INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION

INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
34100 TRIESTE (ITALY) - P.O.B. 586 - MIRAMARE - STRADA COSTIERA 11 - TELEPHONE: 2240-1
CABLE: CENTRATOM - TELEX 460392-1

SECOND SCHOOL ON ADVANCED TECHNIQUES
IN COMPUTATIONAL PHYSICS
(18 January - 12 February 1988)

SMR.282/6

FORTRAN OPTIMIZATION

M.METCALF
CERN, DD Division, Geneva, Switzerland

FORTRAN Optimization: PART I

The rules according to a famous guru:

1. Don't optimize.

2. If you must – do it tomorrow!

FORTRAN Optimization: Part II

In the real world, help is needed to exploit a principal feature of FORTRAN – its stress on object code efficiency.

This course is intended for those writing FORTRAN programs, or those responsible for running them, and should help in understanding:

- why we need to optimize
- the implications of the underlying hardware
- the implications of how compilers work
- the preparatory work (algorithm selection, profiling)
- machine-independent optimization
- use of an optimizing compiler
- machine specific considerations

and also:

- portability of FORTRAN programs
- long-term prospects of FORTRAN

The 3-fold view

The individual user:

- job class
- virtual machine limits
- real-time improvements for long-running jobs
- improved through-put
- huge applications become possible

The management:

- better use of (still) expensive resources
- relieve congestion

↳ provide: measurement tools
consultants

The user community:

- few users affect the many

But measure the cost balance!
short- and long-term.

VARIOUS ISSUES

1 Algorithm Search

MOST IMPORTANT SINGLE METHOD OF OPTIMIZATION

It is useless to optimize code well, if the algorithms being used are inherently slow.

Get the best, most robust algorithms for the job →

search
- literature
- private libraries
- published libraries
- ask colleagues

test
- whole range of data

refine
- don't stop after the first success

consider
- no of iterations
- convergence
- sensitivity of these to data (especially sort algorithms).
- appropriateness

Example

$$S = \sum_{i=1}^{n} (-1)^i \cdot i$$

Direct transcription

```
S = 0.
DO 1 I = 1,N
  S = S + (-1)**I*I
1 CONTINUE
```

costly exponentiation

Odd and even terms

```
S = 0.
DO 1 I = 1,N,2
  S = S - I
1 CONTINUE
DO 2 I = 2,N,2
  S = S + I
2 CONTINUE
```

← overflow?

costly conversion

20 times faster for $N = 10^6$

Pair terms $-1+2 \quad -3+4 \quad -5+6$

```
S = N/2
IF (MOD(N,2).EQ.1) S = S - FLOAT(N)
```

Independent of N (10^5 times faster)

Robust

No exponentiation

No repeated conversion

2 Profiling

Rule of thumb: 10% of code uses 90% of time

↳ For short programs – code first, optimize later

For long programs – plan optimized code from beginning

Tools: Static counts of operations

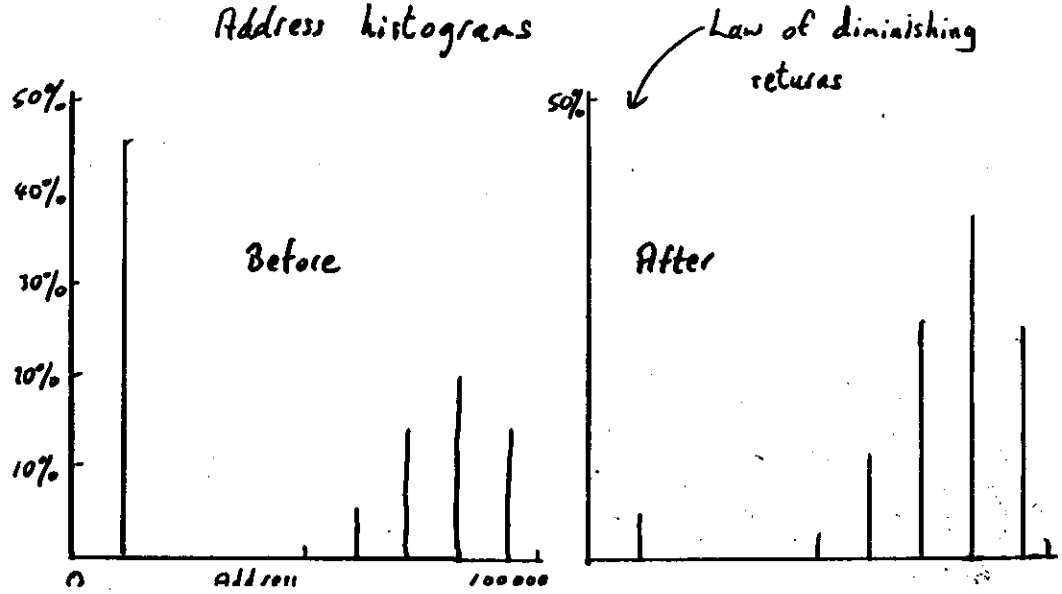
Dynamic counters

Clock routines

Statement counters

Subroutine timers

Address histograms



On the VAX: PCA

Performance and Coverage Analyzer

- collects performance data in a first step
- performs interactive post-processing of statistics in a second step

HELP PCA

On IBM VM/CMS:

GIME TESTSOFT

HELP LOGTIME

3 Clarity

- Clear code helps the compiler to optimize.
- Clear code helps the programmer to optimize.
- Optimization can lead to unclear code.

↳ is it worth it ?-

if so, use plenty of comments!

4 Portability

Use of
- local bit-handling features
- machine language
- fast I/O facilities

leads to non-portable code!

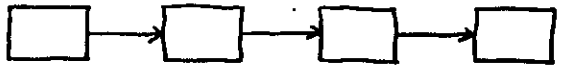
Use
- sparingly
- comment well
- provide standard FORTRAN equivalents

5 Space optimization

Memory is bigger now – but so are the problems!

- Use an optimizing compiler
  (optimized code has fewer instructions)
- In-line vs. external functions
- Scratch common, but dangerous!
- Backing store (use FORTRAN77 direct access)
- Compress large arrays
- Pack large array
- Dynamic memory manager: ZEBRA
  - provides also data structures not otherwise available in FORTRAN:

e.g. linked list:



- Overlays
- Segments

} to share memory between parts of a program (code + data).

MACHINE INDEPENDENT OPTIMIZATION

Advice, in principle, applicable to most processors – compiler + computer.

Initialization of variables

DATA statement more efficient than assignement

- no store
- no instruction
- no word for value

```
VALUE = 1.953
```

WARNING for large arrays – may have object code size problem: use fast copy etc.

BEWARE • interleaving

```
DATA (A(I), B(I), I=1,250)/500*0/
```

• unnecessary and dangerous initialization

```
COMMON /DATA/ A(100), B(100), C(5,20), I, J, K
CALL VZERO (A, 303) !
```

Arithmetic Operations

Basic, but not invariant, order: + −
*
/
**

so, try to reduce strength of operation:

```
(-1)**I* FUNC(I)
```

↓

```
SUM= 0.
DO 1 I = 1, N, 2
   SUM = SUM - FUNC(I) + FUNC(I+1)
1 CONTINUE
```

```
A = B/C/D/E
```

↓

```
A = B/(C*D*E)
```

and reduce their number, for instance by factorising:

```
A = B*(E+F) - C*(E+F) + D*(E+F)
```

↓

```
A = (B-C+D)*(E+F)
```

Also, avoid unnecessary temporary variables, which cost assignments:

```
A = B+C+D*E
```

and not

```
T1 = B+C
T2 = D*E
A = T1+T2
```

Speed of operation depends on data type (especially on machines with no floating-point hardware!).

Double precision sometimes costly (CDC), sometimes free (IBM).

In general:

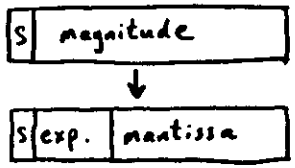
I+J, A+B, I*J, A*B	} fast
A/B	very slow
I/J	conversion
A**3	→ A*A*A
A**I	} function calls
A**B	

Mixed Mode Arithmetic

Type conversion in assignments of expressions may be

- implicit
```
A+I
B=J
```
- explicit
```
A+FLOAT(I)
B=FLOAT(J)
```

but is usually costly (up to 5 instructions):



To avoid:
- correct typing

```
IF (Y.EQ.4) J=10.
```
(4 → integer, 10. → conversion)

- duplicate variable name

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

- grouping

```
X= A+I+B+J+C+K          3 conversions
X= FLOAT(I+J+K)+A+B+C   1 conversion
```


Character Variables

Efficiency of implementation highly system dependent, but good on VAX (on IBM, CDC, function call for each operation or assignment). On IBM, character code does not participate in standard optimizations, even in loops! Nor is it vectorizable on the Vector Facility!

So
- choose your vendor carefully

then:
- avoid hidden overheads

```
CHARACTER*80 CARD
CARD = 'KEY'      77 redundant blank fills
```

- avoid repeated access

```
DO 1 I=1,80
   CARD(I:I)=' '
1 CONTINUE
   ⇓
CARD=' '
```

- avoid overuse of concatenation

```
LEN12=LEN(STR1//STR2)
   ⇓
LEN12=LEN(STR1)+LEN(STR2)
```

DO-loops

Where the biggest savings are normally made

Loop avoidance

initialization overhead:

1 no. of iterations
2 test for zero

3 store control variable

4 execute loop body

completion overhead:

5 decrement iteration counter and increment control variable
6 test for completion

```
      DO 1 I=1,2                  A(1)=0.
        A(I)=0.          ⟶        A(2)=0.
    1 CONTINUE
```

Loop nesting

```
      DO 1 I=1,100                initializations     1
        DO 2 J=1,10                                   100   } 1101
          DO 3 K=1,2                                  1000
          :
    3     CONTINUE                completion tests    2000
    2   CONTINUE                                      1000  } 3100
    1 CONTINUE                                        100
```

Could become: 23 and 2022 by reverse nesting →

longest loops deepest!

Loop parameters

Don't forget the third parameter:

```
      DO 1 I=1,10                      DO 1 I=7,34,3   ← more direct
        J=3*I+4          ⟶              X(I)=Y(I)+C
        X(J)=Y(J)+C                  1 CONTINUE
    1 CONTINUE
```

Loop manipulation

Unrolling:

```
      DO 1 I=1,N                       DO 1 I=2,N,2   ← beware end-effects
        A(I)=B(I)+C(I)       ⟶          A(I-1)=B(I-1)+C(I-1)
    1 CONTINUE                          A(I)=B(I)+C(I)
                                      1 CONTINUE
```

```
      DO 1 I=1,3                       DO 1 I=1,N
        DO 1 I=1,N                       X(1,I)=Y(1,I)+Z(1,I)
          X(I,J)=Y(I,J)+Z(I,J)   ⟶       X(2,I)=Y(2,I)+Z(2,I)
    1 CONTINUE                           X(3,I)=Y(3,I)+Z(3,I)
                                      1 CONTINUE
```

- Reduces overhead per iteration.
- Takes advantage of any pipelining.

Combining:

```
      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
```

→

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

Eliminates overheads.

Unswitching:

```
      DO 1 I = 1,100          <- 100 tests; may be compound.
        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
```

Eliminates invariant tests

```
      IF(FLAG) THEN           <- 1 test
        DO 1 I = 1,100
          A(I) = B(I) + 3. * D(I)
          B(I) = X * D(I)
    1   CONTINUE
      ELSE
        DO 2 J = 1,100
          A(J) = B(J) - 2. * C(J)
    2   CONTINUE
      ENDIF
```

[wrong on p. 81]

Elimination of Common Operations

```
      SUM = 0.
      DO 1 I = 1,N
        SUM = SUM + X * A(I)
    1 CONTINUE
```

→

```
      SUM = 0.
      DO 1 I = 1,N
        SUM = SUM + A(I)
    1 CONTINUE
      SUM = X * SUM
```

Saves operations

```
      DO 1 I = 1,10
        B(I) = A(I) * C + Q
        D(I) = A(I+1) * C
    1 CONTINUE
```

→

```
      T1 = A(1) * C
      DO 1 I = 1,10
        B(I) = T1 + Q
        T1 = A(I+1) * C
        D(I) = T1
    1 CONTINUE
```

Elimination of Redundant Tests

```
      DO 1 I = 1,N
        X(I) = A(I) ** 2
        Y(I) = -B(I) ** 2
        IF (X(I) .LT. 0) GO TO 1
        Z(I) = X(I)
    1 CONTINUE
```

Unnecessary test can be saved.

Invariant Code (Arrays)

```
      DO  1  M=1,N
      :
        DO  2  J=1,K
          DO 2  I=1,L
            R(I,J) = ....
            A(I,J) = ....
2     CONTINUE
      :
      :
        RN=0.
        AN=0.
        DO 3 J=1,K
          DO 3 I=1,L
            RN=RN+R(I,J)+R(I,J)/C(I,J)
            AN=AN+A(I,J)-A(I,J)*C(I,J)
3     CONTINUE
1     CONTINUE
```

Can save reciprocal of each term of C, saving a division.

Can extend principle to any expression involving invariant array elements.

Branches

Fastest
- GO TO 1
- ASSIGN 1 TO K
  :
  GO TO K, (1,2,3)
- IF (I.EQ.0) GO TO 2
- IF(I) 1, 2, 3

Slowest
- GO TO (1,2,3), I-2

↑ but faster than long sequence of logical-IF's.

Fastest form of arithmetic-IF:

```
      IF (I-J) 1, 2, 3
    1 .....
```

Fast to test against zero:

IF(I.EQ.1) GO TO 1 → IF(I.NE.0) GOTO1

(for I = 0, 1)

Sequence in most likely order:

```
IF (I.EQ.1) GO TO 1     90%
IF (I.EQ.2) GO TO 2      9%
IF (I.EQ.3) GO TO 3      1%
```

Reduces no of tests - true also for IF-THEN-ELSE.

Order within test similarly:

```
IF(I.EQ.1.OR.J.EQ.9.OR.K.EQ.15) GO TO 1
```

↑ most likely — ↑ least likely

```
IF(I.EQ.3.AND.M.NE.0.AND.L.EQ.10) GOTO 1
```

↑ most likely to fail — ↑ least likely to fail

Simplify:

```
IF(.NOT.(I.NE.K.OR.J.EQ.4.AND.L.ST.0)) GOTO 1
```

↓ De Morgan

```
IF(I.EQ.K.AND.(J.NE.4.OR.L.LE.0)) GOTO 1
```

- Saves evaluating whole logical expression.
- May break off sooner.

Avoid repeated tests:

```
IF(I.EQ.1) J=K+L
IF(I.EQ.1) L=0
IF(I.NE.1) J=K-J+N
```

→

```
IF(I.EQ.1) THEN
    J=K+L
    L=0
ELSE
    J=K-J+N
ENDIF
```

paired ↑ mutually exclusive

Calling Sequences

```
COMMON /BLOCK/ A,B,C,X
A=1.
B=2.
C=3.
CALL SUB
Y=X+4.
```

⟷

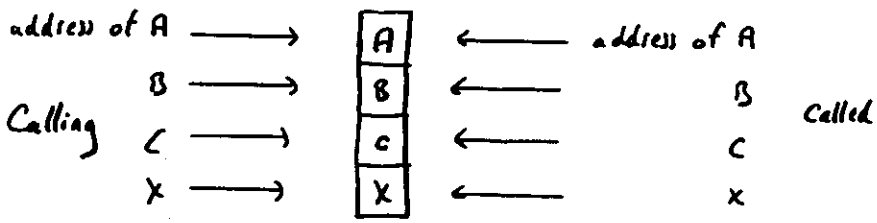
```
A=1.
B=2.
C=3.
CALL SUB(A,B,C,X)
Y=X+4.
```

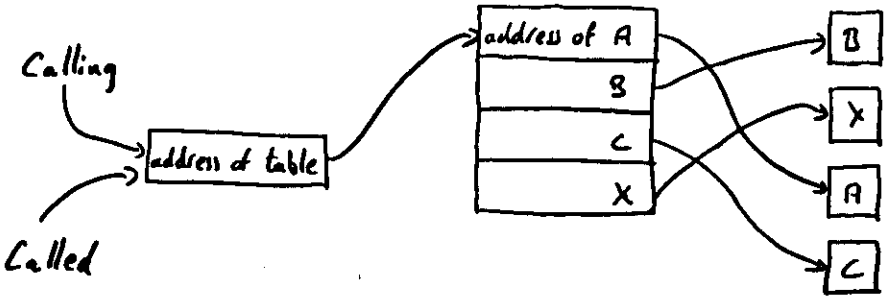
Normally, COMMON is faster:

- store table
- store pointer
- fetch pointer

} calling

called





Be careful with variable dimensions:

```
DIMENSION A(10,5)
CALL SUB (A,10)
      ⋮
SUBROUTINE SUB(A,N)
DIMENSION A(N,*)
```

address calculations based on N can be carried out only at execution-time!

Each CALL involves overheads in register saving and restoring → avoid CALL's in loops where they are comparable to the rest of the loop body:

Either pull the called code into the loop (in-line code),

or push the loop into the called routine.

but, conflict with good program structuring.

Functions

- One of FORTRAN's best structuring features.
- Same considerations as for subroutine calls.
- Ease of use can lead to loss of efficiency

- Use fast intrinsic functions generating in-line code:

  IF(I.LT.0) I=0 → I=MAX(I,0)

- Use statement functions to generate in-line code.
- Use explicit in-line code in loops.
- Use approximations:

  ROOT = 0.5*(1.+A) for A≈1.

- Use look-up tables calculated in advance. ← very important
- Use algebraic reduction:

  X=LOG(A)+LOG(B) → X=LOG(A*B)

- Avoid:

  IF(SQRT(X**2+Y**2).LT.R)... → IF(X**2+Y**2.LT.R**2)...

Input/Output Operations

Optimization can include significant real-time and storage utilization effects:

- for magnetic tape – density and blocking;
- choice between main memory and disc;
- whether to spool from tape to disc;
- pack/unpack v. execution overhead.

- Use unformatted rather than formatted data:

$BE1D7DBF_{16}$ $= -0.00045$ → (E7.1) → $-.5E-3$

↑ · loss of precision
· conversion overhead
· more space (often)

- Use efficient lists:

```
DIMENSION A(50),B(50), C(50), D(150)
EQUIVALENCE (A,D), (B,D(51)), (C,D(101))
WRITE (NOUT) A                                   3 calls
WRITE (NOUT) (A(I), I=1,50)                      -"-
WRITE (NOUT) (A(I),I=1,50),(B(I),I=1,50),(C(I),I=1,50)   5 calls
WRITE (NOUT) (A(I), B(I), C(I), I=1,50)          152 calls!
WRITE (NOUT) D                                   3 calls
```

- Use service routines for variable length arrays:

```
DIMENSION A(100)                    SUBROUTINE OUTPUT(A,L,*)
:                                   DIMENSION A(L)   Error on p.52
LEN = ....                          COMMON /IO/ NOUT, ...
CALL OUTPUT(A,LEN,*10)              WRITE(NOUT,ERR=100) A
:                                   RETURN
10 ERROR RECOVERY                   RETURN 1
....                                100 END
```

- Use temporary storage for non-contiguous elements:

e.g. · `WRITE (NOUT) (A(I), I=1,41,2)`

· `WRITE (NOUT) ((B(I,J), J=1,M), I=1,N)`

```
      DIMENSION B(5,10), C(10,5)
      DO 1 I=1,5
         DO 1 J=1,10
            C(J,I) = B(I,J)
1     CONTINUE
      WRITE (NOUT) C
```

- Avoid excessive use of BACKSPACE, REWIND.

  Often an application for direct-access files.

- Use explicit FORMAT specifications:

```
100 FORMAT (10I10)
or
    PARAMETER (F='(10I10)')        <- CHARACTER variable
or
    WRITE(NOUT,'(10I10)') LIST
not
    WRITE (NOUT,FORM(K)) LIST
```

- Use efficient FORMAT specifications:

```
1 FORMAT(1X,F10.3,1X,F10.3,1X...
1 FORMAT(10(1X,F10.3))

1 FORMAT (10F11.3)
   ^ implicit space
   explicit count
```

- Use asynchronous I/O operations (non-standard!).
  - ↳ • Read direct into memory (no buffer).
    • I/O concurrent with processing (need synchronisation).

Compiler-Dependent Optimization

Consider use of a 'typical' optimizing compiler:

- use highest optimization level
- switch-off debugging aids
- no traceback
- use in-line intrinsic function generation

Arithmetic Operations

Many compilers already optimize

X = Y/2	→	X = Y * 0.5
A**2	→	(A*A) * (A*A) * A
-A-B	→	-(A+B)
2*X	→	X+X

↳ Use explicit or symbolic constants (but latter preferable to former).

Common and constant sub-expressions

- Help the compiler to recognize them:

```
A = X*Y + 2.0 + SQRT(Y*X)  →  A= X*Y + 2.0 + SQRT(X*Y)

A = B-C          A = B-C
D = C-B    →     D = -(B-C)
                   ↑ use parentheses!
```

- Avoid 'hand-optimization':

```
IP1 = I+1
IM1 = I-1        ← temporaries + assignments
X = A(IP1,IM1) + A(IM1,IP1)  →  X = A(I+1,I-1) + A(I-1,I+1)
```

- Regroup

```
X = Y*Z/A        X = (Y/A)*Z
          →
T = Y*V/A        T = (Y/A)*V

Y = 180.*ACOS(V)/3.141  →  Y = (180./3.141)*ACOS(V)
                                ↑
                           compile-time evaluation!
```

DO-loops

- Nesting order

```
DIMENSION A(16,5)
DO 1  J = 1,5                    (J-1)*16+I  increment by 1
  DO 1  I = 1,16
1 CONTINUE A(I,J) = 0.           for reverse nesting increment by 16
```

In each case the subscript expression contains the increment of an induction variable and not the repeated evaluation of the whole expression, including the multiplication, BUT reverse nesting may:

- be less optimized, as constant 'one' may be available anyway, or be in an instruction
- conflict with bank interleaving

  use `m*(N±1)*n` as dimension (N = interleaving factor, n = no. of words per transfer)
- affect paging

- also: use conformable arrays to share subscript expressions:

```
DIMENSION A(10,20), B(10,20)
```
(even if problem needs only 9!)

- Don't confuse the compiler:

```
AA(2*I+2*(2*I+1)*10)  ->  A(2*(I+1), 2*(I+1))
```

↖ horrible substitution of a vector for an array

- Take care with invariant code:

```
      DO 1 J=1,10
        DO 1 I=1,10
          A(I,J) = C(J)*B(I,J)
1     CONTINUE
```

↖ let compiler do the work (efficiently!)

```
      DO 1 I=1,10                      make obvious
        B(I) = 2.*A(I)*Y     ->     B(I)=(2.*Y)*A(I)
1     CONTINUE
```

In general: $A(I,J) = \underbrace{2.0*PI}_{\text{constant}} * \underbrace{X*Y(J)}_{\text{invariant}} * \underbrace{B(I,J)}_{\text{variant}}$

e.g. `(A(I)-B(I))/S  ->  (1/S)*(A(I)-B(I))`

- Beware of de-optimization:

```
      DO 1 I=1,N
        DO 2 J=1,M
          IF(A(J,I).GE.0) GOTO 2
          A(J,I) = SQRT(FLOAT(I))+B(J,I)
2       CONTINUE
1     CONTINUE
```

↖ if A_{ji} normally ≥0, slower!

- Invariant code and external references

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

but `CALL SUB(A,Z+1.)` o.k.

Sometimes need to keep COMMON variable in local variable, to avoid register saving (but poor practice).

- Use scalar accumulators

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

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

- AVOID EXTERNAL REFENCES
  - use in-line code for functions
  - use statement functions for functions
  - don't leave in unused references:

```
IF(FLAG) PRINT *, I,J,K
```

↖ .FALSE.

  - don't include STOP or RETURN
  - push loop into called routine

Data Interference

```
      DIMENSION A(6)
      :
      X = A(2)
      A(I) = 1.
      Y = A(2)        ← must fetch again, as I may = 2


      K = N+1
      DO 10 J=2,N                    ← unrecognized invariant
         A(J) = A(K)/A(1)
10    CONTINUE
```

Often caused by EQUIVALENCE statement – avoid!

```
      COMMON // A(4), N, M, B, C(4)
      DIMENSION AA(11)
      EQUIVALENCE ( A, AA)
      :
      X = A(I)
      B = 1.          ← will B overwrite A(I)?
      Y = A(I)
```

NEVER equivalence:
- scalars to arrays
- loop parameters
- loop indices
- different data types