CFT77 Features Course
Product Set Training Group
Software Training Department
Cray Research, Inc.
Mendota Heights, MN.

SPECIAL MODULE: CFT AND CFT77 DIFFERENCES

Terminal Objective: Recognize product differences between the Cray Fortran compilers, CFT and CFT77, which bear on conversion, debugging and optimization tasks on CRAY X-MP computer systems.

Both Cray Fortran compilers, CFT and CFT77 are full ANSI 77 compilers. However, they differ in specific ways. If you are involved in code conversion between the two compilers, you should be able to identify these differences. Furthermore, where differences may mean that results vary, you need to know how to prevent possible problems.

This special module addresses those issues as they relate to:

* Compilation
* Invocation
* Compiler directives
* Extensions
* Error Detection Differences
* Special debugging considerations
* Optimization
* Vectorization

COMPILATION

CFT and CFT77 differ in compilation goal and process. In general terms, CFT compiles with the primary purpose of detecting vectorizable structures, whereas CFT77 approaches optimization from a broader perspective.

Specific differences follow.

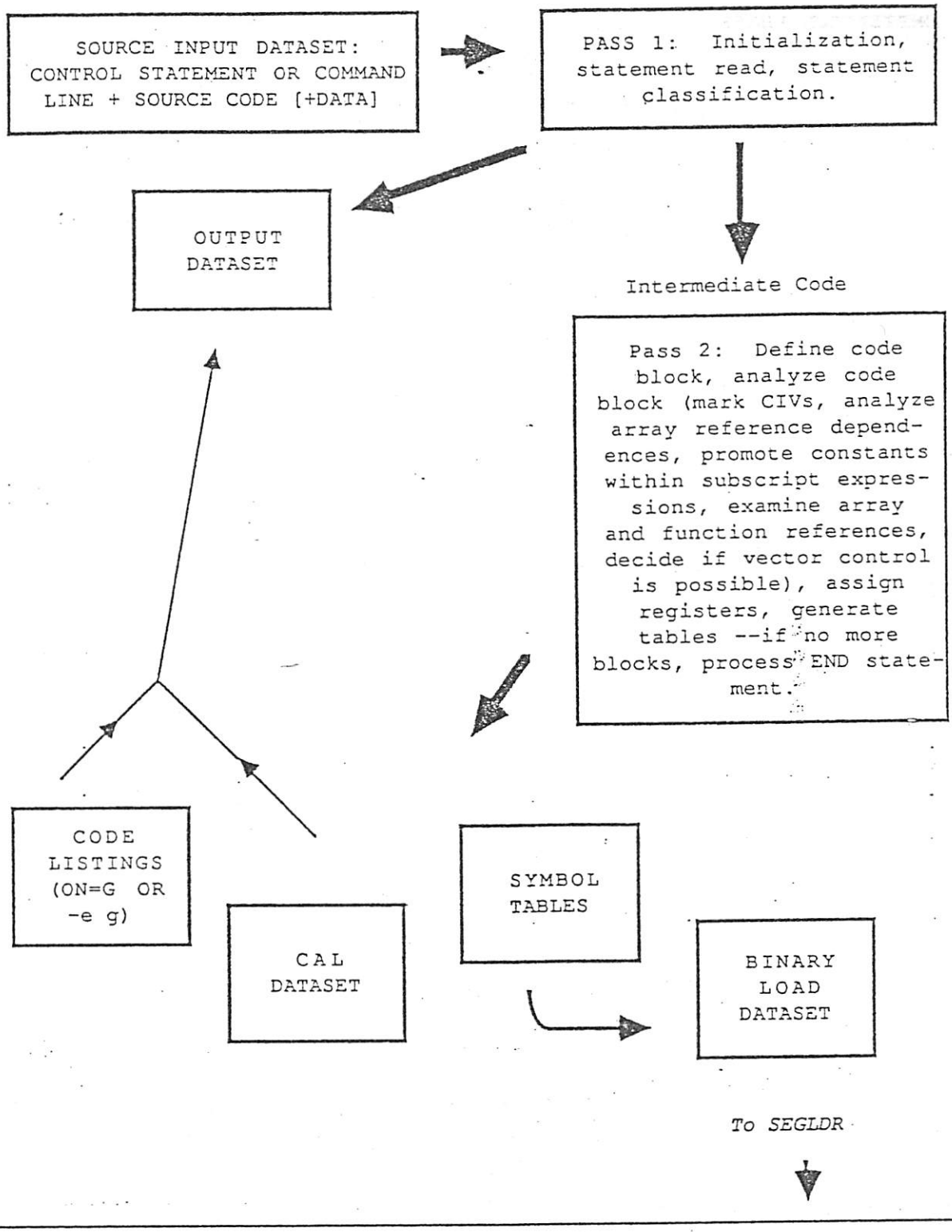
CFT

* Two passes: the first one classifies statements and outputs intermediate code, from which pass two defines code blocks. Code blocks (4000 words of intermediate code, by default) are analyzed for vectorization-geared optimization.
* Block: basic optimization unit; optimization information is retained on a per-block basis.
* Scalar optimization promotes constants, eliminates redundant expressions, hoists and bottom-loads.
* Compiler occupies approximately 200,000 words in the user field.

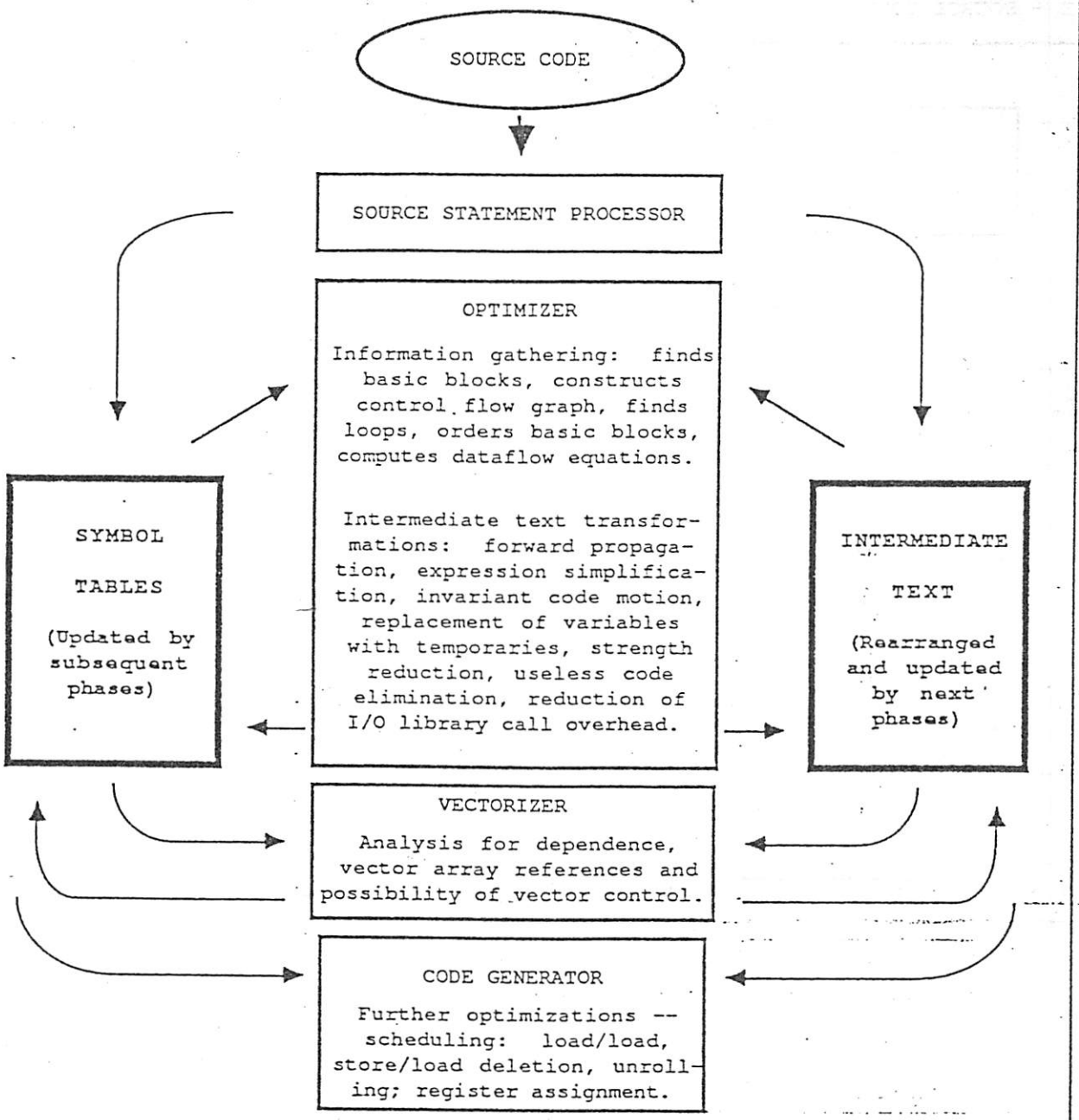
CFT77

* Four phases, each of which updates intermediate code and symbol tables. Phases do not share information directly.
* Dataflow analysis provides the compiler information about the program as a whole and allows it to distribute this information to each block in a flow graph.
* Optimization information retained for the whole program, although program units are optimized individually.
* Scalar optimization is independent from vectorization; vectorization can be skipped altogether. Scalar optimization promotes constants, simplifies algebraic expressions, removes dead code, and substitutes loop induction variables.
* Compiler starts out with 500,000 words in the user field.

CFT INPUT, PASSES and OUTPUT



CFT77 COMPILATION PHASES



INVOCATION

You invoke both the CFT and CFT77 compilers with the respective CFT and CFT77 control statements under the COS operating system, or with the cft and cft77 command lines under the UNICOS operating system. However, there are some differences in formatting and default values between the two compilers.

A specification of the differences follows. (Please note that you need to be familiar with the meaning of keywords on both statements or commands. For additional information, consult CRI publications SR-0018, CFT77 Reference Manual, and SR-0009, CFT Reference Manual.)

The contrast categories in this subsection are default values, specification differences, and compiler options.

Default Values

CFT 1.15	CFT77
COS CFT,	COS CFT77,
AIDS=LOOPPART, ALLOC=STATIC, B=$BLD, CPU=chars, E=3, I=$IN, INT=64 L=$OUT, MAXBLOCK=4000, OFF=ABDFGHIJNOWXZ, ON=CELPQRSTUV, OPT= NOZEROINC BL:NOBTREG:CVL: INVMOV:SLOWMD: KEEPTEMP: NOIFCON:SAFEIF: SAFEDOREP, TRUNC=0, UNROLL=3.	*No equivalent.* ALLOC=STATIC, B=$BLD, CPU=CRAY-XMP:NOCIGS,NOEMA:VPOP E=3, I= $IN, INTEGER=46, L=$OUT, *No equivalent.* OFF=ACFGHJOSX, ON=PQR, OPT=FULL:NOZEROINC, } *No equivalent.* TRUNC=0, *No equivalent.*

Default Values (Continued)

CFT 1.15	CFT77
UNICOS cft	UNICOS cft77
-a static	-a static
-b filename.o	-b filename.o
-d ACDEISabdfghijlnowxz	-d ADILSacfgjosx
-e BLcepqrstuv	-e Bpqr
-i 64	-i 46
-m 3	-m 3
-o nozeroinc, bl,btreg,cvl,slowmd, invmov,keeptemp, safedorep,noifcon, safeif	-o full,nozeroinc
	} *No equivalent*
-t 0	-t 0
	} *No equivalent*
	-C cray-xmp,nocig,noema, novpop

Specification Differences: Compiler and Operating System

CFT		COMMENTS	CFT77	
COS	UNICOS		COS	UNICOS
AIDS	-A	Specifies number of vectorization inhibition messages.	*	*
ANSI	-e A	Prints non ANSI messages at compile time. (Disabled by default.)	STANDARD	-e A

** No equivalent*

CFT		COMMENTS	CFT77	
COS	UNICOS		COS	UNICOS
CPU=	-C	Targets for code generation according to CPU characteristics.	CPU=	-c
DEBUG	-e D	Writes sequence number labels at each executable Fortran statement; necessary for the Symbolic Debugging Package.	DEBUG	-e D
E=	-e m	Error message level	E=	=m
EDN=	-E	Creates an alternate error listing.	*	*
INDEF	-e I	Initializes all stack variables to an indefinite value in CFT; in CFT77, initializes all variables in both static and stack allocation.	INDEF	-e I
*	*	Enables all available kinds of output listings.	LIST	-e L
LOOPMARK	-v	Brackets loops and provides vectorization information on source code listings.	*	*
*	*	Enables logfile messages.	*	-V
OPT=	-o	For CFT77 optimization can be fully active, completely disabled, or in scalar mode only.	OPT=	-o

** No equivalent*

Compiler Options

There are several compiler options available for CFT that CFT77 does not have. CFT77 recognizes CFT options that are not available for CFT77 when you use them, CFT77 issues a warning message.

Most of the CFT options unavailable from CFT77 are unnecessary due to its compiling and optimizing process. Some others are specified similarly for both compilers, but have a different effect, as indicated in the following summary.

COMPILER OPTIONS		
CFT		CFT77
COS	UNICOS	
B	b	No equivalent
D	d	Not available
E	e	No equivalent
I	i	Not available
L	l	No equivalent
N	n	Not available
O	o	Does NOT inhibit vector processing.
S	s	Disabled by default
T	t	No equivalent
U	u	No equivalent
V	v	Not available
W	w	No equivalent
X	x	Not available
Z	z	No equivalent

COMPILER DIRECTIVES

CFT provides several compiler directives that are not applicable to CFT77. When CFT77 finds them in source code it issues a warning message and ignores them.

There are yet other directives that have the same role in both compilers, but you have to specify them differently. Specific differences are:

1. You cannot continue compiler directive lines in CFT77.

2. In CFT77 control statement or command line options that you specify explicitly always have precedence over their corresponding compiler directives. For example, if you specify ON=S (-e s) CFT77 ignores CDIR$ NOLIST.

3. Both compilers support CDIR$ NOVECTOR, but CFT77 does NOT allow you to specify NOVECTOR=n.

4. Both compilers support bounds checking when ON=O (-e o) with compiler directives. However, when CFT77 processes a BOUNDS compiler directive it does not disable variables defined in previous BOUNDS compiler directives. The compiler disables bounds checking for all variables at the end of a program unit. You can disable bounds checking selectively with NOBOUNDS directives.

   In CFT each new BOUNDS directives produces bounds checking for all arrays not named in the directive to be disabled. CDIR$ BOUNDS() disables array bounds checking for all variables.

5. CFT77 supports CDIR$ SUPPRESS, which suppresses scalar optimization by creating a wall that optimization cannot cross. This effect makes it similar to CFT CDIR$ BLOCK. When you use SUPPRESS variables are not carried in registers across the directive; they are stored before the SUPPRESS and read from memory at their next reference. CFT77 also recomputes expressions at their next reference after a CDIR$ SUPPRESS.

The table that follows compares directives for both compilers.

COMPILER DIRECTIVE DIFFERENCES			
Directives	Compiler CFT	CFT77	Comments
ALIGN	X		
ALLOC=STACK/STATIC		X	
BL/NOBL	X		
BOUNDS/BOUNDS()	X	X	BOUNDS() becomes NOBOUNDS in CFT77
CODE/NOCODE	X	X	
CVL/NOCVL	X		
DEBUG/NODEBUG	X		
DYNAMIC	X	X	
EJECT	X	X	
FASTMD/SLOWMD	X		
FLOW/NOFLOW	X	X	
INT24/INT64	X		CFT77: INTEGER=64, INTEGER=46
IVDEP	X	X	(See previous page.)
IVDMO	X		
LIST/NOLIST	X	X	
NEXTSCALAR	X		
NO SIDE EFFECTS	X	X	

COMPILER DIRECTIVE DIFFERENCES (CONT.)			
Directives	Compiler CFT	CFT77	Comments
NODOREP/RESUMEDOREP	X		
NOIFCON/RESUMEIFCON	X		
NORECURRENCE	X		
ROLL/UNROLL	X		
SAFEIF	X		
SHORTLOOP	X	X	
SUPPRESS		X	Suppresses optimization; see BLOCK and previous page.
VECTOR/NOVECTOR	X	X	
VFUNCTION	X	X	

EXTENSIONS

The CFT77 compiler supports most of the CFT compiler's extensions to the ANSI 77 Fortran standard. CFT77 also extends the standard with features of its own. This section presents CFT77 extensions that CFT does not support and differences between extensions that both compilers provide.

CFT77 Extensions

1. Longer identifier names: CFT77 allows up to 31 characters for internal names; names may contain underscores.

2. Array syntax: CFT77 povides a subset of the array syntax proposed for the next Fortran standard. Array syntax allows you to operate on entire arrays or on large parts of arrays with a single statement; in fact, with array syntax you can do the same assignment operations for which you can use DO loops. Array syntax notation is also more readable.

   Examples:

```
DIMENSION A(1000),B(100),C(100),D(100,100)
CHARACTER * 4 CH(10)

A = S       ! Each element of A gets the value of the scalar S

A = B + C   ! A(i) = B(i) + C(i), i = 1 to 100

A = D(1, *)! A gets the first row of D

CH(1:5) (1:3) = 'XYZ'  !  Sets the first five elements of array CH,
                          character positions 1 through 3, to 'XYZ'
```

3. Automatic arrays: Arrays for which storage is allocated when a subprogram starts executing and released when the subprogram returns, which provides temporary arrays whose size is unknown until runtime.

   Example:

```
      SUBROUTINE CLOSURE A(N)
      DIMENSION A(N,N), NEW (N,N)
C  NEW is the same size as A, and is allocated when CLOSURE
C   starts up.
      NEW = A
      DO 10 I = 2, N
        CALL SUBRMULT (A,NEW,NEW,N)
10    CONTINUE
      A = NEW
      RETURN  !  Storage for NEW is released.
      END
```

Differences in Handling Extensions

1. CFT77 generates explicit calls to the system heap manager for automatic arrays, some array syntax statements, and some character concatenations, even when ALLOC = STATIC. CFT never calls the heap manager directly.

2. You can use POINTERs with both CFT and CFT77, but each compiler handles POINTERs differently. With CFT77 POINTER is a separate data type; with CFT a POINTER is an integer.

   CFT77 does not allow assignments to and from reals with POINTERs. You can add or subtract integers from pointers, and the result is an integer. With CFT77 typing a variable integer and then as a pointer produces a fatal error. For the example that follows CFT77 issues an error message, whereas CFT does not:

```
INTEGER P
POINTER (P,A)          ! Variable P is typed twice

POINTER (P,A)
P = P + 1.3            ! Addition of pointer variable and real
```

3. CFT77 allows non integer variables in an array bounds expression. The type of the expression must be integer.

4. IMPLICIT NONE applies to implied-DO variables in CFT77, but not in CFT. Furthermore, with IMPLICIT NONE the CFT77 compiler requires a function name that is declared external also to be declared in a type statement; CFT does not require this.

5. CFT77 does not allow assignments to a DO variable inside a DO loop. In CFT:

```
      N = 100
      DO 10 I = 10, 25, 3
         I = I * N
10 CONTINUE
```

   the iteration count for the loop will be 6, the final value for the loop index will be 28, and the value of I set equal to I * N will be 10030303030300 (100 * 10 + 3, 1003 * 10 + 3, ...). CFT77 flgas this loop as an error.

6. CFT77 does not support the IMPLICIT SKOL satement.

7. Character variables in CFT can be no longer than 16383; furthermore, no more than 511 different lengths of character strings may appear in a single program unit. CFT77 does not have this restriction.

8. Both CFT77 and CFT allow DATA statements to appear before specification statements. CFT77 has an additional restriction: If the type of a variable or array to be defined in a DATA statement is different from the default type, the type must be declared in a type or IMPLICIT statement, before the initial value is defined in a DATA statement.

ERROR DETECTION DIFFERENCES

Error detection at compile time is more extensive the CFT77 than with CFT. The cases summarized in this subsection are errors with both compilers, but are either handled differently or remain undiagnosed under CFT.

1. CFT77 gives an error message when extraneous parentheses appear in PARAMETER statements:

```
PARAMETER (list), (list), ...
```

   is an error for CFT77. CFT does not flag this as an error.

2. CFT77 does not permit extra sets of parentheses around the I/O lists in I/O statements or the implied DO list of array element names in DATA statements. CFT does not detect the error. These examples all have illegal extra parentheses:

```
WRITE (6,12) (A,B)
WRITE (6,12) ((A(I),B(I)), I = 1, 10)
ENCODE (N,12,A) (B,C)
DATA ((A(I),B(I)), I = 1, 10)
```

3. CFT77 issues an error message when you use an array as a statement function dummy argument; CFT does not flag this use. CFT77 will not allow you to use an array name as an actual argument to a statement function: you must use an array element --CFT does not

have this restriction.

4. CFT77 does not allow the use of a logical third argument to intrinsic CVMG functions, whereas CFT does.

5. CFT77 issues an error message when you use values other than real or integer arguments for CLOCK, JDATE, and DATE functions; CFT does not flag this.

6. CFT77 gives an error when you use a name as a function and later as a subroutine, and vice versa; CFT does not issue an error message.

7. Although both compilers ignore REC= or IOSTAT= specifiers in a NAMELIST I/O statement, CFT77 issues a warning message as well. Moreover, CFT77 produces an error message if it detects a NAMELIST group name in the FMT= specifier of a read or write control item list.

8. If you use a statement label on an ELSE statement in CFT77, you receive an error message. CFT produces a warning message and ignores the label.

9. CFT77 produces an error if there is no EXTERNAL statement for a dummy procedure that is passed as an actual argument. CFT does not detect this as an error if you use the name in a CALL statement or as a function reference before you use the name as an actual argument.

10. CFT77 requires that a dummy argument used in an adjustable array bound appear in every dummy argument list in the program unit that contains the array name. CFT does not require this.

11. CFT77 produces an error message for an implied-DO variable in a DATA statement that is not integer; this is not an error for CFT.

12. CFT77 does not allow an out-of-bounds array reference in an EQUIVALENCE statement and gives an error message. (An exception to this is that CFT77 allows an out-of-bounds reference if the array has more than one dimension, but is subscripted as a one-dimensional array.) CFT produces a warning message unless the out-of-bounds subscript expression is less than the lower bound for the dimension, in which case CFT does issue an error message.

13. CFT77 does not let you attempt to dimension a variable that has appeared previously as a scalar in a DATA statement. CFT does not have this restriction.

14. CFT77 detects PARAMETER, DATA, or FORMAT statements that appear before a SUBROUTINE or FUNCTION statement in a program unit, whereas CFT does not detect this.

15. CFT77 detects illegal mixing of logical with other types in masking expressions or as arguments to logical intrinsics, but CFT does not.

This use is invalid with both compilers, because the bit representation used for logical is not guaranteed and may change between releases or machines. In fact, the internal representation for logical is different from the CRAY-2 to the CRAY X-MP.

16. CFT77 issues a CAUTION message when you type an intrinsic function explicitly and that typing differs from the intrinsic's type. CFT issues a CAUTION message whenver you type an intrinsic function explicitly.

17. CFT77 does not allow you to assign different lengths to variables in a type statement for non character types. CFT allows this. In CFT77, for example,

```
COMPLEX * 8X, Y * 16
```

produces an error.

18. When you dimension a character array in a CHARACTER statement with

*"name * length (dimensions)"*

CFT77 issues an error message, while CFT does not.

19. CFT77 does not allow an ENTRY name in a subroutine to appear in a type statement, but CFT allows it.

20. CFT77 only allows integer and Boolean expressions in computed GOTO statements; the compiler issues an error message if you use any other type. CFT allows the use of other arithmetic type and allows the use of character and logical types.

21. CFT77 does not allow the use of an external name as an argument to the LOC function, while CFT does not flag this. Making LOC external lets you work around this restriction if you really need to.

22. If you specify END= on a random access READ statement, CFT77 produces an error message. With CFT the library routine ignores the specifier.

23. The CFT77 optimizer collapses (folds) constant expressions at compile time to produce constant results. Therefore, the computation time for constants takes place at compile time, not at run time. As the compiler collapses the constant expression, it checks to detects operations that would create hardware or library-call error conditions. The compiler issues error messages if it detects any such condition, such as division by zero, illegal exponentiation arguments to system library routines, and arithmetic operations for REAL constants that would generate hardware floating-point errors.

SPECIAL DEBUGGING CONSIDERATIONS

When you convert code from CFT to CFT77 you may find that results vary in some cases. There are some ideas you may wish to consider when faced with these situations. This subsection presents useful reminders and sources of numerical differences between CFT and CFT77, with examples.

Useful Reminders

The topics under this heading identify specific points you need to keep in mind when you debug programs written for CFT and converted to CFT77.

Symbolic Debugging

A call to the UNICOS or COS operating system postmortem symbolic debugger, DEBUG, dumps variables in memory; a call to the symbolic debugger SYMDEBUG interprets the memory of the running program. Additionallly, DRD, the Dynamic Runtime Debugger allows you to debug executing programs either interactively or in batch mode when you call it on the SEGLDR control statement or segldr command line.

The CFT77 compiler's optimization techniques, however, include symbol replacement in cases where the compiler detects redundancy and dead code, as well as heavy use of register storage where memory references are not necessary. It is reasonable to see, for example, only one variable assigned to memory out of 100 used in program. If you dump symbols in memory for such a program, all you can expect to find with the symbolic debugger is the current value for one variable.

In general, CFT77,OPT=OFF or cft77 -o off disables the mechanisms that can make symbolic debugging ineffective. When you disable optimization for symbolic debugging, variables are more likely to be where the debugger indicates them to be.

Register Location

Some optimizing compilers allow you to assign all variables to particular types of registers for the program's life. This simplifies debugging and reduces scalar memory references.

The CFT77 compiler follows a different optimization philosophy. Variables move wherever the program needs them, and their location assignment responds to whatever is more efficient: your intervention is not required to force the choice.

With such patterns, value-tracking can become complex for you. Hence, following variables on the generated-code listing (from cft77 -e g or CFT77,ON=G) may be a challenge. The listing's format for CFT77 is more clearly structured than for CFT; its contents, however, require careful analysis.

Binary Searches

A binary search on the whole program or on a subroutine is a viable debugging technique with the CFT77 compiler.

However, if you are used to binary searches under the CFT compiler, you must modify your search approach. Generally, users who try binary searches on CFT programs separate the source code with end-of-file statements to force different source files. As of the CFT77 compiler's 1.3 release, its compilation time is close to seven times longer than that for the CFT compiler. Separating the source code into different files increases that compilation time factor.

CDIR$ SUPPRESS

Some CFT compiler uses rely on compiler directive BLOCK to isolate sections of code for optimization and debugging. Since the block is the basic compilation unit for the CFT compiler, the user has the option of expanding or reducing the size of blocks with CDIR$ BLOCK. This directive creates a "wall" that the optimizer cannot move. Also, CDIR$ BLOCK sets up a barrier in which all variables are stored in memory.

The CFT77 compiler optimizes code differently. It optimizes modularly, through blocks rather than from block to block (as the CFT compiler does). Nonetheless, you can force it to look at blocked-out segments of code for debugging, especially when the problem may be due to differences in order of evaluation.

The CFT77 compiler provides a functional alternative with CDIR$ SUPPRESS. This directive suppresses optimization by fooling the compiler into interpreting that values could have been redefined. SUPPRESS forces variables in registers to go to memory, from where the variables are read out the next time there is a reference to them. To the CFT77 compiler CDIR$ SUPPRESS is equivalent to a subroutine call with an argument list which contains every variable in the calling program.

Consider, for example, the effect of CDIR$ SUPPRESS in the following subroutine:

```
      SUBROUTINE ORIGBUG (X,Y)
      COMMON /PARA/ OCT90,PCT10,PCT50,ALFA,TON,
     * XBAR,VAR,SDEV,CURT,SKEW,X16,X50,X84
      DIMENSION P(2), X(Y)
      M = (Y+1) / 2
      N2 = Y / 2
      IF(M.EQ.N2) GO TO 10
        PCT50 = X(N2 + 1)
        GO TO 20
```

```
10     PCT50 = (X(N2) + X(N2) + 1)) / 2.00
20     XN = Y
       X10 = 0.10 * XN
       DO 30 I = 1, 2
         M = X10
         Y = X10 - FLOAT(M)
         P(I) = X(M) + (X(M + 1) - X(M))
         X10 = 9.0 * X10
30     CONTINUE
CDIR$  SUPPRESS
       PCT10 = P(1)
       PCT90 = P(2)
       ALFA = (PCT50 * PCT50 - PCT10 * PCT90) /
      *  PCT10 + PCT90 - 2.0 * PCT50
C
C   Do NOT use SUPPRESS if IF or GOTO
C   sequences such as:
C
       IF ((X(1) + ALFA).GT.0.00) RETURN
       ALFA = -0.998 * X(1)
       RETURN
       END
```

SUPPRESS looks liek a call to the calling program. The otpimizer assumes that all variables are stored to memory before SUPPRESS, because they are in the argument list. The compiler has to dump every variable, because it must assume that the referenced routine might be looking at it, and after the call the compiler has to refresh all of its copies in memory (Fortran standards specify that the called routine may have altered the values in the common block).

To use SUPPRESS, remember that it is enabled at the point where it appears, but does not go further than that point. Also, the directive must be on an execution path; including it in a conditional sequence of code cancels out its effect.

Sources of Numerical Differences

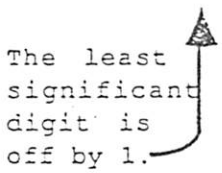
This subsection presents five potential sources of numerical differences in results from CFT77 and CFT: cancellation, rounding, strength reduction in exponentiation, order of evaluation and truncation.

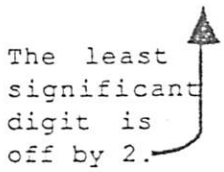
Cancellation

Numeric differences in results can be a symptom of cancellation in operations on approximate values. These differences occur in subtraction of "nearly" equal values or in addition of values with opposite signs and nearly equal absolute values.

Consider, for example, the following subtractions with a nine-digit mantissa and a two-digit exponent on a decimal machine:

```
  1.23456789E+42          1.23456789E+42
- 1.23456788E+42        - 1.23456787E+42
----------------        ----------------
```





When you subtract these, what was a difference in the least significant digit becomes a difference in the most significant digit:

```
  1.23456789E+42          1.23456789E+42
- 1.23456788E+42        - 1.23456787E+42
----------------        ----------------
  1.00000000E+34          2.00000000E+34
```

A difference such as the one the preceding examples create leave you with one or two bits of precision. However, the rest of the program continues to assume it has 48 bits of precision to work with, and proceeds to compute with nonsense.

Similarly, consider the following common loop:

```
      DO 10 I = 1, N

10    DIFFSQ(I) = (X(1) - Y(I)) ** 2
```

Assume the loop to be in a subroutine that has a vector X of known data points. There is an algorithm which tries to compute some parameters; the parameters will allow prediction of values from the value of I.

The program comes up with a vector of actual and predicted data

values. Let's say that the goal is to compute an error function on it. The loop subtracts the prediction from the actual values and squares the results. If the curve fits at all, these numbers should be almost identical. There is nothing uncommon about the computation that produces the values in DIFFSQ; the use of the values in the small array elsewhere, however, can cause problems if other computations build on those values' assumed precision --which machine arithmetic cannot provide. The slight difference between the values is what causes problems. The program could assume 48 bits if precision, whereas precision actually could be of two bits.

Rounding

The way in which CRAY X-MP computer systems handle division influences "strong rounding." Real division in an integer context is particularly sensitive to this process.

In a case such as the following:

```
I = X / Y
```

because CRAY machines do not have divide functional units, X/Y evaluation is as X*(1.0/Y). The reciprocal approximation tends to be too small; therefore, X/Y tends to be too small. Real division in an integer context, such as in I = X/Y, presents situations of concern to CFT77 users converting from CFT.

CFT computes X/Y in an integer context as:

```
X * (1.000 ... 010 ) * (1.0 / Y)
                    2
```

CFT multiplies whenever it finds a real division in a calculation such as I = X/Y (assuming implicit typing). CFT multiplies the numerator by a base 2 factor of 1.000.010, with 10 at the rightmost end, to make the numerator a little larger. Therefore, it attempts to compute a value that is high for the division, instead of a value that turns out too low.

The CFT77 compiler, on the other hand, handles the division as:

```
X * (1.0 / Y)
```

CFT77 takes the reciprocal approximation of the denominator, multiplies it by the numerator, and produces the answer.

Assume, for instance, that X = 6.0 and Y = 3.0. By multiplying 6.0 by the reciprocal approximation of 3.0 you get a mantissa of one bit, which in decimal is 1.9999... (or $1.111 \ldots 111_2$), not 2. In CFT77 this means

that for:

```
I = X / Y

PRINT *, I
```

CFT77 prints 1 (Fortran does not allow conversion to "almost" 2).

Assuming you have a sequence such as:

```
      X = 6.0
      Y = 3.0
      DO 10 J = 1, 59
        I = X/Y
        K = I + J
        A(J) = A(K)
10    CONTINUE
```

under CFT77, K goes from 2 to 60, and I on exit is 1. Under CFT, K goes from 3 to 61 and I is 2 on exit.

The best way to handle this under the CFT77 compiler is with a call to function NINT, which rounds to the nearest integer. For I on I=X/Y above, the result of NINT(X/Y) is 2.

Whenever the problem is possible, a compiler message warns you.

Strength Reduction

Strength reduction in exponentiation can produce numerical differences under the CFT77 compiler.

In a case such as:

```
      DO 10 I = 1, N
10    X = 1.1 ** I
```

CFT77 recognizes the operation as a multiplication. It sets up a temporary register for values and substitutes the temporary for the multiplication within the loop. CFT carries out the operation through calls to exponentiation library routines.

Where CFT77 handles exponentiation through its strength reduction optimization process, it issues a message to warn you that results could vary from those of the CFT compiler's.

Order of Evaluation

Numerical differences can result from order of evaluation of operands under the CFT77 compiler. Since machine arithmetic is not associative, the way you group values for calculation, from left to right or from right to left, can influence results.

For example, where any of the operands is a variable, for machine arithmetic:

```
(A * B) * C
```

is not necessarily equal to A * (B * C).

Hence, it is important that you remember the role of parentheses in Fortran and standard order of evaluation as specified by ANSI 77 Fortran.

The effect of non-associative order of evaluation is limited to multiplication and to the last bit or two; this generally does not affect numerically-stable algorithms, however. In multiplication the differences are usually in the range of a bit or two in the least significant position. For addition and subtraction, nonetheless, cancellation differences may complicate differences from order of evaluation if the values added or subtracted are close.

Optimization also can influence order of evaluation when the compiler's analysis detects that a computation involves invariants. In a case such as:

```
X = 1.0 - A - B
```

the value of X can depend on whether or not A and B are invariant values. With OPT=FULL or -o full, if the CFT77 compiler finds that it is more efficient to group operands from right to left:

```
X = 1.0 - (A + B)
```

Otherwise, evaluation of operands could proceed as if grouped:

```
X= (1.0 - A) - B
```

In the following loop:

```
      DO 100 I = N ,M
       TEMP1 = SIN(X(I,J))
       TEMP2 = COS (X(I,J))
       SUM(I) = SUM(I) + TEMP1 * 4.0 * WEIGHT + TEMP2 * 3.0 * WEIGHT
100   CONTINUE
```

assume that 4.0*WEIGHT and 3.0*WEIGHT are loop-invariant. With full optimization enabled, the CFT77 compiler removes them from the loop. This means that their evaluation for multiplication is from right to left, not from left to right. If you need evaluation for a different grouping and still receive the full benefits of optimization, associate calculations by enclosing them in parentheses. For example:

```
      DO 100 I = N ,M
       TEMP1 = SIN(X(I,J))
       TEMP2 = COS (X(I,J))
       SUM(I) = SUM(I) + (TEMP1 * 4.0) * WEIGHT + (TEMP2 * 3.0) * WEIGHT
100   CONTINUE
```

Truncation

Numerical differences can result from truncation; again, the fact that machine arithmetic is not associative influences truncation. User grouping of values for calculation is extremely important to avoid the problems truncation can produce and propagate throughout the code.

Truncation errors occur when you compute an intermediate result that requires more bits of precision than the machine can give you to represent it.

Assume that in:

```
1.0 - A - B
```

A = 1. 0 and B = 1.0E-27.

1) If you group the calculation from the left:

```
(1.0 - A) - B = (1.0 - 1.0) - 1.0E-27 = -1.0E-27
```

2) If you group the calculation from the right:

```
1.0 - (A + B) = 1.0 - (1.0+1.0E-27) = 1.0 = 1.0 = 0
```

In example 2 above, adding 1.0 - (1.0 + 1.0E-27) requires about 28 decimal

digits of precision. A CRAY X-MP computer system provides you about 15; therefore, the 1.0E-27 part is lost completely and what remains is 1 --which, subtracted from 1 gives you 0.

As with order of evaluation, numerical differences from truncation are good reminders that you need to consider the use of parentheses. It is the best resource Fortran provides you to force calculations the way you need them.

To test if truncation is causing numerical differences in your code, run your program under CFT77, TRUNC=0 or cft77 -t 0 (defaults) and then under CFT77, TRUNC=3 or cft77 -t 3. Differences in results due to truncation problems show up under TRUNC=3 (-t 3) because usually the last two or three bits are the origin of the differences in truncation problems.

OPTIMIZATION

The CFT and CFT77 compilers are both optimizing compilers. They both share optimizing techniques such as constant hoisting and replacement of redundant expressions.

However, each of the Fortran compilers approaches optimization differently (see the section titled "Compilation" in this module). CFT77, for example, goes further in algebraic simplification. This compiler also eliminates dead variables (a variable is live in a program if its value can be used later on; otherwise, it is dead at that point and CFT77 removes it from the code).

As stated under "Compilation," the CFT77 compiler optimizes on the basis of data-flow analysis, whereas the CFT coompiler optimizes according to code block analysis. In data-flow analysis the compiler collects information about the program as a whole and then distributes this information to each block in the flow graph --but CFT77 looks at blocks as regions of closely interrelated code and which may consist of nonsequentially coded statements, not as groups of up to 4000 words of intermediate code the way CFT would. As part of data-flow analysis and live-equation detection, the CFT77 compiler also examines loop induction variables to determine if replacement of user-induced operations on such variables is also possible. These differences are generally transparent to the user; when they cause differences that require debugging efforts, the reminders and techniques under the section titled "Special Debugging Considerations" are effective in dealing with them.

Three additional areas in which the two compilers differ deserve specific attention: constant increment variables and loop induction variables, unrolling, and IF statement optimization.

Constant Increment Variables and Loop Induction Variables

Constant increment variables (CIV) in CFT are real or integer variables which are decremented or incremented by an invariant expression once on each pass through the DO-loop. As of the CFT compiler's 1.15 release, you can look at a CIV more realistically as a variable which increments or decrements constantly, since the CFT compiler sometimes can manipulate a stride even when the stride seems nonlinear. The restrictions on CIVs in CFT are:

* you can use parentheses only on or surrounding an invariant calculation included in an expression which calculates a CIV. In loop 1 below, K is not a CIV; in loop 2, however, K is a CIV:

```
      J = 10                                  J = 10
      DO 1 I = 1, N                           DO 2 I = 1, N
        K = (K + J - 4)                         K = K + (J - 4)
1     DIFF(I) = DELTA(K) * YY           2     DIFF(I) = DELTA(K) * YY
```

* another variable cannot define the CIV recursively in the same loop. L is not a CIV in loop 1 below, but is one in loop 2:

```
      DO 1 I = 1, N                           DO 2 I = 1, N
        L = L + 34                              L = L + 34
        L2 = I + J                              L2 = I + J
1     L = L2 + I                        2     L3 = L2 + I
```

* if the CIV defines itself in the course of the DO loop, its result must be positive. In loop 1 below, M is not a CIV, while it is one in loop 2:

```
      DO 1 I = 1, N                           DO 2 I = 1, N
        M = 3 - M                               K = 3 - M
1   CONTINUE                            2   CONTINUE
```

Additionally, CIVs must follow the formats specified and illustrated as follows:

Format	Example
CIVa = CIVa ± Invariant	I = I + (39 * INV)
CIVa = ± CIVb ± Invariant	I = (39 * INV) - I2
CIVa = CIVa + CIVa	J = J + J
CIVa = CIVa ± CIVb	J = J - I

where INV is invariant and I2 is a CIV.

Official documentation for CFT77 also alludes to constant increment variables. However, restrictions on their use and formatting in source code are different for CFT77; the compiler also uses these for prevectorization analysis and replacements, whereas the CFT compiler need only use these variables for the purpose of setting vector lengths and in determining loop count and stride.

In CFT77 a constant increment variable is also a variable which is incremented or decremented by an invariant expression on each pass through a loop. The invariant expression that defines the constant increment variable must be type integer. Additionally, the only operators allowed on the expression that defines the variable are plus (+) and minus (-); like CIVs in CFT, if the variable defines itself in the course of the loop, the sign of the increment must not alternate (e.g., i = 2 - i is not allowed).

The CFT77 compiler, however, does NOT restrict the use of the constant increment variable to only one appearance on the left side of the equal sign within the same loop. For example, J in loop 1 below is NOT a CIV under CFT, while it is acceptable as an analogous variable under CFT77:

```
      DO 1 I = M, N
         J = I + K
         J = I + L
    1 CONTINUE
```

The usefulness of constant increment variables for CFT77 lies in their potential as loop induction variables. The compiler uses these loop induction variables (LIV) to determine if it can replace or convert some types of operations to vector equivalents. LIVs are computed from CIVs in Fortran source. When CFT77 cannot execute transformations of LIVs to linear equivalents for vectorization, it produces messages to indicate that vectorization is inhibited.

CFT77, for example, takes the following source:

```
      DO 10 J = 2, 25
        K = K + 1
        L = J + 2
        A(K) = B(K) * P
        A(L) = A(K)
10    CONTINUE
```

and, after several levels of variable transformations, generates a sequence equivalent to:

```
      DO 10 j = 2, 25
        A((j * 1) - 1) = B(j) * P
        A(j + 2) = A((j * 1) - 1)
10    CONTINUE
```

where CFT77 replaces auxiliary CIVs K and L with linear equivalents in terms of J. Through these transformations the CFT77 compiler can detect that a dependence due to recurrence inhibits the generation of vector code for the loop.

Unrolling

Unrolling consists of the expansion of an innermost loop into the next higher level of loop nesting, with the purpose of making more efficient use of CPU resources. This process in general is limited to relatively short loops, since past a given point unrolling the loop would not yield time improvements.

The CFT compiler conducts analysis to determine if a loop can be unrolled by default, for loops whose iteration count goes from 3 to 9. You can request that the compiler not do the analysis or unroll the loop by specifying UNROLL=0 on the CFT control statement or -u 0 on the cft command line. For most cases, however, this process is advantageous for code optimization. When this option is enabled, if the vectorizable innermost loop is too short to warrant use of vector resources the compiler expands the loop into the next higher level and, unless something in it inhibits vectorization, generates vector code for the next level.

The CFT compiler unrolls the innermost loop horizontally when such is possible, by default.

Whereas for CFT this user-independent optimization technique is effected for vectorization purposes, the CFT77 compiler analyzes loops that can be unrolled during the code generation phase. For CFT77 the decision to unroll an innermost loop does not depend on whether or not vectorization could provide an improvement on the loop. The CFT77 compiler unrolls a loop if scheduling is improved by it for code generation. The process itself, and the code sequences generated for "unrolling" under the two compilers, are different. Hence, when documentation for both compilers allude to unrolling, you can assume that the reference is to different processes that share a name.

With both compilers you gain from forcing specific calculations to take place through loops unrolled vertically into the outer loop, as well as from vertically and horizontally unrolled outer loops into innermost loops. These user-induced optimization techniques force both compilers to make better use of registers and reduce the ratio of memory references per floating-point operation.

IF Statement Optimization

The CFT and CFT77 compilers optimize IF statements by using different resources. Both compilers also differ in relation to the level of user involvement allowed to control how optimization takes place.

The CFT compiler optimizes IF statements through conversion of MAX/MIN calls and conditional vector merge calls to vectorizable equivalents. Additionally, CFT also uses hardware compress-index and gather-scatter resources to vectorize specific types of IF statements.

You have no control over the optimization of MAX/MIN calls such as:

```
IF(A(I).GT.Y(I)) A(I) = Y(I)
```

under CFT. This conversion takes place even with CFT,OPT=NOIFCON and cft -o noifcon (default).

CFT optimizes IF-THEN/ELSE sequences in one of two ways. If the sequence includes NO conditional division or function calls, the compiler optimizes by making calls to conditional vector merge libraries which evaluate all possible right-hand sides of the IF first and then goes on to run the conditional test. However, when the right-hand side contains division or function calls, the compiler interprets them as potentially unsafe. In these cases the CFT compiler generates vector code for IF-THEN-ELSE sequences as jumps to a block of code inside the innermost loop; that is, the block may or may not

execute depending on the outcome of the test. For this the compiler uses hardware compress-index/gather scatter (CIGS) instructions: the left-hand side of the IF is evaluated first. The compress-index instruction loads only the values needed by using a GATHER (e.g., A(I(J)), where I is an index into J), instead of computing all possible right-hand sides of the IF.

Examples:

CVMG by default for: `IF((A(I).GT.D(I)).OR(A(I).LT.C(I))) A(I) = ABS(C(I)*D(I))`

CIGS by default for:

```
      DO 10 I = 1, N
        IF(MOD(I,2).EQ.0) THEN
         A(I) = B(I) * 2.
        ELSE
         A(I) = C(I) * 2.
        ENDIF
10    CONTINUE
```

You have control over which of these resources the CFT compiler uses through optimization options PARTIALIFCON and FULLIFCON.

Like CFT, the CFT77 compiler can generate optimized code for conditional sequences as long as they are NOT three-branch IFs, assigned GOTOs, computed GOTOs or backward-branching GOTOs. (Whenever the CFT77 compiler finds these, it issues a message such as: "... was not vectorized because there are multiple entries into the loop".)

Unlike CFT, however, the CFT77 compiler does NOT use compress-index instructions to generate optimized code. Consider the following Fortran sequence:

```
      DO 10 I = 1, 100
        IF(A(I).GT.B(I).AND.C(I).GT.C) A(I) = B(I)
10    CONTINUE

      DO 50 I = 1, N
        IF(XX(I).GT.P) GO TO 25
        XX(I) = AA(I)
        GO TO 50
25      XX(I) = BB(I * 2)
50    CONTINUE

      DO 75 IY = 1, 15
        IF(UG(IY).NE.2) UA(IY) = UB(IY) / 2
75    CONTINUE
```

The following table extracts part of the instruction sequences which both compilers would generate for the IF tests in the loops on the opposite page.

Loop	Instructions	
	CFT	CFT77
10	173754 V7 V5-FV4 141367 V3 V6&V7 175237 V2,VM V3,M 073000 S0 VM	003070 VM S7 146710 V7 S1!V0&VM
50	175677 V6,VM V7,M 073000 S0 VM	003010 VM S1 147132 V1 V3!V2&VM
75	170667 V6 S6+FV7 175565 V5,VM V6,N 073000 S0 VM	003060 VM S6 146172 V1 S7!V2&VM ... 164071 V0 S7*RV1 003010 VM S1 147507 V5 V0!V7&VM

The CFT compiler generates compress-index instructions (175ijn), which use a vector mask to set up a dense vector length of cases which tested true for the condition. This in itself is different in CFT77, which uses a vector mask for cases on which operations need to be carried out; however, CFT77 does not use a different vector length. Instead, it does a full-vector merge on the cases in the mask.

An additional difference is that, in all three cases, the CFT77 compiler pulls the test and vector-mask setup out of the loop.

The CFT77 compiler does not allow you partial control over its choice of resources to use. Besides the fact that it does not use compress-index resources, you can disable this type of optimization only by specifying CFT77, OPT=NOVECTOR or OPT=OFF (cft77 -o novector or -o off).

VECTORIZATION

The vectorization process, naturally, is the same for both Fortran compilers. However, there are specific differences in the way either compiler handles dependence identification.

Dependence Identification

CFT77 analyzes for possible data dependences produced by the differences between scalar and vector processing and, although the process is different between the compilers, the result is similar. Both compilers, for example, identify recurrence situations and inhibit vector code. However, there are differences in what the two compilers interpret as a recurrence problem.

The following sequences illustrate differences.

Sequence	Vectorization CFT	Vectorization CFT77	Comments
1) DO 1 J = 100,1,-1 X(J) = Y(J+4) X(J-4) = Z(J) 1 CONTINUE	No	Yes	CFT77 processes this loop using shorter vector lengths (4).
2) DO 2 I = 1, 10 A(I) = X(I) Y(I) = A(I+1) 2 CONTINUE	Yes	No	Subsequent-plus data dependence. CFT77 inhibits vector processing because in scalar mode by the time A(I) needs A(I+1) its old value is available, whereas in vector processing the old value has been updated and destroyed: "CFT77 could not deal with the loop's expression order."
3) KO = 200 DO 3 J = 201,251 SAVE(J) = SAVE(KO) KO = KO - 1 3 CONTINUE	Yes	No	CFT77 interprets this as a case of overlap between the ranges of J and KO.

Vectorization differences, cont.

Sequence	Vectorization CFT	Vectorization CFT77	Comments
4) EQUIVALENCE(J,K) DO 4 J = N,2,-1 F(K) = H(K) G(K) = F(K-4) 4 CONTINUE	Yes	No	CFT77 does not vectorize this because of the scalar store --this applies only to equivalenced array indices, not to arrays.
5) PRINT 101, XL, * (OT(IX),IX=1,JX	No	Yes	CFT77 handles implied DO loops in formatted PRINT statements through a vectorized replacement operation, provided that the implied loop handle one entity only.