optimised
          ------------------- -------------------
          |                  |       L   2,R      |
          |      L   1,X     |       L   1,X      |
          |      BNE $1      |       BNE $1       |
P1 ->     |      L   2,R     |                    |
          |      LIS 3,1     |       LIS 3,1      |
          |      ST  3,A(2)  |       ST  3,A(2)   |
          |      B   $2      |       B   $2       |
P2 ->     |   $1:L   2,R     |    $1:             |
          |      LIS 3,3     |       LIS 3,3      |
          |      ST  3,B(2)  |       ST  3,B(2)   |
          |   $2:            |    $2:             |
          ------------------- -------------------
```

The difficulty with this optimisation is that it requires the branch and the associated condition testing code to be treated as a single unit, so that merged instructions do not split the test and the use of the result. Also the testing instructions must be checked to ensure that they are not able to modify the operands of the merged instructions. This information is easily available to the code-generator as in IMP77 only procedure calls and string resolution can have such side-effects. In a way similar to the other form of merging the two pointers, P1 and P2 are set and adjusted; P1 being moved forward over common code carrying the branch sequence with it (L & BNE), and P2 being advanced, deleting the code it passes over.

3.1.3.3 Advancing

Advancing is the process of moving operations back in the instruction stream so that they are executed earlier and pave the way for improving subsequent statements.

On many machines the statements:

```
 --------------
|  X = X-1     |
|  A(X) = P    |
|  X = X-1     |
|  A(X) = Q    |
 --------------
```

could be compiled to more efficient code if rewritten:

```
 --------------
|  X = X-2     |
|  A(X+1) = P  |
|  A(X) = Q    |
 --------------
```

as only one calculation will need to be done to address both A(X) and A(X+1), the constant, suitably scaled, being added into the displacement field of the appropriate instruction (PDP11):

```
 ----------------
|  SUB   #2,X    |
|  MOV   X,R1    |
|  ADD   R1,R1   |
|  ADD   A,R1    |
|  MOV   P,2(R1) |
|  MOV   Q,(R1)  |
 ----------------
```

3.1.3.4 Factoring

Factoring is the generalisation of merging and involves the removal of common sections of code. Included under this heading is the elimination of common sub-expressions.

At the source level this can be seen in changes such as:

```
 -----------------------------
|  D = SIN(X^2) + COS(X^2)    |
 -----------------------------
```

being replaced by

```
 ----------------------------
|  real T                    |
|  T = X^2                   |
|  D = SIN(T) + COS(T)       |
 ----------------------------
```

At the machine level the optimisation is often available as the result of address arithmetic in the case of simple arrays:

```
 -----------------
|  A(J) = B(J)    |
 -----------------

   Original  (PE3200) Optimised
 ----------------- -----------------
|  L    1,J       |   L    1,J       |
|  SLLS 1,2       |   SLLS 1,2       |
|  AR   1,LNB     |   AR   1,LNB     |
|  L    3,J       |                  |
|  SLLS 3,2       |                  |
|  AR   3,LNB     |                  |
|  L    0,B(3)    |   L    0,B(1)    |
|  ST   0,A(1)    |   ST   0,A(1)    |
 ----------------- -----------------
```

In this case as the code-generator is in complete control the optimisation can be very simple, although rather specific.
The techniques for handling common sub-expressions have been investigated at length by several authors, but measurements indicate that in most programs expressions are so trivial the expense in finding common sub-expressions is not repaid by the resulting improvement in the generated code [Knuth, 1971].

The more general form of factoring can be seen in the transformation of the following statements:

```
 ----------------------------------------
|  if X = 0 then A = 1 else B = 2         |
|  if X = 0 then C = 3 else D = 4         |
 ----------------------------------------
```

into:

```
 -----------------------
|  if X = 0 start       |
|     A = 1             |
|     C = 3             |
|  else                 |
|     B = 2             |
|     D = 4             |
|  finish               |
 -----------------------
```

3.1.3.5 Loop optimisations

3.1.3.5.1 Iteration

Iteration is the process whereby the values in variables from previous iterations of a loop are used to calculate the new values for the current iteration, rather than calculating those values from scratch each time. One of the effects of this optimisation can be the reduction in strength of operations, such as changing multiplications into additions. In this context the IMP77 operators "++" and "--" may be used to great effect. Their action is to adjust the reference on the left by the number of items to the right, hence if X is an integer then X++1 is the next integer and X--2 is the integer two integers before X.

```
 -------------------------------
|  for J = 1,1,1000 cycle       |
|     A(J) = J                  |
|  repeat                       |
 -------------------------------
```

Can be optimised to:

```
 --------------------------------
|  integername T                 |
|  T == A(J)--1                  |
|  for J = 1,1,1000 cycle        |
|     T == T++1                  |
|     T = J                      |
|  repeat                        |
 --------------------------------
```

3.1.3.5.2 Holding

Holding is the process of preloading values used in a loop, into registers or other such temporaries, using those temporaries within the loop and finally storing the values back into the required variables at the end of the loop, if necessary. In the previous example the value in T, the current address of the array element being considered, could be loaded into a register before the start of the loop. In this case, as T is a temporary created by another optimisation, the final value in the register need not be stored once the loop terminates.

The application of most other optimisations will, at worst, have little or no effect on any particular program, however the danger of holding is that it assumes that the values loaded outside the loop will be required within the loop, and this assumption could well be invalid.

For example, consider the following equivalent programs:

```
           A                      B
-------------------- ---------------------
|                    | TEMP = P//Q        |
| while X > 0 cycle  | while X > 0 cycle  |
|    W(X) = P//Q     |    W(X) = TEMP     |
|    X = X-1         |    X = X-1         |
| repeat             | repeat             |
-------------------- ---------------------
```

B will be faster than A if the loop is executed at least twice. If the loop is not executed at all (X <= 0) B will be much slower than A (by an alarming 80 microseconds on the 7/32).

3.1.3.5.3 Removal of invariants

This is the process whereby complex sub-expressions, which do not change their values as the loop progresses, are evaluated outside the loop and held in temporaries:

```
--------------------------------
|  for J = 1, 1, 1000 cycle    |
|     A(J) = LIMIT-MARGIN      |
|  repeat                      |
--------------------------------
```

Can be optimised to:

```
-------------------------------
|  TEMP = LIMIT-MARGIN        |
|  for J = 1,1,1000 cycle     |
|     A(J) = TEMP             |
|  repeat                     |
-------------------------------
```

It is simply a special case of Holding.

3.1.3.6 Expansion

Expansion is the process of rewriting compact representations of parts of a program in a more explicit form, usually resulting in faster execution but at the expense of more code. The two main uses of expansion are to reduce the overheads in loop control by repeating (unrolling) the loop body and hence reducing the number of iterations, and to replace calls on procedures by the body of the procedure, with the necessary substitution for parameters. Extra gains can come from the interaction of the expanded code with the enclosing code as in the following example:

```
for J = 1,1,100 cycle
   A(J) = 0
repeat
```

This can be expanded into:

```
for J = 2, 2, 100 cycle
   A(J-1) = 0
   A(J)   = 0
repeat
```

and can generate the following code (PDP11):

```
    CLR   J
$1:ADD   #2,J
    MOV   J,R1
    ADD   R1,R1
    ADD   LNB,R1
    CLR   A-2(R1)
    CLR   A(R1)
    CMP   J,#100.
    BNE   $1
```

3.1.4 Source optimisations

Source optimisations [Schneck, 1973] are those optimisations which can be effected by changes in the source program. They can be sub-divided into three categories: machine-independent [Hecht, 1973; Kildall, 1973], machine-dependent, and tentative. Tentative optimisations are those which, while unlikely to make the code worse, may improve it on some machines. For example, most machines will handle the comparison "X<1" better if it is rewritten as "X<=0", where X is an integer variable.

3.2 Combination of optimisations

Many of the optimisations described above result in an improvement in the generated code not only by their own effects but also by their interaction with other optimisations, as one improvement often produces the conditions needed for another. As an example consider the compilation of the following, rather unlikely, statements on the Data General NOVA:

```
 ---------------------
|   A = (B&C)<<1      |
|   A = D if A = 0    |
 ---------------------
```

The first statement can generate the obvious code:

```
 ---------------
|   LDA   0,B   |
|   LDA   1,C   |
|   AND   0,1   |
|   MOVZL 1,1   |
|   STA   1,A   |
 ---------------
```

At this stage the value in accumulator 1 (A) can be remembered, and the STA instruction marked as "pending" so that it can be removed later if it is decided that deferring the store will improve the code.

With this knowledge the second statement can be compiled to:

```
 ------------------
|  MOV#   1,1,SZR  |
|  JMP    $1       |
|  LDA    1,D      |
|  STA    1,A      |
| $1:              |
 ------------------
```

Immediately before the label $1 it is known that once again the value of A is in accumulator 1, and that the STA above the label is marked "pending" as before. Following the definition of the label the environment before the jump to that label, can be combined with the environment just before the label, to give the new environment following the label. The information in this environment is that A is in accumulator 1 and that the same store is pending from both old environments. This allows the two marked store instructions to be removed and one store placed after the label (and once again marked as being "pending"). This gives the following code:

```
 --------------------
|     LDA    0,B     |
|     LDA    1,C     |
|     AND    0,1     |
|     MOVZL  1,1     |
|     MOV#   1,1,SZR |
|     JMP    $1      |
|     LDA    1,D     |
|  $1:STA    1,A     |
 --------------------
```

A simple jump optimisation notices that the JMP passes over just one instruction and can therefore be removed by inverting the skip condition on the previous MOVe, giving:

```
 ------------------
¦   LDA   0,B      ¦
¦   LDA   1,C      ¦
¦   AND   0,1      ¦
¦   MOVZL 1,1      ¦
¦   MOV#  1,1,SNR  ¦
¦   LDA   1,D      ¦
¦   STA   1,A      ¦
 ------------------
```

Finally, peephole optimisation combines the AND with the MOVZL giving ANDZL, and then combines this with the following MOV# to give the complete code sequence as:

```
 ------------------
¦   LDA   0,B      ¦
¦   LDA   1,C      ¦
¦   ANDZL 0,1,SNR  ¦
¦   LDA   1,D      ¦
¦   STA   1,A      ¦
 ------------------
```

The most interesting thing to notice about this particular sequence of optimisations is that with the possible exception of the removal of the marked STA instructions, the final code can be generated very simply with local optimisations.

4 The design of the compiler

This section describes the features of the compiler which have had an influence on the form of the intermediate code.

4.1 General structure

One of the aims of this type of compilation strategy is to simplify the production of compilers, and a successful technique for simplifying programs is to divide them into several communicating modules, each largely independent of the others but with well-defined interfaces between them. At the highest level, a compiler can be split up into three major parts:

1 A language processor, which deals with the language-dependent parts such as parsing, semantic checking, and error reporting.

2 A code generator, which takes the decomposed form of the program as generated by 1 above, and constructs the appropriate code sequences to perform the required functions.

3 An object-file generator, which builds an object-file from the code sequences produced by 2, in the form required by the system which is to execute the program.

Commonly, the first two parts of this scheme are combined into one program which generates as its output an assembly-language source file corresponding to the original program.

The third part then becomes the standard system assembler. This approach clearly simplifies the production of the compiler, as one part, the assembler, is provided already and can ease the problems of checking the compiler because the code it generates is presented in a well-known form. Despite these advantages such a scheme was rejected for the following reasons:

1 In order that assembly language can be generated, the compiler must have an internal form of the instructions, which is changed into text, processed by the assembler, and finally converted into the machine representation. These transformations can be eliminated if the compiler works directly with the machine representations.

2 In general, the system-provided assembler will be expecting to receive a much more powerful language than the rather stereotyped text produced by compilers. This will certainly degrade the performance of the assembler. A solution to this is to produce a cut-down version of the assembler which only recognises those constructs generated by the compiler. However, producing a new assembler removes one of the reasons for choosing this route, namely, not requiring extra work in writing the object-file generator.

3 As will be seen later (section 4.7), even after the code sequences have been produced there remain several optimisations which can be performed using knowledge gained during the production of those sequences, for example, generating short forms of jump instructions when the distance between the jump and its destination is small enough. While in certain cases these optimisations can be performed by a standard assembler it is unlikely that the structure of the code-generator would be as simple as if a special-purpose object-file generator were available.

The main interface in such a system is clearly that between the language and machine dependencies, as most languages are largely machine-independent. It is this interface between the language-dependent and machine-dependent parts of the compiler which is termed the INTERMEDIATE CODE. In the following discussion it is assumed that the reader has a reasonable understanding of the structure of the final form of I-code, a definition of which may be found in Appendix.A2.

4.2 The intermediate code

Even while remaining independent of machine architecture, codes can be designed at various levels of abstraction. Roughly, the higher the level of the intermediate-code the closer it is to to the source language, and the lower the level the closer it is to some (possibly hypothetical) processor's instruction set.
The choice as to the level of the intermediate-code eventually comes down to a question of where decisions are to be taken.
If a low-level code is chosen, more decisions will have to be made in the language-dependent phase (making it more complicated) but leaving less choices available to the code-generator (making it simpler, but removing chances for improving the code in the light of particular machine features). If a high-level code is chosen, decisions are left to the code-generator resulting in a simpler language processor but a more complicated code-generator which is better able to adapt to a particular processor.

The design of the intermediate code can also be influenced by its intended role in the complete compiling system. If the code is to be used in the compilation of just one language on many machines, there may be an advantage in increasing the complexity of the code if it results in simpler code generators at the expense of a more complicated, but unique, first phase.

Conversely, if the code is to be generated by several different language processors, a simple intermediate code which is easy to produce may well be more attractive.

As I-code was intended for optimisation, a high-level code was chosen. In addition, as it was hoped that the code could eventually be used in different language processors, it was decided to keep the structure of I-code as simple as possible.

The complete compilation process may be thought of as a sequence of transformations working from the source program to the final object program via a number of intermediate representations. As the transformations are applied, the representations become less dependent on the source language and more dependent on the target machine. In order to simplify the code-generator as much as possible the intermediate code must lie as far from the source language as is possible without straying from the objectives set out below.

4.2.1 Objectives

One of the dangers in designing an intermediate code is that of building into it old techniques and standard expansions of source constructions, which while they may be tried and tested cannot in any way be said to be "the only solutions" or even "the best solutions".

One of the intentions behind the design of I-code was to permit the use of varied implementation strategies. In the same way that the only practical definition of a "good" programming language is that it fits the style of the particular programmer using it, so the measure of the power of an intermediate code must include the ease with which it can adapt to an existing style of code-generator writing. Inevitably, practical constraints prevent total generality: the most general form of a program is a canonical form of itself, but this is little help in compiling it.
It follows that the intermediate code, while remaining true to the original program and distant from "real" machines, must provide enough simplification to make the task of code-generation as easy as possible without inhibiting optimisation.

From the start it was appreciated that an intermediate code suitable for use in optimising compilers would necessarily require more processing than a code such as O-code which was aimed at a quick implementation. The original hope was that although each machine-dependent code generator would not be small, typically about 3000-4000 IMP statements, large portions of one could be taken as a basis for a new implementation. This has proved to be the case, and provision of an existing code-generator as a template greatly simplifies the task of creating a new one (section 6.4.1).

4.2.1.1 Scope

The first and most fundamental objective in the design of I-code was that it should support the compilation of one specific language, IMP-77, on many different machines. Considerations of using the code to implement other languages were secondary to this main aim, but were used to bias the design when a choice had to be made from several otherwise equally suitable possibilities. In retrospect, a few areas of the code could have been made more general without significant overheads in the code generators, mainly in the area of data descriptor definitions, but a detailed discussion of one intermediate code supporting several languages is beyond the scope of this work.

In direct contrast to many intermediate codes, I-code was not designed with the intention of making it convenient to interpret; the prime aim was to permit compilation into efficient machine-code. Nevertheless it is possible to "compile" I-code into threaded code [Bell, 1973] or a form suitable for interpretation, either by generating a conventional interpretive code or by leaving the code in more-or-less its current form but with labels resolved and descriptors expanded into a more convenient representation.

4.2.1.2 Information preservation

As the translation of the source program into intermediate-code is to be machine-independent it will not be possible to know before code generation what details of the program will be of interest to the code-generator. It follows that any loss of information caused by the translation is likely to reduce the scope for optimisation. In addition, not only must the information present in the source be available at the intermediate-code level, but also it must be presented in a form in which it can be recognised easily and used.

For example, the following two program fragments are semantically identical:

```
               A                              B
 ----------------------------------------------------
 |                               |  P = 0          |
 |                               |  cycle          |
 |  TEST for P = 1, 1, 10        |     P = P+1     |
 |                               |     TEST        |
 |                               |  repeat until P = 10 |
 ----------------------------------------------------
```

However, in "B" the information that the fragment contains a simple _for_ construction, while not completely lost, has been scattered through the code, and this dilution of information will increase the complexity of any code-generator wishing to handle _for_ loops specially.

To leave open all avenues for optimisation it is necessary therefore, that all of the semantic information in the source program is preserved in a compact form in the I-code. One sure way of achieving this property is to design the code in such a way as to allow the regeneration of the source program, or at least a canonical form of it which is not significantly different from the original. In this context insignificant differences are the removal of comments and the standardisation of variant forms of statements, such as:

NEWLINE _if_ COUNT = 0

and: _if_ COUNT = 0 _then_ NEWLINE

4.2.1.3 Target machine independence

Most existing intermediate codes are built around a model of a machine which will perform the required computation, and it is this machine which must be mapped onto the actual target computer. In order to simplify this mapping, certain assumptions are made, resulting in the machine being defined in terms of fixed-sized data objects, a fixed way of addressing them, and a fixed set of operations on them, usually involving some kind of stack. When compiling for machines which are similar to this intermediate code machine there is little problem in obtaining a reasonable match, but when there are major differences it becomes impossible to convert the code into an efficient machine representation.

For these reasons it was decided to make I-code independent of actual machine representations: objects would be described once in high-level terms and then all uses would refer to that definition. This immediately removes any assumptions about the sizes of data objects and the ways in which they are addressed, other than those assumptions built in to the source language. One of the main difficulties with existing codes has been their insistence on the store containing a linear array of equally-sized objects, the difference between one object and the next being one address unit. When mapping such a structure onto real machines with (say) byte addressed stores, problems arise with arithmetic involving addresses as the codes frequently pun on addresses and integer values.

Several later versions of such codes have attempted to solve these problems by parameterising the intermediate-code generator so that the characteristics of the target machine may be used to modify the code which is produced. However, they still have built in to them assumptions about how the objects can be addressed.

There are so many constraints which can be imposed on the code to be generated, such as operating system requirements and conventions for communicating with the run-time environment, that a parameterised first phase could not be expected to generate code which was well-suited to every installation. The authors of JANUS [Coleman, 1974] write that they believe that the approach of using a parameterised intermediate code "... is a dead end, and that the adaptability must come in the translation from the intermediate language to machine code".

4.2.1.4 Simplification

For the complexity of the machine-dependent phases of compilation to be kept as low as possible, the machine-independent phase must do as much work as possible while keeping within the constraints imposed by the previous objectives. One way of simplifying the intermediate code is for certain high-level constructions to be expanded into lower-level constructions, but only when there is just one expansion possible under the rules of the language, and that expansion does not scatter information which may be of later use.

The most obvious case of such expansion is in dealing with complex conditional clauses such as:

```
-----------------------------------------------------
|  if (A=B and C#D) or (E<F and G>H) then X else Y  |
-----------------------------------------------------
```

IMP-77 specifies that the condition will only be evaluated as far as is necessary to determine the inevitable truth or falsity of the condition, and so, bearing in mind the modifications to be discussed in section 4.6.1, the statement can be represented more simply as:

```
----------------------------------
|         if A # B  then ->L1    |
|         if C # D  then ->L2    |
|  L1:    if E >= F then ->L3    |
|         if G <= H then ->L3    |
|  L2:    X                      |
|         ->L4                   |
|  L3:    Y                      |
|  L4:                           |
----------------------------------
```

This expansion is tricky and notoriously error prone, and therefore is best done once and for all in the common phase. Similarly it is possible to expand all simple control structures into their equivalent labels and jumps, providing that the structural information is not lost thereby.

4.2.1.5 Decision binding

In any program there will be various options open to a code generator and at some stage in the compilation decisions must be made as to the particular code sequences to be generated. Inevitably these decisions will influence the code which is produced subsequently. On the PDP11, for example, there are two obvious ways of assigning the value in X to the variable Y: either MOVe the value in directly, or move the value into a register first and then assign the register. If the latter way is chosen the value of X will be available in the register for subsequent use, although the former way is better if the value is not required in the near future. In order to make use of information which may well be presented later, it is necessary to be able to defer taking irrevocable decisions until the last possible moment. The structure of I-code permits this delaying in the binding of decisions as it only specifies what needs to be done in abstract terms (using descriptors of arbitrary structure and complexity), and does not give instructions as to how particular results are to be achieved.

4.2.1.6 Ease of use

Of prime importance in the design of the code is the ease with which it may be used to generate good object code. Obviously a high-level code will by its nature be more difficult to handle than a low-level code, but this need not be serious if the code is consistent and results in a convenient expression of the original source. In particular the code should be designed to permit extensive checking to be performed during the compilation process to catch errors in both the intermediate code and the machine-code generator before those errors are passed on to the users. Low-level codes are at a serious disadvantage in this respect as they have lost much of the redundancy present in the source.

4.3 Code layout and addressing

4.3.1 Nested procedure definitions

A common feature of programming languages is the ability to nest the definition of a procedure within another procedure. In addition, several languages imply the definition of procedures within single statements, as in the case of name parameters in ALGOL-60, where the parameter which is actually passed can be a reference to a "thunk", a procedure to evaluate the parameter.
With such nesting, provision must be made for preventing the flow of execution from "falling through" into the procedure from the preceding statements, and this is usually accomplished by planting at the start of the procedure a jump to the statement following the end. While this is simple to implement it does introduce extra instructions which are not strictly necessary. With user-defined procedures the overhead can be minimised when a number of procedures is defined, as one jump instruction can be used to skip them all. Unfortunately thunks will be generated throughout the code in a more-or-less random way, giving little opportunity to coalesce the jumps.

Even if the extra execution time caused by these jumps is insignificant (the jumps round thunks defined in loops get executed repeatedly), the code which they are skipping stretches the code in which they are nested.

On machines with fixed-size jump instructions which can cover the whole machine, such as the DEC PDP10, the stretching causes no problems, but if the addressing is limited, or if several different sizes of jump instruction are provided, the presence of the nested procedure can result in more code being produced later in the generation of large jumps.

4.3.2 Paged machines

On paged machines the overall performance of a program does not depend solely on the efficiency of the code produced by the compiler but includes a factor depending on the locality of references made by the executing program. Traditionally this locality has been improved by monitoring the execution of the program and then re-ordering parts of it in the light of the measurements. Unfortunately not all operating systems provide the user with convenient tools to enable the measurement to be done, leaving only ad hoc methods or intuition for guidance. Without careful control it is all too easy to move one procedure to improve references to it and thereby cause another piece of code to cross page boundaries and counteract any gains in paging performance. Even if the user can obtain the necessary information, a slight change in the program can invalidate the changes.

Notwithstanding these problems, it is evident that by careful structuring of a program significant gains in paging behaviour can be obtained and so this option should not be pre-empted by the intermediate-code (as does Z-CODE which automatically reorders the definitions of procedures).

The possibility of automatic improvement of paging behaviour was investigated by Pavelin who showed that the paging characteristics of a program can be improved by an automatic reordering of the code [Pavelin, 1970].
Pavelin's thesis describes the breaking-up of a program into "chunks", defined by branches and the destinations of branches. At each chunk boundary, extra instructions are planted to cause the updating of a "similarity array" which records the dynamic characteristics of the program. After several runs the similarity arrays are merged and the result is used to specify a reordering of the chunks which should improve the paging performance. In test cases the working-set size of the code was reduced by as much as 40%. The thesis also went on to say that the various compilation problems associated with this "... can be alleviated by operating on an intermediate code which is machine independent with symbolic code addresses".

4.3.3 Events

IMP provides a mechanism for signalling the occurrence of synchronous "events" during the execution of a program. These events are either generated automatically as the result of a program error, or are signalled explicitly by the program. The signalling of the event causes control to be passed back through the dynamic chain of currently active blocks until one is found which has specified a trap for the particular event which has occurred. Execution then continues from a point in that block determined by the trap. In order for this to be implemented it is necessary that the signal routine be able to "unwind" the stack and recover the environment of the block containing the trap.

If the entry and exit sequences of all blocks are identical, as, for example, in the standard procedure entry mechanism specified for the DEC VAX 11/780, the unwinding is fairly trivial. More commonly, however, the recovery is dependent on factors such as the textual level of the procedure and whether it has been optimised or not. In such cases the unwinding can be very expensive or even impossible unless extra information is provided.

For example, on the INTERDATA 7/16 a procedure at the outermost textual level uses register 15 to access its local stack frame, giving the exit sequence:

```
 ------------------
|  LM    7, 4(15)  |
|  BFCR  0, 8      |
 ------------------
```

but a procedure nested within this would use register 14 thus:

```
 ------------------
|  LM    7, 4(14)  |
|  BFCR  0, 8      |
 ------------------
```

It follows that the signal routine must be told which base regiser to use at each stage of the recovery. This can be done either by planting code in the entry and exit sequences of each procedure, or by keeping a static table associating procedure start and finish addresses with the appropriate base register.
The first method is poor as it imposes a run-time overhead on all procedures, whether they trap events or not. The second method is better but can be complicated if procedures are nested as the start-finish addresses alone no longer uniquely define the procedure. One solution is to cause all procedures which use the same exit sequence to be loaded into distinct areas, and to associate the recovery information for the signal routine with each area. This reduces the static overhead to a few words per area, rather than a few words per procedure.

4.4 Data addressing

One of the most important problems which faces the compiler is the addressing of the various data objects used by the program.

As an example of the difficulties which can arise, consider the IMP declarations:

```
 -------------------------------------
 !   integer X                        !
 !   integer array V(0:999)           !
 !   integer Y                        !
 -------------------------------------
```

On a machine such as the INTERDATA 7/16 which uses base+displacement addressing with a 16-bit displacement, the whole of the available storage, (64K bytes), can be addressed with a single instruction. In this case the most efficient implementation of the array is as a row of one thousand integers (halfwords) addressed directly via a local name base (LNB):

```
 ----------------------------------------------------------
 ! LNB {Local Name Base}                                   !
 !  !                                                      !
 !  !     a    a+2     a+4               a+2000   a+2002   !
 !  v    .---.------.------.            .--------.---.     !
 !  .- - ! X ! V(0) ! V(1) ! - - - - -  ! V(999) ! Y ! - - !
 !       .---.------.------.            .--------.---.     !
 ----------------------------------------------------------
```

This implementation has several points in its favour:

i As the size of the array is known at compile-time, no special code is required to create it at run-time; the necessary storage can be claimed on entry to the block along with that for simple variables, return addresses etc.

ii Array references with constant subscripts need no address calculations at run-time. For example using V as declared above, the element V(2) is immediately addressable as the halfword with displacement "a+2 + 2*2" from LNB.

iii In certain more general cases when the subscript is a variable, access can be simplified by remembering previous calculations. For example, the address of the array element V(X) is

```
 --------------------------------------
| addr(V(0)) + X*size of each element |
 --------------------------------------
```

In the example above this becomes:

```
 --------------------------------------
| LNB+a+2    + X*size of each element |
 --------------------------------------
```

which can be rearranged to:

```
 ---------------------------------------------
| a+2        + (X*size of each element+LNB) |
 ---------------------------------------------
```

Hence the following code could be produced (7/16):

```
 --------------------
|  V(X) = 0          |
|                    |
|  LH   1,X(LNB)     | pick up X
|  AHR  1,1          | *2 (2 bytes per integer)
|  AHR  1,LNB        | add in LNB
|  SHR  0,0          | get zero
|  STH  0,a+2(1)     | store in V(X)
 --------------------
```

Noting that the value now in register 1 (X*size+LNB) only depends on the size of each element, X, and the local name base, it is clear that register 1 can be used to address the X'th element of any <u>integer</u> <u>array</u> of one dimension and constant bounds declared at the current level.

Hence if the array W(1:12) were declared immediately after Y in the example above, while register 1 is not changed W(X) can be addressed as a+2002(1).

On the other hand, a machine with limited store cover, such as the Data General NOVA which only has an eight-bit displacement, will almost certainly force the array to be implemented as an immediately addressable pointer which is initialised at run-time to the address of storage claimed explicitly.

```
 ----------------------------------------------------------
| LNB                                                      |
|  |                                                       |
|  |                                                       |
|  v    .---.---.---.                                      |
|  .- -| X | V | Y | - - -                                 |
|       .---.---.---.                                      |
|             |                                            |
|             |                                            |
|             |  .------.------.       .--------.          |
|             +->| V(0) | V(1) | - - - -| V(999) | - - -   |
|                .------.------.       .--------.          |
 ----------------------------------------------------------
```

With this organisation the address of V(X) will be:

```
 ----------------------------
¦ V + X*size of each element ¦
 ----------------------------
```

and there is little that can be done by rearranging the expression to improve on the "obvious" code (7/16):

```
 ---------------
¦  V(X) = 0     ¦
¦               ¦
¦  LH   1,X     ¦    pick up X
¦  AHR  1,1     ¦    double it
¦  AH   1,V     ¦    add in addr(v(0))
¦  SHR  0,0     ¦
¦  STH  0,0(1)  ¦
 ---------------
```

Not only is this second code sequence longer than the first by two bytes, but it will execute more slowly as the second addition involves a store reference whereas the equivalent instruction in the first sequence uses a register.
In both cases, however, some simplification can be done if the subscript is an expression of the form:

```
 ------------------------------
¦   X plus or minus CONSTANT   ¦
 ------------------------------
```

in which case the constant can be removed from the subscript expression evaluation and added into the final displacement.
For example (7/16):

```
 -------------------------
¦  V(X-7) = 0             ¦
¦                         ¦
¦  LH   1,X               ¦     pick up X
¦  AHR  1,1               ¦     double it
¦  AHR  1,LNB             ¦     add in LNB
¦  SHR  0,0               ¦     get zero
¦  STH  0,a+2+(-7)*2(1)   ¦
 -------------------------
```

Unfortunately even this optimisation may not be available. For example, the ICL 2900 series performs array accesses through a DESCRIPTOR REGISTER, and the extra displacement cannot be added into the instruction. Also some machines, such as the IBM 360, only permit positive displacements in instructions.

The examples above pose the following problem: If the intermediate-code is to know nothing of the target machine it cannot know the best way to declare the array, nor the best way to access it. Therefore the code must always produce the same sequences for array declarations and array accesses. It follows that these sequences must remain quite close to the original source and not include any explicit address calculations.

As another example, the DEC PDP11 range has a hardware stack which grows with decreasing store addresses. Because of this it could be convenient to allocate storage for variables in that order, from large addresses to small addresses. However, in several cases it may be necessary to force objects to be created in order of increasing addresses, such as when program structures are to be mapped onto hardware-defined structures in memory, resulting in an implementation which requires to be able to create similar objects in different ways depending on the context.

Finally, some machines provide instructions in which the displacement of the operand is scaled before use, depending on the size of that operand. The GEC 4080 is such a machine, with instructions such as:

```
LDB  1     load byte <1>
LD   1     load halfword, bytes <2> & <3>
LDW  1     load fullword, bytes <4>,<5>,<6> & <7>
```

When producing code for such machines it is convenient to allocate all the local objects of the same size in particular areas, and then arrange the areas in increasing order of the size of the objects they contain. This permits better use of the available displacement field in the instructions.

The solution to these problems which was chosen in I-code was to define a DESCRIPTOR for each object to be manipulated. On input to the code-generator descriptors are converted from their machine-independent form to a new form appropriate to the target machine. As all subsequent reference to the object will be through descriptors the code produced will automatically reflect the decisions made at the time the descriptors were created.

As will be discussed in section 4.5, it may be possible to remove the overhead in setting up addressability for local variables and parameters if the parameters can be held in registers and the local variables are never referenced. After examining many procedures which do use local variables it is clear that a large number of them do not need the complete overhead in setting up a local frame base as they could use the workspace pointer (stack pointer) instead. The criterion is that the position of the locals relative to the workspace pointer must be known at compile time. This reduces to the procedure not having any objects with computed sizes (arrays with computed bounds, for example) and no calls on procedures which access those locals as their global variables.

Consider the compilation of the following procedure on the PDP11:

```
--------------------------------------------
| routine MARK(record(cellfm)name CHAIN) |
|    integer N                           |
|    N = 0                               |
|    while not CHAIN == NULL cycle       |
|       N = N+1                          |
|       CHAIN_INDEX = N                  |
|       CHAIN == CHAIN_LINK              |
|    repeat                              |
| end                                    |
--------------------------------------------
```

The code normally produced for this routine would be:

```
---------------------------
|      MOV   LNB,-(SP)      | remember old LNB
|      MOV   DS,-(SP)       | remember DS
|      MOV   R0,(DS)+       | save the parameter
|      MOV   DS,LNB         | set up local addressing
|      ADD   #20,DS         | reserve local space
|      CLR   10(LNB)        | N = 0
| $1:  MOV   -2(LNB),R1     | test CHAIN
|      BEQ   $2             | branch if NULL
|      INC   10(LNB)        | N = N+1
|      MOV   10(LNB),2(R1)  | CHAIN_INDEX = N
|      MOV   (R1),-2(LNB)   | CHAIN == CHAIN_LINK
|      BR    $1             | repeat
| $2:  MOV   (SP)+,DS       | restore DS
|      MOV   (SP)+,LNB      | restore LNB
|      RTS   PC             | return
---------------------------
```

However, by using workspace pointer (DS) relative addressing this reduces to:

```
 --------------------------
 !      MOV   R0,(DS)+       !
 !      TST   (DS)+          ! reserve local space
 !      CLR   -2(DS)         ! N = 0
 ! $1:  MOV   -4(DS),R1      ! test CHAIN
 !      BEQ   $2             !
 !      INC   -2(DS)         ! N = N+1
 !      MOV   -2(DS),2(R1)   ! CHAIN_INDEX = N
 !      MOV   (R1),-4(DS)    ! CHAIN == CHAIN_LINK
 !      BR    $1             !
 ! $2:  SUB   #4,DS          ! restore DS
 !      RTS   PC             ! return
 --------------------------
```

This optimisation can be performed quite simply by the third phase of compilation.

In the interface between the second and third phases, the code sequences generated by the second phase are made up of items of the form:

<type> <VALUE>

where <type> describes where <VALUE> is to be put, for example in the code area or in the private data area. To achieve the workspace-pointer-relative addressing, extra types are introduced which specify that the associated value is the displacement of a local variable from LNB. Other codes are needed to be able to modify the operation part of the instruction which uses the displacements but these will be ignored here as they cause no difficulty and would just obscure the discussion. In addition, an extra <modify DS> item is output whenever DS is explicitly altered (as when parameters are stacked using MOV ??,(DS)+.

By default the third phase will treat these extra types as being exactly equivalent to <code area> types, and will generate the first sequence of code. However, if when the end of the procedure is processed, the second phase discovers that no dynamic objects or dangerous procedure calls were generated, it marks the end of the procedure accordingly (in the same way as described in section 4.7.2). This mark instructs the third phase to relocate all VALUEs with the appropriate type so as to make them relative to DS. The <modify DS> types are used to keep the third phase's idea of the current position of DS in step with reality.

4.5 Procedure entry and exit

IMP is heavily based on the use of procedures, indeed the only method of communicating with the controlling environment is by means of procedure calls. Also the techniques of structured programming result in the extensive use of procedures. Clearly when writing a compiler for such languages much thought must be given to making procedure entry and exit (and the associated passing of parameters) as efficient as possible.

4.5.1 User-defined procedures

The usual technique for procedure entry and exit is to have standard preludes and postludes which cover all the different types of procedure. For example the EMAS IMP code sequences [Stephens, 1974] are (ICL4/75):

```
 -------------------------
|      STM  4,14,16(WSP)  | save the current environment
|      BAL  15, PROC      | enter the procedure
|       .                 |
| PROC ST   15,60(WSP)    | save the return address
|      LR   LNB,WSP       | set up local stack frame
|      LA   WSP,***(WSP)  | claim local space
|      BALR 10,0          | set up code addressability
|       .                 |
|       .                 |
|      LM   4,15,16(LNB)  | restore calling environment
|      BCR  15,15         | return
 -------------------------
```

While this has proved to be convenient to generate and efficient to execute it has one major problem, part of the housekeeping of the procedure entry is performed at the call itself. This seems undesirable for two reasons:

i Procedures are generally called more often than they are defined. If part of the housekeeping of procedure entry is done at the call that code will be duplicated at each call, thus increasing the size of the program. Putting that code within the procedure reduces the size overhead.

ii If the knowledge of what housekeeping needs to be done for procedure entry is needed outside the procedure it becomes impossible to alter the entry and exit sequences to suit the actual procedure. In particular, on certain machines it is possible to remove the entry and exit sequences altogether when the procedures are simple enough.

If the 4/75 compiler moved the environment-saving STM instruction into the body of the procedure, the storing of the return address would be performed automatically:

```
-------------------------
¦      BAL   15,PROC     ¦
¦-                 -     ¦
¦ PROC STM   4,15,16(WSP) ¦
¦      LR    8,WSP       ¦
¦      ..    ..          ¦
-------------------------
```

This not only saves four bytes per call, very important on a machine with a very severely limited immediate addressing range, but also reduces the overhead in entering the procedure by one instruction.

A further modification would be to pass one or more of the parameters in the registers, leaving the way open for remembering that fact inside the procedure.

Hence a call could be reduced from:

```
 ------------------------
 |     L   1,X           |
 |     ST  1,64(WSP)     |
 |     L   2,Y           |
 |     ST  2,68(WSP)     |
 |     BAL 15,PROC       |    PROC(X, Y)
 |-                     -|
 | PROC STM 4,15,16(WSP) |
 |     ..  ..            |
 ------------------------
```

to:

```
 ----------------------
 |     L   0,X          |
 |     L   1,Y          |
 |     BAL 15,PROC      |
 |-                    -|
 | PROC STM 4,1,16(WSP) |
 |     ..  ..           |
 ----------------------
```

The ability to determine exactly how parameters are to be passed can be of crucial importance in the efficiency of the procedure mechanism.

When compiling for the PDP11 the obvious calling sequence for a procedure with two integer value parameters would be:

```
 -------------------
|   MOV   X,-(SP)   |
|   MOV   Y,-(SP)   |
|   JSR   PC,PROC   |
 -------------------
```

Unfortunately this produces problems inside the procedure as the return address, stacked by JSR, is too far down the stack to permit the use of the RTS instruction to return, for this would leave on the stack the space used by the parameters. Neither can the stack be adjusted before the return, which would then be made indirectly through a location beyond the stack pointer, as space there must be considered volatile, being used by interrupt handling. Extra instructions are needed either at the call or inside the procedure to adjust the stack; the JSR instruction may well not be "a beauty" as claimed by some implementors [Bron, 1976]. A MARK instruction has been introduced in an attempt to overcome this problem, but it is far from helpful as it imposes an arbitrary register convention and puts all of the overhead on the call rather than on the procedure itself.

On the other hand, if all of the parameters can be passed in registers, the JSR will put the return address on a clear stack, permitting the use of RTS for the return. As in practice most procedures have few parameters, usually only one or two, this can give a large saving.

As an example of the power of being able to alter entry and exit sequences, consider a recursive implementation of the IMP routine SPACES:

```
 -----------------------------
| routine SPACES(integer N)   |
|    return if N <= 0         |
|    SPACES(N-1)              |
|    SPACE                    |
| end                         |
 -----------------------------
```

On the PDP10 the straightforward coding for this would be:

```
 -------------------------
|         MOVE  0, X       | pick up X
|         MOVEM 0, 3(SP)   | assign the parameter
|         PUSHJ SP, SPACES | call SPACES
|-                        -|
| SPACES: MOVEM LNB,1(SP)  | save old frame base
|         MOVE  LNB,SP     | pick up new frame base
|         ADDI  SP,3       | reserve stack space
|         SKIPLE 1,2(LNB)  | load, test & skip if X<=0
|         JRST   LAB1      | jump to LAB1
|         MOVE   SP,LNB    | restore stack pointer
|         MOVE   LNB,1(SP) | restore old frame base
|         POPJ   SP        | return
| LAB1:   SOJ    1, 0      | X-1 -> ACC1
|         MOVEM  1,3(SP)   | assign parameter
|         PUSHJ  SP,SPACES | call SPACES
|         PUSHJ  SP,SPACE  | call SPACE
|         MOVE   SP,LNB    | restore stack pointer
|         MOVE   LNB,1(SP) | restore old frame base
|         POPJ   SP        | return
 -------------------------
```

By applying the optimisations of passing the parameter in an accumulator (called ARG) and remembering that the parameter is in this accumulator on entry to the procedure, the code reduces to:

```
 ---------------------------
|         MOVE  ARG,X        | pick up X
|         PUSHJ SP, SPACES   | call SPACES
|-                          -|
| SPACES: MOVEM LNB, 1(SP)   |
|         MOVEM ARG, 2(SP)   | assign the parameter
|         ADDI  SP, 3        |
|         JUMPG ARG, LAB1    | ->LAB1 if ARG > 0
|         MOVE  SP, LNB      |
|         MOVE  LNB, 1(SP)   |
|         POPJ  SP           |
| LAB1:   SOJ   ARG, 0       | parameter = ARG-1
|         PUSHJ SP,SPACES    |
|         PUSHJ SP,SPACE     |
|         MOVE  SP, LNB      |
|         MOVE  LNB, 1(SP)   |
|         POPJ  SP           |
 ---------------------------
```

On inspection it is clear that the local stack frame (pointed at by LNB) is never used within the procedure except by the entry and exit sequences. Hence by reducing those sequences to the absolute minimum, the code becomes:

```
 ---------------------------
|         MOVE  ARG, X       |
|         PUSHJ SP, SPACES   |
|-                          -|
| SPACES: JUMPG ARG, LAB1    |
|         POPJ  SP           |
| LAB1:   SOJ   ARG, 0       |
|         PUSHJ SP, SPACES   |
|         PUSHJ SP, SPACE    |
|         POPJ  SP           |
 ---------------------------
```

Finally, an opportunistic optimisation may be performed [Knuth, 1974; Spier, 1976] by noticing that the final two instructions may be combined so that the procedure SPACE uses the return address pushed onto the stack for the return from SPACES. This results in the tightest form of the code:

```
 ----------------------------
|         MOVE   ARG, X       |
|         PUSHJ  SP, SPACES   |
|-                           -|
| SPACES: JUMPG  ARG, LAB1    |
|         POPJ   SP           |
| LAB1:   SOJ    ARG, 0       |
|         PUSHJ  SP, SPACES   |
|         JRST   SPACE        |
 ----------------------------
```

The final steps in this optimisation can only be performed once the body of the procedure has been compiled. In order that the correct (in this case non-existent) entry sequence can be used, an extra pass over the object code is necessary. This pass can be combined with the process of adjusting labels and jumps which is carried out in the third phase of compilation described in section 4.7. The code generator can mark the position where an extra sequence is required and at the end of the procedure can inform the third phase of any salient features found in the body. The third phase can then decide on the best entry and exit sequences to use.

This ability to tailor the "housekeeping" parts of procedures can be used in many circumstances to limit the inclusion of code which is needed to handle rare constructions to those procedures which use the feature.
As an example of this consider the ICL 2900 series.
The machines of the series are designed around a hardware stack, which resides in one, and only one, segment of the user's virtual memory, and thus limits this data space to 255K bytes. In order to be able to handle programs using very large arrays, space must be available off-stack in another segment or set of consecutive segments. The maintenance of this extra data space will require instructions to be executed on entry to and on exit from procedures which claim space from it, but not from those which only use space from the stack.

These extra instructions can be added to the procedure in a simple manner by the third phase as it now controls the form of the procedure when all the necessary information is available.

For these optimisations to be performed the intermediate code must not lay down rules for procedure entry and exit, rather it should simply mark the points at which suitable code is required.

An additional consideration in the design of the I-code for procedure entry and exit is the requirement of some machines for a "pre-call" to be made the prepare a hardware stack for parameters prior to their evaluation and assignment.

For example (ICL2900):

```
 -------------------
|  PROC(1, 2, 3)    |
|-                 -|
|  PRCL   4         |   pre-call
|  LSS    1         |   load 1
|  SLSS   2         |   stack it and load 2
|  SLSS   3         |   stack it and load 3
|  ST     TOS       |   store it on Top Of Stack
|  RALN   8         |   raise the Local Name Base
|                   |   to point to the new frame
|  CALL   PROC      |   enter the procedure
 -------------------
```

Following these considerations the form of procedure call chosen for I-code was:

```
 -----------------
|  PROC  P        |   stack procedure descriptor
|                 |   \
|  {stack param}  |    : repeated for each parameter
|  ASSPAR         |   /
|  ENTER          |   enter the procedure
 -----------------
```

ASSPAR causes the value described on the top of the stack to be assigned to the next parameter, identified by the procedure descriptor second on the stack, using either ASSVAL or ASSREF as appropriate.
In order to pass some of the parameters in registers all that need be done is for the initial processing of the descriptors for those parameters to define them as the appropriate registers. PROC can then "claim" those registers, the parameter assignment will load them, and finally ENTER can release them for subsequent re-use on return from the procedure.

4.5.2 External procedures

Most useful languages provide means for compiling files of procedures (and less commonly, data objects) which can be accessed from other modules. Also, systems usually provide extensive libraries of procedures which users of high-level languages will want to access. In general an external procedure is identified by a vector of quantities including at least the entry address and a description of the environment in which the procedure is to execute. Depending on the type of operating system in question, the number of quantities in this vector will change. When the system requires a "store image" which has all the addresses fixed before execution, only the entry address is required, as the code of the procedure can be relocated in order to define its environment. As this method demotes code-sharing to a limited facility (making programs shareable is often a privileged operation), several systems have selected a more flexible scheme whereby executing programs have a writeable "linkage area" into which are placed the entry vectors for procedures. The code of these procedures may now be made read-only and shared with only the linkage areas being unique to each user. These vectors are filled in with the references to the externals either prior to program execution, or dynamically when the procedure is first called. Finally, it must be noted that the compiler writer will have little or no control over the standards required by external procedures unless they have been generated with the same compiler.

In particular the parameter passing mechanisms may be different from those used in the intermediate code.
In order to cope with these and other considerations any intermediate code which permits access to external procedures must be sufficiently flexible to allow the variations to be handled efficiently.

4.5.3 Permanent procedures

Most languages define a set of procedures which will be available on any implementation without explicit action by the user (such as the IMP procedures ITOS, REM, READSYMBOL, and READ). Such procedures are termed "permanent procedures". It is common for intermediate codes to provide specific code items to invoke permanent procedures, but this has the problem that the code-generator must know about all such procedures, and the language-dependent phase must be changed and the intermediate-code extended if an implementation wishes to make efficient use of procedures which can be compiled in-line on particular machines. For example many machines provide an instruction for moving blocks of store around and it could be advantageous to have a procedure which invoked this instruction directly.

Before investigating ways of improving the implementation of permanent procedures it is useful to examine in some detail the properties of the procedures mentioned above, which were chosen because they typify the main problems in this area.

ITOS is a fairly complicated string function which returns as its result the decimal character-string representation of the integer value passed to it as a parameter. Because of its complexity this procedure is almost always best implemented as an external procedure which is linked into the program along with any other external entities required.

REM is an integer function which returns the remainder of dividing the first integer parameter by the second, and on many machines can be efficiently compiled in-line, as most integer divide instructions provide both the quotient and the remainder. However, when compiling for machines such as the DATA GENERAL NOVA or the DEC PDP11 when they do not have the optional divide instructions, division has to be performed by a complicated subroutine, suggesting that REM itself should be an external procedure like ITOS.

READSYMBOL falls somewhere between the two, mainly because it is defined to have a general name parameter, that is, the parameter may be a reference to any type of entity: integer, real, byteinteger, etc. To implement READSYMBOL as an external procedure it would have to be passed the general name parameter (comprising both the address of the actual parameter and information about its type and precision), and would have to interpret that parameter in order to be able to store the character, suitably converted, in the appropriate way.

A much more efficient implementation is to convert the statement:

```
 ---------------
 ¦ READSYMBOL(S) ¦
 ---------------
```

into the equivalent form:

```
 ------------------
 ¦ S = F$READSYMBOL ¦
 ------------------
```

where F$READSYMBOL is a function which returns as its result the character value that READSYMBOL would have placed into its parameter. Once this is done, conversions and the choice of store operation can be left to the usual assignment part of the compiler. A further complication can arise if, as in the case of the INTERDATA 7/16 operating system, ISYS [Dewar, 1975], several permanent procedures map directly onto system-provided facilities: the function F$READSYMBOL can be replaced by the supervisor call "SVC 8,0", SELECT INPUT by "SVC 6" etc.

The difficulty caused by READ is mainly one of space. As read can input an integer value, a real value, or a string value depending on the type of its (general name type) argument, it is going to be fairly large, especially if the hardware on which it runs does not provide floating-point instructions, forcing those functions to be performed by subroutine. It follows that on small systems it may be convenient to replace calls on READ by calls on smaller procedures, chosen at compile-time by examining the type of the parameter given to READ, which input solely integer, real, or string values.

Finally it should be noted that the substitutions and modifications discussed above may only be generated as replacements for direct calls on the procedure; if the procedure is passed as a parameter to another procedure no alterations are possible and a "pure" version must be available. As passing a procedure as a parameter is totally distinct from calling the procedure this case does not prevent the improvements being carried out where possible.

It should now be clear that the efficient implementation of permanent procedures will differ greatly from the implementation of user-defined procedures, and the implementation of permanent procedures on different machines. Hence the intermediate-code must make no assumptions about either which permanent procedures are available or how they are to be implemented.
As a side-effect of removing any built-in properties from permanent procedures it becomes possible for a simple code-generator to ignore any possibility of producing special code and compile them all as externals.

These transformations of procedures can only be applied when the procedures are invoked (called) directly. In the case of procedures passed as parameters all calls will of necessity be the same and hence either it will not be possible to pass some permanent procedures as parameters, an unfortunate limitation imposed by several languages, or there must be a "pure" form of the procedures available.

This latter can be done very simply using I-code. The primitive procedure descriptors are defined exactly as if the procedures were truly external, but with an extra marker showing them to be "permanent". The only time that this marker is used is in the procedure-call generating section of the compiler. If the procedure is being passed as a parameter this section of the compiler is not entered and so the procedure will be passed as an external. All that is now necessary is for there to be an external manifestation available when the program executes. This method has the added advantage that there is no compile-time overhead, especially important considering that passing procedures as parameters is one of the least-used features of IMP77.

4.5.4 Primitive Procedures

It is rare for machines to provide simple instructions which can deal directly with all of the requirements of high-level languages and so several constructions will have to be handled by subroutines. The code generator may then refer to these "primitive procedures" as though they were machine instructions.

The cases in which such procedures are required commonly include exponentiation, string manipulation, and array declaration and access.

Given these procedures, the code-generator has a choice between calling them as closed subroutines or expanding them in-line. The former produces dense code but will execute more slowly than the latter (and possibly suffer from not knowing what is corrupted by the routine and therefore having to forget everything it knows). On the other hand while the expansion of primitive procedures in-line will improve the execution speed of the program, it becomes necessary for the code-generator to be able to create the appropriate code sequences and thereby become more bulky.

Once again the choice must be left to the code-generator as the benefits of a particular decision will depend on both the target machine and the use to which the compiler is to be put. If the compiler is to be used for large mathematical problems it is likely that the gains made by putting exponentiation in-line will outweigh the disadvantage of the extra code size, whereas in operating-system work, as exponentiation is probably never needed, the extra complexity of the code generator to expand the routine would not be desirable.

Given that some of the primitive procedures will be referenced often (checked array access, for example) it is important that entry to them is made as efficient as possible and in this area the ability to reorder code can be used to great effect.

In the original Interdata 7/32 IMP77 compiler the primitive routines were gathered together at the end of the user's code, as it was only then that it was known which procedures were required.

```
 ---------
!         ! <- CODE BASE (register 14)
!  USER   !
!  CODE   !
!         !
 ---------
!         !
!  PRIM   !
! PROCS   !
!         !
 ---------
```

With this scheme programs of 16Kbytes or less can reference the primitive procedures with 32-bit instructions (program-counter relative addressing). Unfortunately once the program grew beyond this limit the larger and slower 48-bit form of the instructions had to be used in order to achieve addressability. In the IMP77 code generator there were 352 such large instructions.

In the new compiler the object code is reordered to place the primitive procedures at the head of the user's code where they can be addressed relative to CODE BASE.

```
 ---------
!         ! <- CODE BASE (register 14)
!  PRIM   !
! PROCS   !
!         !
 ---------
!         !
!  USER   !
!  CODE   !
!         !
 ---------
```

The immediate disadvantage of this is that it will push the user's procedures further away from CODE BASE and hence increase the chances of a user procedure reference requiring a long (48-bit) instruction. However in practice this is not a problem as the total size of the primitive procedures is usually quite small, typically less than 800 bytes on the 7/32. The IMP77 code generator mentioned above now needs no long references at all, saving 724 bytes of code, out of about 40Kbytes. The compression of the code so achieved can be enhanced slightly by bringing the destinations of more jumps into the short-jump range, giving an extra saving of 20 bytes the case above. In addition, now that a register (CODE BASE) is pointing to the first primitive procedure, the list of procedures required can be reordered to place the most frequently referenced one first and thereby reduce references to it to 16-bit instructions (BALR_LINK,CODEBASE).

When compiling with checks on, by far the most commonly referenced primitive procedure is the routine which checks for the use of an unassigned variable (over 2000 references to it in the code generator), and this trivial optimisation results in a saving of more than 4000 bytes.

4.6 Language-specified and compiler-generated objects

During compilation, various objects will be manipulated in order to generate code. Some of these objects have a direct representation in the source program and are referred to as "language-specified" objects, whereas others are created by the compilation process itself and are referred to as "compiler-generated" objects. The fact that the compiler-generated objects will be (or can be constrained to be) used in a stereotyped and well-behaved fashion can be used to great advantage to give simple means for optimising parts of the program.

4.6.1 Internal labels

Using most intermediate codes the following program parts would translate into effectively identical sequences:

```
-----------------------------------------------------
|       ->LAB   if X = 0    |     if X # 0 start    |
|       Y = 3               |        Y = 3          |
| LAB:                      |     finish            |
-----------------------------------------------------
```

At first glance this is as it should be, for the two program fragments are semantically identical and could therefore be implemented by the same object code, for example on the PERKIN-ELMER 3200:

```
 ----------------
|   L   1,X     | pick up X and set the condition code
|   BZ  $1      | branch equal (to zero)
|   LIS 0,3     | pick up 3
|   ST  0,Y     | store it in Y
| $1:           | define label $1
 ----------------
```

However, if it is known that the label $1 will only ever be used once, the code-generator may remember that the current value of the variable X will still be in register 1 following the label, and thus remove the need for it to be loaded again if it is required before register 1 gets altered. In the case of user-defined labels no statement can be made about the number of uses of each label without a complete analysis of the parts of the program where the label is in scope.

This suggests that I-code should maintain a clear distinction between user-defined and compiler-generated labels. Also, by making the rule that compiler-generated labels may only be used once, the internal representations of labels may be reused by the code-generator, removing the necessity for large tables of label definitions in this phase of compilation.

This now leaves the question of how to represent conditional jumps in the intermediate code. The first observation is that user-specified jumps need never be conditional, as they can always be surrounded by appropriate compiler-generated conditional jumps. This can be used to restrict the processing of conditions and tests to the compiler-generated jumps. The second observation is that in IMP77 conditionals are always associated with the comparison of two values or the testing of an implied boolean variable (predicates and string resolution).

There are currently three main ways in which processors handle this:

1 "compare" instructions are used to set flags or condition-codes which represent the relationship between two values (one of which is frequently an implied value of zero). These condition-codes are later used to control the execution of conditional branch instructions. This method is used in the PDP11: COMP, BNE etc.

2 Instructions are provided which compare two values as above but instead of setting condition-codes they skip one or more subsequent instructions depending on a specified relationship. By skipping unconditional branches in this way conditional branch sequences may be generated. This method is used in the PDP10: SKIPE etc.

3 Instructions are provided which compare two values and branch to a specified label if a given relationship holds. This method is used in the PDP10: JUMPNE etc.

P-code uses compare instructions to set the boolean value TRUE or FALSE on the stack and then uses this value either as an operand in an expression or to condition a branch (a variant of technique 1 above).
Z-code tests the value in a register against zero and branches accordingly (technique 3 above).

These three techniques have fairly obvious possible representations in I-code:

```
    if X = Y start


    1)  PUSH   X
        PUSH   Y
        COMP            {set condition code}
        BNE    1        {branch not equal}


    2)  PUSH   X
        PUSH   Y
        SKIPE           {compare and skip if equal}
        GOTO   1


    3)  PUSH   X
        PUSH   Y
        JUMP # 1        {compare and branch if not equal}
```

All three of these representations have been tried in different versions of I-code.
Technique 2) was rejected as it proved cumbersome to implement effectively, especially on machines which did not use skips; either the code-generator had to "look ahead" to be able to locate the destination of the skip (which is dependent on the instruction being skipped) or to check before each instruction whether on not a skip had been processed earlier and its destination had not yet been resolved.

Technique 1) was perfect for machines with condition-codes but required look-ahead over subsequent jumps on machines which used skips.

Both 1) and 2) had the additional problem that to generate conditional branches, two separate I-code instructions had to be given. In the case of 1) condition-codes are usually altered by many instructions not directly involved in comparison and hence the compare and its associated branch must be made adjacent. With 2) there is the possibility of generating meaningless constructions such as skipping a line-number definition instruction. These difficulties add complexity to the definition of the intermediate code and require extra checks in the code generator.

Thus the third form was chosen as the most convenient, even though all three forms can be suitably defined to be totally equivalent. In particular the third technique provides all the relevant information to the code-generator in one instruction, and has proved to be simple and effective as a basis for generating code for both condition-code and skip sequences.

Using these ideas the following is the expansion of the statements given at the start of section 4.6.1.

```
 ----------------- -----------------
| PUSH      X   |  PUSH      X    |
| PUSHI     0   |  PUSHI     0    |
| COMP #    1   |  COMP =    1    |
| JUMP      LAB |                 |
| LOCATE    1   |                 |
| PUSH      Y   |  PUSH      Y    |
| PUSHI     3   |  PUSHI     3    |
| ASSVAL        |  ASSVAL         |
| LABEL     LAB |  LOCATE    1    |
 ----------------- -----------------
```

4.6.2 Temporary objects

During the compilation of high-level languages it often becomes necessary to create temporary objects which are not present in the source program. The most common need for temporaries is in the evaluation of expressions. Regardless of the number of accumulators or registers available it is always possible to construct an expression which will require one more. To obtain this register, a register currently in use must be selected and the value currently in it must be saved in a temporary location. One apparent exception to this is a machine in which expressions are evaluated using a stack (e.g. ICL 2900) but in this case the operands are always in temporaries.

Temporary variables may also be required to implement certain high-level constructions, such as the IMP <u>for</u> statement:

```
-----------------------
| for V = A, B, C cycle |
-----------------------
```

which is defined so that the initial values of B and C, and the initial address of the control variable, V, are to be used to control the loop regardless of any assignments to V, B and C. While it is possible for a machine-independent optimiser to discover whether these variables are modified in the loop or not, in the simple case where little optimisation is required the code generator must use temporaries.

In the case of expression evaluation, however, the machine independent phase cannot know how many temporaries will be required. Even giving the first phase knowledge of the number of registers available is not adequate for several reasons. Firstly, the use of registers is commonly tied to the operations being performed, as in the case of integer multiplication on several machines which requires a pair of registers. For a machine-independent first phase to be able to cope with this sort of limitation would require great flexibility of parameterisation.

Secondly, the first phase would have to be given details of the problems encountered in statements such as:

```
 ----------------------------
| LEFT = REM(A,5) + REM(B,7) |
 ----------------------------
```

On a PDP11 equipped with the EIS option, a divide instruction is available which provides both the quotient and the remainder. Hence the statement could be compiled into:

```
 -----------------
|  MOV   A,R1     |
|  SXT   R0       | propagate the sign of A
|  DIV   R0,#5    | remainder to R1
|  MOV   B,R3     |
|  SXT   R2       |
|  DIV   R2,#7    | remainder to R3
|  ADD   R2,R0    |
|  MOV   R0,LEFT  |
 -----------------
```

In this case no temporary store locations are required. However, if the EIS option is not present, no DIV instruction is available and so a subroutine must be used instead. The code becomes:

```
 -----------------
|  MOV   A,R1     |
|  MOV   #5,R2    |
|  JSR   PC,DIV   | result back in R1
|  MOV   R1,T1    | preserve remainder
|  MOV   B,R1     |
|  MOV   #7,R2    |
|  JSR   PC,DIV   | result in R1
|  ADD   T1,R1    |
|  MOV   R1,LEFT  |
 -----------------
```

As the subroutine REM uses R1 (for one of its arguments and to return its result) the result of the first call on REM must be saved in a temporary, T1.

Of course, the function REM could be written so as to preserve the value in, say, R2 and this could be used instead of T1, but this would increase the cost of REM when it is likely that the value in R2 will not be of use as most expressions are trivial [Knuth, 1971].
Unless the machine-independent phase is given intimate knowledge of the target machine (something of a contradiction) it cannot know how many temporaries to use nor when to use them.

The solution adopted by most intermediate codes is to base the code around a stack, thus providing an unlimited number of temporaries which are handled automatically. While this in itself does not hinder the compilation for a machine without a hardware stack, as the code-generator can always simulate the stack internally, its presence invariably results in other parts of the code using it, for example to pass parameters to procedures where the receiving procedure contains built-in knowledge of the layout of the stack.

As a stack does not require the explicit mention of temporaries it has been adopted by I-code, but purely as a descriptive mechanism. Because I-code does not specify the computation but the compilation process needed to produce a program which will perform the computation, this internal stack need have no existence when the final program executes.

The implementors of SIMPL-T describe an intermediate code with some properties similar to I-code, but based on "quadruples" of operators and operands rather than an internal stack [Basili, 1975]. The stack approach was rejected by them because "quads allow more flexibility in the design of the code generator since, for example, no stack is required". The exact meaning of this is not clear but it suggests the misconception that a stack-based intermediate code forces a stack-based object code representation. Regardless of the exact structure of the code generator or the input it takes, some form of internal stack is invariably required for operations such as protecting intermediate values in registers which are needed for other purposes, and it seems reasonable to make this stack more explicit if so doing will simplify the intermediate code and its processing.

.7 Object file generation

Once a program has been compiled into sequences of machine code instructions, there still remains the task of producing an object file in a form suitable for processing by the operating system (if any) under which the program is to be executed. This task was separated from the main part of code generation (the second phase) and has become the third phase of compilation for the following reasons:

i The particular format required in the final object file will vary on any particular machine depending on the operating system in use. As this is to a large extent independent of the code sequences needed to implement the program, it was thought sensible to keep the processes separate.

ii Even following the generation of the code by the second phase there remain many opportunities for further optimisation, both global and structural, which require information only available once the complete program has been compiled. Rather than build global analysis into the second phase these optimisations were left to a third phase.

The third phase takes as its input two data streams generated by the second phase. These streams are:

i The object stream, a sequence of items of the form: <type> <value>* defining the code sequences required in the object file.

ii the directive stream, a sequence of items defining the logical structure of the object stream, that is a specification of label definitions and label references, and details of various code groupings (blocks, procedures etc.).

The third phase starts by taking in the directive stream and constructing a linkage map describing the whole program. This linkage map is processed and then used to control the generation of the final object file from the object stream. The operations performed using the map are:

4.7.1 Reordering

As discussed previously in section 4.3, there are several gains to be made by having the ability to output instructions in an order different from that in which they were implied by the linear structure of the source program. This reordering is performed on the linkage map in a manner controlled by items in the directive stream.

In the most simple case of exbedding procedures (section 4.3.1) this only entails allocating code addresses to the items in the map each time an "end-of-block" control item is input, resulting in the procedures being laid out in "end" order.
To facilitate evaluating references to the reordered areas, all references in the object stream are made relative to the start of the appropriate area.
As this process does not cause the physical moving of the various areas there is an implicit assumption that either the subsequent processing of the object stream can do the reordering (for example by writing its output to specific sections of a direct-access file), or that the object file format can instruct the loader or linker to do the shuffling.

With the linkage map available it becomes possible to make a preliminary pass over the object stream performing structural modifications which require knowledge of the generated code and which alter its size and general appearance. These modifications may be made by passing the object stream through a buffer which is scanned and modified under the control of the linkage table. In this way merging common code sequences and reordering the arms of conditional sequences may be achieved quite simply.

4.7.2 Jumps and Branches

Following the construction of the linkage map structural optimisations may be performed on jumps. The three optimisations which are currently applied are:

i **Use of the smallest instruction**

A common feature of machines is that they provide a variety of sizes of jump instruction, depending on the reason for the jump (conditional or unconditional) and the distance to be jumped.

e.g. PDP11

BEQ (2 byte instruction) conditional jump up to 256 bytes in either direction.

JMP (4 byte instruction) unconditional jump to anywhere.

Perkin-Elmer 3200

BFFS
BFBS (2 byte instructions) conditional jump forward (F) or backward (B) up to 32 bytes away.

BFC (4 byte instruction) conditional jump to within 16Kbytes of the current instruction.

BFC (6 byte variant) conditional jump to anywhere.

In typical programs the frequency of occurrance of such jumps is:

	PDP11	PE3200
2 byte	88%	28%
4 byte	8%	71%
6 byte	2%	<1%

It has been suggested [Brown, 1977] that the problem of deciding which form of jump to use can be eased on certain machines by specifying a "distance" parameter with the intermediate code, e.g. "GOTO LAB,80" informing the code generator that the label LAB is 80 instructions ahead.
It is difficult to think of any case in which this could be of any use as it requires the code generator to be able to predict the amount of target machine-code which will be generated for each intermediate code instruction.

The solution adopted by the IMP compilers has been for the code generator to assume that all jumps are the minimum size, and to let the third phase stretch them where necessary.
The Perkin-Elmer CAL assembler [Interdata, 1974] makes the opposite assumption, namely that jumps are long until proven short. This was rejected as the size of one jump is often dependent on another, so that one of them will be short if and only if both of them are short.

By assuming them long either they will never be found to be short, or the process will have to examine all the jumps repeatedly trying each jump in turn to see if it can be "squeezed". Commonly enabling the "SQUEZ" option in the CAL assembler can double or treble the time to assemble programs. With the assumption that all jumps start short and then grow, all truly short jumps will be found with no possibility of infinite loops, as the process must terminate, in the worse case when all the jumps have been made long.

Several methods for achieving this optimisation have been described [Szymanski, 1978; Williams, 1978].

The technique used by the third phase of the IMP77 compilers for stretching jumps is as follows.
Once the linkage map has been constructed and addresses provisionally allocated, all labels and references to them are grouped according to the block in which they occurred. This is to take advantage of the fact that most references will be local. A procedure STRETCH is now defined which repeatedly attempts to lengthen each reference within a particular group.

If a reference is found which must be stretched, the entry in the linkage map is updated and all subsequent entries are suitably modified to take account of the increased size of the code. The process is repeated until no alterations have been made.
STRETCH is first called once for each group of references in the program. This "local stretch" commonly resolves up to 80% of the references. A final call on STRETCH is then made with all the references lumped together as one group in order to resolve references between blocks, and any local references which, although processed by the local stretch, have become invalidated by changes made by the "global stretch".

The use of a local and a global stretch has a considerable effect on the performance of the compiler: If the calls on "local stretch" are taken out, "global stretch" has to do all the work in ignorance of the block-structure of the labels. This involves repeated searching of the complete label and reference lists in order that changes in the position of these items may be recorded. On the Interdata 7/32 this increases the stretching time for 1968 branches from 2.3 seconds out of a total compilation time of 146 seconds, up to 35 seconds!

The time taken to perform the stretching using both local and global stretch is on average just over 1% of the total compilation time excluding the time for input and output.
Wulf et al. describe an optimisation on the PDP11 which attempts to shorten otherwise long conditional jumps by making them jump to suitable jumps to the same destination, as this is smaller and faster than the six byte instructions which would be generated by default [Wulf, 1975]. This was tried but eventually removed from the PDP11 compiler as finding suitable jumps was a tedious task and of the average 2% of jumps which were long, in compiling many programs only one case was found where the optimisation could be applied. That case was in a program specially constructed to test the optimisation.

At the same time that jumps and labels are being processed, certain operations which depend on the flow of control may be inserted into the code.
The GEC 4080 provides a good example of this problem which can be handled elegantly by the third phase. The machine provides arithmetic instructions which take either fixed point or floating point operands depending on the state of a processor status bit. This bit must be altered by the instructions SIM (Set Integer Mode) and SFM

(Set Floating Mode). During code generation when a label is encountered the state of the status bit will not in general be known, and so a suitable mode switching instruction will need to be planted; frequently this instruction will be redundant. Given the presence of the third phase, the second phase merely needs to mark jumps with the current state of the bit, and to mark labels with the required state (and the previous state of the bit if control can "fall through" past the label). During the process of expanding jumps, these mark bits can be checked. If all references to a label have the same mode, no action needs to be taken, but if the bits differ the appropriate instruction must be added. As an extra improvement if only one jump to a label is from the wrong mode, the mode switching instruction can be planted before that jump rather than after its destination label, so shortening the execution paths when no change is required.

ii <u>Conflating jumps to jumps</u>.

Nested conditional structures in high-level languages often generate jumps which take control directly to another jump. If the second jump can be shown always to be taken whenever the first is, the first can be redefined as jumping directly to the destination of the second.

e.g.

```
---------------------------------------
| while N > 0 cycle                    |
|    N = N-1                           |
|    if N > 5 then TEST1 else TEST2    |
| repeat                               |
---------------------------------------
```

In this program following the call on TEST1 the _else_ causes a jump to be taken to the _repeat_. This statement is simply a jump back to the previous _cycle_.

Hence the following code can be generated (PE3200):

```
--------------------
| $1:  L   1,N     |
|      BLE $3      |
|      SIS 1,1     |
|      ST  1,N     |
|      CHI 1,5     |
|      BLE $2      |
|      BAL 15,TEST1 |
|      B   $1      |
| $3:  BAL 15,TEST2 |
|      B   $1      |
--------------------
```

The danger with this optimisation is that an otherwise short jump can be expanded to a long jump as the following program demonstrates:

```
-----------------------
|  if X = 1 start      |
|     if Y = 1 start   |
|        {A}           |
|     else             |
|        {B}           |
|     finish           |
|  else                |
|     {C}              |
|  finish              |
-----------------------
```

The else following the sequence {A} causes a jump to the next else which jumps past the finish. In that form, the first jump only has to skip {B} and is likely to be a short jump. If it is made to jump directly to the second finish it has to cover {B} and {C}, so reducing the chances of its being short.

Equally, the position can be reversed, resulting in the optimised jump being short when the original was long. If this problem is considered serious the third phase can check the sort of jump which would be generated and act accordingly.

iii Removal of jumps round jumps.

Statements such as:

```
------------------
| ->LABEL if X = Y |
------------------
```

are common, either in the explicit form as given above or in some higher-level representation such as:

```
---------------
| exit if X = Y |
---------------
```

The simple code sequence generated for this would be similar to (PE3200):

```
 --------------
 |    L   1,X   |  pick up X
 |    C   1,Y   |  compare with Y
 |    BNE $1    |  branch not equal
 |    B   LABEL |  jump to LABEL
 | $1:          |
 --------------
```

by combining the two branches the code can be reduced to:

```
 --------------
 |    L   1,X   |
 |    C   1,Y   |
 |    BE  LABEL |
 --------------
```

While it is possible for the code generator to do this immediately, it was decided to leave the optimisation to the third phase for four reasons:

1 The third phase can perform this optimisation simply, almost as a side-effect of constructing the linkage map.

2 The are several cases where the optimisation can be extended in ways which would be awkward for the second phase to deal with. In particular, it would have either to look ahead or to be able to modify code sequences already generated. With a third phase, however, the optimisation reduces to a straightforward inspection of the linkage map.

For example:

```
-----------------
|   exit if X = Y |
| repeat          |
-----------------
```

in which case the optimisation may be applied twice to reduce the code to two instructions.

3 Leaving the optimisation to a later phase simplifies the second phase which is the most complicated part of the compiler.

4 On several machines if the destination of the jump is too far away the original "jump round a jump" may be the only form available (e.g. PDP11). The distance to be jumped will only be known exactly when all labels have been processed.

4.7.5 In-line constants

When compiling for machines such as the Data General NOVA which have a limited direct addressing range and no full-length immediate operands, it is useful if constants can be planted in the code sequence and addressed as program-counter-relative operands. The simplest technique for doing this is for the code generator to maintain a list of required constants and to dump them in-line at a suitable opportunity before the limit of addressability has been exceeded.

Such constants will need to be protected from being executed and so will need to have a jump round them or will have to be planted in a "hole" in the code, that is between an unconditional jump and the next label. As holes occur frequently in high-level languages (for example following every _else_ or _repeat_) and do not require extra code to be planted round the constants, they must be the preferred position for the constants. In order to minimise the number of constants planted it is necessary to delay the dumping of them until the last possible moment, making them as near the forward limit of the addressability of the first outstanding reference. This increases the chance of a subsequent reference to the constant being able to address the previous location.

This poses problems if the second phase is to handle the constants as it cannot know which is the optimum position for the constants in advance of producing the code (especially if the code is to be reordered).

A convenient solution is to utilise the linkage table in the third phase and include in it references to constants and the locations of holes and "forced" holes, that is places where an extra jump is required.

Following the initial resolution of jumps (4.7.2) the list of constants can be examined and holes allocated. The labels are processed again to take account of the extra code and any alignment limitations. During the processing of the object stream the constants are infiltrated into the object file.

4.8 Summary

The major decisions about the design of the compiler were:

a) All information present in the source program should be easily visible in the intermediate code.

b) The intermediate code should be as machine-independent as the source language.

c) The code generator should be split into two distinct phases joined by a stream of code fragments and a linkage map defining the connections between them.

d) The intermediate code should handle objects in terms of language-dependent descriptors which are converted into appropriate machine-dependent descriptors by the second phase.

e) The intermediate code should distinguish clearly between objects explicitly specified in the source program and those implied by the translation.

f) All decisions about code and data addressing must be left to the code-generator.

5 Review of the overall structure

5.1 Division of function

The division of the machine-dependent phase into two parts was motivated by three main considerations:

i to localise the changes necessary to produce different object-file formats,

ii to permit the reordering of sections of the code,

iii to enable the production of short jumps whenever possible.

In addition it turns out that on all of the machines for which this technique has currently been applied points (ii) and (iii) can be handled by almost identical pieces of code, making this phase of compilation machine-independent to a large extent and therefore easing the task of creating new compilers.

Against this must be set the overheads incurred by separating the compilation into two parts which have to communicate. The interface between phases two and three comprises the object file and the directive file, and the third phase needs to process the whole of the directive file before starting to look at the object file.

The ways in which these 'files' will be implemented, and consequently the cost of the communication, will in general vary from system to system. If large virtual memories are available the data may be held in memory as mapped files or arrays, and accessed much more efficiently than on simpler systems using the conventional approach of 'true' files with their more cumbersome transfer operations.

5.2 Testing and development

Although the initial reason for choosing a multi-phase approach to compiling was that of simplifying the generation of new compilers, an extra advantage arose in that the task of checking the compilers so produced, and diagnosing faults in them was very much simplified. This was because of two features of the technique.

Firstly, the programs corresponding to the phases were of managable size, varying from about one thousand statements up to four thousand statements.

Secondly, the phases communicated with each other using well-defined interfaces which could be monitored to narrow down errors to a particular phase and even to specific parts of that phase.

In addition, as the structure of the intermediate code inevitably suggests the general techniques to apply in code generation, many of the complete compilers on different machines had great similarities; usually only the lowest levels of code production and machine-specific optimisation were appreciably different.

This gave rise to three convenient properties with regard to testing and development:

i An error in one compiler will frequently give notice of similar faults in others. Clearly, any faults in the common first phase will be present in all the compilers and only one correction will be required.

ii An improvement in the performance of one compiler, or the code it generates, can suggest similar improvements in others.

iii The third effect on reflection seems obvious yet was noted with some surprise. The systems on which most of the investigation was done, are run with very different operating systems and used by different types of user. These two factors together caused a great spread in the demands placed upon the compiler, resulting in more parts of the compiler being thoroughly tested than would happen when running on one particular system, where users tend to be more stereotyped. Questions of "proper practice" aside, it is a fact of life that all software gets a better testing in the field than at the hands of its creator.

5.3 Diagnostics

As mentioned previously, optimisation is not just a process of improving the storage requirements and speed of a program but also involves fitting a program into the overall framework of the run-time environment. In many applications the provision of extensive run-time checks and post-mortem traces can be of great importance. The ability to generate such diagnostic code has certain implications for the features in the intermediate code.

5.3.1 Line numbers

When producing information about the state of a computation, whether it be an error report following a run-time fault or an execution trace [Satterthwaite, 1972], the data must be presented in a form which is meaningful to the user in terms of the source program. The commonly-provided dump of the machine state, registers, code addresses etc., is a complete failure in this respect, as the correspondence between this and the program state depends on the workings of the compiler and other factors of which the user should not need to be aware.
The simplest way of specifying the point of interest in a program is to give its line number. There are two common techniques for providing line number information at run-time, the choice of which depends on the uses to which the compiler is to be put.

The first is to plant instructions which dynamically update a variable with the current line number whenever it changes. This has the significant advantages that it is extremely cheap to implement and the line number is always immediately available. Its obvious disadvantages are that it increases the execution time for the program, and more significantly, it increases the size of the program, typically by about 6K bytes on the Interdata 7/32 for a 1000 line program, approximately a 50% increase.

The second technique is to build a table giving the correspondence between line numbers and the addresses of the associated code sequences. While this imposes a greater burden on the compiler and takes more time to extract the line number, it has the advantage that it does not increase the code size of the program, nor does it alter its execution speed. Indeed it may even be possible to keep the table out of main memory until it is required.

The choice of technique will have implications on the compiled code. If the line number table approach is used error procedures must have available the address of the point of the error. The effects of this can be seen in the following example of the sort of code generated for unassigned variable checking on the 7/32 using both methods:

```
 -------------
| 17  X = Y   |
 -------------

 -------------  -------------
| LHI  0,17   |             | update line no
| ST   0,LINE |             |
| L    1,Y    |  L    1,Y   |
| C    1,UV   |  C    1,UV  | check value
| BE   ERROR  |             | give the error
|             |  BAL  8,TU  | test for error
| ST   1,X    |  ST   1,X   |
|             |  ..   ..    |
|             |  ..   ..    |
|             |TU:BNER 8    | return if OK
|             |  B    ERROR| give the error
 -------------  -------------
```

As the generated code depends on the method in use it cannot be specified in the intermediate code and so the latter must simply indicate the points in the program at which the line number changes.

5.3.2 Diagnostic tables

In the event of program failure, or when explicitly requested by the user, a trace of the current state of a program, including the values in active variables and the execution history, can be of immense value. For such a trace to be provided the intermediate code must contain the identifiers used in the source program for all the variables, and a source-dependent description of those variables. This latter is needed so that the machine representations may be interpreted in the correct way when giving the values in variables. In I-code all this information is presented in the definitions of descriptors and may be used or discarded at will.

5.3.3 Run-time checks

Most languages define circumstances under which a program is to be considered in error and its execution terminated. These errors include creating a value too large to be represented (overflow), division by zero, use of an array index which is outwith the declared bounds, and so on.
There is a natural division of these errors into those which are detected automatically by the machine and those which must be detected by explicit checks in the program. Commonly, machines catch division by zero automatically but do not provide such a feature for checking array subscripts. The "hardware-detected" errors may be furthur divided into

those which on detection cause the normal flow of control to be interrupted, and those which simply make the knowledge of the occurrance of the error available to the program, for example by setting a condition-code bit. For the purposes of this discussion the second form of hardware-detected error may be considered an error which is not detected automatically, as it still requires explicit instructions to test for the error and to transfer control accordingly. Clearly, the more errors that fall into the automatic category the better, as they do not cause the user's program to grow with sequences of instructions which, in a correct program, will always be testing for conditions which never arise.

These differences complicate the design of intermediate codes as the classification differs from machine to machine: with the VAX all forms of overflow can be made to generate automatic interrupts, but the PDP11 only sets a condition-code bit on some overflows.

There are two basic ways of handling this in the intermediate code: firstly the code can contain explicit requests for the checks to be performed, and secondly the code can be designed in such a way as to give the code-generator enough information to be able to decide where checks are necessary.

Two specific examples can indicate which of these ways should be adopted.

Testing for arithmetic overflow is currently handled by machines in three main ways:

1. An interrupt is generated whenever overflow occurs. This is by far the best method as it requires no overheads in the checked code.

2. A bit is set on overflow and is only cleared when it is tested. This requires explicit checks in the code but several tests may be conflated into a single test at an appropriate point, for example before the final result is stored.

3. A bit is set on overflow, but is cleared by the next arithmetic operation. This again requires explicit checking code but the tests must be inserted after every operation.

For the intermediate code to indicate where overflow testing is to be performed it would have to choose the worst case from the three above, namely case 3. This would result in a test being requested after every arithmetic instruction, which test may just as well be included into the definition of the instructions themselves.

The other area of low-level testing is in implied type conversions such as storing from a 32-bit integer into a 16-bit integer. The VAX provides an instruction which combines the test for truncation with the store (CVTLW). The 7/32 has an instruction (CVHR) which can test the value before assignment, and the 4/75 can most efficiently test following the assignment (CH).

If the request for the check is a separate intermediate code item, the 7/32 case is simple but the other machines will require much more work to be able to generate the efficient check. The problem can be simplified by introducing new assignment instructions which also perform the test, but this adds many new instructions to the code as one instruction will be required for every valid combination of types and every sort of assignment.

The high-level checks such as array bound checking are usually so complicated that the most efficient implementations depend greatly on the particular hardware, so much so that it would be foolish to attempt to express them in the intermediate code. The simplest solution is to ensure that the intermediate code provides enough information to let the code generator decide where and what checks are necessary.

The inclusion of checks against the use of unassigned variables provides a good example of the power of leaving the checking to the code-generator. In a simple-minded approach the code-generator tests every suitable value loaded from store. A minor improvement to this is to mark the descriptor for every local variable in a block when it is first assigned, inhibiting the marking after the first jump. Subsequently, marked objects need not be checked.

A much better improvement may be obtained by making a trivial extension to the register remembering mechanism. If an object is 'known' it must have been used previously, and hence it will have been checked if necessary. Even after the register which held the value of the object has been altered, and hence the association between the register and the object lost, if the compiler remembers that the value *was known* it can suppress any unassigned checks on future references.

At this point a useful property of IMP77 may be used to great effect: once a variable has been assigned it cannot become unassigned. This is not true in many languages, as for example, in ALGOL60 the control variable of a *for* loop is undefined (unassigned) at the end of the loop. This means that in IMP77 the 'was known' property of variables may be preserved across procedure calls, even though all the register content information must be forgotten.

This technique when applied on the 7/32 compiler results in a reduction of 33% in the code required for checking. While it is possible for the unassigned checks to be placed in the intermediate code and for the first phase to remove redundant checks, this supression would require a duplication of the remembering logic which must, in any case, reside in the machine-dependent phase.

6 Observations

6.1 Suitability of I-code for Optimisation

When considering the use of I-code for global optimisation there are two techniques available:

Firstly, the optimisations can be performed using the I-code and going straight into object code, possibly via a third phase. In this case the only real constraint on I-code is that it be powerful enough to be able to carry all the information available in the source and to present it in a compact form.

Secondly, the optimisations can be seen as an extra phase introduced between the first phase (the I-code generator) and what is normally the second phase (the code generator). The optimiser takes in I-code and produces as its output a new I-code stream which can be fed into the code generator. In this case not only must the I-code carry all the source information but it must be able to describe the generation of an optimised program. Clearly the code must be able to reflect the structure of the target machine in some way and hence must be able to lose its machine independence.

The second technique is the more interesting as not only does it permit the optional inclusion of the global optimising without affecting the structure of the other phases, but it removes the optimisations from the low-level details of code production and provides a means for separating the machine-independent and machine-dependent optimisations. In particular in the same way as much of the code generator can be built from a standard "kit" with a few special machine-specific parts, so the global optimiser can utilise code from other optimisers.

The way in which the optimiser can influence the operation of the code generator is by making use of the fact that the intermediate code does not describe a computation but a compilation process. This compilation is driven by the descriptors which are normally translated by the code generator from the machine-independent form in the I-code into the appropriate machine-dependent representation, reflecting the target machine architecture: registers, stacks, memory etc. By short-circuiting this translation a global optimiser can force the use of specific machine features.

For example consider the following fragment of an integer function:

```
 ------------------
|  integer X        |
|  X = A(J)         |
|  X = 0 if X < 0   |
|  result = X       |
 -------------------
```

The standard I-code produced for this fragment would have the form:

```
 -----------------------
|  DEF 12 "X" INTEGER    |
|   SIMPLE DEFAULT NONE  |
|   NONE                 |
|  PUSH      12          | - X
|  PUSH      6           | - A
|  PUSH      7           | - J
|  ACCESS                |
|  ASSVAL                |
|  PUSH      12          | - X
|  PUSHI     0           |
|  COMP >=   1           |
|  PUSH      12          | - X
|  PUSHI     0           |
|  ASSVAL                |
|  LOC       1           |
|  PUSH      12          | - X
|  RESULT                |
 -----------------------
```

On the PDP11 the code generated for this could be:

```
 ------------------
|      MOV   J,R2   |
|      ADD   R2,R2  | Scale the index
|      ADD   A,R2   | Add in ADDR(A(0))
|      MOV   (R2),X | X = A(J)
|      BGE   $1     | ->$1 if X >= 0
|      CLR   X      | X = 0
| $1:  MOV   X,R1   | assign result register
|      {return}     |
 ------------------
```

Here the obvious optimisation is to note that the local variable, X, is eventually to be used as the result of the function and so needs to end in register 1.

By changing the definition of X in the I-code into:

```
 ------------------------------------------------
| DEF x X INTEGER SIMPLE DEFAULT NONE SPECIAL R1 |
 ------------------------------------------------
```

and making no other changes, the code generator will produce code of the form:

```
 --------------------
|      MOV  J,R2     |
|      ADD  R2,R2    |
|      ADD  A,R2     |
|      MOV  (R2),R1  |
|      BGE  $1       |
|      CLR  R1       |
| $1:  {return}      |
 --------------------
```

As this process necessitates the I-code becoming more and more intimately involved with the structure of the target machine, in that it starts referring directly to registers and the like, it is necessary that a new control item be added so that the code generator may be prevented from pre-empting resources which the optimiser is manipulating. The new item is RELEASE and it is used in conjunction with the definition of machine-dependent descriptors. When such a descriptor is introduced (using DEF) the associated target machine component is considered to have been claimed and may only be used in response to explicit direction from the I-code. On receipt of the corresponding RELEASE the component is once again made available for implicit use by the code generator (for temporaries etc.). This mechanism is an exact parallel to the way in which memory locations are claimed by the definition of descriptors and released by the END of the enclosing block.

The main assumption about this style of optimisation is that the code generator has the ability to generate any required instruction, provided that the pertinent information is available at the required time.
As an example, the VAX 11/780 provides addressing modes in which the value in a register may be scaled and added into the effective operand address before the operand is used, hence the following code:

```
 ------------------------
|  integerarray A(1:9)   |
|-                      -|
|  A(J) = 0              |
|                        |
|  MOVL   J,R5           |   pick up J
|  CLRL   12(R3)[R5]     |   A(J) = 0
 ------------------------
```

The operand address generated by the CLRL instruction is:

```
 ------------------
|  12+R3 + R5*4    |
 ------------------
```

as there are 4 bytes (address units) to a longword.
This instruction can be generated naturally during the non-optimised evaluation of array subscripts, and so the optimiser can assume that the index mode of operand will be used whenever a register operand is specified as an array index.
The procedure has the added advantage that in the worst case when the code generator will not produce the instructions that the optimiser hoped, as long as the optimised I-code still describes the required compilation, the code generator will simply produce a more long-winded, but equally valid version of the program.

In other words, as long as some choice is available and some temporary objects are left at the disposal of the code generator, the optimiser cannot force it into a state where working code cannot be produced. In the example above even if the code generator does not produce index mode operands, it can still generate sequences of the form:

```
 ---------------------
| MULL3   R5,#4,R1    | R5*4  -> R1
| ADDL2   R3,R1       | R3+R1 -> R1
| CLRL    12(R1)      |     0 -> (12(R1))
 ---------------------
```

6.2 Performance

The figures in appendix A3 are the results of measuring the effect of various optimisations on the Interdata 7/32 and the DEC PDP11/45.
One problem in choosing programs to be measured is that heavy use of particular language features will increase the overall effect of certain optimisations.

As a trivial example of this consider the following "program":

```
--------------------------------
!  begin                        !
!     integerarray A(1:1000)    !
!     A(1) = 0                  !
!  endofprogram                 !
--------------------------------
```

With all array optimisations enabled, on the 7/32 this generates 30 bytes of code, whereas without the optimisation it results in 170 bytes of code, largely due to the procedure for declaring the array.
Clearly a reduction of 82% is not to be expected on more typical programs.

Similarly the absence of features will bias the results. In particular the smaller programs will not demonstrate the power of the optimisations which only take effect when various size limits have been exceeded: the most obvious such limits being addressing restrictions caused by the size of address fields in instructions.

The major difficulty in producing results which are of any real value is that the effects of the optimisations depend on the individual style in which the programs under consideration were written. Inevitably users get a "feel" for the sort of statement for which the compiler generates good code and they often modify their style of programming accordingly. If at some state in its development a compiler produces poor code for a particular construction, users will tend to avoid that construction, even long after the compiler has been improved and can compile it effectively. This well-known phenomenon [Whitfield, 1973] argues strongly that users should never see the object code generated by the compilers they are using.

The effects of many optimisations are difficult if not impossible to measure with any degree of accuracy as they interact with other optimisations to a great deal. The most obvious interaction is that between the size of jump instruction required and most of the other optimisations. The size of jump is determined by the amount of code generated between the jump and the label it references. If any other optimisation is inhibited this volume of code is likely to increase, decreasing the chances of being able to use the shorter forms of the jump.

Some optimisations depend almost totally on others; it is unlikely that the optimisation of reducing or removing the entry and exit code sequences associated with procedures (section 4.5.1) would have much effect if the parameters were not passed in registers and references to them in the procedures were replaced by references to those registers.
In particular, it must be noted that it is always possible to generate programs which will benefit greatly from those optimisations which do not appear to be of much use from the figures given. However, the test programs used to derive the figures are typical of the programs processed by the compiler, and it is hoped that they give a more realistic and balanced view of the improvements which may be achieved in 'real' cases.

Under some circumstances it may be advantageous to apply all optimisations, even though some may appear to give little benefit, since this 'squeezing the pips' frequently removes one or two instructions from critical loops in a program.
Yet again this shows the difficulty in quantifying the usefullness of optimisations as they are so dependent on the particular circumstances.

One area of measurement has been deliberately omitted from the figures, namely the effect on execution time of the optimisations. This was for several reasons:

1. On the systems used it was impossible to get reliable timing measurements with any accuracy greater than about plus or minus 5%.

2. For the reasons given previously, many programs could benefit greatly from fortuitous optimisations which removed just one crucial instruction, optimisations which could not be expected in every program.

3. Programs which executed for long enough to improve the accuracy of the measurements, invariably lost this accuracy through spending much time in the system-provided procedures, mainly for input and output. This point in particular suggests that as the overhead is beyond the control of the general user, the savings in code space may be much more important. Even with ever-growing store sizes, virtual memory systems will continue to treat smaller programs better than larger ones.

4. Some of the optimisations, particularly passing parameters in registers, prevent the compiled program from running, unless the controlling environment is modified in a parallel way. This would invalidate the timings as the environment is not usually under the control of the compiler.

From the crude measures which were obtained there is a suggestion that the decrease in execution time roughly parallels the decrease in code size.

6.3 Cost of optimisation

The cost of an optimisation is, in general, very difficult to measure, as may be seen by considering the three relevant areas: compile time, space requirement, and logical complexity.

6.3.1 Compile time

In order to generate good code, the compiler must spend time looking for the cases which are amenable to improvement. If no optimisation is performed this time is not used and so the compilation should take less time. However, the non-optimised version commonly requires the production of more code than the optimised version, frequently over fifty percent more when comparing fully diagnostic code with fully optimised code. On all the compilers written so far, the time saved by not having to generate these extra instructions, more than outweighs the time spent in deciding not to generate them.

6.3.2 Space requirement

Several optimisations increase the requirement for workspace, notably all the remembering optimisations. On most machines available at the present, the number of things which may be remembered is fairly small: sixteen registers and one condition-code is probably the maximum. Even if this number is increased by remembering several facts about each •thing, the total amount of space needed will be small when compared with the space needed to hold the information about user-defined objects, information which is required whether optimisation is being performed or not. On large machines the extra memory required will be cheap; on small machines the need for the optimisation will have to be balanced against the size of the largest program which must be compiled.

6.3.3 Logical complexity

The cost of providing an optimisation includes a non-recurrent component, which is the difficulty of performing the optimisation at all because of the logical complexity of discovering the necessary circumstances. In a system which is aimed at portability this cost can often be shared over a number of implementations; the techniques used in one being applicable to others, perhaps after minor modifications.

6.4 Comments on the results

6.4.1 Register remembering

Of all the optimisations tested, a simple remembering of values in registers provided by far the greatest improvement in code size.

One problem in implementing this optimisation is deciding what to remember, as shown by the following code sequence:

```
 -----------
 |   X = Y   |
 |           |
 |   L  1,Y  |
 |   ST 1,X  |
 -----------
```

Following this sequence register 1 will contain both the value in X and the value in Y; should the compiler remember X or Y or both?

The measurements show that the gain in remembering both (2 uses) as opposed to just one (1 use) are quite small. The algorithm used to determine what to remember in the '1 use' case was simply to remember a new piece of information only if nothing else was known about the register in question. This gives the best results in cases such as:

```
A = 0;  B = 0;  C = 0
```

where the value '0' will be remembered, but will perform badly with the more contorted case:

```
A = 0;  B = A;  C = B
```

as again only the value '0' will be remembered. Unless very tight code is required, the cost in maintaining multiple sets of information about each register and searching for particular values will probably rule out such extended remembering optimisations.

Perhaps a surprising result is that the PDP11 on average gains about as much from this optimisation as the 7/32.
This is the result of two interacting effects.
Firstly, the 7/32 dedicates up to five registers to address local variables in the last five levels of procedure nesting, and locks three for other fixed purposes, leaving about ten for intermediate calculations. The PDP11, however, uses a display in store to access intermediate levels, and has to load the address of a particular level each time it is required. In addition the PDP11 implementation fixes the use of four registers, leaving only four for intermediate calculations.
Secondly, the 7/32 needs to use at least one register to move values around while the PDP11 often requires none.
These two effects give a fairly large number of transient values in the registers of the 7/32, and a smaller number of more frequently used values (addresses) in the registers of the PDP11. On average it appears that the number of times necessary values are found is roughly equal in the two cases.

6.4.2 Remembering environments

An environment is the complete knowledge maintained by the compiler at any time. By remembering and merging environments while compiling IF-THEN-ELSE constructions, the effects of the implied labels and jumps on the remembering optimisations can be minimised.

The measurements show that the gains achieved by remembering more and more environments fall off very quickly; two environments seem to be about the best. However, the overhead in providing more than one environment is simply compiler table space, and so a compiler which can handle one environment can easily handle more to get a very small but cheap gain.

One clear result is the difference between the effects on the two machines (sometimes an order of magnitude). This is almost entirely due to the difference in the number of available registers.

6.4.3 Array allocation and use

From monitoring service versions of the compilers is it clear that in IMP77 the vast majority of arrays have constant bounds. Allocating these arrays on the local stack frame at compile time is a simple operation and can save a fair amount of code, much of which would only be executed once, as most arrays are declared in the outermost block.

Remembering array address calculations can reduce the code by about five percent, but it commonly has little effect and is quite tedious and expensive to achieve. The small increase in code size for a few cases is a side-effect of the register allocation mechanism. Registers are chosen by giving priority to those about which the least is known, and then by selecting the least recently used such register. Hence, which register will be used depends on the compilation of previous statements. When a value is required in a specific register, for example during parameter transmission, occasionally it will already be in that register purely by chance. A minor change in the generated code, such as not requiring a new register for an array access, can result in the value not being in the correct register later on.

This instability seems to be undesirable, but alternative strategies, such as biasing the allocation towards or away from particular registers, on average results in worse code.

6.4.4 Common operands

On the 7/32 the only instruction which can be used to simplify statements of the form: X = X op Y is the AM (add to memory) instruction. It is therefore somewhat surprising that its use frequently saves over two percent of the code.

The two possible expansions of a suitable addition statement are:

```
 ------------ -----------
|            |           |
|  L  1,Y    |  L  1,X   |
|  AM 1,X    |  A  1,Y   |
|            |  ST 1,X   |
 ------------ -----------
```

The first saves four bytes and leaves the increment in the register. Even if the incremented value is required immediately afterwards, the extra load instruction will only increase the code size to that of the alternative sequence.

As the PDP11 has many instructions which can be used in this way it is hardly surprising that it benefits much more.

6.4.5 Parameters in registers

This optimisation gives another significant saving in code at little cost to the compiler, simply by moving the store instructions for parameter assignment from the many calls to the unique procedure definitions. The effect is more pronounced on the 7/32 as all assignments require two instructions, a load and a store, whereas the PDP11 can usually make do with one MOV instruction. In the latter case the saving comes from the ability to reduce the size of the procedure entry and exit sequences if all of the parameters can be passed in registers.

6.4.6 Condition-code remembering

On machines with condition codes many instructions set the result of a comparison with zero as a side-effect. Knowledge of this can be used to inhibit explicit code to compare values with zero. However, the small benefit so gained suggests that it is not worth doing, even though it is a very cheap test.

6.4.7 Merging

The large difference between the effect of forward merging on the 7/32 and the PDP11 is mainly due to the addressing modes available on the machines.
On the PDP11 statements of the form "A=B" can be compiled into a single instruction "MOV_B,A", ignoring any extra instructions which may be needed to make A and B addressable. However, on the 7/32 all values must be moved via the registers, resulting in two instructions for the same statement:

```
 ------------
|  L   1,B   |
|  ST  1,A   |
 ------------
```

Hence the following code:

```
 -----------------------------
|  if X=0 then Y=1 else Y=12   |
 -----------------------------
```

```
         7/32                  PDP11
 ------------------  ------------------
|     L     1,X     |     TST   X        |
|     BNE   $1      |     BNE   $1       |
|     LIS   2,1     |                    |
|     ST    2,Y     |     MOV   #1,Y     |
|     B     $2      |     BR    $2       |
|  $1:LIS   2,12    |  $1:               |
|     ST    2,Y     |     MOV   #12.,Y   |
|  $2:              |  $2:               |
 ------------------  ------------------
```

With the 7/32 code, merging can reduce the sequence by one instruction, a "STore", while with the PDP11 no such improvement is possible.

As the techniques for merging and delaying are quite expensive, but not complicated, and have a major influence on the design of the code-generator, the small gains achieved are probably not worth the trouble, unless the last drop of efficiency is required at all costs.

6.5 Criticisms and benefits of the technique

6.5.1 Complexity

The main argument against the use of high-level intermediate codes is that they move the complexity of code generation from the common machine-independent phase into the machine-dependent phase, forcing work to be repeated each time a new compiler is required.

While this is undoubtedly true, the overheads are not as great as they may at first appear.

The extra complexity of the code generators may be split into two parts: an organisational part which builds and maintains the structures used during the compilation, and processes the intermediate code, using it to drive the second part, an operational part which uses the structures to generate code as instructed by the organisational part.

The changes in the organisational part when moving to a new machine are small enough to permit the use of large sections of code from old compilers. Even when considering the operational part, much will be similar from machine to machine, in particular the communication between the second and third phases and the bulk of that latter phase can be taken without change. From examining the compilers produced using I-code it appears that about 60% of the source of the machine dependent parts is common, 20% can be considered as being selected from a standard "kit" of parts, and the final 20% unique to the host machine.

6.5.2 I/O overhead

One of the disadvantages of dividing a compiler into several distinct phases is that it results in an additional cost in communicating between consecutive phases. As discussed in section 5.1 this cost depends on the operating system running the compiler. Even in the worst case where communication is achieved using conventional files the overhead may not be too serious.

The time spent doing input and output on the Interdata 7/32 compiler is about 27% of the total compilation time, and for the PDP11 is about 22%, breaking down as follows:

```
 ---------------------------------------------------------
|                                                         |
|                  ----        ----        ----           |
|                 |    |      |    |-----> |    |         |
| Source ----> | P1 |----> | P2 |        | P3 |----> Object |
|                 |    |      |    |-----> |    |         |
|                  ----        ----        ----           |
|                                                         |
| 7/32:    7%          7%          10%         3%         |
|         (4%)        (4%)        (5%)        (3%)        |
|                                                         |
| PDP11: 9.4%         11%         0.6%        0.5%        |
 ---------------------------------------------------------
```

The figures in parentheses give the percentage of time taken when the input and output requests are made directly to the file manager rather than via the standard subsystem procedures, thus reducing the internal I/O overhead to about 10% of the total compilation time.

6.5.3 Lack of Gains

It has been argued that the increases brought about by adopting a high-level code as opposed to a low level one are not worth the increased effort involved in processing it. Depending on the uses to which the compiler is to be put, small increases in code efficiency can outweigh a reasonable increase in the cost of producing the compiler and using it. A 5% improvement in the execution speed of the compiler itself is not insignificant when the number of times it is used and the cost of each use are considered. However, it cannot be denied that a careful redesign of critical parts of a program can have a greater effect on its performance than any amount of automatic optimisation. Notwithstanding, it seems reasonable that programmers should be able to concentrate on the large-scale efficiencies of program design and have the detailed improvements left to the compiler.

Also it should be noted that measurements indicate that the compilers execute faster when performing certain optimisations than when not performing them, for example passing parameters in registers.

If low-level codes are needed for some reason, the complexity saved from the machine independent phase can be moved into a new phase which converts the high-level code into a low-level one. This provides the low-level code for those who want it while preserving the high-level interface for use when good code is required.

One important gain in using such intermediate codes is that they can ease the difficulties associated with maintaining a number of compilers for different machines, when those compilers are self-compiling.
For several reasons it may not be desirable to permit sites to have the source of the machine-independent phase: commonly to give freedom of choice for the form of the language in which that phase is written and to prevent local "improvements" which rapidly lead to non-standard language definitions. In such cases the intermediate-code generator can be maintained at one site and updated versions can be distributed in the intermediate code form without fear of compromising the quality of the object code generated from it. Such a technique is currently being used in the production of portable SIMULA compilers [Krogdahl, 1980].

6.5.4 Flexibility

At some stage in producing a compiler, the needs of the end user must be considered. The flexibility afforded by the high-level nature of the intermediate code allows the compiler to be adapted to fit its environment. If the compiler is to be used for teaching, the quality of the code it produces can be sacrificed for compilation speed and high-quality diagnostics, particularly as compilation time may well be an order of magnitude greater than the execution time, indeed many of the programs will fail to compile and never reach execution.

If the application is for compiling programs that will be compiled once and then executed many times, more effort can be expended in producing fast code, although this is not to say that diagnostics and fast code must be kept separate as the longer a program runs without failing the more trouble will be caused when it fails without convenient diagnostics.

6.6 Comments on Instruction sets and compilation

Following the production of IMP compilers for several different processors, various features of instruction sets have become evident which influence the generation of code.

i The instruction set should be complete, that is, where an instruction is available for one data type it should be available for all data types for which it is well-defined. Similarly, instruction formats used by one operation should be available for all similar operations. The best example of such an instruction set is that provided by the DEC PDP10. Unfortunately the majority of machines are not so helpful. As an example of the sorts of thing which go wrong, consider the Perkin-Elmer 3200 series. These machines provide three integer data types: fullword (32 bits, signed), halfword (16 bits, signed), and byte (8 bits, unsigned). There are "add fullword" (A) and "add halfword" (AH) instructions but no "add byte" instruction.

There are "add immediate short" and "subtract immediate short" instructions but multiply, divide, and, or etc. do not have short immediate operands.

ii The instructions should be consistent, that is, logically similar instructions should behave in similar fashions.
Again, on the Perkin-Elmer 3200:
Load fullword (L) and load halfword (LH) set the condition code but load byte (LB) does not.
Most register-store instructions can be replaced by a load of the appropriate type followed by a register-register instruction: e.g.

```
 -------------- --------------
|  CH    1,X   |  LH    0,X   |
|              |  CR    1,0   |
 -------------- --------------
```

both result in the same setting of the condition code, but

```
 ------------- --------------
|  CLB 1,B    |  LB    0,B   |
|             |  CLR   1,0   |
 ------------- --------------
```

could result in different settings of the condition code as CLR compares two unsigned 32 bit quantities whereas CLB compares a zero-extended byte from store with the zero-extended least significant byte of register 1. For consistency, either compare halfword (CH) should use the sign-extended less significant half of the register, or better, CLB should not tamper with the value in the register.

iii Complex instructions should be avoided. There are two reasons for this. Firstly, it is easier for a compiler to break down statements into simple operations than it is to build them up into complex ones [Stockton-Gaines, 1965]. Secondly, if the complex instructions do not perform the exact function required by the language, more instructions will be needed to "prepare" for the complex instruction and to "undo" its unwanted effects. As an example, the DEC VAX11/780 is full of complex instructions which seem to be well-suited to high-level languages at first glance, but on closer inspection they are not so useful. A CASE instruction is provided which indexes into a table of displacements and adds the selected value to the program counter. This would seem ideal for compiling SWITCH jumps. Unfortunately, as the table of displacements follows the CASE instruction it would be very expensive to use it each time a jump occurred using a particular switch. Instead all references to the switch must jump to a common CASE instruction. Even this does not help, as in the event of an attempted jump to a non-existent switch label, the diagnostics or the event mechanism will see the error as having occurred at the wrong place in the program.

Although this problem can be "programmed around" it turns out that it is faster to implement switches using sequences of simpler instructions.

iv Machine designers should investigate carefully the full consequences of building-in special fixed uses of machine features. One of the best examples of a clear oversight which causes grief to compiler writers is found in the DATA GENERAL NOVA multiplication instruction. This instruction multiplies the value in register 1 by register 2 and places the double-length result in registers 0 and 1. As only registers 2 and 3 may be used for addressing, and as register 3 is always used for subroutine linkage, it follows that register 2 must be used for addressing the local stack frame, but this is exactly the register which must be corrupted in order to use the multiply instruction!

Although specific machines have been used in the examples, similar problems abound in all machines. Indeed it is clear that machines are most commonly designed for programmers writing in assembler or FORTRAN, and furthermore writing their programs in a particular style.

While it is clear that the problems could be called "mere details" and that they are not difficult to surmount, it remains that they complicate otherwise simple code-generation algorithms, making compilers larger, slower, and correspondingly more difficult to write, debug, and maintain.

In conclusion it appears that the machine most suited to supporting high-level languages should have a small but complete set of very simple instructions, their simplicity permitting rapid execution and great flexibility.

7 Conclusions

7.1 Viability of the technique

The techniques described above have been used to create several IMP77 compilers which are in regular use on a number of systems. In terms of total memory space required for a compilation, about 80K bytes on the 7/32, they compare favourably with other compilers. The major weakness seems to be execution time which can vary from twice as long as other compilers in the worst case, to half as long in the best case. As most of the effort in writing the compilers was spent in investigating the techniques involved and not in minimising compile time, and as the compilers which ran much faster were either totally, or partially written in machine code (the IMP77 compilers are all written exclusively in IMP77), it seems that the technique can be used to produce acceptable service compilers.

7.2 Ease of portability

Although using I-code does not permit compilers to be written in as short a time as with P-code and OCODE, the large amount of code which is common to all of the compilers written so far means that, given a working code generator as a template, a new optimising compiler can be written in the space of a few months, with the final result producing code of high quality.

7.3 Nature of optimisations

During the course of the investigation it became clear that one of the difficulties of optimisation is that gains are achieved by applying a large number of ad hoc rules, especially where peephole optimisations are concerned.
As instruction sets become more complicated and rich, there is a corresponding increase in the variety of ways of implementing high-level language features. This increases the possibilities of optimisation and subsequently the complexity of compilers. By using high-level intermediate codes, such as I-code, it should be possible to concentrate on machine-independent optimisations knowing that the resulting intermediate code can be used to generate efficient code for current machines. Eventually, when better instruction sets are available, hopefully with only one way of doing things and no opportunities for non-trivial optimisation, the same intermediate code can be used to drive code generators which are much simpler and more directly portable.

Appendix A1

The IMP Intermediate Code

A Brief Summary

The IMP intermediate code may be considered a sequence of instructions to a stack-oriented machine which generates programs for specific computers. It is important to note that the intermediate code describes the compilation process necessary to generate an executable form of a program; it does not directly describe the computation defined by the program.

The machine which accepts the intermediate code has two main components:

1 A Descriptor area. This is used to hold descriptors containing machine-dependent definitions of the objects the program is to manipulate. This area is maintained in a block-structured fashion, that is new descriptors are added to the area during the definition of a block and are removed from the area at the end of the block.

2 A Stack. The stack holds copies of descriptors taken from the descriptor area or created specially.

Items on the stack are modified by intermediate code control items to reflect operations specified in the source program. Such modifications may or may not result in code being generated. From the point of view of this definition stack elements are considered to have at least three components:

i Type

ii Value

iii Access rule

The "Access rule" defines how the "Type" and "Value" attributes are to interpreted in order to locate the described object.

For example, the access rule for a constant could be "Value contains the constant" while for a variable it could be "Value contains the address of the variable". Clearly, the access rules are target-machine dependent. Descriptors may be combined to give fairly complex access rules, as in the case of applying "PLUS" to the stack when the top two descriptors are for the variable X and the constant 1, resulting in one descriptor with the access rule "take the value in X and add 1 to it". The complexity of these access rules may be restricted by a code-generator. In the example above code could be generated to evaluate X+1 resulting in an access rule "the value is in register 1", say.

The importance of the code not describing the actual computation which the source program specified but the compilation process required, is seen when attempting to use the code for statements of the form:

A := _if_ B=C _then_ D _else_ E;

This could not be encoded as:

```
PUSH   A
PUSH   B
PUSH   C
JUMP # L1
PUSH   D
BR     L2
LOC    L1
PUSH   E
LOC    L2
ASSVAL
```

The reason is that the items on the stack at the time of the ASSVAL would be (from top to bottom) [E], [D], [A], because no items were given which would remove them from the stack. hence the ASSVAL would assign the value of E to D and then leave A dangling on the stack.

Unless otherwise stated, all constants in the intermediate code are represented in octal.

Descriptors

DEF TAG TEXT TYPE FORM SIZE SPEC PREFIX

This item causes a new descriptor to be generated and placed in the descriptor area. On creation, the various fields of the DEF are used to construct the machine-dependent representation required for the object.

TAG	is an identification which will be used subsequently to refer to the descriptor.
TEXT	is the source-language identifier given to the object (a null string if no identifier was specified).
TYPE	is the type of the object: GENERAL, INTEGER, REAL, STRING, RECORD, LABEL, SWITCH, FORMAT.
FORM	is one of: SIMPLE, NAME, ROUTINE, FN, MAP, PRED, ARRAY, NARRAY, ARRAYN, NARRAYN.
SIZE	is either the TAG of the appropriate record format descriptor for records, the maximum length of a string variable, or the precision of numerical variables: DEFAULT, BYTE, SHORT, LONG.

SPEC	has the value SPEC or NONE depending on whether or not the item is a specification.
PREFIX	is one of: NONE, OWN, CONST, EXTERNAL, SYSTEM, DYNAMIC, PRIM, PERM or SPECIAL. If SPECIAL is given there will follow an implementation-dependent specification of the properties of the object (such as that it is to be a register, for example).

Parameters and Formats

The parameters for procedures and the elements of record formats are defined by a list immediately following the procedures or format descriptor definition:

START	Start of definition list
FINISH	End of definition list
ALTBEG	Start of alternative sequence
ALT	Alternative separator
ALTEND	End of alternative sequence.

Blocks

BEGIN	Start of BEGIN block
END	End of BEGIN block or procedure

PUSH <tag>	Push a copy of the descriptor <tag> onto the stack.
PROC <tag>	This is the same as PUSH except that the descriptor being stacked represents a procedure which is about to be called (using ENTER).
PUSHI <n>	Push a descriptor for the integer constant <n> onto the stack.
PUSHR <r>	Push a descriptor for the real (floating-point) constant <r> onto the stack.
PUSHS <s>	Push a descriptor for the string constant <s> onto the stack.
SELECT <tag>	TOS will be a descriptor for a record. Replace this descriptor with one describing the sub-element <tag> of this record.

Assignment

ASSVAL	Assign the value described by TOS to the variable described by SOS. Both TOS and SOS are popped from the stack.
ASSREF	Assign a reference to (the address of) the variable described by TOS to the pointer variable described by SOS. Both TOS and SOS are popped from the stack.
JAM	This is the same as ASSVAL except that the value being assigned will be truncated if necessary.
ASSPAR	Assign the actual parameter described by TOS to the formal parameter described by SOS. This is equivalent to either ASSVAL (for value parameters) or ASSREF (for reference parameters).
RESULT	TOS describes the result of the enclosing function. Following the processing of the result code must be generated to return from the function.

MAP	Similar to RESULT except that TOS describes the result of a MAP. Again a return must be generated.
DEFAULT <n>	
INIT <n>	Create N data items corresponding to the last descriptor defined, and given them all an initial (constant) value. The constant is popped from the stack in the case of INIT but DEFAULT causes the machine-dependent default value to be used (normally the UNASSIGNED pattern).

Binary operators

ADD	Addition
SUB	Subtraction
MUL	Multiplication
QUOT	Integer division
DIVIDE	Real division
IEXP	Integer exponentiation
REXP	Real exponentiation
AND	Logical AND
OR	Logical inclusive OR
XOR	Logical exclusive OR
LSH	Logical left shift
RSH	Logical right shift
CONC	String concatenate
ADDA	++
SUBA	--

The given operation is performed on TOS and SOS , both of which are removed from the stack, and the result (SOS op TOS) is pushed onto the stack.

e.g. A = B-C

```
PUSH   A
PUSH   B
PUSH   C
SUB
ASSVAL
```

Unary Operators

NEG	Negate (unary minus)
NOT	Logical NOT (complement)
MOD	Modulus (absolute value)

The given operation is performed on TOS.

Arrays

DIM <d> <n>	The stack will contain <d> pairs of descriptors corresponding to the lower and upper bounds for an array. This information is used to construct <n> arrays and any necessary accessing information for use through the last <n> descriptors to have been defined. All of these descriptors will be for similar arrays.
INDEX	SOS will be the descriptor for a multi-dimensional array and TOS will be the next non-terminal subscript. The stack is popped.
ACCESS	SOS will be the descriptor of an array and TOS will be the final/only subscript. Both descriptors are replaced by a descriptor for the appropriate element of the array. E.g. given arrays A(1:5) and B(1:4, 2:6), and integers J,K:

```
A(J) = 0        K = B(J, K)

PUSH   A        PUSH K
PUSH   J        PUSH B
ACCESS          PUSH J
PUSHC  0        INDEX
ASSVAL          PUSH K
                ACCESS
                ASSIGN
```

Internal labels

Internal labels are those labels in the intermediate code which have been created by the process of translating from the source program, and so do not appear explicitly in the source program. The main property of these labels is that they will only be referred to once. This fact can be used to re-use these labels, as, for example, a forward reference to a currently-defined label must cause its redefinition.

LOCATE <l>	define internal label <l>
GOTO <l>	forward jump to internal label <l>
REPEAT <l>	backward jump to internal label <l>

Conditional branches

These branches are always forward.

```
JUMPIF                 <cond> <label>

JUMPIFD                <cond> <label>

JUMPIFA                <cond> <label>

                          Where:   <cond> ::=  =, #,
                                               <, <=,
                                               >, >=,
                                               TRUE, FALSE
```

The two items on the top of the stack are compared and a jump is taken to <label> is the condition specified by <cond> is true. In the case of <cond> being TRUE or FALSE only one item is taken from the stack, and this represents a boolean value to be tested.

User Labels

LABEL <d>	locate label descriptor <d>
JUMP <d>	Jump to the label described by <d>
CALL <d>	Call the procedure described by <d>

Sundry Items

ON <e> <l>	Start of event trap for events <e>. Internal label <l> defines the end of the event block.
EVENT <e>	Signal event <e>
STOP	stop
MONITOR	monitor
RESOLVE <m>	Perform a string resolution
FOR	Start of a for loop
SLABEL <sd>	Define switch label
SJUMP <sd>	Select and jump to switch label
LINE <l>	Set the current line number to <l>

Appendix A2

The IMP77 Intermediate code

Internal representation

In production compilers the mnemonics used in the text are output in an abbreviated form, each mnemonic being translated into a single ASCII printing character.

!	OR	G	ALIAS	c	MCODE
"	JUMPIFD	H	BEGIN	d	DIM
#	BNE	I	unused	e	EVENT
$	DEF	J	JUMP	f	FOR
%	XOR	K	FALSE	g	unused
&	AND	L	LABEL	h	ALTBEG
'	PUSHS	M	MAP	i	INDEX
(	unused	N	PUSHI	j	JAM
)	unused	O	LINE	k	RELEASE
*	MUL	P	PLANT	l	LANG
+	ADD	Q	DIVIDE	m	MONITOR
-	SUB	R	RETURN	n	SELECT
.	CONCAT	S	ASSVAL	o	ON
/	QUOT	T	TRUE	p	ASSPAR
:	LOCATE	U	NEGATE	q	ALTEND
;	END	V	RESULT	r	RESOLVE
<	unused	W	SJUMP	s	STOP
=	unused	X	IEXP	t	unused
>	unused	Y	DEFAULT	u	ADDA
?	JUMPIF	Z	ASSREF	v	MOD
@	PUSH	[	LSH	w	SUBA
A	INIT	\	NOT	x	REXP
B	REPEAT	]	RSH	y	DIAG
C	JUMPIFA	^	PROC	z	CONTROL
D	PUSHR	_	SLABEL	{	START
E	CALL	a	ACCESS	|	ALT
F	GOTO	b	BOUNDS	}	FINISH

Appendix A3

Results from the INTERDATA 7/32 and PDP11

In these results the various test programs are referred to by the following codes:

PDP11	7/32	Program	
P11.1	732.1	TAKEON	The compiler's grammar processor
P11.2	732.2	EDWIN	A graphics package
P11.3	732.3	LAYOUT	A text formatting program
P11.4	732.4	ECCE	A text editor
P11.5	732.5	PILOT	A CAI interpreter
P11.6	732.6	TIMETAB	A schools' timetable generator
P11.7	732.7	DRAFT	A draughts program
P11.8	732.8	SQUARE	A least-squares fitting program
P11.9	732.9	GPM	A macro processor
P11.10	732.10	OS32MT	An operating system emulator
P11.11	732.11	HAL	A high-level assembler
P11.12	732.12	DIRECT	A file and directory handler

Remembering values in registers

		Code Size	Total Reduction	Incremental Reduction
P732.1	0 uses	9504	-	-
	1 use	8194	13.8%	13.8%
	2 uses	8192	13.8%	0.0%
P732.2	0 uses	6500	-	-
	1 use	6126	5.8%	5.8%
	2 uses	6126	5.8%	0.0%
P732.3	0 uses	10960	-	-
	1 use	9968	9.0%	9.0%
	2 uses	9956	9.2%	0.2%
P732.4	0 uses	5288	-	-
	1 use	4970	6.0%	6.0%
	2 uses	4958	6.2%	0.2%
P732.5	0 uses	5468	-	-
	1 use	4990	8.7%	8.7%
	2 uses	4986	8.8%	0.1%
P732.6	0 uses	3424	-	-
	1 use	3208	6.3%	6.3%
	2 uses	3208	6.3%	0.0%
P732.7	0 uses	10736	-	-
	1 use	9880	8.0%	8.0%
	2 uses	9874	8.0%	0.0%
P732.8	0 uses	824	-	-
	1 use	770	6.6%	6.6%
	2 uses	770	6.6%	0.0%
P732.9	0 uses	6448	-	-
	1 use	6148	4.6%	4.6%
	2 uses	6148	4.6%	0.0%
P732.10	0 uses	22968	-	-
	1 use	20656	10.1%	10.1%
	2 uses	20650	10.1%	0.0%
P732.11	0 uses	13996	-	-
	1 use	12470	10.9%	10.9%
	2 uses	12442	11.1%	0.2%
P732.12	0 uses	32600	-	-
	1 use	28532	12.5%	12.5%
	2 uses	28392	12.9%	0.4%

		Code Size	Total Reduction	Incremental Reduction
		----	---------	---------
P11.1	0 uses	9060	-	-
	1 use	7712	14.9%	14.9%
	2 uses	7660	15.4%	0.5%
P11.2	0 uses	6276	-	-
	1 use	6000	4.4%	4.4%
	2 uses	6000	4.4%	0.0%
P11.3	0 uses	9992	-	-
	1 use	9480	5.1%	5.1%
	2 uses	9444	5.5%	0.4%
P11.4	0 uses	5052	-	-
	1 use	4772	5.4%	5.4%
	2 uses	4768	5.6%	0.2%
P11.5	0 uses	5096	-	-
	1 use	4460	12.5%	12.5%
	2 uses	4452	12.6%	0.1%
P11.6	0 uses	3692	-	-
	1 use	3064	17.0%	17.0%
	2 uses	3064	17.0%	0.0%
P11.7	0 uses	7976	-	-
	1 use	7060	11.5%	11.5%
	2 uses	7032	11.8%	0.3%
P11.8	0 uses	668	-	-
	1 use	652	2.4%	2.4%
	2 uses	624	6.6%	4.2%
P11.9	0 uses	4888	-	-
	1 use	4492	8.1%	8.1%
	2 uses	4484	8.3%	0.2%
P11.10	0 uses	20318	-	-
	1 use	19120	5.9%	5.9%
	2 uses	19120	5.9%	0.0%
P11.11	0 uses	12938	-	-
	1 use	12162	6.0%	6.0%
	2 uses	12148	6.1%	0.1%
P11.12	0 uses	12068	-	-
	1 use	10594	12.2%	12.2%
	2 uses	10584	12.3%	0.0%

Remembering sets of registers (environments)

		Code Size	Total Reduction	Incremental Reduction
P732.1	0 environments	8556	-	-
	1 environment	8316	2.8%	2.8%
	2 environments	8238	3.7%	0.9%
	3 environments	8232	3.8%	0.1%
	4 environments	8222	3.9%	0.1%
	5 environments	8218	4.0%	0.1%
	6 environments	8192	4.2%	0.2%
P732.2	0 environments	6202	-	-
	1 environment	6128	1.2%	1.2%
	2 environments	6130	1.2%	0.0%
	3 environments	6126	1.2%	0.0%
	4 environments	6126	1.2%	0.0%
	5 environments	6126	1.2%	0.0%
	6 environments	6126	1.2%	0.0%
P732.3	0 environments	10174	-	-
	1 environment	10062	1.1%	1.1%
	2 environments	9968	2.0%	0.9%
	3 environments	9966	2.0%	0.0%
	4 environments	9964	2.1%	0.1%
	5 environments	9956	2.1%	0.1%
	6 environments	9956	2.1%	0.1%
P732.4	0 environments	5068	-	-
	1 environment	4978	1.8%	1.8%
	2 environments	4958	2.2%	0.4%
	3 environments	4958	2.2%	0.0%
	4 environments	4958	2.2%	0.0%
	5 environments	4958	2.2%	0.0%
	6 environments	4958	2.2%	0.0%
P732.6	0 environments	3262	-	-
	1 environment	3250	0.4%	0.4%
	2 environments	3216	1.4%	1.0%
	3 environments	3208	1.7%	0.3%
	4 environments	3208	1.7%	0.0%
	5 environments	3208	1.7%	0.0%
	6 environments	3208	1.7%	0.0%
P732.7	0 environments	10062	-	-
	1 environment	9970	0.9%	0.9%
	2 environments	9894	1.7%	0.8%
	3 environments	9880	1.8%	0.1%
	4 environments	9874	1.9%	0.1%
	5 environments	9874	1.9%	0.0%
	6 environments	9874	1.9%	0.0%

P732.8	0 environments	806	-	-
	1 environment	782	3.0%	3.0%
	2 environments	782	3.0%	0.0%
	3 environments	770	4.5%	1.5%
	4 environments	770	4.5%	0.0%
	5 environments	770	4.5%	0.0%
	6 environments	770	4.5%	0.0%
P732.9	0 environments	6244	-	-
	1 environment	6202	0.7%	0.7%
	2 environments	6156	1.4%	0.7%
	3 environments	6158	1.4%	0.0%
	4 environments	6148	1.5%	0.1%
	5 environments	6148	1.5%	0.0%
	6 environments	6148	1.5%	0.0%
P732.10	0 environments	21214	-	-
	1 environment	20928	1.3%	1.3%
	2 environments	20748	2.2%	0.9%
	3 environments	20678	2.5%	0.3%
	4 environments	20678	2.5%	0.0%
	5 environments	20668	2.6%	0.1%
	6 environments	20650	2.6%	0.0%
P732.11	0 environments	12772	-	-
	1 environment	12592	1.4%	1.4%
	2 environments	12486	2.2%	0.8%
	3 environments	12472	2.3%	0.1%
	4 environments	12460	2.4%	0.1%
	5 environments	12452	2.5%	0.1%
	6 environments	12442	2.6%	0.1%
P732.12	0 environments	11522	-	-
	1 environment	11418	0.9%	0.9%
	2 environments	11342	1.6%	0.7%
	3 environments	11314	1.8%	0.2%
	4 environments	11314	1.8%	0.0%
	5 environments	11296	2.0%	0.2%
	6 environments	11296	2.0%	0.0%
P11.1	0 environments	7686	-	-
	1 environment	7670	0.2%	0.2%
	2 environments	7660	0.3%	0.1%
	3 environments	7660	0.3%	0.0%
	4 environments	7660	0.3%	0.0%
	5 environments	7660	0.3%	0.0%
	6 environments	7660	0.3%	0.0%
P11.2	0 environments	6012	-	-
	1 environment	6000	0.2%	0.2%
	2 environments	6000	0.2%	0.0%
	3 environments	6000	0.2%	0.0%
	4 environments	6000	0.2%	0.0%
	5 environments	6000	0.2%	0.0%
	6 environments	6000	0.2%	0.0%

P11.3	0 environments	9472	-	-
	1 environment	9440	0.3%	0.3%
	2 environments	9444	0.3%	-0.0%
	3 environments	9444	0.3%	0.0%
	4 environments	9444	0.3%	0.0%
	5 environments	9444	0.3%	0.0%
	6 environments	9444	0.3%	0.0%
P11.4	0 environments	4784	0.2%	0.2%
	1 environment	4776	0.2%	0.0%
	2 environments	4776	0.2%	0.0%
	3 environments	4776	0.2%	0.0%
	4 environments	4776	0.2%	0.0%
	5 environments	4772	0.2%	0.0%
	6 environments	4768	0.3%	0.1%
P11.5	0 environments	4512	-	-
	1 environment	4464	1.1%	1.1%
	2 environments	4456	1.2%	0.1%
	3 environments	4452	1.3%	0.1%
	4 environments	4452	1.3%	0.0%
	5 environments	4452	1.3%	0.0%
	6 environments	4452	1.3%	0.0%
P11.6	0 environments	3076	-	-
	1 environment	3070	0.2%	0.2%
	2 environments	3064	0.4%	0.2%
	3 environments	3064	0.4%	0.0%
	4 environments	3064	0.4%	0.0%
	5 environments	3064	0.4%	0.0%
	6 environments	3064	0.4%	0.0%
P11.7	0 environments	7104	-	-
	1 environment	7048	0.8%	0.8%
	2 environments	7048	0.8%	0.0%
	3 environments	7048	0.8%	0.0%
	4 environments	7048	0.8%	0.0%
	5 environments	7048	0.8%	0.0%
	6 environments	7032	1.0%	0.2%
P11.8	0 environments	640	-	-
	1 environment	624	2.5%	2.5%
	2 environments	624	2.5%	0.0%
	3 environments	624	2.5%	0.0%
	4 environments	624	2.5%	0.0%
	5 environments	624	2.5%	0.0%
	6 environments	624	2.5%	0.0%
P11.9	0 environments	4492	-	-
	1 environment	4484	0.2%	0.2%
	2 environments	4484	0.2%	0.0%
	3 environments	4484	0.2%	0.0%
	4 environments	4484	0.2%	0.0%
	5 environments	4484	0.2%	0.0%
	6 environments	4484	0.2%	0.0%

P11.10	0 environments	19332	-	-
	1 environment	19196	0.7%	0.7%
	2 environments	19158	0.9%	0.2%
	3 environments	19138	1.0%	0.1%
	4 environments	19138	1.0%	0.0%
	5 environments	19120	1.1%	0.1%
	6 environments	19120	1.1%	0.0%
P11.11	0 environments	12280	-	-
	1 environment	12200	0.6%	0.6%
	2 environments	12168	0.9%	0.3%
	3 environments	12160	1.0%	0.1%
	4 environments	12156	1.0%	0.0%
	5 environments	12148	1.1%	0.1%
	6 environments	12148	1.1%	0.0%
P11.12	0 environments	10690	-	-
	1 environment	10616	0.7%	0.7%
	2 environments	10604	0.8%	0.1%
	3 environments	10604	0.8%	0.0%
	4 environments	10594	0.9%	0.1%
	5 environments	10584	1.0%	0.1%
	6 environments	10584	1.0%	0.0%

Simple allocation of arrays and remembering subscripts

	Neither	Allocation Simple	(gain)	Remembering Subscripts	(gain)
P732.1	8596	8476	(1.4%)	8312	(3.3%)
P732.2	6126	6126	(0.0%)	6126	(0.0%)
P732.3	10450	10114	(3.2%)	10426	(0.2%)
P732.4	5056	4958	(1.9%)	5056	(0.0%)
P732.5	5306	5054	(4.7%)	5308	-(0.0%)
P732.6	3384	3254	(3.8%)	3386	-(0.0%)
P732.7	10346	10112	(2.3%)	10344	(0.0%)
P732.8	806	806	(0.0%)	770	(4.5%)
P732.9	6138	6138	(0.0%)	6148	-(0.2%)
P732.10	20806	20684	(0.6%)	20776	(0.1%)
P732.11	12442	12442	(0.0%)	12442	(0.0%)
P732.12	11976	11946	(0.2%)	11326	(5.4%)

	Both optimisations	Total gain
P732.1	8192	4.7%
P732.2	6126	0.0%
P732.3	9956	4.7%
P732.4	4958	1.9%
P732.5	4986	6.0%
P732.6	3208	5.2%
P732.7	9874	4.6%
P732.8	770	4.5%
P732.9	6148	-0.2%
P732.10	20650	0.7%
P732.11	12442	0.0%
P732.12	11296	5.8%

	Neither	Allocation Simple	(gain)	Remembering Subscripts	(gain)
P11.1	8572	8188	(4.5%)	7704	(10.1%)
P11.2	6000	6000	(0.0%)	6000	(0.0%)
P11.3	9764	9556	(2.1%)	9644	(1.2%)
P11.4	4848	4776	(1.5%)	4848	(0.0%)
P11.5	4656	4568	(1.9%)	4452	(4.4%)
P11.6	3356	3202	(4.6%)	3218	(4.1%)
P11.7	7844	7728	(1.4%)	7204	(8.2%)
P11.8	644	624	(3.1%)	644	(0.0%)
P11.9	4796	4796	(0.0%)	4484	(6.5%)
P11.10	19236	19140	(0.5%)	19216	(0.1%)
P11.11	12148	12148	(0.0%)	12148	(0.0%)
P11.12	11094	11060	(0.3%)	10616	(4.3%)

	Both optimisations	Total gain
P11.1	7660	10.6%
P11.2	6000	0.0%
P11.3	9444	3.3%
P11.4	4768	1.6%
P11.5	4452	4.4%
P11.6	3064	8.7%
P11.7	7032	10.4%
P11.8	624	3.1%
P11.9	4484	6.5%
P11.10	19120	0.6%
P11.11	12148	0.0%
P11.12	10584	4.6%

Simplifying: X = X op Y

	Code Without	Code With	Gain
P732.1	8292	8192	1.2%
P732.2	6156	6126	0.5%
P732.3	10068	9956	1.1%
P732.4	5088	4958	2.6%
P732.5	5180	4986	3.7%
P732.6	3368	3208	4.8%
P732.7	11438	11296	1.2%
P732.8	772	770	0.2%
P732.9	6214	6148	1.1%
P732.10	21086	20650	2.1%
P732.11	12590	12442	1.2%
P732.12	11438	11296	1.2%
P11.1	8284	7660	7.5%
P11.2	6220	6000	3.5%
P11.3	10040	9444	5.9%
P11.4	5136	4768	7.2%
P11.5	4800	4452	7.2%
P11.6	3342	3064	8.3%
P11.7	7596	7032	7.4%
P11.8	668	624	6.6%
P11.9	4724	4484	5.1%
P11.10	20634	19128	7.3%
P11.11	12892	12148	5.8%
P11.12	11492	10584	7.9%

Passing some parameters in registers

		Code Size	Total Reduction	Incremental Reduction
P732.1	0 registers	8862	-	-
	1 register	8360	5.7%	5.7%
	2 registers	8192	7.6%	1.9%
P732.2	0 registers	7196	-	-
	1 register	6544	9.1%	9.1%
	2 registers	6126	14.9%	5.8%
P732.3	0 registers	10586	-	-
	1 register	9976	5.8%	5.8%
	2 registers	9956	6.0%	0.2%
P732.4	0 registers	5126	-	-
	1 register	4958	3.3%	3.3%
	2 registers	4958	3.3%	0.0%
P732.5	0 registers	5198	-	-
	1 register	5022	3.4%	3.5%
	2 registers	4986	4.1%	0.7%
P732.6	0 registers	3402	-	-
	1 register	3222	5.3%	5.3%
	2 registers	3208	5.7%	0.4%
P732.7	0 registers	10400	-	-
	1 register	10048	3.4%	3.4%
	2 registers	9874	5.0%	1.6%
P732.8	0 registers	840	-	-
	1 register	810	3.6%	3.6%
	2 registers	770	8.3%	4.7%
P732.9	0 registers	6404	-	-
	1 register	6172	3.6%	3.6%
	2 registers	6148	4.0%	0.4%
P732.10	0 registers	21650	-	-
	1 register	20826	3.8%	3.8%
	2 registers	20650	4.6%	0.8%
P732.11	0 registers	13476	-	-
	1 register	12442	7.7%	7.7%
	2 registers	12442	7.7%	0.0%
P732.12	0 registers	11916	-	-
	1 register	11452	3.9%	3.9%
	2 registers	11296	5.2%	1.3%

		Code Size	Total Reduction	Incremental Reduction
P11.1	0 registers	7796	-	-
	1 register	7756	0.5%	0.5%
	2 registers	7660	1.7%	1.2%
P11.2	0 registers	6192	-	-
	1 register	6072	1.9%	1.9%
	2 registers	6000	3.1%	1.2%
P11.3	0 registers	9564	-	-
	1 register	9448	1.2%	1.2%
	2 registers	9444	1.2%	0.0%
P11.4	0 registers	4776	-	-
	1 register	4768	0.2%	0.2%
	2 registers	4768	0.2%	0.0%
P11.5	0 registers	4508	-	-
	1 register	4452	1.2%	1.2%
	2 registers	4452	1.2%	0.0%
P11.6	0 registers	3098	-	-
	1 register	3064	1.1%	1.1%
	2 registers	3064	1.1%	0.0%
P11.7	0 registers	7124	-	-
	1 register	7096	0.4%	0.4%
	2 registers	7032	1.3%	0.9%
P11.8	0 registers	624	-	-
	1 register	624	0.0%	0.0%
	2 registers	624	0.0%	0.0%
P11.9	0 registers	4520	-	-
	1 register	4488	0.7%	0.7%
	2 registers	4484	0.8%	0.1%
P11.10	0 registers	19302	-	-
	1 register	19166	0.7%	0.7%
	2 registers	19128	0.9%	0.2%
P11.11	0 registers	12364	-	-
	1 register	12152	1.7%	1.7%
	2 registers	12148	1.7%	0.0%
P11.12	0 registers	10734	-	-
	1 register	10648	0.8%	0.8%
	2 registers	10584	1.4%	0.6%

Remembering condition-codes

	Unknown	Remembered	Gain
P732.1	8820	8192	0.3%
P732.2	6134	6126	0.1%
P732.3	9976	9956	0.2%
P732.4	4968	4958	0.2%
P732.5	4988	4986	0.0%
P732.6	3212	3208	0.1%
P732.7	9880	9874	0.1%
P732.8	770	770	0.0%
P732.9	6150	6148	0.0%
P732.10	20684	20650	0.2%
P732.11	12474	12442	0.2%
P732.12	11318	11296	0.2%
P11.1	7732	7660	0.9%
P11.2	6012	6000	0.3%
P11.3	9516	9444	0.8%
P11.4	4792	4768	0.5%
P11.5	4452	4452	0.0%
P11.6	3076	3064	0.4%
P11.7	7064	7032	0.4%
P11.8	624	624	0.0%
P11.9	4496	4484	0.3%
P11.10	19204	19128	0.4%
P11.11	12192	12148	0.4%
P11.12	10626	10584	0.4%

__Forward merging and delaying__

	Neither	Forward Merge	Delaying	Merge & Delay
P732.1	8192	8172 (0.2%)	8160 (0.4%)	8136 (0.7%)
P732.2	6126	6110 (0.3%)	6054 (1.2%)	6044 (1.3%)
P732.3	9956	9948 (0.2%)	9872 (0.8%)	9864 (0.9%)
P732.4	4958	4950 (0.2%)	4942 (0.3%)	4942 (0.3%)
P732.5	4986	4970 (0.3%)	4982 (0.1%)	4966 (0.4%)
P732.6	3208	3194 (0.4%)	3140 (2.1%)	3122 (2.7%)
P732.7	9874	9752 (1.2%)	9834 (0.4%)	9812 (1.6%)
P732.8	770	764 (0.5%)	738 (4.2%)	728 (5.4%)
P732.9	6148	6136 (0.2%)	6132 (0.3%)	6120 (0.4%)
P732.10	20650	20586 (0.3%)	20558 (0.4%)	20490 (0.8%)
P732.11	12442	12406 (0.3%)	12342 (0.8%)	12306 (1.1%)
P732.12	11296	11280 (0.1%)	11272 (0.2%)	11256 (0.4%)
P11.1	7660	7660 (0.0%)	7660 (0.0%)	7660 (0.0%)
P11.2	6000	6000 (0.0%)	5988 (0.2%)	5988 (0.2%)
P11.3	9444	9444 (0.0%)	9434 (0.1%)	9434 (0.1%)
P11.4	4768	4768 (0.0%)	4768 (0.0%)	4768 (0.0%)
P11.5	4452	4448 (0.1%)	4452 (0.0%)	4448 (0.1%)
P11.6	3064	3064 (0.0%)	3060 (0.1%)	3060 (0.1%)
P11.7	7032	7004 (0.4%)	7032 (0.0%)	7004 (0.4%)
P11.8	624	620 (0.6%)	620 (0.6%)	616 (1.3%)
P11.9	4484	4484 (0.0%)	4484 (0.0%)	4484 (0.0%)
P11.10	19128	19128 (0.0%)	19128 (0.0%)	19128 (0.0%)
P11.11	12148	12136 (0.1%)	12136 (0.1%)	12124 (0.2%)
P11.12	10584	10584 (0.0%)	10584 (0.0%)	10584 (0.0%)

All optimisations

	None	All	Gain
P732.1	11300	8136	28.0%
P732.2	7520	6044	19.6%
P732.3	12286	9864	19.7%
P732.4	5782	4942	14.5%
P732.5	6204	4966	19.9%
P732.6	4004	3122	22.0%
P732.7	11750	9812	16.5%
P732.8	988	728	26.3%
P732.9	6848	6120	10.6%
P732.10	24722	20490	17.1%
P732.11	14618	12306	15.8%
P732.12	14064	11256	20.0%
P11.1	9664	7660	20.7%
P11.2	6588	5988	9.1%
P11.3	11092	9434	14.9%
P11.4	5540	4768	13.9%
P11.5	5572	4448	20.2%
P11.6	3666	3060	16.5%
P11.7	8632	7004	18.7%
P11.8	752	616	18.1%
P11.9	5256	4484	14.7%
P11.10	23940	19128	20.1%
P11.11	14328	12124	15.4%
P11.12	12816	10584	17.8%

CPU time in input/output
(as percentages of compile time)

	Phase 1		Phase 2		Phase 3	
	Total CPU	% I/O	Total CPU	% I/O	Total CPU	% I/O
All optimisations:						
P732.1:	53%	8%	32%	6%	14%	5%
P732.2:	53%	8%	35%	8%	12%	5%
P732.3:	51%	7%	34%	7%	15%	5%
P732.4:	54%	8%	32%	6%	14%	5%
P732.5:	49%	12%	36%	5%	14%	5%
P732.6:	50%	7%	37%	7%	12%	6%
P732.7:	48%	7%	39%	7%	13%	7%
P732.8:	54%	7%	36%	7%	10%	5%
P732.9:	50%	8%	36%	8%	14%	6%
P732.10:	54%	6%	28%	5%	17%	4%
P732.11:	52%	7%	32%	6%	16%	5%
P732.12:	52%	8%	32%	8%	17%	7%
No optimisation:						
P732.1	50%	7%	30%	7%	19%	8%
P732.2	55%	8%	31%	9%	14%	8%
P732.3	54%	8%	28%	8%	18%	6%
P732.4	57%	8%	26%	7%	16%	6%
P732.5	52%	8%	30%	8%	17%	7%
P732.6	53%	8%	32%	9%	15%	8%
P732.7	49%	7%	31%	8%	19%	8%
P732.8	56%	7%	32%	9%	12%	6%
P732.9	55%	9%	29%	8%	16%	7%
P732.10	55%	6%	23%	6%	21%	5%
P732.11	55%	8%	26%	6%	20%	6%
P732.12	54%	8%	27%	8%	19%	8%

Overall CPU in input/output

Internal I/O = communication between phases.
External I/O = source input & object file output.

	Internal I/O		External I/O	
	No Opts.	All Opts.	No Opts.	All Opts.
P732.1	16%	13%	7%	7%
P732.2	16%	16%	8%	7%
P732.3	15%	13%	7%	6%
P732.4	14%	13%	7%	7%
P732.5	16%	12%	7%	6%
P732.6	17%	14%	8%	7%
P732.7	17%	15%	6%	6%
P732.8	18%	16%	5%	4%
P732.9	16%	15%	8%	7%
P732.10	13%	11%	6%	6%
P732.11	12%	11%	6%	6%
P732.12	16%	15%	8%	7%

References

Aho, A.V. & Ullman, J.P. (1972)

"Optimisation of straight line programs".

SIAM J. March 1972.

Allen, F.E. & Cocke, J.A. (1971)

"A catalogue of optimising techniques".

Design and optimisation of compilers

Prentice-Hall, R. Rustin (ed). 1971.

Basili, V.R. & Turner, A.J. (1975)

"A transportable extendable compiler".

Software - Practice and Experience,

Vol 5, 1975.

Bell, G. (1973)

"Threaded Code".

CACM Vol 16, No 6, 1973.

Berry, R.E. (1978)

"Experience with the PASCAL P-compiler".

Software - Practice and Experience

Vol 8, No 5, 1978.

Berthaud, M. & Griffiths, M. (1973)
"Incremental compilation and conversational interpretation".
Annual review in automatic programming,
Vol 7, Part 2, 1973.

Bourne, S.R., Birrell, A.D. & Walker, I.W. (1975)
"Z code, an intermediate object code for ALGOL68".
The Computer Laboratory, Cambridge, 1975.

Branquart, P., Cardinael, I. & Lewi, J. (1973)
"Optimised translation process, application to ALGOL68".
Int. Comp. Symp. 1973.

Bron, C. & De Vries, W. (1976)
"A PASCAL compiler for PDP11 minicomputers".
Software - Practice and Experience
Vol 6, 1976.

Brooker, R.A. et al.

"Experience with the compiler-compiler".

Comp. J. Vol 9, 1967.

Brown, P. (1972)

"Levels of language for portable software".

CACM. 15, 1972.

Brown, P. (1977)

"Macro processors".

Software Portability (ed. P. Brown).

Cambridge University Press, 1977.

Buckle, J.K. (1978)

"The ICL 2900 Series".

Macmillan Press Ltd., 1978.

Cocke, J. (1970)

"Global common subexpression elimination".

ACM SIGPLAN notices, Vol 5, No 7. 1970.

Cocke, J. & Schwartz, J.T. (1970)
"Programming languages and their compilers".
Courant Institute of Mathematical Sciences,
New York University, 1970.

Coleman, S.S., Poole, P.C. & Waite, W.M. (1974)
"The Mobile programming system, JANUS".
Software - Practice and Experience.
Vol 4, 1974.

Dewar, H. (1975)
"The ISYS system".
Internal memo, Department of Computer
Science, University of Edinburgh, 1975.

Feldman, J.A. (1966)
"A formal semantics for computer languages
and its application in a compiler-compiler".
CACM 9, 1966.

Feldman, J.A. & Gries, D. (1968)
"Translator writing systems".
CACM 11, 1968.

Gardner, P. (1977)

"An ALGOL68C bootstrap".

Memo CSM-20, University of Essex, 1977.

Gilmore, B.A.C. (1980)

"DEIMOS User's Manual".

Edinburgh Regional Computing Centre.

Grosse-Lindemann, C.O. & Nageli, H.H. (1976)

"Postlude to a PASCAL compiler bootstrap on the DEC system 10".

Software - Practice and Experience,

Vol 6, 1976.

Haddon, B.K. & Waite, W.M. (1978)

"Experience with the Universal Intermediate language JANUS".

Software - Practice and Experience

Vol 8, No 5, 1978.

Halpern, M.I. (1965)

"Machine Independence: its technology and economics".

CACM Vol 8 No 12, 1965.

Hawkins, E.N. & Huxtable, D.H.R. (1963)

"A multi-pass translation scheme for Algol 60".

Annual review in automatic programming, No 3, 1963.

Hecht, M.S. & Ullman, J.D. (1975)

"A simple algorithm for global data flow analysis problems".

SIAM J. Vol 4, 1975.

Huxtable, D.H.R. (1964)

"On writing an optimising translator for Algol 60".

Introduction to System Programming,

P. Wegner (ed.),

Academic Press, 1964.

Interdata (1974)

"Common Assembler Language (CAL) User's Manual".

Interdata publication number 29-375R04. 1974.

Jensen, K. & Wirth, N. (1976)

"PASCAL User Manual and Report".

Springer-Verlag, 1976.

Kildall, G.A. (1973)

"A unified approach to global program optimisation".

CACM Principles of programming languages, 1973.

Klint, P. (1979)

"Line numbers made cheap".

CACM Vol 22, No. 10. 1979.

Knuth, D.E. (1962)

"History of writing compilers".

Proc. ACM 17, 1962.

Knuth, D.E. (1971)

"An empirical study of FORTRAN programs".

Software - Practice and Experience,

Vol 1, 1971.

Knuth, D.E. (1974)

"Structured programming with GOTO statements".

ACM Computing Surveys 6, No 4, 1974.

Krogdahl, S., Myhre, O., Robertson, P.S., & Syrrist, G. (1980)

The SCALA project: S-PORT.

Norwegian Computing Centre working paper.

Lecarme, O. & Peyrolle-Thomas, M. (1978)

"Self-compiling compilers: An appraisal of their implementation and portability".

Software - Practice and Experience

Vol 8, No 2, 1978.

Lowry, E.S. & Medlock, C.W. (1969)

"Object code optimisation".

CACM Vol 12, 1969.

McKeeman, W.M. (1965)

"Peephole Optimisation".

CACM Vol 8, 1965.

Mock, O., Olsztyn, J. et al (1958)

"The problem of programming communication with changing machines".

CACM Vol 1, No 8, 1958.

Nelson, P.A. (1979)

"A comparison of PASCAL intermediate languages".

ACM SIGPLAN Notices, Vol 14, No 8, 1979.

Nori, K.V., Ammann, U. et al (1974 & 1976)

"The PASCAL <P>-compiler: Implementation notes".

Eidgenossische Technische Hochschule, Zurich, 1976.

Pavelin, C.J. (1970)

"The improvement of program behaviour in paged computer systems".

Ph.D. Thesis, University of Edinburgh, 1970.

Poole, P. (1974)

"Portable and adaptable compilers".

Advanced course on compiler construction,

Springer-Verlag 1974.

Richards, M. (1971)

"The portability of the BCPL compiler".

Software - Practice and Experience,

Vol 1, 1971.

Richards, M. (1977)

"Portable Compilers".

Software Portability (ed. P. Brown),

Cambridge University Press, 1977.

Robertson, P.S. (1977)

"The IMP-77 Language".

Internal Report CSR-19-77, Department of Computer Science, University of Edinburgh, 1977.

Russell, W. (1974)

"A translator for PASCAL P-code".

Final year project report, Department of Computer Science, University of Edinburgh, 1974.

Satterthwaite, E. (1972)

"Debugging tools for high-level languages".

Software - Practice and Experience,

Vol 2, 1972.

Schneck, P.B. & Angel, E. (1973)

"A FORTRAN to FORTRAN optimising compiler".

Computer Journal Vol 16, 1973.

Sibley, R.A. (1961)

"The SLANG System".

CACM Vol 4, 1961.

Spier, M.J. (1976)

"Software Malpractice - A distasteful experience".

Software - Practice and Experience,

Vol 6, 1976.

Steel, T.B. (1961)

"A first version of UNCOL".

Proc. AFIPS WJCC 19, 1961.

Stephens, P.D. (1974)

"The IMP language and compiler".

Computer Journal Vol 17, 1974.

Stockton-Gaines, R. (1965)

"On the translation of machine language programs".

CACM Vol 8, No 12, 1965.

Szymanski, T.G.

"Assembling code for machines with span-dependent instructions".

CACM 21, 1978.

Tanenbaum, A.S. (1976)

"Structured Computer Organisation".

Prentice/Hall International, 1976.

Thompson, K. & Ritchie, D.M. (1974)

"The UNIX timesharing system".

CACM Vol 17, No 7, 1974.

Trout, R.G. (1967)

"A compiler-compiler system".

Proc. ACM, 1967.

Waite, W.M. (1970)

"The Mobile Programming System: STAGE 2".

CACM, vol 13, 1970.

Welsh, J. & Quinn, C. (1972)

"A PASCAL compiler for the ICL 1900 series".

Software - Practice and Experience,

Vol 2, 1972.

Whitfield, H. & Wight, A.S. (1973)

"EMAS - The Edinburgh Multi-Access System".

Computer Journal Vol 16, 1973.

Wichmann, B.A. (1977)
"Optimisation".
Software Portability (ed. P. Brown),
Cambridge University Press, 1977.

Wichmann, B.A. (1977)
"How to call procedures, or second thoughts
on Ackermann's function".
Software - Practice and Experience
Vol 7, No 3, 1977.

Williams, M.H.
"Long/short address optimisation in
assemblers".
Software - Practice and Experience,
Vol 9 No 3, 1978

Wulf, W., Johnsson, R.K., Weinstock, C.B., Hobbs, S.O.
& Geschke, C.M. (1975)
"The design of an optimising compiler".
Elsevier Computer Science Library, 1975.