# Optimizing Fortran 90 Shift Operations on Distributed-Memory Multicomputers

*Ken Kennedy  John Mellor-Crummey*
*Gerald Roth*

**CRPC-TR95558-S**
**August 1995**

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

From the *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing* (LCPC95), Columbus, OH, August 1995.

# Optimizing Fortran 90 Shift Operations on Distributed-Memory Multicomputers *

Ken Kennedy, John Mellor-Crummey, and Gerald Roth **

Department of Computer Science MS#132, Rice University, Houston, TX 77005-1892

**Abstract.** When executing Fortran 90 style data-parallel array operations on distributed-memory multiprocessors, intraprocessor data movement due to shift operations can account for a significant fraction of the execution time. This paper describes a strategy for minimizing data movement caused by Fortran 90 `CSHIFT` operations and presents a compiler technique that exploits this strategy automatically. The compiler technique is global in scope and can reduce data movement even when a definition of an array and its uses are separated by control flow. This technique supersedes those whose scope is restricted to a single statement. We focus on the application of this strategy on distributed-memory architectures, although it is more broadly applicable.

## 1 Introduction

High-Performance Fortran (HPF)[11], an extension of Fortran 90, has attracted considerable attention as a promising language for writing portable parallel programs. Programmers express data parallelism using Fortran 90 array operations and use data layout directives to direct partitioning of the data and computation among the processors of a parallel machine.

For HPF to gain acceptance as a vehicle for parallel scientific programming, it must achieve high performance on problems for which it is well suited. To achieve high performance on a distributed-memory parallel machine, an HPF compiler must do a superb job of translating Fortran 90 data-parallel operations on arrays into an efficient sequence of operations that minimize the overhead associated with data movement.

Interprocessor data movement on a distributed-memory parallel machine is typically far more costly than movement within the memory of a single processor. For this reason, much of the prior research on minimizing data movement has focused on the interprocessor case. However, although interprocessor data movement is more costly per element, the number of elements moved within the memory of a single processor may be much larger, causing the cost of local data

* This research supported in part by the NSF Cooperative Research Agreement Number CCR-9120008.

** Supported in part by the IBM Corporation through the Graduate Resident Study Program.

movement to be dominant. Johnsson previously has noted that "eliminating the local data motion by separating the set of data that must move between nodes from the data that stays within local memory may yield a significant performance improvement" [12]. The cost of local data movement becomes more important for distributed arrays as the partition size per processor increases.

In this paper, we focus on the problem of minimizing the amount of intra-processor data movement when computing Fortran 90 array operations. To make our technique as generally applicable as possible, we handle array assignment statements where the right-hand side consists of a call to a Fortran 90 shift intrinsic. The technique can handle all such assignment statements, even when the definition of the array and its uses are separated by control flow. Such a technique supersedes those that are restricted to a single statement.

In the next section we briefly review the Fortran 90 shift operators and their execution cost on distributed-memory machines. In Section 3, we describe the *offset array* strategy for reducing intraprocessor data movement associated with shift operations on arrays with BLOCK or CYCLIC(K) distributions. We also present some empirical results to show the potential profitability of applying the offset array optimization. Section 4 describes a global SSA-based analysis algorithm that restructures programs to use offset arrays where profitable. We close with a look at related work.

## 2 Fortran 90 Shift Operators

The Fortran 90 circular shift operator `CSHIFT(ARRAY, SHIFT, DIM)` returns an array of the same shape, type, and values as `ARRAY`, except that each rank-one section of `ARRAY` crossing dimension `DIM` has been shifted circularly `SHIFT` times. The sign of `SHIFT` determines the shift direction. The end-off shift operator `EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)` is identical to `CSHIFT` except for the handling of boundaries. For the rest of the paper we focus on optimizing `CSHIFT` operations although our techniques can be generalized to handle `EOSHIFT` as well.

### 2.1 Sources of CSHIFT Operations

For HPF, optimizing `CSHIFT` operations is important since `CSHIFT` operations are ubiquitous in stencil-based dense array computations for which HPF is best suited. Besides `CSHIFT` operations written by users, compilers for distributed-memory machines commonly insert them to perform data movement needed for operations on array sections that have different processor mappings [14, 15]. For example, given the statement `X(2:255) = X(1:254) + X(2:255) + X(3:256)` the CM Fortran compiler would translate it into the following statement sequence, where the temporary arrays match the size and distribution of `X`:

```
ALLOCATE TMP1, TMP2
TMP1 = CSHIFT(X,SHIFT=-1,DIM=1)
TMP2 = CSHIFT(X,SHIFT=+1,DIM=1)
```

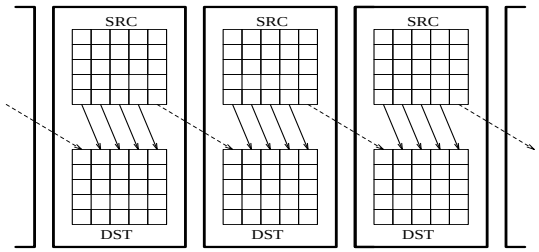


**Fig. 1.** `DST = CSHIFT(SRC,SHIFT=-1,DIM=2)`

```
X(2:255) = TMP1(2:255) + X(2:255) + TMP2(2:255)
DEALLOCATE TMP1, TMP2
```

For the rest of the paper, we assume that all `CSHIFT` operations in a program are explicit (either user-written, or inserted where appropriate during an earlier phase of HPF compilation), that each call to `CSHIFT` occurs as a singleton operation on the right-hand side of an assignment statement, and that each `CSHIFT` in our intermediate form is applied only to a whole array. All other occurrences of `CSHIFT` can be translated into the required form by factoring expressions and introducing array temporaries.

### 2.2 `CSHIFT` Operations on Distributed-Memory Machines

When a distributed array is shifted across a distributed dimension, two major actions take place:

1. Data elements that must be shifted across processing element (*PE*) boundaries are sent to the neighboring PE. This is the interprocessor component of the shift. The dashed lines in Fig. 1 represent this data movement for arrays distributed in a BLOCK fashion.
2. Data elements that stay within the memory of the PE must be copied to the appropriate locations in the destination array. This is the intraprocessor component of the shift. The solid lines in Fig. 1 represent this data movement.

Assuming a BLOCK distribution and that each PE contains a 2D subgrid of size $(g \times g)$, a shift amount of $d$, $d < g$, consists of an interprocessor move of $d$ columns (of size $g$), and an intraprocessor move of $g - d$ columns. The cost of such a shift operation is described by the following model [8]:

$$T_{\text{shift}} = g\,(g-d)\,t_{\text{onpe}} + C_{\text{onpe}} + g\,d\,t_{\text{offpe}} + C_{\text{offpe}} \qquad (1)$$

where $t_{\text{onpe}}$ and $t_{\text{offpe}}$ represent the time to perform an intraprocessor and interprocessor copy respectively, and $C_{\text{onpe}}$ and $C_{\text{offpe}}$ represent the startup time (or latency) for each type of copy. Table 1[3] presents measured values for each of

[3] From Fatoohi [8], ©1993 ACM.

**Table 1.** Measured cost of communication parameters for a 32-bit word (in $\mu$sec).

| Parameter | CM2 | MPP | MP-1 | 610C |
|---|---|---|---|---|
| $t_{\text{offpe}}$ | 9.0 | 3.2 | 2.7 | 3.2 |
| $C_{\text{offpe}}$ | 20.0 | 13.4 | 41.9 | 7.2 |
| $t_{\text{onpe}}$ | 0.7 | - | 5.6 | 9.0 |
| $C_{\text{onpe}}$ | 35.0 | - | 59.1 | 18.0 |

the model's parameters for four different SIMD machines. Different models are required for the cases $d = g$ and $d > g$. CYCLIC(K) distributions also require a different model which include a parameterization for the blocking factor.

The instances in which we are most interested occur when $d$ is small compared to $g$. For such cases Equation (1) is $O(g^2\, t_{\text{onpe}})$ and the execution time $T_{\text{shift}}$ is dominated by the cost of the intraprocessor copies, even when $t_{\text{onpe}} \ll t_{\text{offpe}}$.

## 3 Offset Arrays

The goal of the work described here is to eliminate the intraprocessor copying associated with a Fortran 90 `CSHIFT` operation when it is safe to do so. When we can determine that the intraprocessor copying of a `CSHIFT` is unnecessary, we can transform the program to perform only the interprocessor copying and rewrite references to the shift's destination array to refer to the source array with indexing adjusted by the shift amount. We call such a destination array an *offset array*. In the following subsections, we present criteria for determining when offset arrays are safe and profitable and present the code transformations that avoid intraprocessor copying by exploiting offset arrays.

To hold the data that must move between PEs, we use *overlap areas* [9]. Overlap areas are subgrid extensions to hold data received from neighboring PEs. To limit the impact of allocating this permanent storage, we place an upper bound on their size. This upper bound should be set at compile-time by a heuristic that considers the machine characteristics along with the expected size of the subgrids.

### 3.1 Criteria for Offset Arrays

Given an assignment statement `DST = CSHIFT(SRC,SHIFT,DIM)` within our intermediate representation, the array `DST` may be treated as an offset array if the following criteria can be verified for this statement at compile time:

1. The source array `SRC` is not modified while this definition of `DST` is *live*.
2. The destination array `DST` is not partially modified[4] while `SRC` is *live*.

[4] Any partial modification will require a copy of the shifted array `SRC` and so we simply go ahead and make the copy at the point of the shift. Any full modification of `DST` which *kills* the whole array does not require the copy of `SRC` and thus `DST` may still be treated as an offset array up to the point of the killing definition.

From the work on copy elimination in functional and higher-order programming languages [17], we know that the above two criteria are necessary and sufficient conditions for when the two objects can share the same storage. However, the sharing of storage may not always be profitable. To insure profitability, we add the following efficiency criteria:

3. The `SRC` array and the `DST` array are distributed in the same BLOCK (or CYCLIC(K)) fashion and are aligned with one another.
4. The values `SHIFT` and `DIM` are compile-time scalar constants.
5. The amount of interprocessor data must fit within the bounds placed on the size of the overlap areas.
6. For each use of `DST` that is reached by the given definition, all the definitions of `DST` that reach that use are identical offset arrays of the same source array `SRC`.

These efficiency criteria may be relaxed if we are willing to generate multiple versions of code for statements that use the array `DST`, and then select the appropriate version depending upon run-time conditions. However, due to the drawbacks of multiple versions of code, in particular code growth, we consider these additional criteria as important.

### 3.2 Offset Array Optimization

Once we have determined that the destination array of the assignment statement `DST = CSHIFT(SRC,SHIFT,DIM)` may be an offset array, we perform the following transformations on the code. First we replace the shift operation with a call to a routine that moves the interprocessor data into the appropriate overlap area: `CALL OFFSET_SHIFT(SRC,SHIFT,DIM)`. We then replace all uses of the array `DST`, that are reached from this definition, with a use of the array `SRC`. The newly created references to `SRC` carry along special annotations representing the values of `SHIFT` and `DIM`. In the examples that follow, the annotations are represented by a superscripted vector where the `DIM`-th element contains the value `SHIFT`; *e.g.*, $\texttt{SRC}^{<\ldots,\texttt{SHIFT},\ldots>}$. Finally, when creating subgrid loops during the code generation phase, we alter the subscript indices used for the offset arrays. The array subscript used for the offset reference to `SRC` is identical to the subscript that would have been generated for `DST` with the exception that the `DIM`-th dimension has been incremented by the `SHIFT` amount.

It is possible that offset arrays are themselves used in other shift operations. If these shift operations also meet all of the criteria to be an offset array then the above transformations can again be applied. We call such arrays *multiple-offset arrays*. If one dimension is shifted multiple times, the `SHIFT` amounts are simply added together.

As an example, consider the 5-point stencil routine in Fig. 2(a). The expected intermediate representation in Fig. 2(b) is achieved by separating the communication operations from the computational operations. Once we have determined that the temporary arrays T1–T4 can be offset arrays, we perform the above

```
SUBROUTINE FIVE_PT(a,b,n)
REAL, ARRAY(n,n) :: a,b
DECOMPOSITION decomp(n,n)
ALIGN a,b with decomp
DISTRIBUTE decomp(BLOCK,BLOCK)
REAL cc,cn,ce,cw,cs
COMMON cc,cn,ce,cw,cs

b = cc * a
& + cn * cshift(a,-1,1)
& + cs * cshift(a,+1,1)
& + cw * cshift(a,-1,2)
& + ce * cshift(a,+1,2)

RETURN
END
```

**(a)** Original program

```
SUBROUTINE FIVE_PT(a,b,n)
REAL, ARRAY(n,n) :: a,b
REAL, ALLOCATABLE :: t1,t2,t3,t4
DIMENSION(:,:) :: t1,t2,t3,t4
DECOMPOSITION decomp(n,n)
ALIGN a,b,t1,t2,t3,t4 with decomp
DISTRIBUTE decomp(BLOCK,BLOCK)
REAL cc,cn,ce,cw,cs
COMMON cc,cn,ce,cw,cs

ALLOCATE(t1(n,n),t2(n,n),
&       t3(n,n),t4(n,n))
t1 = cshift(a,-1,1)
t2 = cshift(a,+1,1)
t3 = cshift(a,-1,2)
t4 = cshift(a,+1,2)
b = cc * a
& + cn * t1
& + cs * t2
& + cw * t3
& + ce * t4
DEALLOCATE(t1,t2,t3,t4)
RETURN
END
```

**(b)** Intermediate representation

```
SUBROUTINE FIVE_PT(a,b,n)
REAL, ARRAY(n,n) :: a,b
DECOMPOSITION decomp(n,n)
ALIGN a,b with decomp
DISTRIBUTE decomp(BLOCK,BLOCK)
REAL cc,cn,ce,cw,cs
COMMON cc,cn,ce,cw,cs

CALL offset_cshift(a,-1,1)
CALL offset_cshift(a,+1,1)
CALL offset_cshift(a,-1,2)
CALL offset_cshift(a,+1,2)
b = cc * a
& + cn * a<-1,0>
& + cs * a<+1,0>
& + cw * a<0,-1>
& + ce * a<0,+1>

RETURN
END
```

**(c)** Offset array transformations

```
SUBROUTINE FIVE_PT(a,b,n)
REAL, ARRAY(n/p,n/p) :: b
REAL, ARRAY(0:n/p+1,0:n/p+1) :: a
REAL cc,cn,ce,cw,cs
COMMON cc,cn,ce,cw,cs

CALL offset_cshift(a,-1,1)
CALL offset_cshift(a,+1,1)
CALL offset_cshift(a,-1,2)
CALL offset_cshift(a,+1,2)
do j=1,n/p
 do i=1,n/p
   b(i,j) = cc * a(i,j)
&         + cn * a(i-1,j)
&         + cs * a(i+1,j)
&         + cw * a(i,j-1)
&         + ce * a(i,j+1)
 enddo
enddo

RETURN
END
```

**(d)** Final node program

**Fig. 2.** Offset array optimization on a 5-point stencil computation.

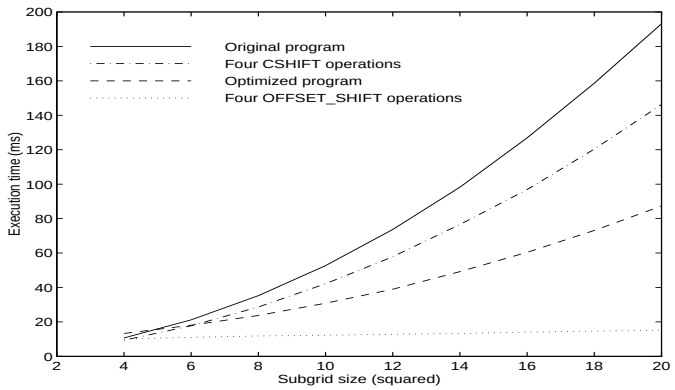


**Fig. 3.** Timings for 5-point stencil computation on 16K MasPar MP-1.

set of transformations. Fig. 2(c) shows the program after the first two transformation steps have been completed. The third step is performed during subgrid loop generation and is shown in Fig. 2(d).

To demonstrate the usefulness of this optimization, we have compiled and executed the code from Fig. 2 on a MasPar MP-1 with 16K processors. Figure 3 compares the execution times of the original program displayed in Fig. 2(a) and the optimized program shown in Fig. 2(d) for varying subgrid sizes. The figure shows that the program exploiting offset arrays gives a speed-up of a factor of two for the larger subgrid sizes. The figure also displays the time to execute the four `CSHIFT`s of Fig. 2(b) and the four calls to `OFFSET_SHIFT` of Fig. 2(d). We can see that the amount of execution time that is spent performing the four `CSHIFT` operations is actually more than the time spent performing the desired computation. In fact, the `CSHIFT` operations account for 75% of the total execution time for the largest subgrid. The corresponding number for the `OFFSET_SHIFT` operations is 17%.

## 4 Offset Array Analysis

Our algorithm for determining offset arrays relies upon the *static single assignment* (SSA) intermediate representation [7]. For our purposes, the SSA representation is an analysis framework which is used in conjunction with the control flow graph. Within SSA, modifications and uses of arrays are represented with `UPDATE` and `REFERENCE` operators, respectively. Since we are analyzing Fortran 90D programs, we have enhanced these operators to handle array sections by incorporating regular section descriptors (RSDs) [1].

In addition to the SSA graph, we generate an interference graph [3]. The interference graph indicates those SSA variables with overlapping live ranges, and is used to check for violations of criteria 1 or 2. The graph is built in the usual manner, but with one exception: all $\phi$-functions occurring at the same merge point are considered to be executed simultaneously. This prevents the

detection of spurious interferences that may prevent the use of offset arrays across merge points. We also enhance the accuracy of the interference graph by exploiting the RSDs in `UPDATE` operations to identify statements that *kill* the entire array.

### 4.1 Offset Array Algorithm

In this section, we present our algorithm for identifying offset arrays and transforming the program to reference them. In describing the algorithm, we typically refer to the SSA variable names rather than their CFG counterparts. The algorithm is shown in Fig. 4.

We begin by traversing the CFG in a reverse depth-first order looking for shift operations. Upon encountering a shift operation which satisfies criteria 3–5, we check the interference graph to see if criterion 1 is not violated. If all the required criteria are satisfied, then it is safe for the destination array to be an offset array. Given such a shift operation, we rename the destination array $\texttt{DST}_i$ by giving it the same SSA name as the source array $\texttt{SRC}_j$. The new name is annotated as described in Section 3.2. By using the same name we do not violate the spirit of SSA. This is because the shift really does not create any new values but rather just specifies a new indexing method for existing values. We change the use of the `SHIFT` intrinsic into a use of the `OFFSET_SHIFT` routine, and update the interference graph by renaming the changed variable.

After we make this change, we propagate the information in an optimistic manner. This will insure that criterion 6 is satisfied wherever possible. This is accomplished by simply following the SSA def-use edges and replacing all uses of $\texttt{DST}_i$ with uses of $\texttt{SRC}_j^{<\ldots,\texttt{SHIFT},\ldots>}$. Depending upon the type of use, further propagation may be possible. Several different cases must be handled during this propagation; we discuss them next. When the propagation of a change has completed, we continue the traversal of the program looking for the next offsetable array. The reverse depth-first traversal order is important so that multiple-offset arrays can be correctly handled in a single pass of the program.

As stated in the previous paragraph, we propagate the offset array information by changing all uses of the original destination array $\texttt{DST}_i$ into uses of the new offset array $\texttt{SRC}_j^{<\ldots,\texttt{SHIFT},\ldots>}$. Since we are dealing with arrays, these uses can only occur in three places: a `REFERENCE` operation, an `UPDATE` operation, or a $\phi$-function. If this change is propagated into an array `REFERENCE` operator, there are no more opportunities for propagation. If the use is at an `UPDATE` operator we propagate the offset array through the operation when possible. It is valid to propagate through the `UPDATE` as long as it does not violate criterion 2 and does not realign or redistribute the array. The propagation is accomplished by generating a new instance of the `SRC` array, call it $\texttt{SRC}_k$, to be the target of the `UPDATE` operation in place of the existing `DST` array instance. This new $\texttt{SRC}_k$ receives the same annotations as $\texttt{SRC}_j^{<\ldots,\texttt{SHIFT},\ldots>}$, and then is propagated to all its uses in a similar manner. If it is not possible to propagate through the `UPDATE` operation then a copy of the offset array may be required prior to the

**Procedure** *Offset_Arrays*
**Input:** *CFG*, the control flow graph for the procedure.
**Output:** *CFG* optimized with offset arrays.
/* See Fig. 5 for auxiliary routines. */

$SSA = Create\_SSA\_Form(CFG)$
$IG = Build\_Interference\_Graph(SSA, CFG)$

**for** each SHIFT operation *stmt* in a depth-first traversal of the *CFG* **do**
  push *stmt* onto stack $S$
**endfor**

**while** stack $S$ is not empty **do**
  pop *stmt* off of stack $S$
  **switch** (*stmt*)

  **case** `SHIFT` operation: $Dst_{Dsub} = shift(Src_{Ssub}^{Sanot}, shift, dim)$:
    **if** criteria 3, 4, or 5 is violated **then break endif**
    **if** $Check\_Interferences(Dst_{Dsub}, Src_{Ssub}, IG)$ **then break endif**
    calculate new annotation *Nanot* from *Sanot*, *shift*, and *dim*
    replace *stmt* with $Src_{Ssub}^{Nanot} = offset\_shift(Src_{Ssub}^{Sanot}, shift, dim)$
    **call** $Replace\_Uses(Dst_{Dsub}, Src_{Ssub}^{Nanot}, S)$
    **call** $Update\_Graphs(Dst_{Dsub}, Src_{Ssub}^{Nanot}, SSA, IG)$
    **break**

  **case** $\phi$-function: $Dst_{Dsub} = \phi(Lvar_{Lsub}^{Lanot}, Rvar_{Rsub}^{Ranot})$:
    **if** criteria 3 is violated **then break endif**
    **if** $Lvar^{Lanot} \neq Rvar^{Ranot}$ **then break**
    **elseif** $Dst_{Dsub} = Lvar_{Lsub}$ or $Dst_{Dsub} = Rvar_{Rsub}$ **then break**
    **elseif** $Lvar_{Lsub}^{Lanot} = Rvar_{Rsub}^{Ranot}$ **then**
      $New_{Nsub} = Lvar_{Lsub}$
    **else**
      $New_{Nsub} = Find\_Phi(Lvar_{Lsub}, Rvar_{Rsub}, stmt)$
    **endif**
    **if** $Check\_Interferences(Dst_{Dsub}, New_{Nsub}, IG)$ **then break endif**
    replace *stmt* with $New_{Nsub}^{Lanot} = \phi(Lvar_{Lsub}^{Lanot}, Rvar_{Rsub}^{Ranot})$
    **call** $Replace\_Uses(Dst_{Dsub}, New_{Nsub}^{Lanot}, S)$
    **call** $Update\_Graphs(Dst_{Dsub}, New_{Nsub}^{Lanot}, SSA, IG)$
    **break**

  **case** `UPDATE` operation: $Dst_{Dsub} = Update(Lvar_{Lsub}^{Lanot}, section, values)$:
    **if** criteria 3 is violated **then break endif**
    $New_{Nsub} = Gen\_Next\_SSA\_Var(Lvar)$
    **if** $Check\_Interferences(Dst_{Dsub}, New_{Nsub}, IG)$ **then break endif**
    replace *stmt* with $New_{Nsub}^{Lanot} = Update(Lvar_{Lsub}^{Lanot}, section, values)$
    **call** $Replace\_Uses(Dst_{Dsub}, New_{Nsub}^{Lanot}, S)$
    **call** $Update\_Graphs(Dst_{Dsub}, New_{Nsub}^{Lanot}, SSA, IG)$
    **break**

  **endswitch**
**endwhile**

**call** $Insert\_Copies(CFG, SSA)$
**return** *CFG*

**Fig. 4.** Offset array propagation algorithm

**Function** *Check_Interferences*($Dst_{Dsub}$, $Src_{Ssub}$, *IG*)
/* Return TRUE if there exists an interference between */
/* $Dst_{Dsub}$ and some $Src_i$, $i \neq Ssub$. */

**Procedure** *Replace_Uses*(*Dest*, *Src*, *S*)
/* Replace each reference to *Dest* with a reference to *Src*. */
/* If the use is in a $\phi$-function or `UPDATE` operation, then */
/* push operation on stack *S*. */

**Procedure** *Update_Graphs*(*Old*, *New*, *SSA*, *IG*)
/* Replace node *Old* with node *New* in both *SSA* and *IG*. */

**Function** *Find_Phi*($Lvar_{Lsub}$, $Rvar_{Rsub}$, *stmt*)
/* Find the $\phi$-function merging $Lvar_{Lsub}$ and $Rvar_{Rsub}$ */
/* at the same merge point as *stmt* and return it. */
/* If it does not exist, return a new instance of *Lvar*. */

**Fig. 5.** Auxiliary procedures.

`UPDATE`. This copy is inserted by a subsequent phase which we describe in the next subsection.

When propagating an offset array into a $\phi$-function, in addition to verifying criteria 1 and 3, it is only valid to continue the propagation if the other input to the $\phi$-function is an equivalent offset array (see criterion 6). Two offset arrays are equivalent if they are from the same SSA family and have identical annotations. The one exception to this rule is if a cycle has been created (*i.e.*; one of the inputs to the $\phi$-function, when its annotation is removed, is the same SSA variable being defined by the $\phi$-function). When it is possible to propagate through the $\phi$-function, we need to select the correct SSA variable to receive the definition of this $\phi$-function. If the $\phi$-function happens to be merging identical values, we simply use one of its inputs as the target variable. Otherwise we look for a $\phi$-function at the same merge point whose inputs are the unannotated variables of the current $\phi$-function. If found, we use the SSA variable that it defines, otherwise we generate a new instance of the SSA variable in the same manner as we did for `UPDATE`. In any case, the variable is annotated with the same annotation as the input variables and is propagated forward. If it is not possible to propagate through a $\phi$-function, then array copy statements must be inserted on the appropriate branches leading to the $\phi$-function. These copies are added by the *Insert_Copies* routine, which is the last function called by the *Offset_Arrays* procedure and which we describe next.

### 4.2 Inserting Array Copies

Once we have found all the offset arrays and propagated them as far as possible through the program, it may be necessary to insert some array copy statements to maintain the original semantics. The copy statements may be needed at points where an offset array is used to define, via an `UPDATE` operation or a $\phi$-function, a non-offset array.

```
Procedure Insert_Copies(CFG, SSA, stmt_set)
/* stmt_set is produced in Offset_Arrays as described */
for each stmt in stmt_set do
  switch (stmt)
  case UPDATE operation: Dst_Dsub = Update(Lvar_Lsub^Lanot, section, values):
    if (section does not specify the entire array) then
      insert Dst = Lvar^Lanot immediately preceding stmt in CFG
    endif
    break
  case φ-function: Dst_Dsub = φ(Lvar_Lsub^Lanot, Rvar_Rsub^Ranot):
    if (Lanot ≠ nil) then /* Rvar is handled similarly. */
      insert Dst = Lvar^Lanot on appropriate branch in CFG
      /* optimize placement when possible. */
    endif
    break
  endswitch
endfor
return
```

**Fig. 6.** Algorithm to insert array copy statements.

It is quite easy to determine the statements that may require a copy while we are propagating offset arrays in the *Offset_Arrays* procedure. To track these statements, we maintain a set (list) of such statements (the code has been omitted from Fig. 4). An `UPDATE` operation which is processed by the algorithm but determined not to be offsetable is added to the set. A $\phi$-function which is determined not to be offsetable is also added to the set, and marked so that it is added only once. If it is later determined to actually be offsetable (after the other input parameter has been processed), then it is removed from the set if it is marked as being included. The use of *pruned-SSA form*, where dead $\phi$-functions have been eliminated, can greatly reduce the number of $\phi$-functions added to the set.

After the propagation of offset arrays has completed, the procedure *Insert_Copies*, shown in Fig. 6, is called to add the required array copies to the program. It examines each statement in the set that was produced to determine if an array copy is actually needed and to select the best placement for it. If the array copy is truly required, it will take as input the offset array and will define the array originally used by the statement. This copy statement will perform all the intraprocessor data movement that was avoided at the shift operation. Note however that no interprocessor data movement is required.

Given an `UPDATE` operation from the set, an array copy statement is not required if the `UPDATE` is a *killing* definition. Otherwise, we insert an array copy immediately preceding the `UPDATE`.

For a $\phi$-function which defines a non-offset array, an array copy statement will need to be generated for each input parameter that is an offset array. In general, the copy statements are placed on the appropriate branches leading to the merge point represented by the $\phi$-function. It is possible to optimize this

placement in the case of some loop structures. If a copy must be made for the array values coming around from the previous iteration but there is no use within the loop of the values defined by the $\phi$-function, then it is possible to move the array copy out of the loop by placing it on the loop exit branch. The copy is moved to the shallowest nesting level such that it still dominates all uses. This is advantageous in situations where it is allowable for an array to be an offset array inside a loop nest but not outside. The full copy is only performed when the loop nest is exited rather than on each iteration.

The insertion of such array copy statements into the program raises a concern. We must answer the question of whether these inserted copy statements can generate more data movement than was specified in the original program. The following theorem states that this is not possible.

**Theorem 1.** *Given a copy statement $C$ created for an offset array* $\texttt{SRC}^{anot}$ *which was generated by a set of shift operations $\{S_1, S_2, S_3, \ldots\}$, $C$ is never executed more often than $\{S_1, S_2, S_3, \ldots\}$.*

*Proof.* The theorem follows directly from the following two lemmas. □

**Lemma 2.** *Any path from the beginning of the program (Root) to $C$ must go through at least one $S_i$.*

*Proof.* Assume there exists a path $P_1$: *Root* $\xrightarrow{*} C$ that does not contain a shift operation $S_i$. Since $C$ is an inserted copy statement for an offset array, there must exist a shift operation $S_j$ and a path $P_2$: $S_j \xrightarrow{*} C$. Since both $P_1$ and $P_2$ end at $C$, and $P_1$ does not contain $S_j$ (by assumption), then there must exist a merge point $X$ that joins $P_1$ and $P_2$ prior to $C$. $X$ must contain a $\phi$-function which merges the values of $\texttt{SRC}^{anot}$ generated at $S_j$ with the other values of SRC that reach $C$ along *Root* $\xrightarrow{*} X$. But our algorithm only propagates an offset array through a $\phi$-function when the $\phi$-function merges identical offset arrays. Thus there must exist a shift operation $S_i$ identical to $S_j$ on *Root* $\xrightarrow{*} X$ which contradicts our original assumption. □

**Lemma 3.** *$C$ cannot be more deeply nested than all $S_i \in \{S_1, S_2, S_3, \ldots\}$.*

*Proof.* Assume $C$ is contained in a loop which does not contain a shift operation $S_i$. Since $C$ is an inserted copy statement for an offset array, there must exist a shift operation $S_j$ outside the loop and a path $P$: $S_j \xrightarrow{*} C$. The path $P$ must contain a $\phi$-function to merge the values reaching $C$ from $S_j$ with those that reach $C$ from the back edge of the loop. By the same argument used in the proof of Lemma 2, there must exist a shift operation $S_i$ within the loop that reaches the back edge of the loop, thus contradicting our original assumption. □

### 4.3 Cost Analysis

During our offset array algorithm each SSA def/use edge is processed at most once. Thus our algorithm is guaranteed to terminate. This also means that our

algorithm is quite efficient. The cost of the algorithm is actually dominated by the cost of generating SSA form and building the interference graph, both of which are $O(n^2)$ in the worst case (although building SSA is $O(n)$ in practice [7]). Once these structures are built, the rest of the algorithm is linear. Finding offsetable arrays is $O(n)$ and their propagation through the program is $O(e)$. In addition, the checking of interferences is $O(i)$. Here $n$ is the size of the program, $e$ is the number if edges in the SSA graph, and $i$ is the number of edges in the interference graph.

## 5 Related Work

**Stencil Compiler:** The stencil compiler [2, 4] for the CM-2 avoids the memory-to-memory copying for shift operations that occur within specific, stylized, array-assignment statements. These statements, or *stencils*, must be in the form of a weighted sum of circularly-shifted arrays. Not only does the compiler eliminate intraprocessor data movement for these statements, it also optimizes interprocessor data movement by using the CM-2's multidimensional and bidirectional interconnect, and exploits hand-optimized library microcode to minimize data movement between local memory and registers. However, use of this special-purpose compiler requires that the user identify these stylized assignment statements in the source program and separate them into their own subroutine.

Our compiler scheme is a superset of the stencil compiler. We hoist all shift operations, whether implicit or explicit, out of expressions and assign them to array temporaries. This allows us to handle all shift operations, whether part of larger expression or not, in a uniform manner. Since hoisted temporaries have short life spans and thus never have conflicting uses, we will always be able to make them into offset arrays.

Currently we do not plan to exploit multidimensional and bidirectional communication, since they are exclusive to the CM-2's slicewise model. Although, it would not be difficult to scan adjacent communication operations looking for opportunities. Our context partitioning optimization [13] groups together as many such operations as possible, thus maximizing the possibilities of finding such opportunities.

Finally, to match the performance of the stencil compiler, we would exploit a highly-optimizing node compiler to perform final code generation. Such a compiler would consider the memory hierarchy and attempt to minimize data movement between local memory and registers [5].

**Scalarizing Compilers:** Previous work on Fortran 90D [6], like the stencil compiler, is capable of avoiding some intraprocessor data movement for stylized expressions. In this case, the expressions have to use array syntax. The compiler translates the array syntax expressions into equivalent Fortran 77D code using FORALL statements. It is then the job of the Fortran 77D back end, using dependence information, to determine the exact amount of interprocessor communication required. Unfortunately, any call to `CSHIFT`, whether in an assignment statement or as part of an expression, still makes a full copy of the

array. As with the stencil compiler, our work is a superset of this work.

**Functional Languages:** Functional and many high-level languages have value semantics, and thus do not have the concept of state and variable as in Fortran. Naive compilation of such languages causes the insertion of many copy operations of aggregate objects to maintain program semantics. It is imperative that compilers for such languages eliminate a majority of the unnecessary copies if they hope to generate efficient code. This task is known as *copy optimization*.

Schwartz [17] characterizes the task of copy optimization as the destructive use (reuse) of an object $v$ at a point $P$ in the program where it can be shown that all other objects that may contain $v$ are *dead* at $P$. He then develops a set of *value transmission functions* that can be used to determine the safety of a destructive use within the language SETL.

Gopinath and Hennessy [10] address the problem of copy elimination by *targeting*, or the proper selection of a storage area for evaluating an expression. For the lambda calculus extended with array operation constructs, they develop a set of equations which, when solved iteratively to a fixpoint, specify targets for array parameters and expressions. Unfortunately, solving their equations to a fixpoint is at least exponential in time.

Schnorf *et al.* [16] describe their efforts to eliminate aggregate copies in the single-assignment language SISAL. Their work analyzes edges in a data flow graph and attempts to determine when edges, representing values, may share storage. Our work has some similarities to parts of their work.

## 6 Conclusion

In this paper, we have presented a unified framework for analyzing and optimizing shift operations on distributed-memory multicomputers. The framework is capable of handling all such operations, whether written by the user or generated internally by the compiler. This work supersedes prior work by others that only handled shifts embedded within expressions. And although this paper has concentrated on distributed-memory machines, the optimizations presented are also applicable to scalar and shared-memory machines.

An implementation of the algorithms described in this paper is currently in progress.

## 7 Acknowledgments

We'd like to thank Cliff Click, Paul Havlak, and Mike Paleczny for the many fruitful discussions regarding our usage of SSA. We'd also like to thank Ralph Brickner for sharing some of his knowledge on compiling stencils.

## References

1. J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Dept. of Computer Science, Rice University, April 1983.

2. R. G. Brickner, W. George, S. L. Johnsson, and A. Ruttenberg. A stencil compiler for the Connection Machine models CM-2/200. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.
3. P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Dept. of Computer Science, Rice University, April 1992.
4. M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.
5. S. Carr. *Memory-Hierarchy Management.* PhD thesis, Dept. of Computer Science, Rice University, September 1992.
6. A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and C.-W. Tseng. Unified compilation of Fortran 77D and 90D. *ACM Letters on Programming Languages and Systems*, 2(1–4):95–114, March–December 1993.
7. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
8. R. Fatoohi. Performance analysis of four SIMD machines. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.
9. M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.
10. K. Gopinath and J. L. Hennessy. Copy elimination in functional languages. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1989.
11. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.
12. S. L. Johnsson. Language and compiler issues in scalable high performance scientific libraries. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.
13. K. Kennedy and G. Roth. Context optimization for SIMD execution. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.
14. K. Knobe, J. Lukas, and M. Weiss. Optimization techniques for SIMD Fortran compilers. *Concurrency: Practice and Experience*, 5(7):527–552, October 1993.
15. G. Sabot. A compiler for a massively parallel distributed memory MIMD computer. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.
16. P. Schnorf, M. Ganapathi, and J. Hennessy. Compile-time copy elimination. *Software—Practice and Experience*, 23(11):1175–1200, November 1993.
17. J. T. Schwartz. Optimization of very high level languages – I. Value transmission and its corollaries. *Computer Languages*, 1(2):161–194, 1975.

This article was processed using the LaTeX macro package with LLNCS style