SLAC–TN-91–5
May 1991
(TN)

INTRODUCTION TO VECTORIZATION USING THE IBM 3090 VF AND VS FORTRAN RELEASE 2*

BEBO WHITE

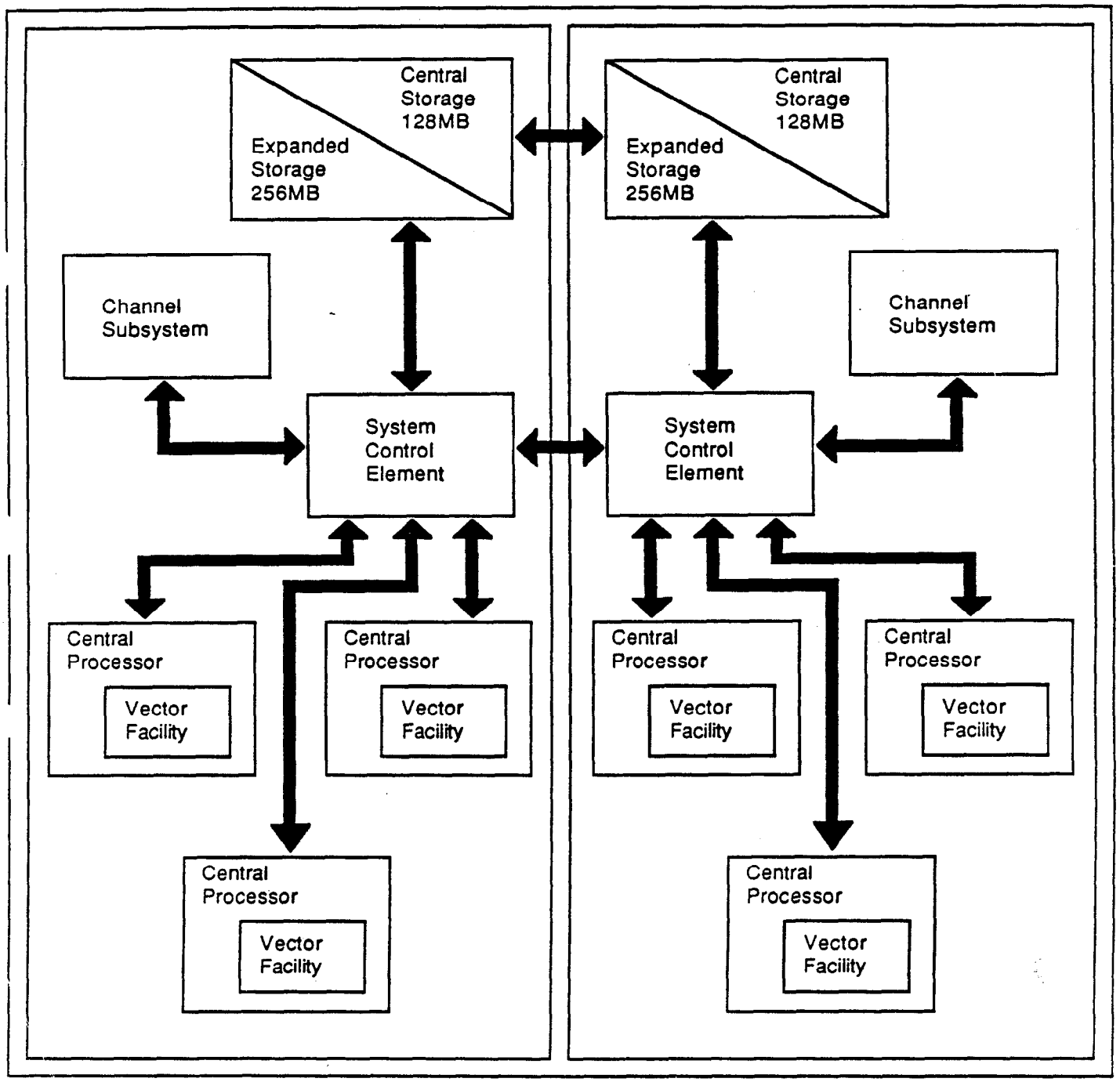
Stanford Linear Accelerator Center,
Stanford University, Stanford, CA 94309

and

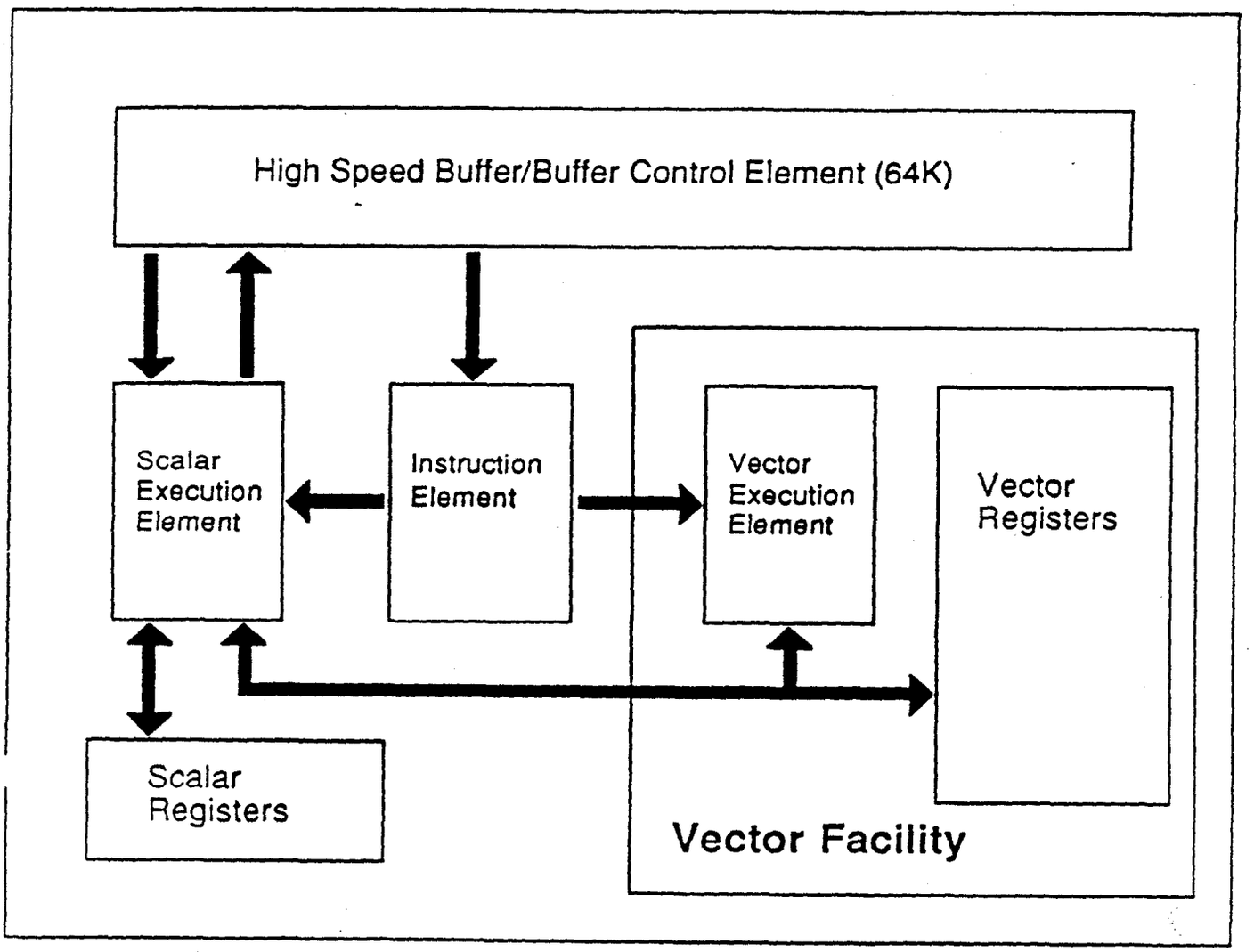
CERN, CH-1211, Geneva 23, Switzerland

*Work supported by the Department of Energy, contract DE-AC03-76SF00515

The IBM 3090-600 Processor Unit Design



The IBM 3090 Central Processor



What is Vector Processing

"Vector processing is a complication to computing, invented to make number crunchers go faster."

Most of the elementary vector operations consist of a series of independent calculations for all elements of the operand vectors, and so may be performed in parallel. Vector processing may thus be seen as one particular form of parallel computing.

The IBM 3090-600E Vector Facility

- fast scalar performance for compute intensive applications
- six processors, each with a vector facility and 64 KB cache memory
- 256 megabytes of memory
- 1 gigabyte of expanded storage
- 115 gigabytes of disk storage
- each application may use up to 999 megabytes of virtual memory

The IBM 3090-600E Vector Facility

- the dynamic range is 10^{+75} to 10^{-78}
- provides a decimal precision from 6 to 7 (short) digits to 13 to 14 (long) decimal digits
- cycle time of 17.2 nanoseconds
- theoretical peak performance of 116 megaflops
- likely ESSL peak performance of 75 megaflops
- realistic vector program performance goal of 40 to 50 megaflops

The IBM 3090 Vector Facility

- 16 32-bit data vector registers or 8 64-bit registers (for single or double precision data)

- these 16 vector registers operate on up to 128 data elements (the section size) of 4 bytes each

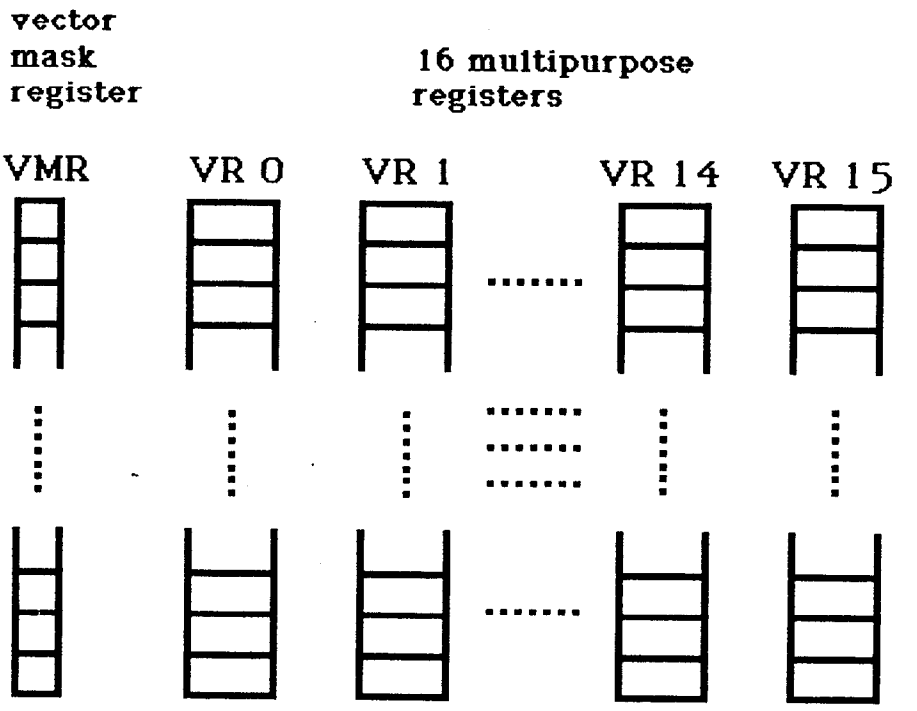
- three other vector registers:
  - vector mask register
  - vector activity count
  - vector status register

- 171 vector assembler instructions

- FORTRAN code using REAL*8 data has access to three compound vector instructions, which execute two FLOPs per cycle (after pipeline startup):
  - multiply and add
  - multiply and subtract
  - multiply and accumulate

- most other vector instructions execute one FLOP per cycle (after pipeline startup)

Vector Facility Registers



- the vector mask register is 1 bit wide
- the vector registers are 32 bits wide and may be paired for a width of 64 bits
- the section size Z is 128 elements

What is a Vector?

- a **VECTOR** is a group of elements in an array

- a vector is partitioned into a **SECTION** in order to execute on the vector hardware. The section size on the IBM 3090E is 128 elements.

- the spacing between successive elements in a vector is called **STRIDE**. For example, the vector A(1), A(2), A(3)... has stride 1.

```
THE ARRAY A(100,200) IS LAID OUT IN STORAGE AS:

A(1,1) A(2,1) A(3,1)....A(100,1) A(2,1) A(2,2)...

      A(1,1) A(2,1) <=== STRIDE 1
      A(1,1) A(1,2) <=== STRIDE 100
```

- an **INDUCTION VARIABLE** is any INTEGER*4 variable that is incremented or decremented by a fixed amount each time through a loop, such as with the index of a DO loop. This is also referred to as an **INDUCTIVE SUBSCRIPT**.

THE SUBSCRIPT EXPRESSION I IS AN INDUCTION VARIABLE HERE:

```
      DO 10 I = 1,N
10    A(I) = SCAL
```

THE SUBSCRIPT EXPRESSION I*I IS A NON-INDUCTION SUBSCRIPT HERE:

```
      DO 10 I = 1,N
10    A(I*I) = SCAL
```

What Does Vectorizable Mean?

- only DO loops can be vectorized
- the basic unit of vectorization is the statement – there is no partial vectorization within a FORTRAN statement
- in a DO loop, the calculations in one iteration of the loop must not depend on a previous iteration.

  For example, this loop vectorizes

```
      DO 10 I = 1,90
      C(I) = A(I) + C(I) * 3
10    CONTINUE
```

  while this one does not

```
      DO 20 I = 1,90
      C(I+1) = A(I) + C(I) * 3
20    CONTINUE
```

Scalar Computation For a DO Loop

- registers for scalar arithmetic hold only one element at a time

- to add two vectors A and B, each element in vector B has to be added individually to the appropriate element in the vector A, and then assigned to the appropriate element in vector C.

  For example,

```
      DO 10 I = 1,N
      C(I) = A(I) + B(I)
10    CONTINUE
```

  the sequence of instructions for this DO loop, executed in scalar mode would be:

1. LOAD ELEMENT COUNT(N)
2. LOAD a(i) INTO scalar register
3. ADD b(i) INTO scalar register
4. STORE c(i) FROM scalar register
5. DECREMENT COUNT BY 1

Vector Computation for a DO Loop

- vector registers can hold up to 128 elements

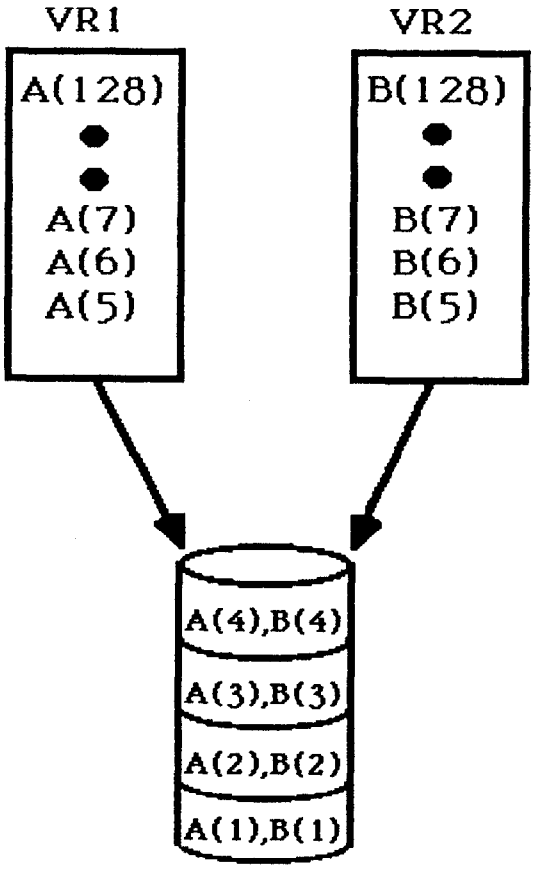
- vectorizing a DO loop produces instructions that operate on groups of data elements.

  The sequence of instructions in vector mode is:

  1. LOAD ELEMENT COUNT (N)
  2. LOAD a(1) – a(128) INTO vector register
  3. ADD b(1) – b(128) TO VECTOR register
  4. STORE c(1) – c(128) FROM vector register
  5. DECREMENT COUNT by 128

Vector Registers

Provide a FAST storage location for operands, available to the pipeline on a one cycle per operand set basis.



Vector Sectioning — the Basic Action of Vectorization

```
      DO 10 J = 1,N
10    A(J) = B(J)
```

becomes sectioned as:

```
      DO 10 J = 1,N,Z
      DO xx JV = J,J+MIN(N-J,Z-1),1
xx       A(JV) = B(JV)
10    CONTINUE
```

- the innermost (DO xx) loop is executed in the vector registers in groups of Z (128) elements at a time
- the outer loop increment is Z instead of 1 so that the vector instructions in the loop are executed approximately N/Z times rather than the N times required by the equivalent scalar loop.
- the remaining iterations (i.e., when N is not an integer multiple of Z) are also processed in the vector registers
- the MIN is the "sectioning overhead."

Tools for Vectorization

- VS FORTRAN Version 2 Release 3 Compiler
- Interactive Debugger (IAD)
- Engineering and Scientific Subroutine Library (ESSL)
- Assembler Listing

Vectorization Strategy

- time your program to find where it spends most of its time (the ***hot spots***)

- compile your program with VS FORTRAN Version 2, using the vector option on all or just key routines and then run it.

- look at the vector report to see which loops were vectorized

  1. were key loops vectorized?

  2. what prevented vectorization?

- compare vector to scalar execution times

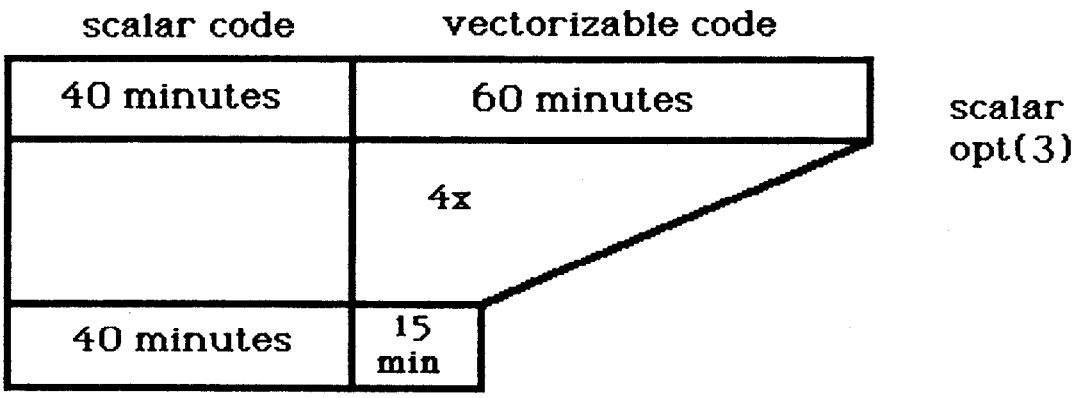
- assess performance expectations

- if necessary and potentially fruitful, modify your program to increase vectorization

Vector Content

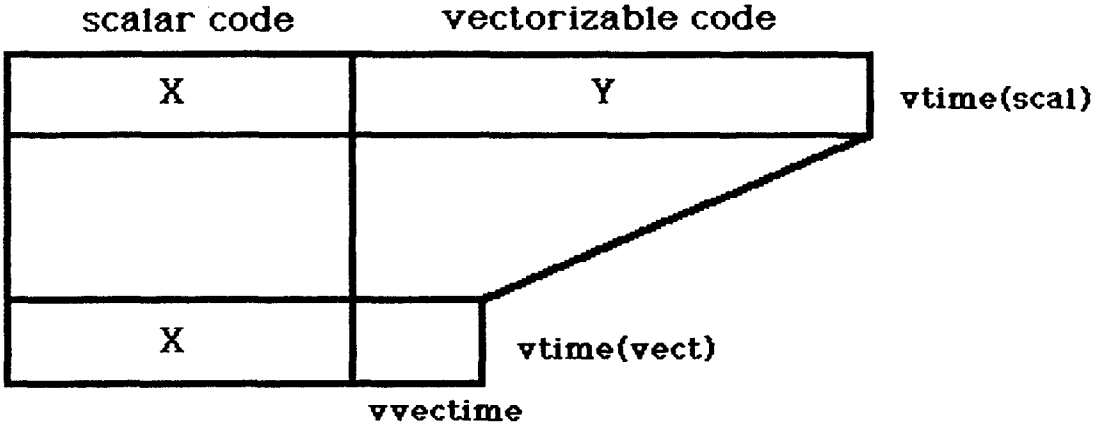
the **Vector Content** of a program is that percentage of the scalar code that vectorizes.

- assume, for example, that 60% of your scalar code vectorizes
- assume further that this 60% has a vector to scalar speedup of 4



$$\text{program speedup} = \frac{\text{scalar time}}{\text{vector time}} = \frac{100}{55} = 1.82$$

Vector Performance Formulas



Y = vtime(scal) - X
= vtime(scal) - (vtime(vect) - vvectime)
= vtime(scal) - vtime(vect) + vvectime

% vectorizable = Y / vtime(scal) * 100
good vector content = 75% +

vector speedup = Y / vvectime
good vector/scalar speedup = 3 to 5

program speedup = vtime(scal) / vtime(vect)
good program speedup = 1.5 to 3.0

AMDAHL's LAW

$$Y = \frac{1}{1 - V + \frac{V}{a}}$$

Y = PROGRAM SPEEDUP
V = % VECTORIZABLE
a = SPEEDUP VECTOR/SCALAR

FOR $a \to \infty$ $\quad Y = \frac{1}{1-V}$

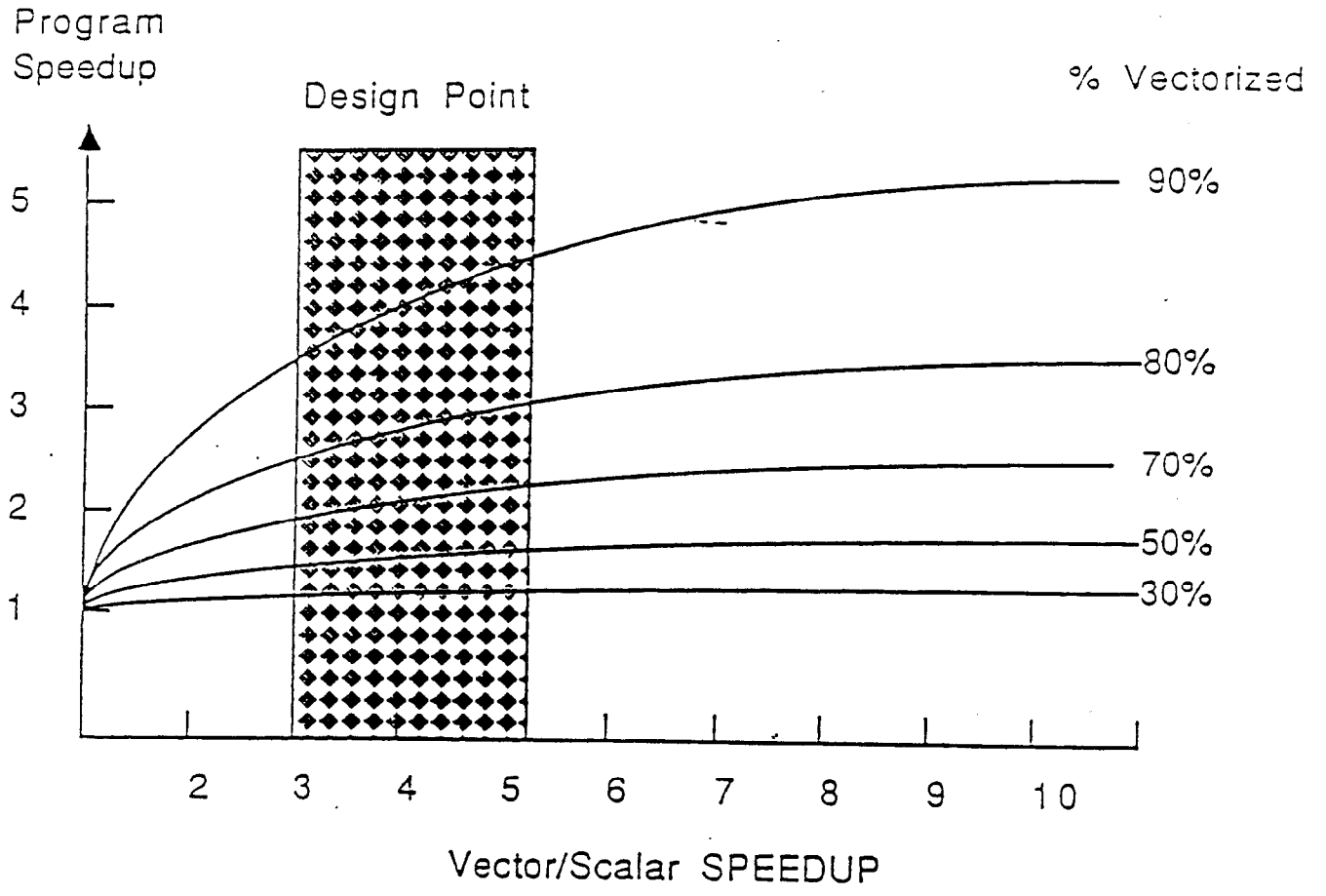
$\begin{cases} V = 1/4 & Y = 4/3 \\ V = 1/2 & Y = 2 \\ V = 3/4 & Y = 4 \end{cases}$ HEAVENLY

FOR $a \to 5$

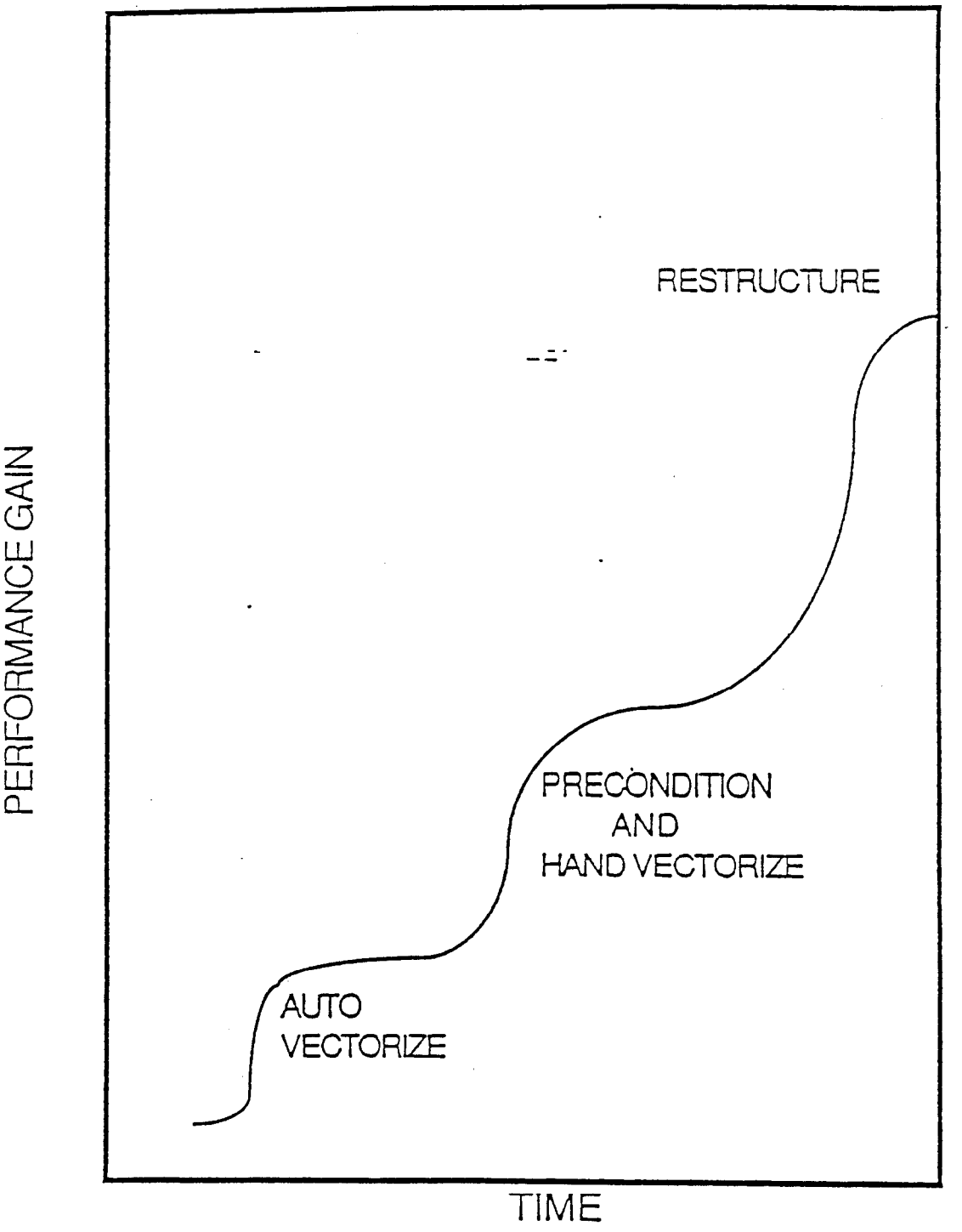
$\begin{cases} V = 1/4 & Y = 5/4 \\ V = 1/2 & Y = 5/3 \\ V = 3/4 & Y = 5/2 \end{cases}$ REALISTIC

FOR $V \to 1$ $\quad Y = a$ HEAVENLY

Vector Performance Considerations — Amdahl's Law



Level Of Effort



Quick Timing

- **READY MESSAGE**

  when no errors occur, the CMS ready message is of the form:

```
R; T=M.MM / N.NN HH:MM:SS
```

  where m.mm is elapsed CPU in seconds and n.nn is elapsed CPU plus overhead in seconds (since the last CMS ready message).

- **INDICATE USER**

  issue the command **INDICATE USER** before and after running a program to determine approximate overall time and vector time.

  VTIME elapsed CPU since LOGON in mmm:ss
  VVECTIME
  elapsed vector CPU since LOGON in mmm:ss (a subset of VTIME)

The FORTRAN Version 2 Compiler

- can automatically vectorize eligible statements in DO loops
  - only statements in DO loops can be vectorized
  - will select the **single** DO loop in a nest of loops whose vectorization will lead to the fastest execution
- will use vector versions of most intrinsic math functions
- can use optimization level 2 or 3 with vectorization; default is OPT(3)
- generates a vector report which shows the vectorization decisions made by the compiler

Compiling with the Vector Options

NOVECTOR is the FORTVS2 default. The VECTOR option and suboptions must be specified.

Syntax:

```
FORTVS2 PROGNAME (OPT(213)
                  VECTOR (VECTOR SUBOPTIONS)
                  OTHER COMPILER OPTIONS...
```

Example:

```
FORTVS2 MULT (OPT(3) VECTOR (REPORT (XLIST))
```

Vector Suboptions

- REPort (TERM LIST XLIST SLIST STAT)

  TERM Flags vectorized loops and shows how those loops were restructured. Display is at the terminal.

  LIST Same as TERM, but information is placed in the LISTING file.

  XLIST Produces detailed information about why loops were not vectorized, put in the LISTING file.

  SLIST Shows vectorized loops and statements in the format of the entire source program; placed in the LISTING file.

  STAT A vector statistics table is placed in the LISTING file.

- IVA

  Produces a Program Information File, which is required by IAD to use Interactive Vectorization Aid

functions.

- SIZE (ANY|LOCAL|n)

  Specifies the section size to be used.

  ANY uses the section size of the machine on which the routine is running.

  LOCAL uses the section size of the machine that compiled the program

  n used to specify an explicit section size. Must be the same as the machine's actual section size.

Vector Suboptions Example

```
FORTVS2 TEST (OPT(3) VECTOR(REPORT(TERM))
```

WOULD DISPLAY AT THE TERMINAL:

```
SCAL ----- DO 10 I = 1,N
|____________A(I+500) = A(I) + 1.0
```

```
FORTVS2 TEST (OPT(3) VEC(SIZE(LOCAL)REP(XLIST))
```

WOULD PLACE IN THE LISTING FILE:

```
VECT ----- DO 10 I = 1,N
|____________A(I+500) = A(I) + 1.0
```

Sample Timing Analysis

```
      PROGRAM FTVECT
      PARAMETER (N=20000, M1=1200, M2=175, M3=425)
      REAL*4 D(N), E(N), DOTPR, SUM
      REAL*4 A(M1,M2), B(M2,M3), C(M1,M3)

      DO 10 I=1,M1
        DO 10 J=1,M2
          A(I,J) = NINT(FLOAT(I-J))
10    CONTINUE

      DO 15 I=1,M2
        DO 15 J=1,M3
          B(I,J) = 1.0/SQRT(FLOAT(I)/FLOAT(J))
15    CONTINUE

      DO 20 I=1,N
        D(I) = SIN(FLOAT(I) / 2.0)
        E(I) = COS(FLOAT(I) * 2.0)
20    CONTINUE

      DOTPR = 0.0
      SUM = 0.0

      DO 30 I=1,5
```

```
        SUM = SUM + D(I) / E(I)
 30   CONTINUE

      DO 35 I=1,M1
        DO 35 J=1,M3
          DO 35 K=1,M2
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
 35   CONTINUE

      DO 40 I=1,N
        DOTPR = DOTPR + (D(I) * E(I))
 40   CONTINUE

      DO 50 I=1,1200
        DO 50 J=1,200
C         WRITE (10,51) C(I,J)
 50   CONTINUE
 51   FORMAT (F15.5)

      DO 55 I=5000, N, 5000
        WRITE (6,56) I, D(I), E(I)
 55   CONTINUE
 56   FORMAT(' I = ',I5,'   D(I) = ',F8.2,'   E(I) =
',F8.2)

      WRITE (6,*) 'SUM:              ',SUM
```

```
      WRITE (6,*) 'DOT PRODUCT:      ',DOTPR
      WRITE (6,*) 'MATRIX MULTIPLY: ',C(M1,M3)

      STOP
      END
```

```
FORTVS2 FTVECT (OPT(3)
VS FORTRAN VERSION 2 ENTERED.  09:48:11

**FTVECT** END OF COMPILATION 1 ******

VS FORTRAN VERSION 2 EXITED.   09:48:11

READY; T=0.12/0.15 09:48:11
LOAD FTVECT (CLEAR
READY; T=0.08/0.11 09:48:23
IND USER
USERID=BEBO     MACH=370 STOR=0006M VIRT=V XSTORE=NONE
IPLSYS=CMSR5C   DEVNUM=0015
PAGES: RES=000914 WS=000590 LOCK=000000 RESVD=000000
NPREF=000035 PREF=000000 READS=000040 WRITES=000047
XSTORE=000048 READS=000436 WRITES=000630 MIGRATES=000047
CPU 00: CTIME=00:47 VTIME=000:42 TTIME=000:44 IO=001721
        RDR=000000 PRT=000053 PCH=000000
        VVECTIME=000:07 TVECTIME=000:07
```

```
READY; T=0.01/0.01 09:48:28
START
DMSLIO740I EXECUTION BEGINS...
I =  5000   D(I) =    -0.65   E(I) =    -0.95
I = 10000   D(I) =    -0.99   E(I) =     0.81
I = 15000   D(I) =    -0.85   E(I) =    -0.60
I = 20000   D(I) =    -0.31   E(I) =     0.32
SUM:              -8.36326027
DOT PRODUCT:     -0.393103242
MATRIX MULTIPLY:   587375.500
READY; T=26.01/26.16 09:49:15
IND USER
USERID=BEBO     MACH=370 STOR=0006M VIRT=V XSTORE=NONE
IPLSYS=CMSR5C   DEVNUM=0015
PAGES: RES=000899 WS=000864 LOCK=000000 RESVD=000000
NPREF=000034 PREF=000000 READS=000040 WRITES=000047
XSTORE=000048 READS=000436 WRITES=000630 MIGRATES=000047
CPU 00: CTIME=00:48 VTIME=001:08 TTIME=001:10 IO=001743
        RDR=000000 PRT=000073 PCH=000000
        VVECTIME=000:07 TVECTIME=000:07
READY; T=0.01/0.01 09:49:25

FORTVS2 FTVECT ( VECTOR (LEVEL(2) REPORT(TERM))
VS FORTRAN VERSION 2 ENTERED.  09:50:18
```

```
(I) USE OF VECTOR REQUIRES OPT(2) OR OPT(3). OPTIMIZATION
LEVEL HAS BEEN
SET TO 3.

SCAL    +------- DO 10 I=1,M1
SCAL    |+------   DO 10 J=1,M2
        ||______    A(I,J) = NINT(FLOAT(I-J))
        |_______

VECT    +------- DO 15 I=1,M2
SCAL    |+------   DO 15 J=1,M3
        ||______    B(I,J) = 1.0/SQRT(FLOAT(I)/FLOAT(J))
        |_______

VECT    +------- DO 20 I=1,N
        |         D(I) = SIN(FLOAT(I) / 2.0)
        |_______  E(I) = COS(FLOAT(I) * 2.0)

SCAL    +------- DO 30 I=1,5
        |_______  SUM = SUM + D(I) / E(I)

VECT    +------- DO 35 I=1,M1
SCAL    |+------   DO 35 J=1,M3
SCAL    ||+-----    DO 35 K=1,M2
        |||_____      C(I,J) = C(I,J) + A(I,K) * B(K,J)
        ||______
```

```
        |_______

VECT    +------- DO 40 I=1,N
        |_______   DOTPR = DOTPR + (D(I) * E(I))

UNAN             DO 55 I=5000, N, 5000

THE DO-LOOPS HAVE BEEN PROCESSED AS INDICATED.

**FTVECT** END OF COMPILATION 1 ******

VS FORTRAN VERSION 2 EXITED.   09:50:34

READY; T=0.18/0.23 09:50:34
LOAD FTVECT (CLEAR
READY; T=0.10/0.13 09:50:56
IND USER
USERID=BEBO     MACH=370 STOR=0006M VIRT=V XSTORE=NONE
IPLSYS=CMSR5C   DEVNUM=0015
PAGES: RES=000916 WS=000695 LOCK=000000 RESVD=000000
NPREF=000034 PREF=000000 READS=000040 WRITES=000047
XSTORE=000047 READS=000489 WRITES=000712 MIGRATES=000047
CPU 00: CTIME=00:49 VTIME=001:09 TTIME=001:11 IO=001858
        RDR=000000 PRT=000130 PCH=000000
        VVECTIME=000:07 TVECTIME=000:07
READY; T=0.01/0.01 09:51:02
```

```
START
DMSLI0740I EXECUTION BEGINS...
I =  5000   D(I) =    -0.65   E(I) =    -0.95
I = 10000   D(I) =    -0.99   E(I) =     0.81
I = 15000   D(I) =    -0.85   E(I) =    -0.60
I = 20000   D(I) =    -0.31   E(I) =     0.32
SUM:              -8.36326027
DOT PRODUCT:     -0.393156052
MATRIX MULTIPLY:   587375.500
READY; T=7.87/7.94 09:51:18
IND USER
USERID=BEBO     MACH=370 STOR=0006M VIRT=V XSTORE=NONE
IPLSYS=CMSR5C   DEVNUM=0015
PAGES: RES=000904 WS=000856 LOCK=000000 RESVD=000000
NPREF=000034 PREF=000000 READS=000040 WRITES=000047
XSTORE=000044 READS=000492 WRITES=000712 MIGRATES=000047
CPU 00: CTIME=00:50 VTIME=001:17 TTIME=001:19 IO=001888
        RDR=000000 PRT=000150 PCH=000000
        VVECTIME=000:14 TVECTIME=000:14
READY; T=0.01/0.01 09:51:24
```

Sample Timing Analysis (cont.)

vtime (scalar) = 26.01
vtime (vector) = 7.87
vvectime = 7.0

$$\text{program speedup} = \frac{\text{vtime (scalar)}}{\text{vtime (vector)}} = \frac{26.01}{7.87} = 3.3$$

$$Y = \text{vtime(scalar)} - (\text{vtime(vector)} - \text{vvectime})$$
$$= 26.01 - 7.87 + 7.0$$
$$= 25.14$$

$$\%\ \text{vectorizable} = \frac{Y}{\text{vtime(scalar)}} * 100$$

$$= \frac{25.14}{26.01} * 100$$

$$= 96.6\%$$

$$\text{vector speedup} = \frac{Y}{\text{vvectime}}$$

$$= \frac{25.14}{7.0}$$

$$= 3.59$$

Sample Hot Spot Analysis

```
FORTVS2 FTVECT
VS FORTRAN VERSION 2 ENTERED.  17:12:25

**FTVECT** END OF COMPILATION 1 ******

VS FORTRAN VERSION 2 EXITED.   17:12:25

READY;
Q TXTLIB
TXTLIB   = NPACKLIB VSF2FORT CMSLIB   TSOLIB
READY;
LOAD FTVECT
READY;
START (DEBUG
DMSLIO740I EXECUTION BEGINS...
AFF010I VS FORTRAN VERSION 2 RELEASE 3 INTERACTIVE DEBUG
AFF011I 5668-806 (C) COPYRIGHT IBM CORP. 1985, 1988
AFF013I LICENSED MATERIALS-PROPERTY OF IBM
AFF296E THE AFFON FILE CANNOT BE READ; FILE IGNORED.
AFF995I WHERE: FTVECT.5
AFF001A FORTIAD
ENDDEBUG SAMPLE(4)
I =  5000   D(I) =    -0.65   E(I) =    -0.95
I = 10000   D(I) =    -0.99   E(I) =     0.81
```

```
I = 15000   D(I) =    -0.85   E(I) =    -0.60
I = 20000   D(I) =    -0.31   E(I) =     0.32
SUM:              -8.36326027
DOT PRODUCT:     -0.393103242
MATRIX MULTIPLY:   587375.500
AFF306I PROGRAM HAS TERMINATED; RC ( 0)
AFF001A FORTIAD
LISTSAMP *.*
AFF550I PROGRAM SAMPLING INTERVAL WAS 4 MS; TOTAL NUMBER
OF SAMPLES WAS 42627.
AFF551I DIRECT SAMPLES:
AFF555I STATEMENT                    SAMPLES  %UNIT %TOTAL
AFF557I FTVECT.ENTRY/EXIT                  0   0.00   0.00
AFF557I FTVECT.5                           0   0.00   0.00
AFF557I FTVECT.6                           3   0.01   0.01
AFF557I FTVECT.7                          82   0.19   0.19
AFF557I FTVECT.8/10                        4   0.01   0.01
AFF557I FTVECT.9                           0   0.00   0.00
AFF557I FTVECT.10                          1   0.00   0.00
AFF557I FTVECT.11                         42   0.10   0.10
AFF557I FTVECT.12/15                       5   0.01   0.01
AFF557I FTVECT.13                          0   0.00   0.00
AFF557I FTVECT.14                          4   0.01   0.01
AFF557I FTVECT.15                          5   0.01   0.01
AFF557I FTVECT.16/20                       1   0.00   0.00
AFF557I FTVECT.17                          0   0.00   0.00
```

```
AFF557I FTVECT.18                      0    0.00    0.00
AFF557I FTVECT.19                      0    0.00    0.00
AFF557I FTVECT.20                      0    0.00    0.00
AFF557I FTVECT.21/30                   0    0.00    0.00
AFF557I FTVECT.22                      0    0.00    0.00
AFF557I FTVECT.23                      4    0.01    0.01
AFF557I FTVECT.24                   1268    2.98    2.97 *
AFF557I FTVECT.25                  37729   88.60   88.51
******************
AFF557I FTVECT.26/35                3421    8.03    8.03
**
AFF557I FTVECT.27                      0    0.00    0.00
AFF557I FTVECT.28                      4    0.01    0.01
AFF557I FTVECT.29/40                   0    0.00    0.00
AFF557I FTVECT.30                      0    0.00    0.00
AFF557I FTVECT.31                      5    0.01    0.01
AFF557I FTVECT.32/50                   7    0.02    0.02
AFF557I FTVECT.34                      0    0.00    0.00
AFF557I FTVECT.35                      0    0.00    0.00
AFF557I FTVECT.36/55                   0    0.00    0.00
AFF557I FTVECT.38                      0    0.00    0.00
AFF557I FTVECT.39                      0    0.00    0.00
AFF557I FTVECT.40                      0    0.00    0.00
AFF557I FTVECT.41                      0    0.00    0.00
AFF557I FTVECT.42                      0    0.00    0.00
AFF001A FORTIAD
```

```
QUIT
READY;
```

FTVECT LISTING

```
1LEVEL 2.3.0 (MAR  1988)        VS FORTRAN              JUN
12, 1989  17:12:25
PAGE:    1
OOPTIONS IN EFFECT: NOLIST NOMAP NOXREF NOGOSTMT NODECK
SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NOSYM NORENT
                    SDUMP(ISN) NOSXM NOVECTOR IL(DIM)
NOTEST NODC NOICA NODIRECTIVE NODBCS NOSAA
                         OPT(0) LANGLVL(77) NOFIPS
FLAG(I)  AUTODBL(NONE)  NAME(MAIN)    LINECOUNT(56)
CHARLEN(500)
0 IF DO   ISN
*....*...1.........2.........3.........4.........5.........6...
0           1         PROGRAM FTVECT
            2         PARAMETER (N=20000, M1=1200,
M2=175, M3=425)
            3         REAL*4 D(N), E(N), DOTPR, SUM
            4         REAL*4 A(M1,M2), B(M2,M3), C(M1,M3)

            5         DO 10 I=1,M1
      1     6           DO 10 J=1,M2
      2     7             A(I,J) = NINT(FLOAT(I-J))
      2     8    10   CONTINUE

            9         DO 15 I=1,M2
```

```
      1    10           DO 15 J=1,M3
      2    11             B(I,J) =
1.0/SQRT(FLOAT(I)/FLOAT(J))
      2    12    15   CONTINUE

           13         DO 20 I=1,N
      1    14           D(I) = SIN(FLOAT(I) / 2.0)
      1    15           E(I) = COS(FLOAT(I) * 2.0)
      1    16    20   CONTINUE

           17         DOTPR = 0.0
           18         SUM = 0.0

           19         DO 30 I=1,5
      1    20           SUM = SUM + D(I) / E(I)
      1    21    30   CONTINUE

           22         DO 35 I=1,M1
      1    23           DO 35 J=1,M3
      2    24             DO 35 K=1,M2
      3    25               C(I,J) = C(I,J) + A(I,K) *
B(K,J)
      3    26    35   CONTINUE

           27         DO 40 I=1,N
      1    28           DOTPR = DOTPR + (D(I) * E(I))
```

```
      1    29    40   CONTINUE

           30         DO 50 I=1,1200
      1    31           DO 50 J=1,200
                C          WRITE (10,51) C(I,J)
      2    32    50   CONTINUE
           33    51   FORMAT (F15.5)

           34         DO 55 I=5000, N, 5000
      1    35           WRITE (6,56) I, D(I), E(I)
      1    36    55   CONTINUE
           37    56   FORMAT(' I = ',I5,'   D(I) =
',F8.2,'   E(I) = ',F8.2)

1LEVEL 2.3.0 (MAR  1988)        VS FORTRAN              JUN
12, 1989  17:12:25  NAME:FTVECT
PAGE:    2
0 IF DO   ISN
*....*...1.........2.........3.........4.........5.........6....
0          38         WRITE (6,*) 'SUM:              ',SUM
           39         WRITE (6,*) 'DOT PRODUCT:
',DOTPR
           40         WRITE (6,*) 'MATRIX MULTIPLY:
',C(M1,M3)

           41         STOP
```

```
            42          END
0*STATISTICS*   SOURCE STATEMENTS = 42, PROGRAM SIZE =
3340064 BYTES, PROGRAM NAME = FTVECT    PAGE:    1.
0*STATISTICS*    NO DIAGNOSTICS GENERATED.
0**FTVECT** END OF COMPILATION 1 ******
TIME STAMP: 89.16317.12.25
```

The Stages of Vector Compilation

A DO loop must pass **four** stages of qualification before it can be compiled into vector instructions:

1. **ANALYSIS ELIGIBILITY STAGE 'UNAN'**

   the compiler determines whether or not the DO loop can be analyzed

2. **RECURRENCE DETECTION STAGE 'RECR'**

   loops are analyzed for data dependences that inhibit vectorization

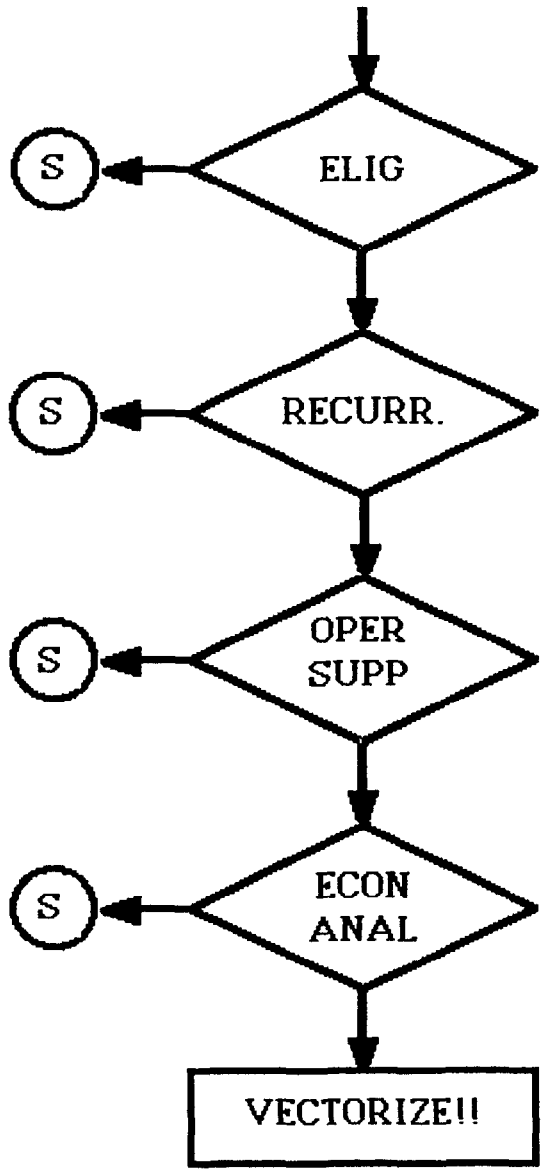
3. **OPERATIONS SUPPORT STAGE 'UNSP'**

   loops are checked for hardware and compiler support of all operations

4. **ECONOMIC ANALYSIS STAGE 'ELIG'**

   the compiler makes decisions about which loops to vectorize based upon whether scalar mode or vector mode is faster

The Stages of Vector Compilation (Graphically)



Terminology

Dependence

A dependence exists when the order in which statements are executed is important to the results of the program. Data dependencies are caused by multiple references to the same location in storage.

Indirect Addressing

The situation when the subscript of an array is itself an array element.

Induction Variable

any integer variable that is incremented or decremented by a fixed amount, such as the index of a DO loop. Induction variables other than the DO loop variables are called **auxiliary induction variables**.

Loop Distribution

the process of rewriting a DO loop into two or more smaller DO loops.

Recurrence

A data dependence that inhibits vectorization.

Scalar Expansion

a scalar variable that is replaced with a temporary vector.

Section Size

the number of elements used by the vector registers (128 on the 3090)

Statement Inhibitors

constructs for which no vector instructions exist or for which the compiler does not have the ability to generate the required instruction sequence.

Stride the interval between the data elements as they are fetched or stored by a program.

Vector a group of elements obtained by subscripting through an array.

Vector Inhibitors

constructs that restrict vectorization analysis either for entire loops or for individual statements.

Vector Length

the number of elements of an array that are referenced by a vector instruction. It may also be thought of as the number of iterations of a loop that is being vectorized.

Vector Report

- UNAN – rejected for vectorization analysis
- UNSP – unsupported for vectorization by the compiler or hardware
- RECR – ineligible for vectorization because of recurrence
- ELIG – eligible, but not chosen for vectorization
- VECT – vectorized

The Analysis Eligibility Stage

- DO loops are checked for operations which inhibit further analysis of the loop. Up to eight innermost levels of a nest of loops are analyzed.

- a loop will be flagged 'UNAN' if it contains any of the loop inhibitors:

    - loops other than DO loops

    - branches out of a loop, around an inner loop, or backwards within a loop

    - I/O statements

    - subroutine calls

    - external, non-intrinsic function references

    - ASSIGN, ENTRY, RETURN, PAUSE or STOP statements

    - computed or assigned GOTO statements

- DO loop parameters which are not INTEGER*4
- DO loop parameters which are in EQUIVALENCE statements
- character data
- comparisons of COMPLEX data
- loops with more than 8 nested levels

Loops Other Than DO Loops

- loops other than DO loops are not vectorized.
- recognize constructs which can be stated as DO loops and re-write them for vectorization.
- re-write this:

```
      I = 1
25    IF(I.GT.N) GOTO 26
      B(I) = X(I) ** A(I) * C
      I = I + 1
      GOTO 25
26    CONTINUE
```

as a DO loop:

```
      DO 25 I = 1,N
      B(I) = X(I) ** A(I) * C
25    CONTINUE
```

Branch Out of Loop (Pre-mature Exit)

- try distributing/re-structuring the loop. Re-write this:

```
UNAN+--------   DO 40 J = 1,N
    |             X(J) = Y(J) - Z(J)
    |             IF (X(J).LT.0.) GOTO 50
    |             ROOT(J) = SQRT(X(J))
    +-----40    CONTINUE
         50     JLAST = J - 1
```

as this (if it executes faster):

```
VECT+--------   DO 41 J = 1,N
    |
    +----- 41   TEMPX(J) = Y(J) - Z(J)

UNAN+--------   DO 42 J = 1,N
    |
    +----- 42   IF ( TEMPX(J).LT.0) GOTO 51

          51    JLAST = J - 1
                IF ( JLAST.EQ.0) GOTO 52
```

```
VECT+--------    DO 43 J = 1,JLAST
    |            X(J) = TEMPX(J)
    +----- 43    ROOT(J) = SQRT(X(J))

                 IF ( JLAST.EQ.N ) GOTO 53
           52    X(JLAST + 1) = TEMPX(JLAST + 1)
           53 CONTINUE
```

I/O Statements

- I/O statements are not analyzed by the compiler for vectorization
- move the I/O statement out of the loop
- Re-write this:

```
UNAN+--------   DO 30 I = 1,N
    |             A(I) = C(I) ** 2
    |             B(I) = C(I) ** 0.5
    |             WRITE (6,*) A(I),B(I)
    +-----30    CONTINUE
```

as this:

```
VECT+--------   DO 30 I = 1,N
    |           A(I) = C(I) ** 2
    |           B(I) = C(I) ** 0.5
    +----- 30   CONTINUE

UNAN+--------   DO 31 I = 1,N
    |           WRITE (6,*) A(I), B(I)
    +----- 31   CONTINUE
```

Subroutine Calls

- the compiler can't analyze a loop with a subroutine call, because vector inhibitors could be present in the subroutine. If a DO loop containing a subroutine call is a hotspot, try bringing the subroutine in line.

- Re-write this:

```
                    COMMON Y
                    . . . . .
UNAN+--------       DO 40 J = 1,N
    |                 X(J) = Y(J) - Z(J)
    |                 CALL SUB( J, X(J), Z(J))
    |                 . . . . .
    +-----40        CONTINUE
                      . . . . .
                    SUBROUTINE SUB( IND, A, B)
                    COMMON Y
                    Y(IND) = A + B
                      . . . . .
```

as this:

```
                        COMMON Y
                          . . . . .
VECT+--------           DO 40 J = 1,N
    |                   X(J) = Y(J) - Z(J)
    |                   Y(J) = X(J) + Z(J)
    |                     . . . . .
    +----- 40           CONTINUE
```

Recurrence Detection Stage

- a **data dependence** occurs when two statements (or iterations of the same statement) refer to the same data location

- some data dependences inhibit vectorization; they are called **recurrences**.

- a recurrence is flagged as 'RECR' on the XLIST-ing.

- by changing your code, it may be possible to eliminate a recurrence and vectorize the changed code.

Forms of Recurrence

- a reference in one iteration of a DO loop to an array element whose value was changed in an earlier iteration. For example,

```
      DO 100 I = 1,1000
      C(I+1) = C(I) * 3
100   CONTINUE
```

- an induction variable that modifies inner DO loop parameters. For example,

```
      DO 500 J = 1,1000
      DO 400 K = 1,J
        . . . . .
```

- any dependences that prevent interchanging the order of nested DO loops.

Operations Support Stage

Loops are checked for hardware and compiler support of all operations.

These operations PREVENT vectorization (loop inhibitors) and the loops containing them will be flagged as 'UNSP':

- data types

```
REAL*16
COMPLEX*32 (EXCEPT COMPARES)
LOGICAL*1
```

- any intrinsic functions with REAL*16 or COMPLEX*32 arguments

- INTEGER*2 governed by an IF statement

- relational expressions that need to be stored. For example,

```
L = A.GE.B
```

- intrinsic functions from the families: DIM, MOD, SIGN, NINT, ANINT, MAX, MIN
- non-inductive subscripts governed by an IF statement
- non-inductive subscripts to an INTEGER*2 array
- misaligned data
- IF statements with redundant parentheses
- any intrinsic function when the NONINTRINSIC option is specified

These Use the Vector Hardware

- data types

REAL*4	REAL*8
COMPLEX*8	COMPLEX*16
some INTEGER*2	
INTEGER*4	LOGICAL*4

- mathematical operations

REAL**REAL	DOUBLE**DOUBLE

- intrinsic functions

SQRT	DCOTAN	ABS	NOT	DPROD	DSQRT	ATAN
DABS	AIMAG	REAL	EXP	DATAN	IAND	DIMAG
DREAL	DEXP	CABS	IBCLR	AINT	SNGL	ALOG
CDABS	IBSET	DINT	DBLE	DLOG	ALOG10	IDINT
COMPLX	AMAX1	SIN	DLOG10	IEOR	DCOMPLX	DMAX1
DSIN	ATAN2	IFIX	CONJG	MAX1	DATAN2	COS
INT	DCONJG	AMIN1	DCOS	HFIX	IOR	FLOAT
DMIN1	DTAN	IABS	ISHFT	DFLOAT	MIN1	

These Do Not Use the Vector Hardware

- these operations and functions are evaluated using **scalar** routines. Their use in vector mode could slow down your program.
- mathematical operations

```
INTEGER ** INTEGER
INTEGER / INTEGER
REAL ** INTEGER
DOUBLE ** INTEGER
COMPLEX ** INTEGER
(DOUBLE COMPLEX) ** INTEGER
COMPLEX ** COMPLEX
(DOUBLE COMPLEX) ** (DOUBLE COMPLEX)
COMPLEX DIVIDE
DOUBLE COMPLEX DIVIDE
```

- intrinsic functions

ACOS	SINH	DGAMMA	CDCOS	CSQRT	DACOS	DSINH
ALGAMMA	CSIN	CDSQRT	ASIN	ERF	DLGAMA	CDSIN
IBCLR	DASIN	DERF	TANH	CEXP	IBSET	COTAN
ERFC	DTANH	CDEXP	ISHFT	COSH	DERFC	TAN
CLOG	DCOSH	GAMMA	CCOS	CDLOG		

Some Examples — Taking Advantage of the Vector Hardware

- indirect addressing to handle non-constant stride or randomly ordered elements. This is sometimes called **scatter/gather**.

```
VECT+--------   DO 10 J = 1,N
    |           B(J) = A(J) + P * C(J**2)
    +----- 10   CONTINUE

VECT+--------   DO 15 J = 1,N
    |           Y(J) = Z( IND(J))
    +----- 15   CONTINUE
```

- operations under mask.

```
VECT+--------   DO 20 J = 1,N
    |           IF( B(J).GT.XOLD ) THEN
    |              B(J) = A(J) + P * C(J)
    |           ENDIF
    +----- 20   CONTINUE
```

Non-Inductive Subscripts Governed by IF

- non-inductive subscripts governed by an IF prevent vectorization

```
UNSP+--------   DO 20 J = 1,N
    |           IF( B(J).GT.XOLD ) THEN
    |              B(J) = A(J) + P * C(J**2)
    |           ENDIF
    +----- 20   CONTINUE
```

```
    UNSP     THE ARRAY(S) "C" ARE USED IN
CONDITIONALLY EXECUTED
    CODE AND HAVE NON-INDUCTIVE SUBSCRIPT EXPRESSIONS
```

- recode with instructions which are supportable for vectorization:

```
VECT+--------   DO 20 J = 1,N
    |           CT = C(J**2)
    |           IF( B(J).GT.XOLD ) THEN
    |              B(J) = A(J) + P * CT
    |           ENDIF
    +----- 20   CONTINUE
```

The Economic Analysis Stage

- the Economic Analyzer is the name given to the code in the VS FORTRAN Version 2 compiler that estimates the number of cycles (cost) that will be expended to execute given sections of code.

- the choice of which regions to vectorize, if any, is based upon the calculations of the cycles for all possible combinations of nested loops (to a level of 8).

- 'ELIG' indicates that the loop was found eligible for vectorization, but has been chosen to run in scalar mode.

Example of Economic Analysis

Consider the following program involving integer divisions:

```
      INTEGER*4 K(100), J(100)
      REAL*4 X(100), Z(100)

      DO 30 I = 1,100
      J(I) = J(I)/K(I)
      Z(I) = K(I)/X(I)
30    CONTINUE

      STOP
      END
```

the Economic Analyzer determines the following,

```
      INTEGER*4 K(100), J(100)
      REAL*4 X(100), Z(100)

ELIG+--------   DO 30 I = 1,100   SCALAR FASTER
    +--------   J(I) = J(I)/K(I)  THAN VECTOR
```

```
VECT+--------   DO 30 I = 1,100
    +--------   Z(I) = K(I)/X(I)
                STOP
                END
```

ILX0148K 0004 ELIG CODE THAT WAS ELIGIBLE TO EXECUTE IN VECTOR MODE
WAS DETERMINED TO EXECUTE MORE EFFICIENTLY IN SCALAR.

Good Vector Programming Practices

- time your program so you know where to spend your efforts.
- check that your data and intrinsic functions can use the vector hardware.
- use ESSL whenever possible.
- try to eliminate vector inhibitors.

Succeeding in the Recurrence Detection Stage

- during the Recurrence Detection Stage the compiler REJECTS any DO loop for vectorization and flags it as 'RECR' if it contains:
    - an induction variable that modifies inner DO loop parameters
    - any dependencies that prevent loop interchange.
    - unbreakable recurrences
- the first two points have to do with the way outer loop vectorization is executed. No matter which loop is chosen as the vector loop, vectorization actually occurs at the innermost loop level, in sections of 128 (or fewer) data elements.

An Induction Variable that Modifies an Inner Loop Parameter

If an induction variable in an outer loop modifies an inner DO loop parameter, that outer loop cannot be moved to the innermost loop level. Therefore, vectorization cannot occur on that outer loop.

Consider, for example,

```
RECR +-------   DO 10 J = 1,100
VECT |+------   DO 10 K = 1,J
     ||         C(J,K) = A(J,K) * B(J,K)
     +------10  CONTINUE
```

Loop Interchange — Preventing Dependencies

If program results would change by moving an outer loop to the innermost level, vectorization is prohibited on the outer loop. This is called a **loop interchange preventing dependence**.

Consider the following two pieces of code, which differ only in their DO loop order:

```
      DO 15 I = 1,N                    DO 15 J = 1,M
      DO 15 J = 1,M                    DO 15 I = 1,N
15    A(I-1,J+1) = A(I,J)        15    A(I-1,J+1) = A(I,J)
```

their execution in scalar mode would be as follows:

```
A(0,2) = A(1,1)                  A(0,2) = A(1,1)
A(0,3) = A(1,2)                  A(1,2) = A(2,1)
A(1,2) = A(2,1)                  A(0,3) = A(1,2)
A(1,3) = A(2,2)                  A(1,3) = A(2,2)
```

Data element A(0,3) contains different values, depending upon the order of the DO loops. The outer DO loop cannot be moved to the innermost level, and therefore it cannot be vectorized.

What Exactly is a Recurrence?

- for example:

```
      DO 99 J = 1,100
      A(J+1) = A(J) + B(J)
99    CONTINUE
```

- results:

SCALAR EXECUTION	VECTOR EXECUTION
FETCH A(1)	FETCH A(1)
COMPUTE A(1) + B(1)	FETCH A(2)
STORE A(2)	ETC.
FETCH A(2)	COMPUTE A(1) + B(1)
COMPUTE A(2) + B(2)	COMPUTE A(2) + B(2)
STORE A(3)	ETC.
ETC.	STORE A(2)
	STORE A(3)
	ETC.

- note that in scalar execution, A(2) is stored before it is fetched.

- in vector execution, A(2) would be fetched before it is stored. The wrong value of A(2) would be used for the computation!

- vectorization is prohibited due to the recurrence on A.

- a recurrence is a data dependence which prevents vectorization.

Data Dependencies

- a **DEPENDENCE** exists when the order in which statements are executed may change the results of the program.

- data dependences are caused by multiple references to the same location in storage.

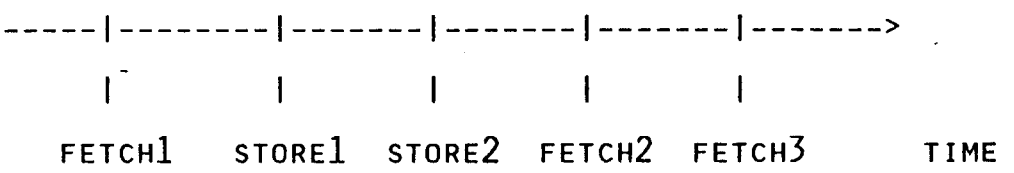
- a dependence occurs by:
  - the execution of successive statements or

  - the successive execution of a single statement during different iterations of a DO loop.

Data Dependences

- data dependences are caused by multiple references to the same location in storage.

- this is a time-shot of one storage location:



STORE FOLLOWED BY FETCH:	TRUE DEPENDENCE
FETCH FOLLOWED BY STORE:	ANTI-DEPENDENCE
STORE FOLLOWED BY STORE:	OUTPUT DEPENDENCE
FETCH FOLLOWED BY FETCH:	INPUT DEPENDENCE

- the recurrence analysis stage examines storage reference patterns. The order in which stores and fetches are done in scalar mode has to be maintained in vector mode.

True Dependences

- a true data dependence is a store to a memory location followed by a fetch from that location. Statement T depends upon statement S if S defines a value and T references it:

```
S: X =
T:   = X
```

- S must execute before T, because S defines a value used by T. The execution of T depends on the execution of S being completed.

- a single statement true dependence is of the form:

```
A(J+1) = . . .A(J). . .
```

- a single statement true dependence is a recurrence. It prevents vectorization.

Anti-Dependencies

- an anti-dependence is a fetch from a memory location followed by a store to that location. Statement T depends upon statement S if S references a value and T defines it:

```
S:    = X
T: X =
```

- S must execute before T because S must reference X before T redefines it.

- a single statement anti-dependence is of the form:

```
A(J-1) = . . .A(J). . .
```

Anti-Dependencies (cont.)

- for example,

```
VECT+--------   DO 30 J = 1,N
    |           A(J-1) = A(J) + B(J)
    +-------30  CONTINUE
```

-

SCALAR EXECUTION	VECTOR EXECUTION
FETCH A(1)	FETCH A(1)
COMPUTE A(1) + B(1)	FETCH A(2)
STORE A(0)	ETC.
FETCH A(2)	COMPUTE A(1) + B(1)
COMPUTE A(2) + B(2)	COMPUTE A(2) + B(2)
STORE A(1)	ETC.
ETC.	STORE A(0)
	STORE A(1)
	ETC.

- the order of fetches and stores is preserved in vector execution. A single statement anti-dependence WILL vectorize.

Single Statement Dependencies

a **true dependence** is a store to a memory location followed by a fetch.

a single statement true dependence is of the form:

```
RECR+--------   DO 10 J = 1,N
    |           A(J+1) = . . .A(J). . .
    +-------10  CONTINUE
```

a single statement true dependence WILL NOT vectorize.

an **anti-dependence** is a fetch from a memory location followed by a store.

a single statement anti-dependence is of the form:

```
VECT+--------   DO 10 J = 1,N
    |           A(J-1) = . . .A(J). . .
    +-------10  CONTINUE
```

a single statement anti-dependence WILL vectorize.

Multiple Statement Dependences

- a dependence can occur by the execution of successive statements.

- the compiler will consider all valid statement re-orderings within a loop when it does the recurrence analysis.

- the compiler examines the order of fetches and stores in a DO loop to determine whether it can safely vectorize the loop.

Multiple Statement Dependences (cont.)

EXAMPLE 1: an anti-dependence on A

```
VECT+--------   DO 30 J = 1,N
    |           A(J) = B(J) + C(J)
    |           E(J) = A(J+1)
    +-------30  CONTINUE
```

- the compiler will reorder the two statements and thereby preserve the order of fetches and stores on A! The loop WILL vectorize.

EXAMPLE 2: a true dependence on A and an anti-dependence on B

```
VECT+--------   DO 30 J = 1,N
    |           A(J+1) = B(J) + C(J)
    |           B(J) = A(J)
    +-------30  CONTINUE
```

- the compiler determines that the order of fetches and stores is preserved with vector execution and WILL vectorize the loop.

Multiple Statement Dependencies : Two Anti-Dependencies

- example:

```
RECR+--------   DO 30 J = 1,N
    |           A(J) = B(J) + C(J)
    |           B(J) = A(J+1)
    +-------30  CONTINUE
```

- scalar execution:

```
A(1) = B(1) + C(1)        FETCH B(1) AND STORE A(1)
B(1) = A(2)               FETCH A(2) AND STORE B(1)
A(2) = B(2) + C(2)        FETCH B(2) AND STORE A(2)
B(2) = A(3)               FETCH A(3) AND STORE B(2)
ETC.
```

- vector execution (1st attempt):

```
A(1) = B(1) + C(1)        FETCH B(1) AND STORE A(1)
A(2) = B(2) + C(2)        FETCH B(2) AND STORE A(2)
ETC.
B(1) = A(2)               FETCH A(2) AND STORE B(1)
B(2) = A(3)               FETCH A(3) AND STORE B(2)
ETC.
```

- the order of fetches and stores on A has changed!!

- vector execution (2nd attempt – re-ordered DO loop):

```
B(1) = A(2)              FETCH A(2) AND STORE B(1)
B(2) = A(3)              FETCH A(3) AND STORE B(2)
ETC.
A(1) = B(1) + C(1)       FETCH B(1) AND STORE A(1)
A(2) = B(2) + C(2)       FETCH B(2) AND STORE A(2)
ETC.
```

- the order of fetches and stores on B has changed!

- a forward and a backward anti-dependence form a cycle of dependences. This is a recurrence that prevents vectorization.

- however, a scalar temporary may be used to "break" this type of recurrence. This technique is known as **node splitting**.

Scalar Expansion

Scalar Expansion is the replacement of a scalar variable T by a vector temporary whose elements are all equal to the original scalar.

Some Rules:

- the scalar variable must be local to the loop in which it is used.
- it cannot use values defined before the loop. The first reference to T must be a store (i.e., T = ...).
- it cannot be used after the loop. The first reference to T after the loop, if any, must also be a store.
- it cannot be in COMMON or EQUIVALENCEd.

The Model:

```
      DO 30 J = 1,N
      . . . . .
      T = . . . . .
      . . . . .
      . . . . . = T
      . . . . .
30    CONTINUE
```

Node Splitting

- scalar temporaries can be used to break recurrences. This technique is known as **node splitting**.

- the compiler expands the scalar temporaries into vector temporaries.

-

```
   RE-WRITE THIS:                          AS THIS:
   --------------                          --------

RECR+--   DO 30 J = 1,N          VECT+--   DO 30 J = 1,N
   I      A(J) = B(J)+C(J)           I      T = B(J)+C(J)
   I      B(J) = A(J+1)              I      B(J) = A(J+1)
   +-30   CONTINUE                   I      A(J) = T
                                     +-30   CONTINUE
```

- scalar execution:

```
A(1) = B(1) + C(1)        FETCH B(1) AND STORE A(1)
B(1) = A(2)               FETCH A(2) AND STORE B(1)
A(2) = B(2) + C(2)        FETCH B(2) AND STORE A(2)
B(2) = A(3)               FETCH A(3) AND STORE B(2)
ETC.
```

Node Splitting (cont.)

•

RE-WRITE THIS:

```
RECR+--   DO 30 J = 1,N
   |      A(J) = B(J)+C(J)
   |      B(J) = A(J+1)
   +-30   CONTINUE
```

AS THIS:

```
VECT+--   DO 30 J = 1,N
   |      T = B(J)+C(J)
   |      B(J) = A(J+1)
   |      A(J) = T
   +-30   CONTINUE
```

- vector execution with node splitting:

```
T(1) = B(1) + C(1)          FETCH B(1)
T(2) = B(2) + C(2)          FETCH B(2)
ETC.
B(1) = A(2)                 FETCH A(2) AND STORE B(1)
B(2) = A(3)                 FETCH A(3) AND STORE B(2)
ETC.
A(1) = T(1)                 STORE A(1)
A(2) = T(2)                 STORE A(2)
```

- the order of fetches and stores has been preserved.

Partial Sums

- GIVEN:

```
                          SUM = 0.0
          VECT+--------   DO 30 J = 1,N
              |              SUM = SUM + A(J) * B(J)
              +-------30  CONTINUE
```

- the accumulation on SUM is called a reduction operation.
- SUM carries a recurrence: a single statement true dependence.
- there is a hardware solution called **partial sums** which works around this inherent recurrence.
- integer partial sums are not vectorized because they are faster in scalar. To allow the rest of a loop to vectorize, change to REAL*8.
- the order in which data elements are added using partial sums is not the same as scalar addition. Since floating point addition is not commutative, results are

slightly different in vector and scalar modes. To prevent vectorization, use the compiler option NOREDUCTION.

The Use of Scalar Temporaries

- accumulators should be scalar temporaries rather than array references since temporaries don't have to be stored.

- Re-write this:

```
VECT+--------   DO 15 I = 1,LEN
    |           DO 15 J = 1,LEN
    |           C(I,J) = 0.0
    |             DO 15 K = 1,LEN
    |             C(I,J) = C(I,J) + A(I,K) * B(K,J)
    +-----15    CONTINUE
```

as this:

```
VECT+--------   DO 15 I = 1,LEN
    |           DO 15 J = 1,LEN
    |           TEMP = 0.0
    |              DO 17 K = 1,LEN
    |              TEMP = TEMP + A(I,K) * B(K,J)
    |     17       CONTINUE
    |           C(I,J) = TEMP
```

```
      +-----15    CONTINUE
```

- or use this ESSL subroutine:

```
      CALL DGEMUL(A,LEN,'N',B,LEN,'N',C,LEN,LEN,LEN,LEN)
```

Summary on Recurrence

- Accurate recurrence detection requires that the compiler know as much as possible about the nature of subscript calculations for the array variables used within a loop. This requires information about:

  - the dimensionality of arrays
  - the parameters of the DO loops
  - expressions used to calculate the subscripts of each array reference

  If information about these factors is not available to the compiler, the optimum degree of vectorization may not be achieved.

- the compiler determines when it is safe to interchange loops, when it is safe to distribute a loop into multiple loops and when it is safe to reorder statements within a loop.

- if an outer loop cannot safely be moved to the innermost loop level, vectorization cannot occur on the outer loop.

- a single statement true dependence of the type

```
A(J+1) = ...A(J)...
```

  is a recurrence that prevents vectorization.

- a single statement anti-dependence of the type

```
A(J-1) = ...A(J)...
```

  vectorizes.

- if the compiler flags a loop with multiple statements as a recurrence, you can try introducing temporaries to break that recurrence.

- the compiler often cannot analyze complicated array subscripts, EQUIVALENCEd arrays, or arrays using indirect addressing. In such instances, the compiler may flag a loop as a recurrence, even though no recurrence occurs. You can override these "fake" recurrences with compiler directives, so long as you are sure that no recurrences actually occur.

Vector Compiler Directives

- compiler directives are used to override decisions made by the compiler and to give additional information to the compiler.

- there are three compiler directives:

    - ASSUME COUNT (n) : specifies a value that the compiler can use as an estimate for the iteration count of a loop

    - PREFER

        - VECTOR – specifies that a particular loop in a nest will be the best choice for a vector loop (if eligible)

        - SCALAR – specifies that a particular loop should not be chosen for vector execution

    - IGNORE

        - RECRDEPS – specifies that potential recurrences can be ignored in determining

eligibility for vectorization

- EQUDEPS – specifies that the compiler should assume that variables used in EQUIVALENCE statements do not give rise to recurrences

- ON and OFF keywords may be used with ASSUME COUNT and PREFER. Otherwise, a directive applies only to the DO loop immediately following it.

How To Use Vector Directives

- a directive is used with a so-called trigger-string, which is a character string defined by the user. Its purpose is to allow the compiler to distinguish a comment from a directive.

- the syntax of a vector compiler directive is :

```
Ctrigger-string keyword additional-information
```

C indicates a comment line and is immediately followed by the trigger-string. The keywords are ASSUME COUNT, PREFER and IGNORE.

- a directive is activated by the @PROCESS DIRECTIVE statement. The @PROCESS statement is placed before the first statement of EACH program unit (main program or subprogram) that uses a directive. The @ must be in column one.

- a directive can be treated as a comment by omitting the @PROCESS DIRECTIVE statement or by specifying @PROCESS NODIRECTIVE.

- each type of directive pertains to just one stage:

DIRECTIVE	STAGE
ASSUME COUNT	ECONOMIC ANALYSIS
PREFER	ECONOMIC ANALYSIS
IGNORE	RECURRENCE DETECTION

Directives : Where Added Information is Useful

use ASSUME COUNT for:

- unknown trip count:

```
DO 20 J = M,N,L          <--- HOW MANY ITERATIONS?
```

use PREFER for:

- overriding the compiler's economic decision: by timing your code, you might determine that the compiler made the wrong decision.

```
COMPLEX C,D              <--- COMPLEX DIVISION
DO 20 K = 1,N                 IS SLOW IN VECTOR MODE
D(K) = C(K) / D(K)
```

use IGNORE RECRDEPS for:

- unknown loop index upper bound : recurrence conditions may depend on its value.

```
DO 10 J = 1,N            <--- WHAT IS THE SIZE OF N?
A(J+50) = A(J) * B(J)    <--- RECURRENCE IF N > 50
```

- unknown DO increment : recurrences may depend on the direction of the increment.

```
DO 10 J = M,N,L          <--- WHAT IS THE SIGN OF L?
A(J-1) = A(J) + B(J)     <--- RECURRENCE IF L IS
                              NEGATIVE
```

Directives : Where Added Information is Useful

- unknown auxiliary induction variable : recurrences may depend on its value.

```
DO 10 J = 1,N
A(J) = A(K) * B(J)        <--- RECURRENCE IF K < J
K = K + M                 <--- WHAT ARE K AND M?
```

- unknown subscript offset : recurrences may depend on the value of the offset.

```
DO 10 J = 1,N
A(J+M) = A(J)             <--- RECURRENCE IF 0 < M < N
```

- when arrays are EQUIVALENCEd : the compiler always assumes dependence among equivalenced arrays.

```
EQUIVALENCE (A(50), B(1))
DO 10 J = 1,N             <--- NO RECURRENCE, BUT THE
A(J) = B(J)                    COMPILER THINKS THERE
                               IS!
```

- unknown indirect addressing subscripts :

```
DO 10 J = 1,N
A(J) = A(K(J)) + B(J)      <--- IS THERE A RECURRENCE?
A(K(J)) = A(K(J)) + B(J) <--- ARE THE A'S
                                INDEPENDENT?
```

Using Directives : Rules of Thumb

- use them for hotspots. Don't clutter your program where they are not needed.
- use ASSUME COUNT rather than PREFER where appropriate.
- double check to insure that IGNORE is used safely.

Summary : When to Use Directives

- the ASSUME COUNT and PREFER directives will not affect program results. Use them for:
  - unknown trip counts
  - vector loop selection
  - when the compiler makes the wrong economic decision

- make sure that you ARE outsmarting the compiler before you use PREFER.

- program results could change if you use IGNORE incorrectly. It can be used for:
  - unknown loop index upper bound
  - unknown DO increment
  - unknown DO auxiliary induction variable increment
  - unknown subscript offset
  - unknown equivalence-induced dependencies
  - unknown indirect-addressing dependencies

Poor Vector Performance

If vectorization gives poor performance gains, consider the following:

1. the storage reference pattern is poor (stride considerations)

2. the vector lengths are too short

3. there are too many IF statements

4. too many loop structures are inappropriate for vectorization

5. inefficient handling of sparse arrays

Stride Considerations

stride is a very important consideration for vector performance since arrays with small strides can be moved from virtual storage to vector registers and back much more efficiently than arrays with large strides.

the stride can be positive, negative or zero. For positive and negative strides, it is possible to specify vector elements beyond the range of an array thereby leading to unpredictable results and/or program errors.

Methods:

- data re-structuring – re-organize arrays to optimize stride

- data re-structuring using temporaries

Data Re-Structuring to Minimize Stride

since FORTRAN multi-dimensional arrays are stored in column-major form, the first subscript of an element always varies most rapidly and the last subscript always varies the least rapidly.

therefore, one way of minimizing stride is to insure that the dimension of an array that is the desired target for vectorization is the left-most dimension.

Given:

```
      PROGRAM STRIDE
      REAL*4 A(5,10,1000), B(5,10,1000)
      . . . . .
      DO 10 K = 1,1000
      DO 10 J = 1,10
      DO 10 I = 1,5
         A(I,J,K) = A(I,J,K) + B(I,J,K)
10    CONTINUE
```

re-write as:

```
      PROGRAM STRIDE
      REAL*4 A(1000,10,5), B(1000,10,5)
```

```
      . . . . .
      DO 10 K = 1,1000
      DO 10 J = 1,10
      DO 10 I = 1,5
         A(K,J,I) = A(K,J,I) + B(K,J,I)
10    CONTINUE
```

Vector Length Considerations

vectorization of a loop with a large vector length has a much greater payoff than vectorization of a short loop.

for very short loops, vectorization may result in poorer performance than scalar

Methods:

- use the ASSUME COUNT directive
- use dual path code
- create longer vectors through EQUIVALENCE, copying into temporary vectors, etc.
- eliminate loop unrolling

Dual Path Directives

if the loop count varies from small to large, depending upon your initial data, you could code a dual path to select scalar or vectorized loops.

for example:

```
@PROCESS DIRECTIVE ('*VDIR')
     . . . . .
     IF (N.LT.20) GOTO 30

C*VDIR  ASSUME COUNT (100)

     DO 10 K = 1,N
        COMPUTATIONS
10   CONTINUE
     GOTO 40

C*VDIR  ASSUME COUNT (5)
30   DO 11 K = 1,N
        COMPUTATIONS
11   CONTINUE
40   CONTINUE
```

Using Equivalence to Combine Multiple Dimensions

re-write this:

```
      DIMENSION A(10,8,9), B(10,8,9)
      . . . . .
      DO 99 I = 1,10
      DO 99 J = 1,8
      DO 99 K = 1,9
99    A(I,J,K) = A(I,J,K) + B(I,J,K)
```

as this:

```
      DIMENSION A(10,8,9), B(10,8,9)
      DIMENSION AA(80,9), BB(80,9)
      EQUIVALENCE (A(1,1,1), AA(1,1))
      EQUIVALENCE (B(1,1,1), BB(1,1))
      . . . . .
      DO 99 IJ = 1,80
      DO 99 K = 1,9
99    AA(IJ,K) = AA(IJ,K) + BB(IJ,K)
```

IF Statement Considerations

- a vectorized IF uses the vector mask register.
- all computations, for every iteration of the loop, are performed for every IF, THEN and ELSE clause.
- at the end of the loop, only the results corresponding to the correct IF conditions are stored, using the vector mask register.
- vectorized IFs perform well when there is no ELSE clause and the IF condition is usually true.
- because all computations are performed, a vectorized IF may result in divide-by-zero interrupts or subscripts out of range.

Methods for Dealing with IFs

- try eliminating the need for IFs.
- try moving IFs outside the vector loop.
- try using separate loops for each IF condition
- try creating temporary vectors containing values which satisfy the IF conditions. Do computations on the temporary vectors, then copy the results back to the original vectors.
- you might have to use the PREFER SCALAR directive if you determine that a loop containing IF statements is faster in scalar mode.

Eliminating IFs

This example shows how one might eliminate an IF whose purpose is to test for some boundary condition.

re-write this:

```
      DO 10 K = 1,N
      . . . . .
      DO 20 J = 1,M
      . . . . .
      IF ((J.EQ.1).OR.(J.EQ.M)) THEN
         X(J,K) = 0.
      ELSE
         X(J,K) = A(J,K) * B(J,K)
      ENDIF
      . . . . .
20    CONTINUE
      . . . . .
10    CONTINUE
```

as this:

```
      DO 10 K = 1,N
```

```
      . . . . .
      X(1,K) = 0.
      DO 20 J = 2,M-1
      . . . . .
      X(J,K) = A(J,K) * B(J,K)
      . . . . .
20    CONTINUE
      X(M,K) = 0.
      . . . . .
10    CONTINUE
```

Separate Loops for IFs

Generating an identity matrix can be handled like this:

re-write this:

```
      DO 10 I = 1,N
      DO 10 J = 1,N
      IF (I.EQ.J) THEN
         X(I,J) = 1.
      ELSE
         X(I,J) = 0.
      ENDIF
10    CONTINUE
```

as this:

```
      DO 10 I = 1,N
      DO 10 J = 1,N
         X(I,J) = 0.
10    CONTINUE
      DO 20 I = 1,N
         X(I,I) = 1.
20    CONTINUE
```

Inner vs. Outer Loop Considerations

- vectorizing a loop means that sectioning occurs on (according to) that loop's index.

- conceptually, this may be viewed as creating another loop at the innermost level.

- for example, this DO loop:

```
                    REAL*8 A(1000,100)
VECT+--------       DO 15 I = 1,1000
    |               DO 15 J = 1,100
    |               A(I,J). . .
    +-------15      CONTINUE
```

is treated by the compiler as:

```
        DO 15 I = 1,1000,128
        DO 15 I = 1,100
        . . .A(I:128,J). . .
15      CONTINUE
```

Inner vs. Outer Loop Considerations (cont.)

- the left-most array dimensions should have the largest values.
- with two-dimensional arrays, make the outer loop correspond to the left-most array subscript.
- for example, re-write this:

```
                                REAL*8 A(1000,100),
     B(1000,100)
                                REAL*8 X(1000)
                                DO 10 J = 1,100
                  VECT+-------- DO 10 I = 1,1000
                      |         A(I,J) = X(I) + B(I,J)
                      +-------10 CONTINUE
```

as this:

```
                  VECT+-------- DO 10 I = 1,1000
                      |         DO 10 J = 1,100
                      |         A(I,J) = X(I) + B(I,J)
                      +-------10 CONTINUE
```

- there is an advantage to OUTER loop vectorization if it reduces the number of times the vector X has to be loaded thereby optimizing vector register usage.

- the compiler will ordinarily vectorize on the left-most dimension.

Sparse Array Considerations

programs that deal with sparsely stored data can sometimes show a performance degradation when vectorized depending upon the methods used to manipulate the data.

Methods:

- indirect addressing
- compress and expand
- inhibit vectorization

Indirect Addressing

Given:

```
      SUBROUTINE SPARSE(MASK,A,B,C)
      LOGICAL*4 MASK(1000)
      REAL*4 A(1000), B(1000), C(1000)
      . . . . .
      DO 10 I = 1,1000
         IF (MASK(I)) THEN
             A(I) = B(I) + C(I)
         ENDIF
10    CONTINUE
```

Re-write as:

```
      SUBROUTINE SPARSE(MASK,A,B,C)
      LOGICAL*4 MASK(1000)
      REAL*4 A(1000), B(1000), C(1000)
      INTEGER*4 TCOUNT, INDX(1000)
      . . . . .
      TCOUNT = 0
      DO 9 I = 1,1000
         IF (MASK(I)) THEN
             TCOUNT = TCOUNT + 1
             INDX(TCOUNT) = 1
```

```
      ENDIF
 9    CONTINUE

      DO 10 I = 1,TCOUNT
            A(INDX(I)) = B(INDX(I)) + C(INDX(I))
10    CONTINUE
```

Interactive Vectorization Analysis (IVA)

Vector tuning can be assisted by gathering vector length and stride information at run time using IAD.

Before IAD can gather vector tuning information, you must create a Program Information File (PIF) by using the IVA suboption.

```
FORTVS2 FILENAME (OPT(3) VECTOR(IVA))
```

To collect and view the vector length and stride information, use the following IAD commands:

VECSTAT

activates recording of vector length and stride for all loops (VECSTAT *.* ON)

LISTVEC

displays average length and stride for vectors (actual vs. compiler estimates). (LISTVEC *.*)

Summary : Your Vector Migration Effort

- time your program

- local program modifications
  - ESSL calls
  - workarounds for vector inhibitors
  - reorder DO loops
  - use temporaries
  - vector directives

- global program restructuring
  - re-think program organization
  - re-think data organization
  - algorithmic changes

Set Expectations

- keep efforts focused on good payback potential: work with hotspots
- be realistic: remember that good vector program speed-ups are 1.5-3.
- analyze program performance:
  - program speed-up
  - percent vectorized
  - vector speed-up
- know when to quit!!

Test Case 1: Avoid Variable Offsets in Arrays

Given:

```
      SUBROUTINE TEST(A,N,IBASE1,IBASE2)
      REAL*4 A(1000)
      INTEGER*4 N,IBASE1,IBASE2
      . . . . .
      DO 10 J = 1,N
         A(I) = A(I+IBASE1) * A(I+IBASE2)
10    CONTINUE
```

Re-write:

```
      SUBROUTINE TEST(A0,A1,A2,N,ISIZE0,ISIZE1,ISIZE2)
      REAL*4 A0(ISIZE0), A1(ISIZE1), A2(ISIZE2)
      INTEGER*4 N,ISIZE0,ISIZE1,ISIZE2
      . . . . .
      DO 10 J = 1,N
         A0(I) = A1(I) * A2(I)
10    CONTINUE
```

Test Case 2: Avoid Indirect Addressing

Given:

```
      DO 10 I = 1,N
10       A(INDX(I)) = A(INDX(I)) + . . .
```

Re-write:

```
      DO 9 I = 1,N
9        TEMPA(I) = A(INDX(I))

      DO 10 I = 1,N
10       TEMPA(I) = TEMPA(I) + . . .

      DO 11 I = 1,N
11       A(INDX(I)) = TEMPA(I)
```

Test Case 3: Using Variable Increments

Given:

```
      IVAR = 1
      DO 10 I = 1,N
         A(IVAR) = A(IVAR) + . . .
         IVAR = IVAR + ISTEP
10    CONTINUE
```

Re-write:

```
@PROCESS DIRECTIVE('DIR')
      . . . . .
      IVAR = 1
*DIR  IGNORE RECRDEPS(A)
      DO 10 I = 1,N
         A(IVAR) = A(IVAR) + . . .
         IVAR = IVAR + ISTEP
10    CONTINUE
```

Test Case 4: Using Adjustably Dimensioned Arrays

Given:

```
      SUBROUTINE TEST(A,N,M)
      REAL*4 A(N,M)
      . . . . .
      DO 10 J = 1,M
      DO 10 I = 1,N
         A(I,J) = A(I,J) + . . .
   10 CONTINUE
```

Re-write:

```
@PROCESS DIRECTIVE('DIR')
      SUBROUTINE TEST(A,N,M)
      REAL*4 A(N,M)
      . . . . .
*DIR IGNORE RECRDEPS(A)
      DO 10 J = 1,M
      DO 10 I = 1,N
         A(I,J) = A(I,J) + . . .
   10 CONTINUE
```

Test Case 5: Array EQUIVALENCE

Given:

```
      SUBROUTINE TEST
      REAL*4 A(100), B(1000)
      EQUIVALENCE (A(1),B(101))
      . . . . .
      DO 10 I = 1,100
         A(I) = B(I) * 10.0
10    CONTINUE
```

Re-write as:

```
      SUBROUTINE TEST
      REAL*4 A(100), B(1000)
      EQUIVALENCE (A(1),B(101))
      . . . . .
      DO 10 I = 1,100
         B(I+100) = B(I) * 10.0
10    CONTINUE
```

or:

```
@PROCESS DIRECTIVE('DIR')
      SUBROUTINE TEST
```

```
      REAL*4 A(100), B(1000)
      EQUIVALENCE (A(1),B(101))
      . . . . .
*DIR  IGNORE RECRDEPS
      DO 10 I = 1,100
         A(I) = B(I) * 10.0
 10   CONTINUE
```

Test Case 6: Scalar EQUIVALENCE

Given:

```
      SUBROUTINE TEST
      REAL*4 A(100),B(100)
      INTEGER*4 PARAM,P1,P2,. . .
      COMMON /PCOM/ PARAM(10)
      EQUIVALENCE (PARAM(1),P1), (PARAM(2),P2),. . .
      . . . . .
      DO 10 I = 1,M
         A(P1) = A(P1) + B(I)
10    CONTINUE
```

Re-write:

```
      SUBROUTINE TEST
      REAL*4 A(100),B(100)
      INTEGER*4 PARAM,P1,P2,. . .
      COMMON /PCOM/ PARAM(10)
      P1 = PARAM(1)
      P2 = PARAM(2)
      . . . . .
      DO 10 I = 1,M
         A(I+P1) = A(I+P1) + B(I)
10    CONTINUE
```

```
      .  .  .  .  .
      PARAM(1) = P1
      PARAM(2) = P2
```

Test Case 7: Restrict Optimization to Improve Partial Vectorization

Given:

```
      SUBROUTINE TEST(A,B,X,Y)
      REAL*4 A(100),B(0:100),X(100),Y(100)
      . . . . .
      DO 10 I = 1,100
         A(I) = A(I) + X(I) * Y(I)
         B(I) = B(I-1) + X(I) * Y(I)
10    CONTINUE
```

Re-write:

```
      SUBROUTINE TEST(A,B,X,Y)
      REAL*4 A(100),B(0:100),X(100),Y(100)
      . . . . .
      DO 10 I = 1,100
1        A(I) = A(I) + X(I) * Y(I)
2        B(I) = B(I-1) + X(I) * Y(I)
10    CONTINUE
```

Test Case 8: Scalar Expansion for Partially Vectorizable Loops

Given:

```
      SUBROUTINE TEST(A,B,X,Y)
      REAL*4 A(100),B(0:100),X(100),Y(100)
      . . . . .
      DO 10 I = 1,100
         T = X(I) * Y(I)
         A(I) = A(I) + T
         B(I) = B(I-1) + T
   10 CONTINUE
```

Re-write:

```
      SUBROUTINE TEST(A,B,X,Y)
      REAL*4 A(100),B(0:100),X(100),Y(100)
      REAL*4 TT(100)
      . . . . .
      DO 10 I = 1,100
         TT(I) = X(I) * Y(I)
         A(I) = A(I) + TT(I)
         B(I) = B(I-1) + TT(I)
   10 CONTINUE
      T = TT(M)
```

Test Case 9: Scalar Expansion for Non-Local Scalars

Given:

```
      SUBROUTINE TEST(A,B)
      REAL*4 A(100),B(101)
      . . . . .
      T = B(1)
      DO 10 I = 1,100
         A(I) = T
         T = B(I+1)
10    CONTINUE
```

Re-write:

```
      SUBROUTINE TEST(A,B)
      REAL*4 A(100),B(101)
      REAL*4 TT(0:100)
      . . . . .
      TT(0) = B(1)
      DO 10 I = 1,100
         A(I) = TT(I-1)
         TT(I) = B(I+1)
10    CONTINUE
```

References

- VS FORTRAN Version 2 Programming Guide (SC26-4222)
- VS FORTRAN Version 2 Language and Library Reference (SC26-4221)
- Engineering and Scientific Subroutine Library Guide and Reference (SC26-0184)
- Designing and Writing FORTRAN Programs for Vector and Parallel Processing (SC23-0337)
- Vectorization and Vector Migration Techniques (SR20-4966)

Recommended Additional Reading

1. Agarwal, R., et.al., *New Scalar and vector elementary functions for the IBM System/370*, IBM Journal Research Development, Vol.30, No.2, March 1986

2. Clark, R.S., et.al., *Vector system performance of the IBM 3090*, IBM Systems Journal, Vol.25, No.1,1986

3. Ellersick, R., *Vector Coding Techniques for VS FORTRAN Version 2*, SHARE 68, March 1987

4. Ellersick, R., *Vector Coding Techniques for VS FORTRAN Version 2*, SHARE 69, August 1987

5. Ellersick, R., *Tuning Vector Programs with VS FORTRAN*, SEAS AM 88, September 1988

6. Lipps, H., *Introduction to Vector Processing*, CERN, date unknown

7. Liu, B. and Strother N., *Programming in VS Fortran on the IBM 3090 for Maximum Vector Performance*, IEEE Computer, June 1988

8. Metcalf, M., *Vectorization of HEP Programs*, CERN-DD/88/21, November 1988

9. Padegs, A., et.al., *The IBM System/370 Vector Architecture: Design Considerations*, IEEE Transactions on Computers, Vol.37, No.5, May 1988

10. Scarborough R., and Kolsky, H., *A vectorizing Fortran compiler*, IBM Journal Research Development, Vol.30, No.2, March 1986

Acknowledgements

Special acknowledgement must be given to the vector training material from the following sources:

- "Introduction To Vectorization," Cornell National Supercomputer Facility, Cornell University, Ithaca, New York, September 1988; written and edited by Helen Doerr and Francesca Verdier.

- "CERN Vector Processing Workshop," IBM European Center for Scientific and Engineering Computing, Via Giorgione 159, 00147 Rome, Italy, October 1988; particularly the material developed and written by Francesco Antonelli, Paolo Di Chio and David Soll.