FORTRAN PROGRAM OPTIMIZATION

Michael Metcalf

Lectures given in the
Academic Training Programme of CERN
1979–1980

GENEVA
1980
Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

© Copyright CERN, Genève, 1980

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use. If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.
ABSTRACT

Techniques are outlined which indicate how FORTRAN programs may be optimized, particularly on large CDC and IBM systems, whose compilers and CPU's are briefly described.
CONTENTS

1. INTRODUCTION ................................................................................. 1

2. WHAT DOES A COMPILER DO ? .................................................. 2
   The basic task of a compiler ......................................................... 2
   The additional tasks of the FORTRAN H compiler ..................... 2
   The case of the CDC compiler ..................................................... 3

3. WHAT TO OPTIMIZE ? ................................................................. 4
   Algorithm selection ....................................................................... 4
   Optimization versus Portability .................................................... 4
   Optimization versus Clarity .......................................................... 5
      Clear code helps the compiler to optimize ............................... 5
      Clear code helps the programmer to optimize ......................... 5
      Effect of Optimization on Clarity .............................................. 6
      The identification of the most used parts of the code ............... 7

4. GENERAL HINTS FOR OPTIMIZATION ......................................... 9
   Compiler options .......................................................................... 9
   Initialization of variables ............................................................ 9
   Arithmetic operations ............................................................... 10
   Mixed data types ......................................................................... 10
   Repeated calculations ................................................................ 11
   DO-loops ..................................................................................... 11
      Avoidance of loops ................................................................ 12
      Loop organization .................................................................. 12
      Subscripts .............................................................................. 13
      Invariant code ....................................................................... 14
      Loop manipulations ................................................................ 15
      Use of scalar accumulators ..................................................... 16
   Branches ..................................................................................... 17
   Calling sequences ...................................................................... 17
   EQUIVALENCE statement .......................................................... 18
   Input/Output operations ............................................................. 18

5. SPECIFIC HINTS WITH THE CDC FTN COMPILER ..................... 19
   Object code optimization ............................................................ 19
   Examination of DO-loops ............................................................ 20
   Program size and structure ....................................................... 20
   Data Interference ....................................................................... 22
   Basic external functions ............................................................ 23
   Basic intrinsic functions ............................................................ 23
6. THE IBM FORTRAN (H EXTENDED) OPTIMIZATION ENHANCEMENT COMPILER . 24

The optimization produced by the compiler ..... 24
Commuation of subscript dimension computations ..... 24
Elimination of duplicate computations ..... 24
Backward movement of invariant computations ..... 24
Optimization of subscript component combinations ..... 24
Strength reduction of induction variable computations ..... 25
Local register optimization ..... 25
Global register optimization ..... 25
Global register remapping ..... 25
Global register purging ..... 25
Global register scavenging ..... 25
Section-oriented branch optimization ..... 26
Program size and structure ..... 26
I/O operations ..... 27
Special data types ..... 27
EQUIVALENCE statements ..... 28
Array subscripts ..... 28
Logical IF-statements ..... 29
Use of storage ..... 29
Interleaving ..... 29
The cache memory ..... 29
Virtual Storage and Paging ..... 30

7. A BRIEF LOOK AT THE CRAY-1 VECTOR PROCESSOR ..... 31

The CPU ..... 31
Vectorization ..... 31

8. ACKNOWLEDGEMENTS ..... 34

9. SOURCES ..... 35

10. FIGURES ..... 36
1. INTRODUCTION

Although the unit costs associated with computer hardware have continuously dropped over the past two decades, and the costs of producing software have steadily risen, it is nevertheless necessary, in a context such as CERN's, where the limited capacity is sometimes saturated by a relatively small number of programs, to examine the efficiency of the large physics codes which between them use 80% of the machine time.

The aim of this report is to describe techniques by which it is possible to write efficient FORTRAN code from the outset, and to improve upon the efficiency of existing programs. Efficiency here means execution-time use of the CPU, and to a lesser extent real-time efficiency, even though these are obtained at the expense of the compile-time speed which, however, represents only a small fraction of CERN's computer usage. Unfortunately, optimization is often more akin to an art than a science, involving not only knowledge of an individual processor and its compiler, but also the acquisition of a feeling for the effect of different modifications to a section of code. We are helped considerably, however, by the timing tools available to us, and these play an important role in the identification of the problem areas and in the evaluation of the detailed changes brought about by attempts to modify a program. One of the early lessons we learn is that often a change to the code is counter-productive, and attempts to perform 'hand' optimization can lead to a lessening of the ability of the compiler to optimize. The path to an optimized program is, then, often fraught with difficulties and studded with disappointments, but if this document helps in the removal of some of the major inefficiencies in our programs, then it will have served its purpose of helping to obtain more useful units of computing from our fixed capacity. It must be stressed that it is intended principally as a pragmatic review for the physicist-programmer, and not as a treatise in computer science.

The examples and hints which follow are related principally either to large CDC series computers with their FTN compiler, or to large IBM 370 and 303X machines and their FORTRAN IV (HE Extended) Optimization Enhancement compiler, or to both, and are not necessarily relevant on other systems than these. It must also be stressed that some of the points mentioned here reflect recent changes in compilers, and would not have been valid a few years ago.
2. WHAT DOES A COMPILER DO?

2.1 THE BASIC TASK OF A COMPILER

Any discussion of code optimization must have as its basis a knowledge, however slight, of what a compiler does and within what framework it has to operate. The question as to how a compiler performs its task is a different and much more complex one, but an idea can be gained from Fig. 1, which shows a block diagram of the IBM FORTRAN H compiler. If we consider just the highest optimization level path, it may be seen that the program begins by initialising itself and then reads in the source code, generating an intermediate text for internal use. The syntax of this text is checked and any error messages generated. On the path OPT=3, the compiler breaks the code down into smaller blocks and optimizes the text of each block (see also Section 5); it then assigns registers for the variables and addresses generated by the source code, and attempts to make optimum use of these registers. Within each section of code it attempts to optimize the way in which branch instructions are generated, before finally producing the object code in the form of an object module.

The object module consists of all the information necessary for a linkage editor or loader to join the module to others in a form ready for execution in the same or in a subsequent job or job step. At load- or execution-time, depending on the system, the relative addresses contained in the modules are changed to absolute addresses in memory. This may be seen in Figs. 2 and 3, illustrating CDC and IBM object modules, where each contains a section referring to 'address substitution' or 'relocation dictionary' holding the relevant information to perform this operation.

2.2 THE ADDITIONAL TASKS OF THE FORTRAN H COMPILER

For our purposes the important part of the object code is that labelled 'machine instructions' or 'executable statements'. These consist mainly of a series of binary codes which define the operations to be performed on the contents of registers, the retrieval and storage of data words in memory, and branch instructions. If we examine the basic structure of the IBM processor in Fig. 4 we see immediately some of the constraints under which the compiler has to operate.

The 16 general registers are used for 24-bit base addresses, as index registers and as accumulators in integer arithmetic and logical operations. Of these 16, only 12 at the most are available to the compiler for general use, as four have predetermined functions associated with the calling sequence between two sub-programs. The four floating-point registers can store four 32- or 64-bit floating-point numbers, and so it becomes clear that a major task of an optimizing compiler is to arrange the flow of operations so as to keep as many as possible of the most used operands in the small number of registers, thereby minimizing the number of time consuming references to main storage.
2.3 THE CASE OF THE CDC COMPILER

The CDC 7600 and larger CYBER models present an even more complex challenge. Fig. 5 shows not only three sets of eight registers, for indices, addresses and operands, but also the nine separate functional units, which are able to process independent operands simultaneously (parallel processing). Thus, an instruction to multiply the contents of two operand registers can be issued to the floating multiply unit one 27.5 nsec. clock cycle after an instruction to process two other unrelated operands has been issued to, say, the divide unit. Apart from the divide unit, the micro-operations in the functional units are segmented in such a way that they can accept new independent operands one or, in the case of the multiply unit, two clock cycles after accepting the previous instruction. This ability of a functional unit to accept a stream of independent instructions at a rate greater than its single operation speed is known as pipelining. It is then evident that to exploit the power of the CPU to the fullest possible extent, not only must the use of registers be optimized but instructions must be scheduled in such a way that the parallel processing and pipelining facilities are used in an optimal manner.

The CDC systems have another interesting hardware feature known as an instruction stack, which holds 12 words of instructions, contained two, three or four to a word. A pointer points to the current instruction word, and when it reaches the second but last word the words in the stack are shifted up by one position, and a new word of instructions is brought from memory into the bottom position in the stack. If one imagines the operation of a DO-loop of length ten words or less, it is evident that after one trip through the loop its instructions will be contained completely in the stack, and no further memory references are necessary to fetch instructions during the subsequent execution of the loop. This can improve the execution speed of a loop by a factor of two or three under certain conditions depending on its contents, and ways of exploiting this facility via the compiler will be given in later sections.
3. WHAT TO OPTIMIZE?

There are a number of general aspects of optimization which need to be considered, especially as effort needs to be applied only to those sections of existing code which are heavily used. These points should act too as a guide to the writing of new code.

3.1 ALGORITHM SELECTION

This report is devoted to the optimization of code, and not to the selection of the most efficient algorithm for the problem involved. In other words, it presupposes that a choice of the most suitable algorithm has already been made by a study of the problem and a search for existing code in the literature or in public or private program libraries. (At CERN, see the Program Library reference manual including the algorithm supplement at the end of each chapter, and the NAG Library manuals which are available at most terminal clusters. The routines you find may not always be the most efficient but provide a starting point for optimization based on reliable code). A knowledge of the available algorithms is an important first step to efficient code as many basic routines are coded in machine language (for instance, many matrix manipulation packages) and can in some cases, e.g. sorting, be many times faster than what can be produced quickly starting from scratch.

3.2 OPTIMIZATION VERSUS PORTABILITY

To achieve the very highest level of optimization it is often necessary to resort to the use of machine dependent features which rapidly render a program non-portable. Such features include the use of

- machine language, admissible as long as a FORTRAN version is available,

- machine dependent facilities for shifting and masking like

\[
I = \text{AND}(J,K) \quad (I,J,K \text{ type INTEGER, CDC}) \\
I = \text{LAND}(J,K) \quad (IBM);
\]

- fast I/O routines such as

\[
\text{BUFFER IN} \quad (CDC) \\
\text{IOPACK} \quad (CERN \ IBM).
\]

In a given case, the use of such features may be justified by a very large increase in efficiency, but as far as possible code of this type should be localised to ease conversion problems, and not be scattered throughout the whole program. Nor should these features be used unnecessarily in non-critical areas.
3.3 OPTIMIZATION VERSUS CLARITY

The question of the mutual interaction between optimization and clarity has to be viewed from three aspects: how clarity helps the compiler to optimize, how clarity helps the programmer to optimize and how optimization can on the other hand lead to less clear code.

3.3.1 Clear code helps the compiler to optimize

A compiler divides code into basic blocks and tries to optimize code locally in the blocks and globally between the blocks. It is a great help to the compiler in this task if the code lends itself to an efficient analysis and certain coding practices help towards this end. As a general rule, programs should be split into sub-programs each of which carries out a single or a few similar functions. Within a sub-program the flow of control should be from top to bottom, with as few statement labels, GO TO's and especially backward jumps as possible. Code which is clear to a reader is also clearer to the compiler, and this assists in the generation of better object code.

The data structure adopted should be as simple as the problem allows, with particular avoidance of the EQUVALENCE statement, which makes the meaning of the code less clear and acts as an impediment to the generation of efficient code.

3.3.2 Clear code helps the programmer to optimize

Clearly written code and simple data structures not only help the compiler to analyze the logic flow better, but also help the programmer working on its optimization to grasp more quickly its meaning, thereby allowing a more effective optimization effort. Program changes can be applied with more confidence, whereas one hesitates to touch a single line of 'spaghetti' programs. If the author of

IF(X.LT.Y) GO TO 30
IF(Y.LT.Z) GO TO 50
SMALL=Z
GO TO 70
30 IF(X.LT.Z) GO TO 60
SMALL=Z
GO TO 70
50 SMALL=Y
GO TO 70
60 SMALL=X
70 ......

had written what was intended, namely

SMALL=AMIN1(X,Y,Z)
neither understanding nor optimization effort need have been invested.

As an aid to readability, it is sometimes useful to consider the layout of the code and to improve this before starting to optimize. The IBM compiler provides the PARM=FMT option, which produces a source listing which is indented and divided according to the logical structure of the code as determined by the compiler. On the CERN CDC 7600, the library routine TIDY (Q900) is useful for cleaning-up poor code, and there is a facility for automatically indenting DO-loops in ANSI-FORTRAN code provided by:

```
FIND(INDENT,ID=PULIB)
INDENT(oldcode,newcode)
```

where 'oldcode' and 'newcode' are FORTRAN codes, UPDATE source or PAM card images.

Some tidying is often a necessary first step in the optimization process.

### 3.3.3 Effect of Optimization on Clarity

In constrast to the last sub-section, the case has also to be considered that the effect of optimizing code is often to make it less clear, at least on a line-by-line basis. FORTRAN derives its name from FORMula TRANslation, and has achieved widespread acceptance as a programming language because of the direct way in which it can be used to express mathematical formulae. Optimization can have the side-effect of causing a section of code to become less clear in its meaning, making it more difficult to understand and maintain. Even the well known Horner's Rule for the evaluation of a polynomial:

\[
Y = C(1) + X*(C(2) + X*(C(3) + X*(C(4) + X*(C(5) + X*C(6)))))
\]

is not as immediately obvious as

\[
Y = C(1) + (C(2)*X + C(3)*X^2 + C(4)*X^3 + C(5)*X^4 + C(6)*X^5)
\]

and the use of algebraic factorization over a complex section of code causes it to become quite opaque, giving further reason to follow the cardinal rule of optimizing only those parts of a program in which appreciable amounts of computing time are spent. Optimization of scarcely used code impedes understanding and represents wasted effort in terms of the results it brings, compared to the work devoted to it.
3.4 THE IDENTIFICATION OF THE MOST USED PARTS OF THE CODE

At CERN we have access to powerful but simple-to-use tools to analyse the distribution of time over the field length of a program. On the CDC 7600 the timing histograms may be obtained from a normal job whose control cards are modified to use SPY (L300) in the following way:

a) the job card becomes SPYnn or XnSPY
b) generate a load-map with MAP(PART)
c) before the LGO or program execution card add a card of the form

   SPY76A,lowadr,highadr

   where 'lowadr' and 'highadr' define the field length to be analysed
d) add a card SPY76B immediately after the execution step.

A FORTRAN callable interface

   CALL SPYON
   CALL SPYOFF

may be used to time the individual overlays of an overlaid program. The output provided by SPY is largely self-explanatory and identifies immediately any problem areas through significant peaks in the distributions; the aim is to reduce their size.

On the CERN IBM system, SPY is used in the following manner:

   // EXEC NFORTCL
   //C.SYSIN *

   user code

   // EXEC SPYIBMG
   //G.FT01F001 DD ...........

   etc.

   //G.SYSIN DD *

   user data cards

   //EXE SPYIBMT
   //T.SYSIN DD *
   50

The last two cards are optional, showing how to define the histogram bin width in units of bytes (default is 100 HEX).

By specifying the parameters

   SPYDNS='nn,ggg.SPYDATA',SPYDSP='(NEW,CATLG)',SPYUNT=SYSDA

on the SPYIBM card, and
TDSN='nn.ggg.SPYDATA'

on a SPYIBMT card in a separate job, the histograms may be stored and attached for later analysis.

The IBM version produces a number of histograms, the one of main interest being marked MAIN, and similar to that produced on the CDC.

Whereas CDC SPY is a monitor program running in a peripheral processor which does not affect the execution of the spied program in any way, SPYIBM assumes control of the execution of the program which runs as a sub-task, and absolute execution times are lengthened somewhat, and even a great deal if the G step has to wait for resources, such as the mounting of tape. The histograms, however, are correct in all cases.

A further aid to timing tests is the use of the subroutine TIMED (2007) which returns the elapsed time since the previous call. This is useful for timing individual routines or other sections of code.

On the CDC 7600 the package TIME76 (L310) is available for detailed timing, especially of limited sections of code where instruction scheduling and memory conflicts are of interest.
4. GENERAL HINTS FOR OPTIMIZATION

The following points are generally true for modern optimizing compilers, but should always be tested in specific cases, especially on systems other than the large CDC and IBM computers dealt with in later sections.

4.1 COMPILER OPTIONS

Production code should always be compiled with the highest optimization level available, and without options which produce possible traceback information at the expense of efficient sub-program calling sequences. Thus, on the IBM at CERN the NFORE procedure should be used for compilation, as this has OPT=3 as default, and if the code is well tested the option PARM=NOGOSTM may be specified. On the CDC systems, FTN should be used with OPT=2 and without the T (traceback) option which suppresses call-by-value calls to basic external functions (argument values in registers) in favour of call-by-name (just the start address of the whole argument list in a register).

4.2 INITIALIZATION OF VARIABLES

Variables are often repeatedly initialized at execution-time, when perhaps a single initialization at compile time (DATA statement) would be sufficient. Sometimes variables are initialized unnecessarily 'for safety' and this can lead, as a side-effect, to the calculation of an erroneous result if the code contains a fault, whereas an uninitialized variable (on the CERN systems), when incorrectly referenced, will result in a much more useful run-time error and diagnostic.

4.3 ARITHMETIC OPERATIONS

It is important to know that the various arithmetic operations are performed at very different speeds, and the so-called strength of operations should be reduced to take account of these differences. The order, starting with the fastest is:

- integer    +    I+J
- floating   +    A+B
- integer    *    I*J
- floating   *    A*B
- floating   /    A/B
- integer    /    I/J
- exponentiation ** A**B

In general, divisions should be avoided. In certain cases, in complicated code (see sub-section 4.5), a repeatedly used reciprocal should be calculated once and the division replaced:
RA = 1.0/A
:
:
B = C * RA
100 D = E * RA

When raising floating-point variables to an integer power, the exponent should be provided in the form of an integer constant where possible, as the compiler will substitute a repeated multiplication for exponents up to at least 12; thus

A = B ** 3

is faster than

N = 3
A = B ** N

4.4 MIXED DATA TYPES

A data type conversion such as

A = I

is always surprisingly expensive, especially if repeated in loops, as it involves several extra instructions. This overhead can be avoided by following these guidelines:

a) Try to eliminate such conversion by appropriate typing of variables.

b) Identify the remaining conversions by explicit use of the FLOAT and IFIX functions (then at least the overhead becomes visible):

A = FLOAT(I).

c) Where necessary, use a duplicate variable; compare:

DO 1 I=1,N
   A(I) = I
1 CONTINUE

DO 1 I=1,N
   A(I)=X
   X=X+1.
1 CONTINUE

where the extra increment can be faster than the saved conversion.

d) The number of mode conversions occurring in a mixed mode expression should be kept to a minimum, bearing in mind that the compiler is governed by the left-to-right evaluation rule of FORTRAN. In

X = A+I+B+J+C+K
the expression involves three conversions whilst executing, whereas

\[ X = \text{FLOAT}(I+J+K)+A+B+C \]

contains just one.

4.5 **REPEATED CALCULATIONS**

Modern compilers make an attempt to identify repeated calculations and to remove the repetitions. It falls to the programmer to help the compiler by writing such expressions in identical forms, thus

\[ A = X*Y+2.0+Y*X \]

should appear as

\[ A = X*Y+2.0+X*Y, \]

and

\[ A = B*B*C*C \]

as

\[ A = (B*C) * (B*C); \]

and in

\[ A = B-C \]

\[ D = C-B \]

the second statement should be

\[ D = -(B-C) \]

Parentheses are often needed to identify such repeated calculations, and should be used freely.

This type of optimization can be inhibited by the presence of statement labels in the sequence of code, as this indicates a possible alternative path into the block, and therefore possibly other values of the variables involved.

4.6 **DO-LOOPS**

Most time in programs is spent in the execution of DO-loops, and it is here that the greatest gains in optimization are to be achieved.
4.6.1 Avoidance of loops

The first rule of DO-loops is not to use them unnecessarily. At CERN, many short loops can be replaced by calls to the WECMAN package (F121) or to UCOPY (V301) which are written in machine language. Neither these routines nor loops should be used for trivial operations like

\[
\begin{align*}
&\text{DO 1 I=1,2} \\
&\quad \text{A(I)=0.} \\
&\quad 1 \text{ CONTINUE}
\end{align*}
\]

and other short loops such as

\[
\begin{align*}
&\text{Y=C(4)} \\
&\text{DO 1 I=1,3} \\
&\quad Y=Y*X+C(4-I) \\
&\quad 1 \text{ CONTINUE}
\end{align*}
\]

are more costly than their explicit equivalent

\[
Y = C(1)+X*(C(2)+X*(C(3)+X*C(4))),
\]

as long as the number of iterations is small. This is because the loop has to be initialized and terminated, and each passage through the loop requires a test for completion. In addition, registers may be unnecessarily assigned for use in the loop.

4.6.2 Loop organization

The effect of these loop overheads can be seen in the following example of nested loops:

\[
\begin{align*}
&\text{DO 1 I=1,100} \\
&\quad \text{number of initializations} = 1 \\
&\quad \text{DO 2 J=1,10} \\
&\text{\quad DO 3 K=1,2} \\
&\text{\quad :} \\
&\text{\quad :} \\
&\text{\quad 3 CONTINUE} \\
&\text{\quad number of tests on completion} = 2000 \\
&\text{\quad 2 CONTINUE} \\
&\quad \text{total} = 100 \\
&\quad 1 \text{ CONTINUE} \\
&\quad \text{total} = 3100
\end{align*}
\]

These loops, if written in the reverse order (assuming there are no more valid reasons for not doing so) would involve only 23 initializations and terminations and 2022 tests for completion, and this leads to the rule that the most repeated loops should be the innermost ones.
4.6.3 Subscripts

If we consider the case of a multi-dimensioned array referenced inside nested loops, we have the rule that the first subscript should normally vary fastest:

\[
\begin{align*}
& \text{DO 1 J=1,10} \\
& \text{DO 1 I=1,10} \\
& \quad A(I,J) = B(I,J) \times C \\
& \text{1 CONTINUE}
\end{align*}
\]

A former reason for this is no longer valid, and used to concern the calculation of the subscript expression which is

\[I+J^{(N-1)}\]

\((N=\text{column length}).\)

In this expression \(J^{(N-1)}\) is invariant to the loop, and can be moved outside leaving a simple increment only in the loop. The alternative expression \(J+I^{(N-1)}\) could, depending on the compiler and the context, generate slower code.

Secondly, in some cases where there is an increment by one, the FORTRAN compiler will generate code to pre-fetch the next operand and this facility is inhibited when the increment is different from unity.

Thirdly, stepping through consecutive memory locations can be more efficient than stepping through from column to column, which will be shown later to be even potentially disastrous.

Combining now these two nesting rules, we can derive a third - that arrays should, if the problem allows, be dimensioned with their largest subscripts first.

The calculation of subscript expressions is an area in which compiler writers have invested a great deal of effort, and the code generated often contains none of the multiplications one would expect from the basic subscript expressions, but rather a series of constant increments. This means that attempts by the programmer to perform explicit subscript evaluation will normally only reduce the efficiency of the code, rather than increase it, especially as any temporary variables used may be stored and fetched from memory rather than being possibly kept in registers. Thus

\[X = A(I+1,I-1) + A(I-1,I+1)\]

should never be written as

\[
\begin{align*}
& \text{IP1=I+1} \\
& \text{IM1=I-1} \\
& X = A(IP1,IM1) + A(IM1,IP1),
\end{align*}
\]

and this applies outside loops too. Additionally, several arrays in the same loop and with identical subscripting should have the same shape (dimensions) so that the subscript needs to be calculated only once for several references.

There is sometimes also a tendency to forget that DO-loops have three
loop parameters, and to perform simple subscript calculations which would
be better left to the compiler. Compare

\[
\begin{align*}
\text{DO } & 1 \ I=1,10 \\
& J=4+3*I \\
& X(J)=Y(J)+C \\
1 & \text{ CONTINUE}
\end{align*}
\]

Another example of poor code would be the case where an array is stored as
a vector, and the element subscripting performed explicitly.

4.6.4 Invariant code

It has become common practice to hand optimize DO-loops by attempting
to remove invariant expressions. Except in the case of some very long
loops, this should not normally be done if the invariant expression is
recognisable as such to the compiler, as it is able to perform this opera-
tion more efficiently. In an example of the type :

\[
\begin{align*}
\text{DO } & 1 \ I=1,10 \\
& A(I)=A(I) \\
\text{DO } & 1 \ J=1,10 \\
& C(J,I)=B(J)*A(I) \\
1 & \text{ CONTINUE}
\end{align*}
\]

the compiler may generate and store in a memory location a value for \( A(I) \),
whereas the straightforward and apparently inefficient version of this
loop is well optimized by the compiler, which simply places the value of
\( A(I) \) in a register during execution of the inner loop.

Once again it is necessary to order terms and use parentheses in
order help the compiler to recognize invariant expressions; in

\[
\begin{align*}
\text{DO } & 1 \ I=1,10 \\
& B(I)=2.+A(I)+Y \\
1 & \text{ CONTINUE}
\end{align*}
\]

the right-hand side should be rewritten as

\[(2.+Y) + A(I)\]

Another example is :

\[A(I,J) = 2.0*PI*X*Y(J)*B(I,J)\]

where the constants and the invariants are correctly grouped together.
The degree of invariant code optimization performed in a loop containing external references depends on the extent to which the invariant code is obscured by references to variables in COMMON or in an argument list. In

```
DO 1 I=1,10
   A(I)=Y/Z
   CALL SUB(A,Z)
1 CONTINUE
```

the division cannot be removed from the loop because SUB may change Z; CALL SUB(A,Z+1) however cannot change Z and the invariant can be removed. In a difficult case this is a situation where a temporary variable can be used to good effect. For the same reason, COMMON should not be used for the scratch storage of strictly local variables.

### 4.6.5 Loop manipulations

In critical situations it can be useful to 'unroll' a loop to reduce the number of overheads per operation. The loop

```
DO 1 I=1,N
   A(I)=B(I)*C(I)
1 CONTINUE
```

can be written as

```
DO 1 I=2,N,2
   A(I-1)=B(I-1)*C(I-1)
   A(I)=B(I)*C(I)
1 CONTINUE
```

obtaining two multiplications per loop test. This can be especially effective on machines where the multiplications are pipelined, and the loop can be unrolled even further (but should not go beyond the stack size on a machine with an instruction stack). Care must be taken if N is not a multiple of the chosen degree of unrolling; in this case some extra code is necessary after the loop.

In a similar way, loops can be 'combined' to save on overheads:

```
DO 1 I=1,100
   A(I)=B(I)*C(I)+4.
1 CONTINUE.
DO 2 J=1,100
   D(J)=E(J)+5.
2 CONTINUE
```

becomes

```
DO 1 I=1,100
   A(I)=B(I)*C(I)+4.
   D(I)=E(I)+5.
1 CONTINUE
```
but combination of loops should be used with care if loops are no longer short enough to fit in the stack of a stack machine.

The reverse technique is 'unswitching', useful in the case of loops containing an invariant logical test:

```
DO 1 I=1,100
   IF(FLAG) GO TO 3
   A(I)=B(I)-2.*C(I)
   GO TO 1
3   A(I)=B(I)+3.*D(I)
   B(I)=X*D(I)
1 CONTINUE
```

can be written as

```
IF(FLAG) GO TO 3
DO 1 I=1,100
   A(I)=B(I)-2.*C(I)
1 CONTINUE
GO TO 4
3 DO 2 J=1,100
   A(J)=B(J)+3.*D(J)
   B(J)=X*D(J)
2 CONTINUE
4 ..........
```

which avoids repeated testing of the invariant, and increases the possible degree of optimization by creating shorter loops.

### 4.6.6 Use of scalar accumulators

Inside short loops it is better to use scalars on the left-hand side of statements, rather than vector references, as the scalar can be kept in a register during the loop, whereas the array reference may be stored for each iteration:

```
SUM=0.
DO 1 I=1,100
   SUM=SUM+A(I,J)
1 CONTINUE
B(J)=SUM
```

is better than:

```
DO 1 I=1,100
   B(J)=B(J)+A(I,J)
1 CONTINUE.
```

Similarly, scaling factors should be applied to the final sum outside a loop and not to the individual terms inside it.
4.7 BRANCHES

All the various branch instructions execute at different speeds; the slowest is the computed GO TO, then the arithmetic IF, the logical IF and fastest of all is the assigned GO TO. The computed GOTO becomes more efficient once more than four or five branches can be taken from a given point.

As a two branch IF, the logical test is less confusing as well as faster; compare:

\[
\text{IF(I.EQ.2) GOTO 1 and IF(I-2)2,1,2 2 \ldots...}
\]

Logical tests should always be ordered for efficient testing:

\[
\begin{align*}
\text{IF(I.LT.10) GOTO 1} \\
\text{IF(I.GT.20) GOTO 2}
\end{align*}
\]

not

\[
\begin{align*}
\text{IF(I.GT.20) GOTO 2} \\
\text{IF(I.LT.10) GOTO 1}
\end{align*}
\]

if I is normally less than 10.

4.8 CALLING SEQUENCES

It is less costly to pass values between two sub-programs in the form of COMMON variables rather than in arguments. In the latter case, both the calling and the called routines have to perform some additional work to store and fetch the variables into some form of local storage, which is avoided if COMMON variables are used.

The transfer of arrays via arguments can be even more costly, and the use of variable dimensioned arrays is extremely inefficient as certain address calculations can be performed only at execution time, rather than being optimized at compile-time. Sometimes a compromise is possible; in

\[
\begin{align*}
\text{SUBROUTINE(A,N)} \\
\text{DIMENSION A(5,N)}
\end{align*}
\]

some addressing can be optimized as the column length of A is known to the compiler, whereas in the case of A(N,5) this would not be so.

The overheads involved in calling an external sub-program make it generally necessary to avoid writing very tiny routines which are called very frequently to perform rather limited tasks. In this case it is many times faster to use statement functions or to write the code out directly in-line. The use of external references inside loops can also have disastrous consequences for optimization, as the compiler has to store the current register contents for each call, and to retrieve them afterwards, and the loop execution time suffers accordingly.
For short sub-programs, where the subroutine call is equally well suited to being written as a function call, this latter form is to be preferred, as the function value will be left in a register rather than being stored in a memory location.

4.9 EQUIVALENCE STATEMENT

Variables kept in EQUIVALENCE classes frequently become unoptimizable and the use of this statement should normally be avoided. Especially bad are the use of scalars equivalenced to array elements, the equivalencing of loop indices and parameters, and the equivalencing of variables of differing type.

4.10 INPUT/OUTPUT OPERATIONS

At CERN, almost all large scale I/O is performed using binary unformatted I/O operations. It is sufficient to stress here that this is far more efficient than formatted I/O, which, apart from standard card image input and line-printer output, should be reserved for special applications only. The difference in efficiency derives from the fact that a conversion of each variable as specified by the FORMAT statement is necessary for formatted I/O, whilst a binary operation copies the data words in their unchanged form between memory and the external storage medium.
5. SPECIFIC HINTS WITH THE CDC FTN COMPILER

These points are valid for the larger CDC CYBER 170 series machines as well as the CDC 7600.

5.1 OBJECT CODE OPTIMIZATION

At compile time the FTN compiler evaluates constant subexpressions, and redundant instructions and expressions within a statement are eliminated.

Next the following optimizations take place:

a) Redundant instructions and expressions within a sequence of statements are eliminated.

b) Subscript calculations are simplified, and values of simple integer variables are stored in machine registers throughout loop execution, for innermost loops satisfying all of the following conditions:

No entries other than by normal entry at the beginning of the loop.

No exits other than by normal termination at the end of the loop.

No external references (user function references or subroutine calls, input/output, STOP, or PAUSE statements, or basic external function references) in the loop.

No IF or GOTO statement in the loop branching backwards to a statement appearing previously in the loop.

The compiler then collects information about the program unit as a whole and the following optimizations are attempted:

a) Values of simple variables are not retained when they are not referenced by succeeding statements.

b) Invariant (loop-independent) subexpressions are evaluated prior to entering the loops containing them.

c) For all loops, the evaluation of subscript expressions containing a recursively defined integer variable (such as I when I=I+1 appears within the loop) is reduced from multiplication to addition.

d) Array addresses, values of simple variables in central memory, and subscript expression are stored in machine registers throughout loop execution for all loops.
e) In all loops and in complicated sections of straight-line code, array references and subscript values are stored in machine registers.

f) In small loops, indexed array references are prefetched after safety checks are made to ensure that the base address of the array and its increment are reasonable and should not cause an out-of-bounds reference (mode 1 error).

Lastly, a critical path scheduling is performed to utilize the multiple functional units efficiently.

5.2 EXAMINATION OF DO-LOOPS

It is especially instructive to inspect the cross reference map produced by the R=n (n=1,2 or 3) option of the FTN compiler. Under the heading PROPERTIES we find for each loop one or more indications as to whether the loop was optimized and any reasons for not optimizing it. The properties are:

- OPT - loop has been optimized,
- INSTACK - loop fits into the instruction word stack,
- EXITS - loss of optimization because it contains a reference to an external sub-program,
- ENTRIES - loss of optimization because it contains entries from outside its range,
- NOT INNER - loss of optimization because it is not the innermost loop in a nest,
- EXITS - loss of optimization because it contains references to statement labels outside its range.

The list should be examined for important loops in critical routines to ensure that the compiler is being helped as much as possible. In particular, loops just too long for the stack (longer than 12B words on the CDC 7600) could possibly be reworked to fit them in, and loops containing EXITS which are simple functions could have these functions replaced by in-line code, or by statement functions which also produce in-line code.

Lastly, the FORTRAN listing should be examined for informative (I) diagnostics relating to optimization, and the relevant action taken.

5.3 PROGRAM SIZE AND STRUCTURE

The FTN compiler decomposes the source code into basic blocks, which are segments of code having a single entry and exit; this enables it to eliminate unnecessary operations within a block. For instance, in

\[
\begin{align*}
1 & \quad X = Y+Z \\
2 & \quad A = X+B \\
2 & \quad X = X/R
\end{align*}
\]
the compiler can eliminate the first store of the value of X as it is redefined immediately afterwards in the same basic block. The compiler should not be allowed to become confused in its basic block analysis by the presence of redundant or unnecessary statement labels.

Similarly, DO-loops and IF-loops are identified and invariant code and common sub-expressions are removed, where they can be identified as such, and IF-loops should always be written to resemble DO-loops as closely as possible.

These procedures are helped by a straightforward logic, with a minimum of statement labels and a simple path through the code, the total length of the sub-program should not exceed 600 executable statements. An effort should be made not to link artificially parts of the code which are independent through the use of identical temporary variable names. The two blocks

\[
\begin{align*}
\text{TEMP} &= A \ast \ast 2 \\
C &= \text{TEMP} + B \\
Q &= X \ast \ast \text{TEMP} \\
1 &\quad \cdots \\
\vdots &
\end{align*}
\]

are badly written because the common expression elimination is anyway unnecessary, and the use of the name TEMP twice can confuse the compiler into believing there is some link between the two blocks, and to saving the value of TEMP at the end of each block. The compiler can perform the same optimization using a register in each case.

Similarly DO-loops should always have different control variables, as these do not materialize if they are not in COMMON, if their values are not required outside the loop and if they are used as a simple index; in

\[
\begin{align*}
\text{DO } 1 &\quad I = 1,10 \\
A(2,I) &= 2 \ast \ast B \\
1 &\quad \text{CONTINUE}
\end{align*}
\]

only a register and no storage location is ever used for the value of I, and so short loops should always have unique control variable names.

The FTN compiler also has the feature that any multiplications used in subscript calculations are replaced by shifts if all but the last dimension of an array are powers of two, and by shifts and additions if they are sums of powers of two. Thus it can be an advantage to write an array dimension as, for instance, A(16,5) rather than A(15,5).
5.4 DATA INTERFERENCE

Data interference is the interaction between memory references which prevents the elimination during optimization of at least one reference. For instance, in

\[
\text{DIMENSION A(5)}
\]

\[
X = A(2) \\
A(I) = 1. \\
Y = A(2)
\]

the possible values of I are unknown at compile time, and the second fetch of A(2) cannot be eliminated.

Unintentional data interference may be introduced by over-zealous use of the EQUIVALENCE statement. If there are two separate equivalence classes located in a COMMON block, the compiler may merge them, as well as any intervening block members, into one large class. Thus the practice of equivalencing simple variables such as DO-loop control variables and parameters into a scratch COMMON is false economy.

In the following example

\[
\text{COMMON/SCRATCH/A(100)} \\
\text{DIMENSION ITEMP(50)} \\
\text{EQUIVALENCE(ITEMP(1),A(1)),(ITEMP(51),N),(ITEMP(52),M)} \\
\text{DO 1 N=1,M} \\
\text{1 IF(ITEMP(N).EQ.-1) GO TO 10} \\
\text{1 ITEMP(N)=0} \\
\text{10 CONTINUE}
\]

there is unintended data interference between the store in ITEMP(N) and the use of N and M. This interference prevents loop optimization such as the assignment of loop counters to registers throughout the loop.

Similarly, in

\[
\text{COMMON/A(4),N,M,B,C(4)} \\
\text{DIMENSION AA(11)} \\
\text{EQUIVALENCE (A(1),AA(1))} \\
\text{X=A(I)} \\
\text{B=1.} \\
\text{Y=A(I)}
\]

the second fetch of A(I) may not be eliminated because of data interference with B, introduced by the EQUIVALENCE statement.

In:

\[
K=N+1 \\
\text{DO 10 J=2,N} \\
10 A(J)=A(K)/A(1)
\]
A(K)/A(1) should be evaluated once before the loop and used as a temporary variable since the compiler does not recognize the fact that K is greater than N and that the loop cannot therefore change the value of A(K), and hence does not remove this section of invariant code from the loop.

Similarly:

```
DO 10 I=1,N
  10 X(J)=X(J)+X(J+1)*B(I)
```

should be rewritten as

```
T=X(J)
DO 10 I=1,N
  10 T=T+X(J+1)*B(I)
X(J)=T
```

since the compiler does not recognize that J+1 may not equal J, and does not assign a register for X(J).

5.5 BASIC EXTERNAL FUNCTIONS

It is more efficient, where possible, to consolidate algebraically references to basic external functions:

```
X = ALOG(A)+ALOG(B)
```

should appear as

```
X = ALOG(A*B)
```

5.6 BASIC INTRINSIC FUNCTIONS

The use of basic intrinsic functions, which appear as in-line code, should be considered wherever they can replace an external call or long-hand code, e.g. AMAX1 provides a much faster way of determining the maximum value of a string of numbers than a DO-loop, and the function RANF is ten times faster than RNMDM. However, unnecessary use of non-ANSI functions (grey shaded parts of the FTN manual, revision C, pp. 8-2 to 8-5) should be avoided.

5.7 TEST FOR ZERO

The CDC instruction set contains tests on equality and inequality with zero. This means that in all tests against zero, a suitable instruction will be used, whereas for other values this does not happen. If J can have only the value 0 or 1,

```
is faster than
```

```
IF(J.NE.0) .........
```

```
IF(J.EQ.1) .........
```
6. THE IBM FORTRAN (H EXTENDED) OPTIMIZATION ENHANCEMENT COMPILER

These points are valid for large IBM 370 and 303X series processors running with virtual storage and offering the above named compiler.

6.1 THE OPTIMIZATION PRODUCED BY THE COMPILER

The compiler working at its highest optimization level, OPT=3, proceeds through 11 steps in optimization. Each of these is summarized briefly.

6.1.1 Commutation of subscript dimension computations

In this step, the subscript expressions are analysed and the constant parts are removed from loops. This can involve some reordering as can be seen in the example A(4,J,6) where the first and last sub-expressions in the evaluation are constant in a loop over J, and can be consolidated and removed.

6.1.2 Elimination of duplicate computations

If an expression is computed more than once, the first value can be retained and used in place of the subsequent ones, as long as none of the operands change between the appearances of the expression and they lie on a common path. This is especially true for common subscript expressions in a loop.

6.1.3 Backward movement of invariant computations

Here, an expression, assignment or subscript evaluation which is invariant to a loop will be moved outside the loop.

6.1.4 Optimization of subscript component combinations

Array addressing on the IBM can involve up to four terms, a base address BASE for the subprogram, a displacement address DISP for the zeroth element of the array combined with any numeric constant part of the subscript, a constant part XCON which is invariant in the current loop, and a variable part XVAR which varies within the loop. In non-optimized code the address evaluation is performed as

\(((XVAR+XCON)+DISP)\)+BASE

whereas in optimized code this is reordered as

\[XVAR+(XCON+(DISP+BASE))\]
enabling the expression to be evaluated partly outside the loop, and also

to use more efficient machine instructions, in most cases.

6.1.5 Strength reduction of induction variable computations

An induction variable is one which is altered only once in a loop,
and by a constant increment or decrement. In this step, expressions of the
type c*I are replaced by a new variable which is incremented by c on each
iteration.

6.1.6 Local register optimization

A sub-set of registers is now assigned for keeping results until they
are used in a following reference.

6.1.7 Global register optimization

Another sub-set of registers is assigned for the storage of those
variables and constants which are most frequently referenced in the sub-
program.

6.1.8 Global register remapping

In this step, variables which have been assigned to registers both
locally and globally have the global assignments substituted for the local
ones, allowing use too of more advantageous machine instructions.

6.1.9 Global register purging

Here unnecessary initialization of registers is eliminated, this
occurring when a register is not used between two initializations, as may
happen particularly in certain nested loop configurations.

6.1.10 Global register scavenging

At this stage it may happen that there are now some general registers
which remain unused, especially after the remapping, and these are
assigned for use by address constants which have not been assigned to
registers otherwise.

(It is interesting to note the important role of register optimization
in the IBM compiler, contrasting with the importance given to func-
tional unit scheduling by the CDC compiler).
6.1.11 Section-oriented branch optimization

That part of the program which is beyond the range of the two registers kept for fast direct branch addressing is divided into sections which internally use direct branch addressing, but which for branches between sections use a third register for the section address. In this manner only three registers at most are taken for branch optimization, whilst normally using the fast machine instruction for branching.

6.2 PROGRAM SIZE AND STRUCTURE

The compiler breaks up the source sub-program into blocks which begin and end with statement labels. Unlike the case of FTN, it is not always advantageous to break the link between blocks, as in the IBM case the creation of too many local variables can create problems. As a rule, in large program units it is better to use common names for the various temporary variables created, whereas in small units different names should be chosen.

The choice of names should be such as to use equally each variable name length (one to six characters), as the compiler keeps separate tables for each name length, and compiles faster if there is an equal distribution of lengths.

Addressing on the IBM is performed within a sub-program by adding a 12-bit local address to a 24-bit base address contained in a register. A sub-program whose total length (excluding some large arrays which are treated as if in COMMON) exceeds 8192 (2000 HEX) bytes requires an additional register for addressing, resulting in a loss of roughly 5% in speed as the register is not available for optimization. The program load-map should be checked to see that no sub-program is longer than 8192 bytes. (With the old H Extended compiler up to three extra registers could be taken).

A further degradation in optimization can result from the use of variables from too large a number of COMMON blocks within a given DO-loop. Each COMMON block reference requires a register, and if there are insufficient registers those which otherwise would be used for optimization have to be taken.

Similarly, COMMON blocks should be structured with scalars first, followed by small arrays and with long arrays last; e.g.

\[ \text{COMMON/DATA/XIN,XOUT,NAME(10),BUFFER(4000)} \]

This is because addresses of scalars and arrays in COMMON blocks are referenced with respect to the beginning of the block, and a second register is required beyond 4096 bytes (1024 words). A register can be saved too if a frequently referenced COMMON variable is considered for storage in a local variable, but this can be a dangerous practice if the value is changed inside the sub-program.
6.3 I/O OPERATIONS

As already mentioned, unformatted I/O should always be preferred to formatted I/O. Data are most efficiently written from a single array; in

```
DIMENSION A(100)
WRITE(NOUT) A
```
or
```
WRITE(NOUT)(A(I),I=1,50)
```
each WRITE instruction generates just a single output request.

Interleaving, in contrast, is very costly:

```
WRITE(NOUT)(A(I),B(I),C(I),I=1,50)
```
generates one call per array element, whereas

```
WRITE(NOUT)(A(I),I=50),(B(I),I=1,50),(C(I),I=1,50)
```
generates only three calls.

This instruction provides an example in which an EQUIVALENCE statement can be used to advantage - if A, B and C are equivalenced in the appropriate way, just one call would be required.

On the CERN machines, RECFM=U formats which are used extensively for experimental data are supported by IOPACK (DD/US/45).

6.4 SPECIAL DATA TYPES

The use of the non-ANSI data types LOGICAL*1, INTEGER*2, etc. not only lead to a loss of portability, but to a considerable loss in efficiency. Before each reference to a LOGICAL*1 variable a register is cleared, either by allocation of a register just for the variable, or by issuing an instruction for this purpose; this is not necessary for a LOGICAL*4 variable which fills the register. An INTEGER*2 variable is optimized less well than an INTEGER*4 one - no strength reduction is performed and use as a loop index is especially bad.

If DO-loops contain vectors of differing element length, any apparently identical subscripts will, in fact, have to be evaluated separately for each array; in

```
INTEGER*2 I(10)
INTEGER*4 J(10)
:
:
DO 1 K=1,10
   J(K)=I(K)
1 CONTINUE
```
K could be calculated just once if I were promoted to INTEGER**n.

The mis-alignment of full-word variables across half-word boundaries leads to severe performance loss, but is fortunately reported by the system at compile-time; e.g.

```
COMMON/DATA/ IN,A(1000)
INTEGER*2 IN
```

The same argument applies to the mis-alignment of double-word variables (REAL*8) across word boundaries.

6.5 EQUIVALENCE STATEMENTS

The EQUIVALENCE statement leads to a loss of optimization. In certain cases, the statement will be interpreted literally by the optimizing compiler to produce unexpected results. In :

```
COMMON/JIM/COM1,COM2,COM3,......
DIMENSION ARRAY(1)
EQUIVALENCE (ARRAY(1),COM1)
DATA N/T/
    :
    COM1=
    COM2=
    :
    DO 10 I=1,N
         ARRAY(I)=
10 CONTINUE
    :
    IF(......) GO TO 99
    COM2=
    :

99 RETURN
```

the compiler assigns COM2 to a register until the RETURN, ignoring the assignment in the loop so that on returning from the subroutine the value thought to be placed in COM2 by the loop will not be found.

6.6 ARRAY SUBSCRIPTS

The constant of an array subscript, e.g. 6 in A(I+6), is usually placed directly in a machine language instruction in a 12-bit field. If the constant component is either negative or greater than 4092 bytes it is kept separately and added to the array reference, and so positive constants less than 4092 bytes (1023 words) should be used wherever possible.
6.7 **LOGICAL IF-STATEMENTS**

The evaluation of a logical IF-statement containing only `.OR.'s is broken off as soon as its value is definitely established. Thus in

```
IF(I.EQ.J.OR.L.EQ.K) GO TO 1
```

the evaluation is stopped after the first component if I is equal to J. This feature means that the components of logical expressions containing `.OR.'s should be grouped separately and in an order such that they can be tested in the most efficient manner, working from left to right. Parentheses as well as the operators `.AND.` and `.NOT.` inhibit this feature.

6.8 **USE OF STORAGE**

6.8.1 **Interleaving**

The main memory of the CERN IBM's is 4-fold interleaved such that consecutive double-words (8 bytes) are stored in different nodules. Care must be taken not to step through an array in such a manner as to access every fourth double-word repeatedly, as in this way only one memory module is being addressed, leading to long wait times. In

```
DIMENSION A(8,10)
DO 1 I=1,10
   A(1,I)=FLOAT(I)
1 CONTINUE
```

the memory would be accessed more efficiently by changing the first dimension of A to 9. A similar situation can arise in any row-wise manipulation of an array, as in

```
A(I,J)=A(I,J-1)+A(I,J+1)
```

6.8.2 **The cache memory**

Between the main memory and the CPU, the CERN IBM 370/168 and IBM 3032 each have a 32 Kbyte cache (buffer) memory. This memory lessens the interleaving problems for fetching but not for storing, and imposes the additional programming constraint that the volume of data referenced during a series of computations should be restricted to the size of the buffer. Certain patterns of subscripting in large arrays can cause continual flushing of the cache and should be avoided. In practice, this problem is rarely met.
6.8.3 Virtual Storage and Paging

It is difficult to assess the inefficiencies which can arise in a system possessing a large virtual store which pages into real memory. However, it is possible to define some guide-lines which will minimise these overheads, whatever their absolute magnitudes may be:

a) Locality of reference - processing should be sequential for code and data, so sub-programs treating a certain set of data should be grouped together, and references should not jump around wildly inside the set of data.

b) Minimum instantaneous real storage - there will be less paging if there are fewer pages being referenced at any point in the program's execution, so data should be grouped logically and code written in a modular fashion.

c) Validity of reference - paging is reduced if unnecessary data accesses are eliminated, and if long searches through data can be avoided. This implies the use of data structures like arrays which can be directly addressed, rather than chains which must be searched. Indirect addressing is harmful.
7. A BRIEF LOOK AT THE CRAY-1 VECTOR PROCESSOR

The introduction of vector processors means a radical change in the approach to optimization, as illustrated by these examples from the Cray Research CFT compiler running on the CRAY-1 computer.

7.1 THE CPU

The CRAY-1 (Fig.6) is a 12.5 nsec. cycle time vector processor, which has the special capability of being able to handle 64-bit, 64-word vectors in vector functional units. It has also extra sets of registers which act as a buffer between main memory and the main operating registers. The instruction stack is 64 words long. The computer's very high speed is achieved by a number of special hardware features, those concerning the CPU including not only the parallel operation of functional units and pipelining as in the CDC 7600, but also a feature known a chaining whereby the results emerging from one functional unit are fed directly into another, with no intermediate store or wait.

7.2 VECTORIZATION

Although the CRAY-1 operates at a speed of about double that of a CDC 7600 for scalar operations, it can obtain factors of 10 or more for vector operations. For the FORTRAN programmer this can imply a completely new way of structuring problems and their coded algorithms, in order to take advantage of the vector facilities and to avoid inadvertently rendering DO-loops non-vectorizable. Any loop which contains the statements GOTO, a CALL or any reference to an extend function or, in most cases, an IF cannot be vectorized. In addition, all variables within a loop must fall into one of these categories:

a) Invariant - remains unchanged

b) Constant increment integer (CII) - this is an induction variable, an integer which at one point only in the loop is incremented by a constant value on each pass through the loop. It may reference a single CII in its definition.

c) Vector array references - the subscripts must all be invariants, except one which may have the general form

\[ [+\text{invariant}\star] \text{CII}[+\text{invariant expression}] \]

Thus the following loop is vectorizable:

\[
\text{DO 10 I=3,101,2}\\
\text{K=K-KDELT}\text{A}\\
\text{J=107-I}\\
\text{A=(3,I-2)=COS(B(J))*C(M-2*K+L*M/7,L,M/L)*X*D(L,M)}\\
\text{10 CONTINUE}
\]
(I,J and K are CII's, A,B, and C are vector array references, KDELTA, X, D, L, and M are invariants and COS is an intrinsic function).

Clearly a temporary variable defined inside a loop and used to contain a repeated sub-expression falls into none of these categories and inhibits vectorization.

Great attention to the structure of loops is required. In the matrix multiplication

```
DO 1 I=1,N
  DO 1 J=1,M
    A(I,J)=0.
    DO 1 L=1,K
      A(I,J)=A(I,J)+B(I,L)*C(L,J)
  1 CONTINUE
```

we see that A(I,J) is independent of the inner DO-loop index, rendering the loop non-vectorizable. The same sequence, written with the inner loops exchanged and the setting to zero performed then on a row basis, is six times faster:

```
DO 1 I=1,N
  DO 2 J=1,M
    A(I,J)=0.
  2 CONTINUE
  DO 1 L=1,K
    DO 1 J=1,M
      A(I,J)=A(I,J)+B(I,L)*C(L,J)
  1 CONTINUE
```

Care is needed to avoid interdependencies which can inhibit the vectorization of loops. In the loop

```
JMINUS1=J-1
DO 1 I=2,100
  A(I,J)=A(I-1,JMINUS1)
  1 CONTINUE
```

the compiler cannot determine whether JMINUS1 is ever equal to J, whereas in

```
DO 1 I=2,100
  A(I,J)=A(I-1,J-1)
  1 CONTINUE
```

this is clearly never the case, and vectorization can take place.

The final example illustrates in a very clear way the problems that arise in coding for a vector machine in an efficient way, and shows also how vectorization is sometimes achieved by substituting a vector for a scalar. In the simple loop

```
SUM=0.
DO 1 I=N,M
  SUM=SUM+A(I)
  1 CONTINUE
```
the scaler accumulator SUM does not fall into one of the three required categories, and the loop does not vectorize; when rewritten in the vectorized form below it is less fast for 10 iterations, 22 % faster for 100, and 87 % faster for 100,000:

```
C
C INITIALIZE VECTOR ACCUMULATOR
   DO 10 I=1,MINO(64,M-N+1),1
       HELPER(I)=A(N+I-1)
   10 CONTINUE
C
C DEFINE NO. OF BLOCKS OF 64 WORDS, AND NO. LEFTOVER
   JS=I+N-2
   KOUNT=(M-JS)/64
   LAST=(M-JS)-64*KOUNT
C
C SUM ARRAY IN BLOCKS
   DO 20 J=1,KOUNT
      DO 15 I=1,64
         HELPER(I)=HELPER(I)+A(I+JS)
      15 CONTINUE
   JS=JS+64
   20 CONTINUE
C
C ADD LEFTOVERS
   DO 30 I=1,LAST
      HELPER(I)=HELPER(I)+A(M+1-I)
   30 CONTINUE
C
C FINAL SUM
   SUM=HELPER(I)
   DO 40 I=2,MINO(64,M-N+1),1
      SUM=SUM+HELPER(I)
   40 CONTINUE

(Note that the CFT compiler accepts zero-trip loops).

Cray Research provides an extensive program library to enable users to access vectorized code in a simple way.)
8. ACKNOWLEDGEMENTS

I would like to thank the many colleagues in DD Division who have helped me in the preparation of this report, and particularly Dr. R. Matthews for his valuable advice on the IBM part of the subject.

I thank also F.G. De Bilio for his preparation of this text.
9. SOURCES

CRAY-1 FORTRAN (CFT) Reference Manual, CRAY Research publication 2240009


COMPASS Version 3 Reference Manual, CDC Publication 60492600

FORTRAN Extended Version 4 Reference Manual, CDC Publication 60497800

FORTRAN Extended Version 4 User's Guide, CDC Publication 60499700

SCOPE 2.1 User's Guide, CDC Publication 60372600

IBM OS FORTRAN IV (H Extended) Compiler, Programmer's Guide

IBM System/370 FORTRAN IV (H Extended) Optimization Enhancement, Program Description and Operations Manual

IBM System/370 Principles of Operation

Introduction to Virtual Storage in System/370, Student's Text

'The IBM System/370 Model 3033', Datamation, May 1977

'The IBM 3033, an inside look', Datamation, May 1979

Figure 1: Flow diagram of the major phases of the Fortran H Extended compiler. The optimization enhancements are mainly in the OPT(3) path.
<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY statement names</td>
</tr>
<tr>
<td>FORMAT statements</td>
</tr>
<tr>
<td>Compiler generated constants</td>
</tr>
<tr>
<td>COMMON block information</td>
</tr>
<tr>
<td>External references</td>
</tr>
<tr>
<td>Source module constants</td>
</tr>
<tr>
<td>Compiler generated external references</td>
</tr>
<tr>
<td>Machine instructions</td>
</tr>
<tr>
<td>Branch list</td>
</tr>
<tr>
<td>Relocation Dictionary</td>
</tr>
</tbody>
</table>

*Figure 2: IBM object module*
<table>
<thead>
<tr>
<th>Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address substitution code</td>
</tr>
<tr>
<td>ENTRY</td>
</tr>
<tr>
<td>Executable statement</td>
</tr>
<tr>
<td>Parameter list for external references</td>
</tr>
<tr>
<td>Compiler generated entities</td>
</tr>
<tr>
<td>Simple variables, FORMAT statements, program constants</td>
</tr>
<tr>
<td>Local arrays</td>
</tr>
<tr>
<td>Hollerith constants</td>
</tr>
<tr>
<td>Formal parameters</td>
</tr>
</tbody>
</table>

*Figure 3: CDC object module*
Figure 4: IBM 370 basic structure

The IBM has three levels of memory, the unit of transfer between virtual storage and main memory being the 4Kbyte page, and that between main memory and the cache being the 32 byte block.

The Instruction Pre-Processing Function (IPPF) fetches instructions from the cache, decodes them and prepares the addresses, registers and operand fetches for the Execution Function. Should an instruction be a branch, then instructions along the branch path are pre-fetched; for conditional branches this involves a guess as to which path will be followed.
Large Core Memory
LCM
512K words

Small Core Memory SCM
64K words

Instruction stack

18-bit index registers

18-bit address registers

60-bit operand registers

Boolean
Shift 2

Normalize
Floating add 2

Long add 3

Floating multiply 4

Floating divide 20

Population count 2

Increment 2

Functional Units with cycle times per single instruction

4th operand pair

→ a segmented functional unit (pipeline)

→ 1st result

Figure 5: CDC 7600 basic structure
**Figure 6 : CRAY-1 Vector processor**