CF90™ Co-array Programming Manual

SR–3908 3.1

Document Number 004–3908–001

Record of Revision

Version	*Description*
3.1	August 1998 Original Printing. This manual supports the Programming Environment 3.1 release.

Contents

Page

About This Manual

This document describes the co-array extension to the CF90 compiler on CRAY T3E systems, for the Programming Environments (PE) release 3.1. Co-arrays can be used to perform data passing in single-program-multiple-data (SPMD) programs on UNICOS/mk systems.

Related CF90 Publications

The following documents contain additional CF90 compiler information that may be helpful:

- *CF90 Commands and Directives Reference Manual*
- *Fortran Language Reference Manual, Volume I*
- *Fortran Language Reference Manual, Volume II*
- *Fortran Language Reference Manual, Volume III*
- *Intrinsic Procedures Reference Manual*
- *Message Passing Toolkit: MPI Programmer's Manual*
- *Message Passing Toolkit: PVM Programmer's Manual*

Related Co-array Publications

The following technical papers may be of use to you when implementing co-arrays:

- R. W. Numrich and J. Reid, *Co-array Fortran for Parallel Programming*, Silicon Graphics, Inc., Rutherford Appleton Laboratory, ACM Fortran Forum, 1998.
- R. W. Numrich, J. L. Steidel, B. H. Johnson, B. D. de Dinechin, G. W. Elsesser, G. S. Fischer, and T. A. MacDonald, *Definition of the F−− Extension to Fortran 90*, Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computers, Lectures on Computer Science Series, Number 1366, Speinger-Verlag, 1998, pages 282–306.

Obtaining Publications

The *User Publications Catalog* describes the availability and content of all Cray Research hardware and software documents that are available to customers. Customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a printed copy of this document, either call the Minnesota Distribution Center at +1–651–683–5907, or send a facsimile of your request to fax number +1–651–452–0141. Silicon Graphics employees may send electronic mail to `orderdsk@cray.com` (UNIX system users).

Silicon Graphics maintains information on publicly available Cray Research documents at the following URL:

`http://www.cray.com/swpubs/`

This Web site contains information that allows you to browse documents online and send feedback to Silicon Graphics.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
...	Ellipses indicate that a preceding element can be repeated.
[ ]	Typically, brackets enclose optional portions of a command or directive line in UNICOS/mk

documentation. In this manual, however, brackets denote the co-dimensions of a co-array.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail at the following address:

  techpubs@sgi.com

- Contact your customer service representative and ask that an SPR or PV be filed. If filing an SPR, use PUBLICATIONS for the group name, PUBS for the command, and NO-LICENSE for the release name.

- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–651–683–5600

- Send a facsimile of your comments to the attention of “Software Publications Group” in Eagan, Minnesota, at fax number +1–651–683–5599.

We value your comments and will respond to them promptly.

Introduction [1]

Data passing, also known as one-sided communication, has proven itself to be an effective method for programming single-program-multiple-data (SPMD) parallel computations on UNICOS and UNICOS/mk platforms. Its chief advantage over message passing is lower latency for data transfers, which leads to better scalability of parallel applications. *Co-arrays* are a syntactic extension to the Fortran Language that offers a method for programming data passing.

Data passing can also be accomplished by using the shared memory (SHMEM) library routines. Using SHMEM, the program transfers data from an object on one processing element (PE) to an object on another via subroutine calls.

Co-arrays provide an alternative syntax for specifying these transfers. With co-arrays, the concept of a PE is replaced by the concept of an *image*. When data objects are declared as part of a co-array, data on different images can be read or written in a fashion similar to the way in which arrays are read and written in Fortran. This is done by adding additional dimensions, or *co-dimensions*, within brackets ([ ]) to an object's declarations and references. These extra dimensions express the image upon which the object resides.

Co-arrays offer the following advantages over SHMEM:

- Co-arrays are syntax-based, so programs that use them can be analyzed and optimized by the compiler. This offers greater opportunity for hiding data transfer latency.
- Co-array syntax can eliminate the need to create and copy data to local temporary arrays.
- Co-arrays express data transfer naturally through the syntax of the language, making the code more readable and maintainable.
- The unique bracket syntax allows you to scan for and to identify communication in a program easily.

Consider the following SHMEM code fragment from a finite differencing algorithm:

```
CALL SHMEM_REAL_GET(T1, U, NROW, LEFT)
CALL SHMEM_REAL_GET(T2, U, NROW, RIGHT)
NU(1:NROW) = NU(1:NROW) + T1(1:NROW) + T2(1:NROW)
```

Co-arrays can be used to express this fragment simply as:

```
NU(1:NROW) = NU(1:NROW) + U(1:NROW)[LEFT] + U(1:NROW)[RIGHT]
```

Notice that the resulting code is more concise, easier to read, and that the copies to local temporary objects `T1` and `T2` are eliminated.

Co-arrays can interoperate with the other message passing and data passing models available on UNICOS/mk systems. This interoperability allows you to introduce co-arrays gradually into codes that presently use the Parallel Virtual Machine (PVM), the Message Passing Interface (MPI), or SHMEM.

Writing Programs With Co-arrays [2]

This chapter describes the syntax and semantics of the co-array extension to CF90.

2.1 Execution Model and Images

Programs with CF90 co-arrays use the *single-program-multiple-data (SPMD)* execution model. In the SPMD model, the program and all its data are replicated and executed asynchronously. Each replication of the program is an *image.*

Caution: Like PEs, images are numbered. Unlike PEs, which are numbered starting with zero, images are numbered starting with one.

The total number of images that are executing can be accessed through the NUM_IMAGES(3) intrinsic function. An image can access its own image number through the THIS_IMAGE(3) intrinsic function. Images can synchronize through the SYNC_IMAGES(3) intrinsic subroutine.

2.2 Specifying Co-arrays

A *co-array* is a data object that is identically allocated on each image and, more significantly, can be directly referenced by any other image syntactically.

A co-array specification consists of the local object specification and the co-dimensions specification. The *local object* is the data object to be replicated on each image. The *co-dimensions* are the dimensions of the co-array. They are specified within brackets ([ ]) and appended to the specification for the local object.

Example 1. The following statements show co-array declarations:

```
REAL, DIMENSION(20)[8,*] :: A, C
REAL :: B(20)[8,*], D[*], E[0:*]
INTEGER :: IB(10)[*]
```

Note: Generally, a co-dimension specification in brackets takes the same form as a dimension specification in parentheses. The exception is that for co-dimensions, the upper bound of the rightmost co-dimension must be an asterisk (*). This is because co-array objects are replicated on all images, so co-size is always equal to NUM_IMAGES(3I).

Elements of co-arrays on other images can be refererenced by appending square brackets to the end of a reference to the local object. As Example 2 shows, the brackets contain subscripts, one for each co-dimension:

Example 2:

```
A(5)[3,7] = IB(5)[3]
D[11] = E
A(:)[2,3] = C(1)[1,1]
```

The co-dimension specification of a co-array creates a mapping of subscripts to images. This mapping is identical to the mapping that parenthesized array dimensions create between subscripts and elements of an array. For example, the following table lists the image number for some references of the objects declared in Example 1:

Reference	Image
IB(5)[3]	3
A(5)[3,7]	31
D[11]	11
E[11]	12

The terms *local rank*, *local size*, and *local shape* refer to the rank, size and shape of the local object of a co-array. The terms *co-rank*, *co-size*, and *co-shape* refer to those properties implied by the co-dimensions of a co-array. For example, for co-array A declared in the preceding list, its local rank is 1; its local size is 20; its co-rank is 2; and its co-size is equal to NUM_IMAGES(3I). The co-rank of a co-array cannot exceed 7.

The local object of a co-array can be of a derived type, but a co-array can never be a component within a derived type. For example:

```
TYPE DTYPE1
   REAL :: X
   REAL :: Y
END TYPE DTYPE1

TYPE(DTYPE) :: DT(100)[*] ! PERMITTED: CO-ARRAY OF DERIVED TYPE

TYPE DTYPE2
   REAL :: X
   REAL :: Y[*]           ! NOT PERMITTED:
                            ! CO-ARRAY IN DERIVED TYPE
```

```
END TYPE DTYPE2
```

Most objects can be the local object of a co-array, but the following list indicates restrictions on co-array specifications:

- The smallest referenceable element of the local object of a co-array must have a type such that the size is 64 bits and is aligned on a 64-bit boundary. For example:

```
REAL*4 :: X(100)[*]            ! NOT SUPPORTED: 32-BIT TYPE
CHARACTER(LEN=80) :: C[*]      ! NOT SUPPORTED: 8-BIT TYPE
COMPLEX(KIND=8) :: CX(100)[*] ! NOT SUPPORTED: 32-BIT ALIGNMENT
COMPLEX(KIND=16) :: CX2(100[*]! NOT SUPPORTED:  128-BIT TYPE
```

- Co-arrays with assumed-size local size are not supported. For example:

```
REAL :: Y(*)[*]     ! NOT SUPPORTED: LOCAL OBJECT ASSUMED SIZE
```

- Co-arrays with deferred-shape local shape or co-shape are supported, but the co-array must be allocatable. Co-array pointers are not supported. For example:

```
REAL, ALLOCATABLE :: WA(:)[:]   ! SUPPORTED: ALLOCATABLE
REAL, POINTER     :: WP(:)[:]   ! NOT SUPPORTED: POINTER
```

- Co-arrays with assumed-shape local shape or co-shape are not supported. For example:

```
SUBROUTINE S1( Z1, Z2 )
REAL :: Z1(:)[*]    ! NOT SUPPORTED: ASSUMED-SHAPE LOCAL SHAPE
REAL :: Z2(:)[:]    ! NOT SUPPORTED: ASSUMED-SHAPE CO-SHAPE
```

- Automatic co-arrays are not supported. For example:

```
SUBROUTINE S2( A, N )
REAL :: A(N)[*]             ! SUPPORTED: CO-ARRAY ACTUAL ARGUMENT
REAL :: W(N)[*]             ! NOT SUPPORTED: AUTOMATIC LOCAL OBJECT
```

2.3 Referencing Co-arrays

Co-arrays can be referenced two ways: with brackets and without brackets.

When brackets are omitted, the object local to the invoking image is referenced; this is called a *local reference*. For example:

```
REAL, DIMENSION(100)[*] :: A, B, C, D, E

A(I) = B(I) + C(I)              ! LOCAL REFERENCES TO A, B, C
D = E                           ! LOCAL REFERENCES TO D, E
```

When brackets are specified, the object on the image specified by the subscripts within the brackets is referenced. This is called a *bracket reference.* For example:

```
A(I)[IP] = B(I) + C(I)          ! REFERENCE TO A ON IMAGE "IP";
                                ! LOCAL REFERENCES TO B, C

D(:) = E(:)[IP2]                ! REFERENCES TO E ON IMAGE "IP2"
                                ! LOCAL REFERENCES TO D
```

Components of derived type co-arrays are specified by appending the component specification after the brackets. For example:

```
TYPE DTYPE3
  REAL    :: X(100)
  INTEGER :: ICNT
END TYPE DTYPE3

TYPE (DTYPE3) :: DT3[*]

DT3%ICNT = DT3[IP]%ICNT   ! SUPPORTED: BRACKET IN DERIVED TYPE
DT3%X(J) = DT3[IP]%X(J)   ! COMPONENT REFERENCES
```

The co-subscripts of a co-array reference must translate to an image number between 1 and NUM_IMAGES(3I), otherwise the behavior of the reference is undefined.

The following additional restrictions exist for co-array references:

- Specification of subscripts for co-dimensions generally follows the specification of subscripts within parentheses. However, support for triplet subscript notation within brackets is not supported. For example:

  ```
  D(K)[1:N:2] = E(K)[1:N:2]       ! NOT SUPPORTED:
                                  ! TRIPLET NOTATION IN []S
  ```

- While brackets are supported in references to components of derived type co-arrays, bracket references of derived types are not supported. For example, consider the declaration for DT3 as stated previously in this subsection:

```
   DT3 = DT3[IP]     ! NOT SUPPORTED: DERIVED TYPE
                     ! BRACKET REFERENCES
```

2.4 Initializing Co-arrays

Co-arrays can be initialized using the DATA statement, but only the initialization of the local object of the co-array can be specified. Bracket references are not allowed in a DATA statement. For example:

```
REAL :: AI(100)[*]
DATA AI(3)      /1.0/   ! PERMITTED
DATA AI(3)[11]  /1.0/   ! NOT PERMITTED
```

When the program is executed, the co-array local objects on every image are initialized identically, as specified.

2.5 Using Co-arrays with Procedure Calls

If a procedure with a co-array dummy argument is called, the called procedure must have an explicit interface, and the actual argument must be a local reference to a co-array. If the actual argument has subscripts, their values should be the same across all images, otherwise the program behavior is undefined. For example:

```
INTERFACE
   SUBROUTINE S3( A, N )
   REAL :: A(N)[*]
END INTERFACE

REAL :: X(100,100), Y(100,100)[*]

CALL S3( X(1,K), 100 )  ! NOT PERMITTED:
                        ! LOCAL ACTUAL, CO-ARRAY DUMMY

CALL S3( Y(1,K), 100 )  ! PERMITTED: CO-ARRAY ACTUAL AND DUMMY;
                        ! UNDEFINED IF "K" NOT SAME VALUE ON
                        ! ALL IMAGES
```

Bracket references cannot appear as actual arguments in subroutine calls or function calls. For example:

```
CALL S3( Y(1,K)[IP], 100 )  ! NOT PERMITTED: BRACKET ACTUAL
```

Co-array bracket references can appear within an actual argument, but only as part of an expression that is passed as the actual argument. Parentheses can be used to turn a bracket reference into an expression. For example:

```
CALL S3( ( Y(1,K)[IP] ), 100 )  ! PERMITTED: ACTUAL IS EXPRESSION
```

The rules of resolving generic procedure references are the same as those in the Fortran 90 standard.

The following restrictions affect co-arrays used in procedures:

- A function result is not permitted to be a co-array.
- A pure procedure is not permitted to contain any co-arrays.

2.6 Specifying Co-arrays in COMMON and EQUIVALENCE Statements

Co-arrays can be specified in COMMON statements. For example:

```
COMMON /CCC/ W1(100)[*], W2(100)[16,*] ! PERMITTED:
                                       ! CO-ARRAYS IN COMMON
```

The layout of the common block on any one image is as if all objects of the common block were declared without co-dimensions.

Data objects that are not co-array data objects can appear in the same common block as co-arrays.

Co-arrays can be specified in EQUIVALENCE statements, but bracket references cannot appear in EQUIVALENCE statements. For example:

```
REAL :: V1(100)[*], V2(100)[*], V3(100)

EQUIVALENCE ( V1(50), V2(1) )          ! PERMITTED: CO-ARRAYS
EQUIVALENCE ( V1(1)[16], V2(1)[1] )    ! NOT PERMITTED:
                                       ! SQUARE BRACKETS
```

Data objects that are not co-array data objects cannot be equivalenced to co-array data objects. For example:

```
EQUIVALENCE (V1(50), V3(1)) ! NOT PERMITTED: V3 NOT
                            ! CO-ARRAY OBJECT
```

2.7 Specifying Allocatable Co-arrays

A co-array can be allocatable. Co-dimensions are specified by appending brackets containing the co-dimension specification to the co-array local specification in an ALLOCATE statement. For example:

```
REAL, ALLOCATABLE :: A1(:)[:], A2(:)[:,:]

ALLOCATE ( A1(10)[*])          ! PERMITTED: ALLOCATABLE CO-ARRAY
ALLOCATE ( A2(24)[0:7,0:*] )
```

As with the specification of statically allocated co-arrays, the upper bound of the final co-dimension must be an asterisk (*) and the values of all other bounds must be identical across all images.

Caution: Execution of ALLOCATE and DEALLOCATE statements containing co-array objects causes an implicit barrier synchronization of all images. All images must participate in the execution of these statements, or deadlock can occur.

2.8 Using Pointers in Derived Type Co-arrays

A pointer cannot be declared as a co-array, but a co-array can be of a derived type containing a pointer member. This enables construction of irregularly sized data structures across images. For example:

```
TYPE DTYPE4
   INTEGER :: LEN
   REAL, POINTER :: AP(:)
END TYPE DTYPE4

TYPE(DTYPE4) :: D4[*]          ! PERMITTED: CO-ARRAY OF DERIVED
                               ! TYPE CONTAINING POINTER
```

A bracket reference to a pointer in a derived type co-array returns the value from the object on the specified image. For example, the reference D4[7]%AP(22) returns the value of D4%AP(22) as evaluated on image 7.

To help prevent the possibility of pointers being assigned invalid data, co-array bracket references cannot appear in pointer assignment statements. For example:

```
REAL :: Q(100)

D4[IP]%AP => Q                  ! NOT PERMITTED: BRACKET IN
Q => D4[IP]%AP                  ! POINTER ASSIGNMENT
```

Note: Pointers in derived type co-arrays cannot appear in pointer assignment statements. They can be assigned only by using the ALLOCATE statement. For example:

```
ALLOCATE ( D4%AP(100) ) ! PERMITTED: ALLOCATE OF CO-ARRAY POINTER
                        ! MEMBER
D4%AP => Q              ! NOT PERMITTED: POINTER ASSIGNMENT
                        ! INTO CO-ARRAY ELEMENT
```

2.9 Intrinsic Procedures

The following intrinsic procedures support co-arrays:

- LOG2_IMAGES(3I), which returns the base 2 logarithm of the number of executing images truncated to an integer
- NUM_IMAGES(3I), which returns the number of executing images
- REM_IMAGES(3I), which returns
  `MOD( NUM_IMAGES(), 2**LOG2_IMAGES() )`
- SYNC_IMAGES(3I), which synchronizes images
- THIS_IMAGE(3I), which returns the index of, or co-subscripts related to, the invoking image

Only NUM_IMAGES(3I), LOG2_IMAGES(3I), and REM_IMAGES(3I) can appear in specification statements. None of the intrinsics are permitted in initialization expressions.

For more information on these intrinsic procedures, see the online man pages for each. Copies of the online man page appear in Appendix A, page 21, of this manual.

2.10 Ensuring Data Coherence Across Images

An image can guarantee to another image that it has completed all its references to co-array data by executing and coordinating a SYNC_IMAGES(3I) intrinsic procedure call with the other image.

SYNC_IMAGES(3I) guarantees the completion of references to data executed by that image within procedures for which the data is declared as part of a co-array. The reference must either be a direct read or write of the data or a procedure call that references the data.

For example, consider the following subroutine:

```
SUBROUTINE TST(A,B,C,D,N,IP)
REAL :: A(N)[*], B(N)[*], C(N)[*], D(N)

A(:) = B(:)[IP]
CALL SUB1(C,N)
D(:) = 0.0

CALL SYNC_IMAGES()

END
```

When an image executes the SYNC_IMAGES(3I) call in the preceding example, it guarantees to all images executing a coordinating SYNC_IMAGES(3I) call that its references to A and B are complete and that all references to C by SUB1 are complete. It does not, however, guarantee that its references to D are complete, since D is not declared as a co-array. This is true even if the actual argument for D is a co-array.

Behavior of references to the same data by different images without such coordinating SYNC_IMAGES(3I) calls is undefined.

2.11 Controlling Execution Across Images

The execution of a STOP statement by any image halts the execution of all images.

2.12 Input and Output (I/O)

Each image has its own set of independent I/O units. A file can be opened on one image when it is already open on another, but only the BLANK, DELIM, PAD, ERR, and IOSTAT specifiers can have values that differ from those in effect on other images.

Caution: For a unit identified by an asterisk (*) in a `READ` or `WRITE` statement, there is a single position for all images. Only one image executes a statement for such a unit at any one time. The system introduces synchronization when necessary. Otherwise, each image positions each file independently. If the access order is important, the program must provide its own synchronization between images.

Executing Programs with Co-arrays [3]

This chapter describes the relationships between co-array programs and various commands, tools, and products available in the UNICOS/mk programming environment.

3.1 Using the `f90`(1) and `mpprun`(1) Commands

The `-Z` option on the `f90`(1) command line must be specified in order for co-array syntax to be recognized and translated. Otherwise, co-array syntax generates `ERROR` messages.

Upon execution of an `a.out` file that has been compiled and loaded with the `-Z` option, an image is created and executed on every PE assigned to the job. Images `1` through `NUM_IMAGES`(3I) are assigned to PEs `0` through `N$PES-1`, consecutively.

You can set the number of PEs assigned to a job at either compile time or load time by specifying the `-X` option on the `f90`(1) command. The number of PEs can also be set at run time by executing the `a.out` file by using the `mpprun`(1) command with the `-n` option specified.

Bounds checking is performed by specifying the `-Rb` option on the `f90`(1) command line. This feature is not implemented for co-dimensions of co-arrays.

For more information on the `f90`(1) and `mpprun`(1) commands, see the `f90`(1) and `mpprun`(1) man pages.

3.2 Using the CrayTools Tool Set with Co-array Programs

The CrayTools tool set, which includes `totalview`(1), `apprentice`(1), `xbrowse`(1), and other tools, does not contain special support for co-arrays and does not support the bracket notation. In most cases, however, these tools can still be used effectively to analyze programs containing co-arrays.

The following sections discuss issues related to the interaction of these tools with programs containing co-arrays.

3.2.1 Debugging Programs Containing Co-arrays

The `totalview`(1) debugger does not support the bracket notation. Co-arrays generally appear as their corresponding local object with co-dimensions stripped off.

Co-array data can be viewed and referenced by switching the `totalview`(1) `Process` window to the PE corresponding to the desired image and accessing the co-array with local references.

3.2.2 Analyzing Co-array Program Performance

To the CrayTools performance tools, which include `apprentice`(1), `pat`(1), and others, co-arrays generally appear as their corresponding local object with co-dimensions stripped off.

Caution: References to co-arrays on different images appear to the performance tools as local data references. This may skew the remote reference statistics of these tools.

3.3 Interoperating with Other Message Passing and Data Passing Models

Co-arrays can interoperate with all other message and data passing models available on UNICOS/mk systems: MPI, PVM, and SHMEM. This allows you to introduce co-arrays into existing application codes incrementally.

These models are implemented through procedure calls, so the language interaction between co-arrays and these models is well defined. For more information on passing co-arrays to procedure calls, see Section 2.5, page 7.

Caution: MPI, PVM, and SHMEM generally use PE numbers, which start at zero, but the co-array model generally deals with image numbers, which start at one. For information on the mapping between PE and image numbers, see Section 3.1, page 13

Co-arrays are *symmetric* for the purposes of SHMEM programming. Pointers in co-arrays of derived type, however, may not necessarily point to symmetric data.

For more information on the the other message passing and data passing models, see one of the following publications:

- The *Message Passing Toolkit: PVM Programmer's Manual*
- The *Message Passing Toolkit: MPI Programmer's Manual*
- The `intro_shmem`(3) and man page.

Optimizing Co-arrays [4]

Programs containing co-arrays benefit from all the usual steps you can take to improve run-time performance of code that runs on one PE. This chapter describes additional steps you can take to improve the performance of programs that contain co-array references across images on UNICOS/mk systems.

4.1 Splitting Co-array References

In order to cover the latency and increase the bandwidth of off-image co-array references on UNICOS/mk systems, the compiler aggressively splits these references from the original COMPUTE loop into special GET and PUT loops that precede and follow the COMPUTE loop.

The following example illustrates this:

```
DO I = 1, N
A(I)[IP1] = B(I)[IP2] + C(I)[IP3]
ENDDO
```

The compiler restructures the preceding loop into the following three loops:

```
DO I = 1, N             ! GET LOOP
E1(I) = B(I)[IP2]
E2(I) = C(I)[IP3]
ENDDO

DO I = 1, N             ! COMPUTE LOOP
E3(I) = E1(I) + E2(I)
ENDDO

DO I = 1, N             ! PUT LOOP
A(I)[IP1] = E3(I)
ENDDO
```

The temporary arrays E1, E2, and E3 are subsequently mapped to CRAY T3E E-registers, and the reads and writes of the co-arrays are translated to E-register GET and PUT instructions. This restructuring and translation allows for nearly optimal overlap of remote references in the execution of the GET and PUT loops. Some overlap of GET and PUT operations with operations in the compute loop is also enabled.

The compiler splits co-array references only when it can detect, through classic data dependence analysis, that the behavior of the original loop can be preserved. More precisely, a co-array reference is split only if the compiler can determine that its result does not depend on any other co-array reference within that loop.

In addition, the compiler splits co-array references only when it can split all references within a loop. For all co-array references that cannot be split, less optimal, straight-line E-register instructions are generated.

Only inner loops are considered for co-array reference splitting. The compiler issues an optimization-level message when a loop's co-array references are split.

4.2 Vectorizing Co-array References

Once split, certain co-array references are eligible for further optimization. In particular, vector versions of the E-register `GET` and `PUT` operations can be generated. Eight elements can be referenced with one vector E-register instruction, so bandwidth is increased by reducing the number of E-register instructions issued.

A co-array reference can be vectorized only if its co-subscripts are invariant with respect to the inner loop; in other words, all references during execution of the loop are to the same image. In addition, the stride of the reference on that image must be an invariant with respect to the loop.

Furthermore, a `GET` operation can only be vectorized if it is the only `GET` in its loop. This restriction applies to `PUT` operations as well. Likewise, a `PUT` operation can only be vectorized if it is the only `PUT` in its loop.

Finally, in order for vector E-register operations to be generated, unrolling must be enabled by specifying the `-O unroll2` option on the `f90`(1) command line.

The compiler issues an optimization-level message when co-array references in a loop are vectorized.

4.3 Using CRAY T3E Data Streams

On some UNICOS/mk systems (other than the CRAY T3E 900 and CRAY T3E 1200E system models), the default behavior of programs that contain E-register code (which is generated for co-array references) is that the hardware feature called *data stream buffers* is not enabled. Enabling data stream buffers can

improve run-time performance of on-image memory references. You can enable data streams for these programs by using the `set_d_stream`(3) library routine.

Contact your site representative to see whether your system exhibits this behavior. The `streams_guide`(7) man page contains a pointer to information on CRAY T3E programming with coherent memory streams. The information there explains how data streams can be safely enabled in your program.

Intrinsic Procedure Man Pages [A]

This appendix contains copies of the following man pages:

- LOG2_IMAGES(3I)
- NUM_IMAGES(3I)
- REM_IMAGES(3I)
- SYNC_IMAGES(3I)
- THIS_IMAGE(3I)

Note: The following man pages are designed for online use. In certain cases, the text of the man pages online may be more readable than the text that appears in this appendix.

```
LOG2_IMAGES(3I)

NAME
     LOG2_IMAGES - Returns the base 2 logarithm of the number of executing
     images truncated to an integer

SYNOPSIS
     LOG2_IMAGES()

IMPLEMENTATION
     UNICOS/mk systems

STANDARDS
     CF90 compiler extension to Fortran 90

DESCRIPTION
     LOG2_IMAGES is a CF90 intrinsic procedure that returns the base 2
     logarithm of the number of executing images, truncated to an integer.
     It is an inquiry function.

RETURN VALUES
     The LOG2_IMAGES intrinsic function returns a scalar value of type
     default integer.

SEE ALSO
     CF90 Co-array Programming Manual
```

```
NUM_IMAGES(3I)

NAME
     NUM_IMAGES - Retrieves the total number of images that are executing

SYNOPSIS
     NUM_IMAGES()

IMPLEMENTATION
     UNICOS/mk systems

STANDARDS
     CF90 compiler extension to Fortran 90

DESCRIPTION
     NUM_IMAGES is a CF90 intrinsic procedure that retrieves the total
     number of images that are executing.  It is an inquiry function.

RETURN VALUES
     The NUM_IMAGES intrinsic function returns a scalar value of type
     default integer.

SEE ALSO
     CF90 Co-array Programming Manual
```

```
REM_IMAGES(3I)

NAME
     REM_IMAGES - Returns MOD(NUM_IMAGES(), 2**LOG2_IMAGES())

SYNOPSIS
     REM_IMAGES()

IMPLEMENTATION
     UNICOS/mk systems

STANDARDS
     CF90 compiler extension to Fortran 90

DESCRIPTION
     REM_IMAGES is a CF90 intrinsic procedure that returns
     MOD(NUM_IMAGES(), 2**LOG2_IMAGES()).  It is an inquiry function.

RETURN VALUES
     The REM_IMAGES intrinsic function returns a scalar value of type
     default integer.

SEE ALSO
     CF90 Co-array Programming Manual
```

```
SYNC_IMAGES(3I)

NAME
     SYNC_IMAGES - Synchronizes images

SYNOPSIS
     SYNC_IMAGES([[IMAGE=]image])

IMPLEMENTATION
     UNICOS/mk systems

STANDARDS
     CF90 compiler extension to Fortran 90

DESCRIPTION
     SYNC_IMAGES is a CF90 intrinsic procedure that synchronizes images.
     It is an intrinsic subroutine.  SYNC_IMAGES accepts the following
     argument:

     image     A scalar integer or an integer array.  The behavior of
               SYNC_IMAGES differs, as follows, depending on whether image
               is specified and whether it is a scalar or an array:

               * If image is absent, the image waits for all other images
                 to call SYNC_IMAGES with no argument.

               * If image is a scalar and the invoking image has index i,
                 the invoking image waits for the image with index image to
                 execute a SYNC_IMAGES call with a scalar argument with
                 value i.  After both calls are made, execution proceeds.
                 If image is i, the call has no effect.  If image has a
                 value that is less than 1 or greater than NUM_IMAGES(3I),
                 the behavior is undefined.

               * If image is an array, the invoking image has index i and
                 array image contains the value i, the invoking image waits
                 for those images whose values are contained in array image
                 to execute a SYNC_IMAGES call with an array argument that
                 contains the value i.  Any values in array images that are
                 outside the range 1 through NUM_IMAGES are ignored.  If
                 image does not contain the value i, the behavior is
                 undefined.
```

```
SEE ALSO
     CF90 Co-array Programming Manual
```

```
THIS_IMAGE(3I)

NAME
     THIS_IMAGE - Retrieves an image number

SYNOPSIS
     THIS_IMAGE([[ARRAY=]array[,[DIM=]dim]])

IMPLEMENTATION
     UNICOS/mk systems

STANDARDS
     CF90 compiler extension to Fortran 90

DESCRIPTION
     THIS_IMAGE is a CF90 intrinsic procedure that allows an image to
     retrieve its own image number.  It returns the index of, or
     co-subscripts related to, the invoking image.  It is an inquiry
     function.  THIS_IMAGE accepts the following arguments:

     array     The name of a co-array.

     dim       An integer value.

RETURN VALUES
     The results differ, as follows, depending on the optional arguments
     specified:

     * If array is absent, the result is a default integer scalar with a
       value equal to the index of the invoking image.

     * If array is present with a co-rank of one and dim is absent, the
       result is a default integer scalar with a value equal to the co-
       subscript of the elements of array that resides on the invoking
       image.

     * If array is present with a co-rank greater than one and dim is
       absent, the result is a rank one default integer array of a size
       equal to the co-rank of array.  Element k of the result has a value
       equal to the co-subscript k of the elements of array that reside on
       the invoking image.

     * If array and dim are present, the result is a default integer scalar
```

with value equal to co-subscript dim of the elements of array that reside on the invoking image.

EXAMPLES

Assume that the following declaration exists in a program:

```
REAL :: A(100)[8,4]
```

The following table lists various THIS_IMAGE calls and the values they return on images 3 and 13:

Call	Image 3	Image 13
THIS_IMAGE()	3	13
THIS_IMAGE(A)	/ 3, 1 /	/ 5, 2 /
THIS_IMAGE(A,1)	3	5
THIS_IMAGE(A,2)	1	2

SEE ALSO

CF90 Co-array Programming Manual

Index