

# Enhanced Parallelization via Analyses and Transformations on Array SSA Form

Kathleen Knobe  
Compaq Cambridge Research Lab  
One Kendall Sq., Building 700, Ste 721  
Cambridge, MA 02139, USA  
knobe@crl.dec.com

Vivek Sarkar  
IBM Thomas J. Watson Research Center  
P.O. Box 704  
Yorktown Heights, NY 10598, USA  
vsarkar@us.ibm.com

## Abstract

*Array SSA form is a version of SSA form that captures precise element-level data flow information for array variables. As an example of program analysis using Array SSA form, we presented a conditional constant propagation algorithm that can lead to discovery of a larger set of constants than previous algorithms that analyze only scalar variables [7]. As an example of program transformation using Array SSA form, we showed that making its element-level  $\phi$  functions manifest at run-time can increase the set of parallelizable loops [6].*

*In this paper, we extend our past work by showing how Array SSA form can be used to perform analyses and transformations that go beyond loop parallelization. These transformations (which include region/task parallelism, pipeline parallelism, speculation, and general forms of code reordering) rely on two important properties of Array SSA form. First, because the  $\phi$  operation is truly functional in Array SSA form, it can be subjected to classical optimizations and transformations (e.g., copy propagation, reassociation, code reordering) like other operations. Second, Array SSA form, like scalar SSA form, creates a factoring of reaching definitions. However, in the array case this factoring can be refined by using element-level information. Specifically, we present a static analysis technique called resolution that reduces the number of definitions that appear to reach a use. Resolution is global in scope and is distinct from traditional dependence analysis which focuses on array references in loops.*

## 1 Introduction

A long-standing goal for parallelizing compilers and program analysis tools is to find efficient algorithms that can reveal the flow of data in a program as precisely as possible. Static single assignment (SSA) form [4] has been a significant advance in this regard for scalar variables.

In addition to enabling efficient scalar analysis algorithms, the renaming of scalars in SSA form has been useful for enhancing parallelism by eliminating storage-related dependences on scalars. However, an array is viewed as a single object in scalar SSA form. This is a serious limitation because the kinds of analyses that sophisticated parallelizing compilers need to perform on arrays e.g., array privatization and loop parallelization, need to be performed at the element level.

In past work, we introduced an Array SSA form that captures precise element-level data flow information for array variables. It is general and simple, and coincides with scalar SSA form when applied to scalar variables. Array SSA form provides for renaming of array variables and uses a  $\phi$  function to identify the defining assignment for each array element. The combination of array renaming and the ability to specify element-level merges of distinct definitions in Array SSA form significantly increases the potential for parallelization and code reordering. As an example of program analysis using Array SSA form, we presented a conditional constant propagation algorithm that can lead to discovery of a larger set of constants than previous algorithms that analyze only scalar variables [7]. As an example of program transformation using Array SSA form, we showed that making its element-level  $\phi$  functions manifest at run-time can increase the set of parallelizable loops [6].

In this paper, we extend our past work by showing how Array SSA form can be used to perform analyses and transformations that go beyond loop parallelization. These transformations (which include region/task parallelism, pipeline parallelism, speculation, and general forms of code reordering) rely on two important properties of Array SSA form. First, the introduction of “@ variables” in Array SSA form enable its  $\phi$  operations to be *pure functions* i.e., functions whose results depends only on the arguments. Therefore,  $\phi$  operations can be subjected to classical optimizations and transformations (e.g., copy propagation, reassociation, code reordering) like other operations. Second, Array SSA

```

{X initialized here.}
do i...
  if (C[i]) then
    X[f(i)] := ...
  endif
enddo
... := X[...]
```

**Figure 1. Loop L with Conditional and Indirection**

form, like scalar SSA form, creates a factoring of reaching definitions. However, in the array case, this factoring can be refined by using element-level information. Specifically, we present a static analysis technique called *resolution* that reduces the number of definitions that appear to reach a use. Resolution is global in scope and is distinct from traditional dependence analysis which focuses on array references in loops.

The rest of the paper is organized as follows. Section 2 summarizes Array SSA form and discusses the placement and semantics of  $\phi$  functions. Section 3 discusses the impact of applying classical optimizations to  $\phi$  functions in Array SSA form. Section 4 gives examples of enhanced parallelization using Array SSA form. Section 5 discusses resolution of  $\phi$  functions. Section 6 discusses related work, and Section 7 contains our conclusions and indicates possible directions for future work.

## 2 Placement and Semantics of $\phi$ functions

The salient properties of Array SSA form are as follows [6]:

1. Each array (and scalar) definition is given a new name.
2. Each use of an array (or scalar) variable refers to a unique name/variable.
3.  $\phi$  functions are introduced at certain program points to generate new definitions that combine the results of several definitions. Since an array-element assignment does not kill the entire array, a new *definition*  $\phi$  function is used to represent the merge of the new array-element assignment with the prior array value.

Section 2.1 addresses the issue of  $\phi$  function placement in Array SSA form, and Section 2.2 discusses the semantics of these functions and how they can be implemented at runtime when so desired.

### 2.1 $\phi$ Function Placement

This section addresses the question of where to place  $\phi$  functions in Array SSA form. As in scalar SSA, the key

```

/* X0 is the initializing definition
   from figure 1 */
X0[...] :=
do i := 1, n
  X1 :=  $\phi(X4, X0)$ 
  if (C[i]) then
    X2[f(i)] := ...
    X3 :=  $\phi(X2[f(i)], X1)$ 
  end if
  X4 :=  $\phi(X3, X1)$ 
end do
X5 :=  $\phi(X4, X0)$ 
... := X5[...]
```

**Figure 2. Partial Array SSA form for loop from figure 1**

requirement on  $\phi$  placement is that it enable each use to refer to a single name. The initial placement that we present here is an extension of the  $\phi$  placement used by scalar SSA; however, this choice is not fundamental, and subsequent optimization of  $\phi$  nodes (addressed in Section 3) may result in a different but semantically equivalent placement of  $\phi$  functions.

Initial placement of  $\phi$  functions follows two rules:

#### 1. **definition** $\phi$

A  $\phi$  function is inserted immediately after each definition of an array variable that does not completely kill the array value. This *definition*  $\phi$  merges the values of the element(s) modified in the definition with the values available immediately prior to the definition. Definition  $\phi$ 's need not be inserted for definitions of scalar variables.

For example, consider the def,  $X[f(i)] := \dots$ , in figure 1, and assume that the def has been renamed to  $X_2[f(i)] := \dots$ . Also, let  $X_1$  be the def of variable  $X$  that reaches the point just prior to the def of  $X_2$ . Then, the  $\phi$  function,  $X_3 := \phi(X_2, X_1)$  is inserted immediately after the def of  $X_2$  to represent an element-level merge of  $X_2$  and  $X_1$ ; any subsequent use of variable  $X$  (before an intervening def) will simply refer to  $X_3$  instead.

#### 2. **control** $\phi$

A  $\phi$  function is inserted at exactly at the same locations where scalar SSA would have inserted a  $\phi$  function (at the dominance frontier [4]). As in scalar SSA, the purpose of this *control*  $\phi$  function is to merge values computed along distinct control paths.

As an example, we can see that placement of these definition  $\phi$  and control  $\phi$  functions would convert the code

```

/* All @ array elements have
   initial value =  $\perp$  */
@X0[1:m] :=  $\perp$  ; @X1[1:m] :=  $\perp$  ;
@X2[1:m] :=  $\perp$  ; @X3[1:m] :=  $\perp$  ;
@X4[1:m] :=  $\perp$  ; @X5[1:m] :=  $\perp$  ;

X0[...] :=
@X0[...] := ()
do i := 1, n
  X1 :=  $\Phi(X_4, @X_4, X_0, @X_0)$ 
  @X1 := max(@X4, @X0)
  if (C[i]) then
    X2[f(i)] := ...
    @X2[f(i)] := (i)
    X3 :=  $\Phi(X_2, @X_2, X_1, @X_1)$ 
    @X3 := max(@X2, @X1)
  end if
  X4 :=  $\Phi(X_3, @X_3, X_1, @X_1)$ 
  @X4 := max(@X3, @X1)
end do
X5 :=  $\Phi(X_4, @X_4, X_0, @X_0)$ 
@X5 := max(@X4, @X0)
... := X5[...]

```

**Figure 3. Full Array SSA form — after insertion of @ arrays**

$$X_3[j] = \begin{array}{ll} \text{if} & @X_2[j] \succeq @X_1[j] \text{ then } X_2[j] \\ \text{else} & X_1[j] \\ \text{end if} & \end{array}$$

**Figure 4. Conditional Expression Semantics for  $X_3 = \Phi(X_2, @X_2, X_1, @X_1)$**

for loop  $L$  in Figure 1 to the Array SSA form shown in figure 2. The algorithm for computing the initial Array SSA form for a program uses the  $\phi$  placement algorithm for scalar SSA form [4] to determine the placement of control  $\phi$  functions. In addition, definition  $\phi$  functions are inserted as described above.

The Array SSA form just described is called *partial Array SSA form*. It is used for static analysis such as the conditional constant propagation algorithm described in [7] and the  $\phi$  function resolution discussed in section 5. *Full Array SSA form* is used when the form is made manifest at run-time, and also for defining the semantics of Array SSA form as described below.

Unoptimized Version:  
 $X_2[f(i)] := \dots$   
 $@X_2[f(i)] := (i)$   
 $X_3 := \Phi(X_2, @X_2, X_1, @X_1)$

Optimized Version:  
 $X_{2f(i)} := \dots$   
 $X_3 := \Phi(X_{2f(i)}, @X_2, X_1, @X_1)$

**Figure 5. Optimization of Definition  $\Phi$**

$$X_3[j] = \begin{array}{ll} \text{if} & j = i \text{ then } X_{2i} \\ \text{else} & X_1[j] \\ \text{end if} & \end{array}$$

**Figure 6. Optimized Semantics for Definition  $\Phi$ ,  $X_3 = \Phi(X_{2i}, X_1, @X_1)$**

## 2.2 $\Phi$ Function Semantics and Implementation

To define the semantics of  $\phi$  functions, we introduce the concept of an @ array. For each static definition  $X_k$  the @ array  $@X_k$  identifies the most recent “time” at which each element of  $X_k$  was modified by this definition. The initial value of each @ array element is  $\perp$ .  $@X_k[j] = \perp$  indicates that element  $j$  has not (yet) been modified by static definition  $X_j$ .

For an *acyclic* control flow graph, a static definition  $X_k$  may execute either zero times or one time. These two cases can be simply encoded as  $@X_k[j] = \perp$  and  $@X_k[j] \neq \perp$  (for each element  $j$ ). For a control flow graph with *cycles* (loops), a static definition  $X_k$  may execute an arbitrary number of times. Therefore, we need more detailed information for the  $@X_k[j] \neq \perp$  case so as to distinguish among different dynamic execution instances of static definition  $X_k$ . Specifically,  $@X_k[j]$  contains the *iteration vector* at which element  $j$  was last modified by static definition  $X_k$ .

The *iteration vector* of a static definition  $X_k$  is a single point in the iteration space of the set of loops that enclose the definition. Let  $n$  be the number of loops that enclose a given definition. (The loops need not be counted loops and need not be perfectly nested. Our only assumption is that all loops are single-entry, or equivalently, that the control flow graph is *reducible* [5, 1].) A single point in the iteration space is specified by the iteration vector  $\vec{i} = (i_1, \dots, i_n)$ , which is an  $n$ -tuple of iteration numbers one for each enclosing loop.

The values of @ array elements can be computed as follows. We assume that all @ array elements,  $@X_k[j]$ ,

have the value  $\perp$  at the start of program execution. For each real (non- $\phi$ ) definition,  $X_k[s]$ , we assume that a statement of the form  $@X_k[s] := \vec{i}$  is inserted immediately after the def (where  $s$  is an arbitrary subscript value and  $\vec{i}$  is the current iteration vector for all loops that surround  $X_k$ ). Each  $\phi$  definition also has an associated  $@$  array. Figure 3 shows the result of inserting  $@$  array computations into figure 2 so as to obtain the full Array SSA form for loop  $L$ . We use the notation,  $\Phi$ , for the  $\phi$  function augmented with  $@$  variable arguments. The semantics of a  $\Phi$  function can be specified by a conditional expression that is a pure function of its input arguments, as shown in Figure 4 ( $\succeq$  represents the *lexicographic*  $\succeq$  relationship on iteration vectors).

We now discuss a simple optimization that can be performed when an array assignment only updates a single element (which is the common case for array assignments). Consider the assignment to  $X_2[f(i)]$  in Figure 3. Note that only a single element of  $X_2$  is assigned, after which  $X_2$  is immediately incorporated into  $X_3$ . Therefore, we replace the array definition of  $X_2[f(i)]$  by the definition<sup>1</sup> of a scalar temporary  $X_{2f(i)}$ , as shown in figure 5. The conditional expression semantics of this optimized definition  $\Phi$  can then be rewritten using  $X_{2f(i)}$ , as shown in Figure 6.

In addition to removing storage-related dependences by renaming scalars and arrays, Array SSA form introduces a new dimension for reordering by converting the  $\phi$  operation to a functional version, the  $\Phi$  function, whose results depend only on its arguments. The next section shows how this conversion enables a variety of classical optimizations to be performed on  $\Phi$  functions.

### 3 Classical Optimizations Applied to $\phi$ Operations

This section describes a set of classical transformations that can be applied to  $\Phi$  operations in Array SSA form. These transformations can be useful for setting the stage for parallelization. The key feature of Array SSA form that enables these transformations is that the inclusion of the  $@$  variables as arguments to the  $\Phi$  operation ensures that  $\Phi$  is a pure function. Traditional optimizations such as copy propagation, loop-invariant code motion, common subexpression elimination, elimination of partial redundancies and dead store elimination can all be used to eliminate or re-position computations for  $@$  arrays and  $\Phi$  functions.

The transformations discussed in this section are often performed in combination. For this reason, the figures illustrating code prior to a given transformation will not

<sup>1</sup>The only purpose of the  $f(i)$  subscript in the name,  $X_{2f(i)}$ , is to associate a unique scalar temporary with the static  $X_2[f(i)]$  array definition. The runtime value of  $f(i)$  has no impact on the scalar temporary that's used.

```

{X1 in effect here.}
do i1 = 1, n
...
X2[i1] := rhs2(i1)
@X2[i1] := < i1 >
...
enddo
X3 = Φ(X2, @X2, X1, @X1)
do i2 = 1, n
...
lhs3[i2] := ... X3[i2] ...
...
enddo

```

Figure 7. Before copy propagation.

always show unoptimized Array SSA form but may instead show the output of some other transformations.

#### 3.1 Copy Propagation

In this section, we describe copy propagation applied to the  $\Phi$  function. Consider the code in figure 7. Notice that the  $\Phi$  operation has the effect of requiring completion of the loop on  $i1$  before the start of  $i2$  since the  $\Phi$  takes  $X_2$  as input and generates  $X_3$ . However, this example can be converted to the code in figure 8 by performing copy propagation on  $X_3$ . We use the notation,  $\Phi(X_2[j], @X_2[j], X_1[j], @X_1[j])$  to refer to the  $\Phi$  computation applied to a given element,  $j$ . Note that a producer-consumer form of pipeline parallelism can now be set up between the two loops in Figure 8.

As discussed in [6], copy propagation on Array SSA form can also enable “element-level dead code elimination” in which an array assignment such as  $X_2[i] := rhs(i)$  is propagated into a use,  $X_2[j]$ , by rewriting  $X_2[j]$  as  $rhs(j)$  (provided that the  $rhs$  computation has no side effects). If all uses of  $X_2$  are propagated in this manner, then the  $X_2[i] := rhs(i)$  assignment can be eliminated.

#### 3.2 Associativity

In this section, we describe reordering based on associativity of the  $\Phi$  operator. This associativity applies both among distinct static  $\Phi$  operations and among distinct dynamic executions of a given static  $\Phi$ .

##### 3.2.1 Static Reassociation

We observe that  $\phi$  functions in scalar and Array SSA form provide a factoring of use-def chains. For example, if there are assignments to array  $X$  named  $X_1$ ,  $X_2$  and  $X_3$  in straightline code then the factoring can be represented

```

{X1 in effect here.}
do i1 = 1, n
  ...
  X2[i1] := rhs2(i1)
  @X2[i1] := < i1 >
  ...
enddo
do i2 = 1, n
  ...
  lhs3[i2] := ... Φ(X2[i2], @X2[i2], X1[i2], @X1[i2]) ...
  ...
enddo

```

**Figure 8. After propagation of definition  $X_3$  into its use.**

```

{X1 in effect here.}
do i2 = 1, n
  ...
  X2[i2] := rhs2(i2)
  @X2[i2] := < i2 >
  ...
enddo
X3 = Φ(X2, @X2, X1, @X1)
do i4 = 1, n
  ...
  X4[i4] := rhs4(i4)
  @X4[i4] := < i4 >
  ...
enddo
X5 = Φ(X4, @X4, X3, @X3)

```

**Figure 9. Before Reassociation**

```

{X1 in effect here.}
do i2 = 1, n
  ...
  X2[i2] := rhs2(i2)
  @X2[i2] := < i2 >
  ...
enddo
do i4 = 1, n
  ...
  X4[i4] := rhs4(i4)
  @X4[i4] := < i4 >
  ...
enddo
X6 = Φ(X4, @X4, X2, @X2)
X7 = Φ(X6, @X6, X1, @X1)

```

**Figure 10. After Reassociation**

as<sup>2</sup>  $\phi(\phi(X_1, X_2), X_3)$ . We use this notation as shorthand to indicate that the outer  $\phi$  takes the results of the inner  $\phi$  as an argument. Typically, the two  $\phi$  operations occur at distinct locations in the code, though the compound expression shown above could be obtained as a result of copy propagation.

In unoptimized Array SSA, all  $\phi$  expressions represent the cumulative state from the beginning of the program *i.e.*, from the name that represents the initial value of the array. However,  $\phi$  functions can be associative in many cases, making it possible to reassociate the terms in the factoring *e.g.*, to use an equivalence such as  $\phi(\phi(X_1, X_2), X_3) \equiv \phi(X_1, \phi(X_2, X_3))$ . Note that the version,  $\phi(X_1, \phi(X_2, X_3))$ , is not cumulative from the start, since its computation begins with  $X_2$  and  $X_3$ , rather than  $X_1$ . Although the results of reassociated  $\phi$  expressions are identical, some of their characteristics may differ as follows:

- The “cumulative from program start” property may not hold for some versions.
- Some versions are inherently more concurrent than others.
- The intermediate values available differ among the distinct versions, *e.g.*, consider  $\phi(X_1, X_2)$  vs.  $\phi(X_2, X_3)$

An example of reassociation is given in Figures 9 and 10.  $\phi$  computations in unoptimized Array SSA form guarantee that each array access refers to the correct values. We must make the same guarantee after reassociation. The  $\phi$  structure identifies reaching definitions. The transformed

<sup>2</sup>We use the  $\phi$  notation in this section, since reassociation can be applied to  $\phi$  functions in Partial Array SSA form as well as  $\Phi$  functions in Full Array SSA form.

program after reassociating  $\phi$  functions will be correct if each reference is reached by the same set of source definitions (under the same conditions) both before and after the transformation. This is trivially true if intermediate arrays are not referenced. Section 5 shows that it is possible to perform reassociation even when intermediate arrays are referenced.

A secondary benefit of allowing reassociation to eliminate the “cumulative from start” property, is that it provides a foundation for summarizing the computation of a region of code such as a procedure, a loop or a loop nest in a manner that’s independent of its context. The summary can then be combined with the values in effect on entry to the region.

### 3.2.2 Dynamic Reassociation

The associativity transformations described above produce distinct *static factorings*, that is factoring on the basis of lexically distinct assignments. But we might also consider transformations based on *dynamic factoring*. Just as static factoring is naively assumed to be cumulative from the beginning of the program, multiple executions of a single static  $\phi$  operation within a loop are assumed to occur in a sequential order as specified by the original loop. Instead, we can sometimes reorder these dynamic executions for more effective computation.

Consider a loop containing a  $\phi$  function that incorporates new values each time through the loop cumulatively from the beginning in order of the loop iterations. This  $\phi$  function appears to involve a loop carried true dependence which inhibits parallelization. However, if the iterations are distributed among  $n$  processors (say), each processor could accumulate local results prior to an  $n$ -way merge of the final results. This amounts to reassociating the dynamic executions of a static assignment.

### 3.3 Solving Recurrences

In this section, we outline how the Array SSA form of a program can be transformed by solving recurrences on @ arrays and  $\Phi$  functions. Consider the code in Figure 11, which is a transformed version (after copy propagation) of the original Array SSA form in Figure 3. The max computation inside loop  $i$  in Figure 11 forms a recurrence as shown below, where the superscript,  $i$ , indicates the current iteration.

$$@X_4^i := \max(@X_2^i, @X_4^{i-1}, @X_0).$$

Note that the value of an @ array must be monotonically nondecreasing as a function of  $i$ , i.e.,  $@X_2^i[j] \succeq @X_2^{i-1}[j]$  for each element  $j$  and each iteration  $i \geq 1$  (assuming that  $@X_2^0[j]$  represents the initial  $\perp$  value of  $@X_2[j]$ ).

```

X0[...] :=
@X0[...] := ()
do i := 1, n
  if (...) then
    X2[f(i)] := ...
    @X2[f(i)] := (i)
  endif
  X4 := Φ(X2, @X2, X4, @X4, X0, @X0)
  @X4 := max(@X2, @X4, @X0)
end do
X5 := Φ(X4, @X4, X0, @X0)
@X5 := max(@X4, @X0)
... := X5[...]

```

Figure 11. A recurrence on  $X_4$

```

X0[...] :=
@X0[...] := ()
do i := 1, n
  if (...) then
    X2[f(i)] := ...
    @X2[f(i)] := (i)
  end if
end do
X5 := Φ(X2, @X2, X0, @X0)
@X5 := max(@X2, @X0)
... := X5[...]

```

Figure 12. Optimized version of Figure 11

Therefore the recurrence can be solved to obtain  $@X_4^i[j] = \max(@X_2^i[j], X_0[j])$ , for each element,  $j$ .

Similarly, the conditional expression for the  $\Phi$  definition  $X_4 := \Phi(X_2, @X_2, X_4, @X_4, X_0, @X_0)$  in Figure 11 is really a recurrence that defines  $X_4^i[j]$  as a function of  $X_4^{i-1}[j]$  as follows:

$$X_4^i[j] = \begin{array}{ll} \text{if} & @X_2^i[j] \succeq @X_4^{i-1}[j] \\ \text{then} & X_2^i[j] \\ \text{else if} & @X_4^{i-1}[j] \succeq @X_0[j] \\ \text{then} & X_4^{i-1}[j] \\ \text{else} & X_0[j] \\ \text{end if} & \end{array}$$

Again, observing that  $@X_2^i[j]$  is a monotonically nondecreasing function of  $i$  leads to the following solution to the recurrence:

$$X_4^i[j] = \begin{array}{ll} \text{if} & @X_2^i[j] \succeq @X_0[j] \\ \text{then} & X_2^i[j] \\ \text{else} & X_0[j] \\ \text{end if} & \end{array}$$

$$\Rightarrow X_4^i = \Phi(X_2^i, @X_2^i, X_0, @X_0)$$

Thus, the recursive  $\Phi$  definition for  $X_4$ ,  $X_4^i := \Phi(X_2^i, @X_2^i, X_4^{i-1}, @X_4^{i-1}, X_0, @X_0)$  is equivalent to the non-recursive  $\Phi$  definition  $X_4^i := \Phi(X_2^i, @X_2^i, X_0, @X_0)$ , assuming that  $X_2$  and  $@X_2$  contain all the values that were written during iterations  $1 \dots i$  of the loop. If we use the non-recursive definitions, we observe that there is no use of  $X_4$  or  $@X_4$  inside this loop and hence the final values,  $X_4^n$  and  $@X_4^n$ , can both be propagated outside the loop to obtain  $X_5 := \Phi(\Phi(X_2, @X_2, X_0, @X_0), X_0, @X_0)$  and  $@X_5 := \max(\max(@X_2, @X_0), @X_0)$ . Because the  $\Phi$  definition for  $X_5$  includes the final values computed by the loop, we also refer to this  $\Phi$  function as a *finalization*  $\Phi$ .

Figure 12 shows the transformed version of the code from Figure 11, after solving recurrences. Note that there are no longer any loop carried dependences on  $X_4$  and  $@X_4$  in Figure 12.

## 4 Parallelization Examples

In this section, we discuss several examples illustrating how Array SSA form can be transformed to enable enhanced parallelization.

### 4.1 Loop-level Parallelization

Consider the loop in figure 1. This loop is not typically parallelized by existing compilers. The default way to ensure that the correct values are visible upon completion, given the combination of the data dependent control flow

1. /\* INITIALIZATION. Allocate array temporaries. Note that array  $X_2$  has been expanded. \*/  
`allocate  $X_2[1 : m, 1 : n]$ ,  $@X_2[1 : m, 1 : n]$ ,  $X_5[1 : m]$`   
 Also initialize  $@X_2[*,*] := 0$
2. /\* EXECUTION. Execute loop in parallel using array temporaries  $X_2$  and  $@X_2$ . At this abstract level, the computation model is one processor per iteration. \*/  
`doall  $i := 1, n$`   
`if ( $C[i]$ ) then`  
`$X_2[f(i), i] := \dots$`   
`$@X_2[f(i), i] := i$`   
`end if`  
`end doall`
3. /\* FINALIZATION. Compute final value in  $X_5$  \*/  
`doall  $j := 1, m$`   
`$temp := \text{MAX}(@X_2[j, 1 : n])$`   
`if ( $temp > 0$ ) then`  
`$X_5[j] := X_2[j, temp]$`   
`else`  
`$X_5[j] := X_0[j]$`   
`end if`  
`end doall`
4. `free  $X_2[1 : m, 1 : n]$ ,  $@X_2[1 : m, 1 : n]$ ,  $X_5[1 : m]$`

**Figure 13. Abstract parallelization of figure 1**

and data dependent overwriting of elements in the array, is simply to execute the loop in order. However, if we start with Array SSA form, and perform copy propagation followed by recurrence solving (as discussed in Section 3), then standard array expansion can be enabled to obtain the abstract parallel code shown in figure 13. This abstract parallel form is then converted to concrete parallelism with serial overwriting within each processor and parallel execution followed by a merge across processors. For details of this transformation, see [6].

### 4.2 Region-level Parallelization

Figure 14 exhibits an output dependence between the two assignments to  $X$ . Both left hand sides have unanalyzable indicies. Nevertheless, the Array SSA renaming capability can enable the two regions corresponding to the two loops to execute in parallel, as shown in Figure 15. The  $\Phi$  operation in this form can handle reordering because it takes the  $@$  variables as explicit arguments to accurately merge the two sets of values. (A similar example can be

```

do i2 = 1, n
  ...
  X[f(i2)] := rhs2(i2)
  ...
enddo
do i4 = 1, n
  ...
  X[g(i4)] := rhs4(i4)
  ...
enddo

```

**Figure 14. Region Example 1 - Source**

```

{X1 in effect here.}
k := ...
do i2 = 1, n
  ...
  X2[i2, k] := rhs2(i2)
  @X2[i2, k] := < i2 >
  ...
enddo
X3 = Φ(X2, @X2, X1, @X1)
do i4 = 1, n
  ...
  ... := X3[j, k + 1]
  ...
enddo

```

**Figure 16. Region Example 2 - Array SSA Form**

```

{X1 in effect here.}
PARALLEL SECTIONS
  BEGIN SECTION
    do i2 = 1, n
      ...
      X2[f(i2)] := rhs2(i2)
      @X2[f(i2)] := < i2 >
      ...
    enddo
  END SECTION
  BEGIN SECTION
    do i4 = 1, n
      ...
      X4[g(i4)] := rhs4(i4)
      @X4[g(i4)] := < i4 >
      ...
    enddo
  END SECTION
END PARALLEL SECTIONS
X5 := Φ(X4, @X4, X2, @X2)

```

**Figure 15. Region Example 1 - Array SSA Form**

```

{X1 in effect here.}
k := ...
do i2 = 1, n
  ...
  X2[i2, k] := rhs2(i2)
  @X2[i2, k] := < i2 >
  ...
enddo
do i4 = 1, n
  ...
  ... := X1[j, k + 1]
  ...
enddo
X3 = Φ(X2, @X2, X1, @X1)

```

**Figure 17. Region Example 2 - Resolved Array SSA Form**

```

do  $i = 1, n$ 
  ...
  R0:
  if (c0[i]) then
     $a := X[i] + X[i - 1]$ 
  endif
  ...
  R1:
  if (c1[i]) then
     $X[i] := rhs_1(a)$ 
  endif
  ...
  R2:
  if (c2[i]) then
     $X[i] := rhs_2(a)$ 
  endif
  ...
  {More definitions}
  Rn:
  if (cn[i]) then
     $X[i] := rhs_n(a)$ 
  endif
enddo

```

**Figure 18. Source Code for pipelining**

constructed in which the definition of  $X$  in the first loop replaced by a use of  $X$ . This new example would exhibit an anti-dependence. Again, the Array SSA renaming capability would enable the two regions corresponding to the two loops to execute in parallel.)

Now, consider the example in Figure 16. It exhibits an apparent true dependence from the definition of  $X_2[i_2, k]$  to the use of  $X_3[j, k + 1]$  since the definition of  $X_3$  takes  $X_2$  as an argument. However, the resolution process (discussed in Section 5) can leverage past work on data dependence analysis [10] to determine that use  $X_3[j, k + 1]$  cannot be resolved to definition  $X_2$  (since the use is in column  $k + 1$  and the definition is to column  $k$ ). Hence use  $X_3[j, k + 1]$  can be resolved to (rewritten as) use  $X_1[j, k + 1]$ , as shown in figure 17.

### 4.3 Pipelined Parallelization

In the previous sections, we showed how Array SSA form can be used to transform serial code into parallel code by eliminating output- and anti-dependences via renaming and by eliminating some true dependences via resolution.

In general, if a loop contains a dependence cycle, some form of synchronization among iterations will be required for correct parallelization *i.e.*, the loop can be executed with pipeline parallelism. Recall that the maximum throughput

```

{X initialized here.}
do  $i := imin, imax$ 
s2: if (  $f(X[g(i)])$  ) then
s3:    $X[h(i)] := rhs(i)$ 
      endif
enddo
... := X[...]

```

**Figure 19. Cycle of Dependences**

1. /\* INITIALIZATION. Allocate array temporaries. Note that array  $X_2$  has been expanded. \*/  
allocate  $X_2[1 : m, 1 : n]$ ,  $@X_2[1 : m]$ ,  $X_5[1 : m]$   
Also initialize  $@X_2[*] := 0$
2. /\* The EXECUTION phase consists of a parallel loop for statement s3 and a sequential loop for statements s1, s2, and s4. \*/

```

doall  $i := imin, imax$ 
s3:  $X_2[h(i), i] := rhs(i)$ 
enddo

```

```

do  $i := imin, imax$ 
s1: /* Set temp :=  $X_1[g(i)]$  */
      $j := g(i)$ 
     if ( $@X_2[j] > 0$ ) then
       temp :=  $X_2[j, @X_2[j]]$ 
     else
       temp :=  $X_0[j]$ 
     end if
s2: if  $f(temp)$  then
s4:    $@X_2[h(i)] := i$ 
     endif
enddo

```

3. /\* FINALIZATION. Compute final value in  $X_5$  \*/

```

doall  $j := 1, m$ 
  if ( $@X_2[j] > 0$ ) then
     $X_5[j] := X_2[j, @X_2[j]]$ 
  else
     $X_5[j] := X_0[j]$ 
  end if
end doall

```

4. free  $X_2[1 : m, 1 : n]$ ,  $@X_2[1 : m, 1 : n]$ ,  $X_5[1 : m]$

**Figure 20. Abstract parallelization for figure 19**

for a pipelined loop is limited by the longest cycle in the dependence graph for the loop. Array SSA form can be used to improve the efficiency of pipeline parallelism by either breaking dependence cycles or by reducing the length of the longest cycle.

Consider the example loop in Figure 18. This code has output-dependences among all  $n$  definitions of  $X$  and anti-dependences between the definitions at the bottom of the iteration and the use of  $X[I]$  at the top of the iteration. It also contains a true dependence from all the definitions on the previous iteration to the use of  $X[I - 1]$  in the current iteration. Finally, there are true dependences from the assignment to scalar  $a$  to the uses of  $a$  in  $rhs_1, rhs_2$ , etc. This loop has a dependence cycle of length  $n + 1$  which results in a software pipeline that has a maximum throughput of performing one iteration every  $n + 1$  time steps, just like the original loop (assuming that a “time step” is the time required to execute a single if-block). However, the renaming, merging and reassociativity transformations in Array SSA form can reduce the dependence cycle length to three stages: one stage for the  $a := X[i] + X[i - 1]$  computation, one stage in which all  $n$  definitions are performed concurrently, one stage for the  $\Phi$  function that combines the outputs of all definitions, thus leading to a software pipeline with a maximum throughput of performing one iteration every 3 time steps.

Further, if there was no true dependence on scalar variable  $a$  (say) in the example, then the Array SSA form transformations could be used to break the dependence cycle, and perform the computations using synchronization-free parallelization.

#### 4.4 Parallelism via Speculative Execution

Consider the loop in figure 19. Here there is a cycle that includes a control dependence and a true dependence. If the  $rhs$  computation is significant it is more effectively performed in parallel. Using Array SSA we can compute the assignment for all iterations of the loop speculatively and in parallel. Then we can use the power of the  $\Phi$  function to generate the parallel version shown in figure 20. For details of this transformation see [6].

### 5 Resolution: A $\phi$ -Specific Optimization

Resolution is a  $\phi$ -specific optimization. It is a general mechanism for analyzing the propagation of values through array element accesses. Array SSA form identifies the SSA names that can reach any use. At run-time the  $\phi$  computation resolves the set of reaching definitions to determine a single value. The goal of compile-time resolution is to minimize the number of SSA names that, at compile-time, appear to reach any use. This will make compile-

time analysis and reordering more effective. In past work, we introduced a limited form of resolution in a constant propagation algorithm based on Array SSA form [7].

In this section, we discuss general mechanisms for resolution that can be used by a wide range of analyses and optimizations. Since resolution is a static (compile-time) analysis, it can be used for both  $\phi$  functions in partial Array SSA form and  $\Phi$  functions in full Array SSA form. For simplicity, the discussion in this section is confined to partial Array SSA form with  $\phi$  functions (but no @ variables).

There are two forms of uses in array SSA form:

- Source-level uses — a use is a source-level use if it refers to an assignment in the source.
- Intermediate uses — a use is an intermediate use if it refers to the result of a  $\phi$  operation.

Consider the code fragment labeled “Partial Array SSA” in Figure 21. The source-level uses ( $a1_5$ ,  $a3_5$  and  $a5_{10}$ ) are shown in bold. All other uses ( $a0$ ,  $a2$ ,  $a4$  and  $a6$ ) are intermediate uses. In this form the use of  $a6[10]$  is reached by all three source-level definitions. The goal of Array SSA resolution is to minimize the number of source-level definitions that can reach any use.

Under “Resolved Partial Array SSA” in the same figure, the use of  $a6[10]$  is replaced by a use of  $a5_{10}$ . In this case we have resolved the use to a single source-level definition,  $a5_{10}$ . In other cases, resolution may result in additional  $\phi$  definitions. We may replace a use of a  $\phi$  definition,  $n1$ , reached by three source-level definitions with a new one,  $n2$ , reached by only two of those three source-level definitions.  $n1$  may or may not still be needed for other uses. The result may be a more complex structure with more  $\phi$  definitions and more names but the number of source-level definitions reaching uses will be reduced. SSA form itself increases the potential for reordering by eliminating output dependences and reducing anti-dependences. Resolution reduces true dependences and further increases the potential for reordering.

We introduce two types of resolution. *Forward resolution* considers two definitions,  $d1$  and  $d2$ , and may determine that  $d1$  is *occluded* by  $d2$ . Forward resolution does not consider uses, only the definitions. *Backward resolution*, on the other hand applies to a specific use and takes the indices at that use into account. It can therefore be more aggressive but the result applies only to a specific use.

#### 5.1 Basic Idea

We would like to be able to associate any array element use,  $A[k]$ , with a specific static source-level definition. For two reasons we may not have sufficient knowledge at compile-time to do this.

Source:	Partial Array SSA:	Resolved Partial Array SSA:
$a[5] := \dots$	[ $a0$ is in effect here] $\mathbf{a1}_5 := \dots$	[ $a0$ is in effect here] $a1_5 := \dots$
$a[5] := \dots$	$a2 := d\phi(\mathbf{a1}_5, a0)$ $\mathbf{a3}_5 := \dots$	$a2 := d\phi(a1_5, a0)$ $a3_5 := \dots$
$a[10] := \dots$	$a4 := d\phi(\mathbf{a3}_5, a2)$ $\mathbf{a5}_{10} := \dots$	$a4 := d\phi(a3_5, \mathbf{a0})$ $a5_{10} := \dots$
$\dots := a[10]$	$a6 := d\phi(\mathbf{a5}_{10}, a4)$ $\dots := a6[10]$	$a6 := d\phi(a5_{10}, a4)$ $\dots := \mathbf{a5}_{10}$
$\dots := a[5]$	$\dots := a6[5]$ [ $a6$ is in effect here]	$\dots := \mathbf{a3}_5$ [ $a6$ is in effect here]

**Figure 21. Straight line Code - Constant Indices**

Source:	Partial Array SSA:	Resolved Partial Array SSA:
$a[j] := \dots$	[ $a0$ is in effect here] $a1_j := \dots$	[ $a0$ is in effect here] $a1_j := \dots$
$a[j] := \dots$	$a2 := d\phi(a1_j, a0)$ $a3_j := \dots$	$a2 := d\phi(a1_j, a0)$ $a3_j := \dots$
$a[k] := \dots$	$a4 := d\phi(a3_j, a2)$ $a5_k := \dots$	$a4 := d\phi(a3_j, \mathbf{a0})$ $a5_k := \dots$
$\dots := a[k]$	$a6 := d\phi(a5_k, a4)$ $\dots := a6[k]$	$a6 := d\phi(a5_k, a4)$ $\dots := \mathbf{a5}_k$
$\dots := a[j]$	$\dots := a6[j]$ [ $a6$ is in effect here]	$\dots := a6[j]$ [ $a6$ is in effect here]

**Figure 22. Straight line Code - Symbolic Indices**

- A def  $A[j]$  may or may not definitely define same location as referenced by the use  $A[k]$ .
  - data:  $j$  and  $k$  may or may not be the same.
  - control: The definition of  $A[j]$  may or may not occur on the path to the use  $A[k]$ .
- An intervening definition  $A[i]$  (between the def of  $A[j]$  and the use of  $A[k]$ ) may or may not interfere.
  - data:  $i$  may or may not be the same as  $j$  and  $k$ .
  - control: The intervening definition of  $A[i]$  may or may not occur on the path from the def of  $A[j]$  to the use  $A[k]$ .

The control issues outlined above exist in classic SSA form for scalars as well. However the data issues are unique

to Array SSA. If the program is straight line code and all indices are constants then we can resolve each naive intermediate use to a single source-level definition.

Initially we will ignore the inaccuracies indicated above by restricting our attention to straight line code (control flow is known) and to constant indices (data flow is known). Then in section 5.2 we will relax the constant index requirement. In section 5.3 we will relax the straight line code requirement.

When comparing two constants, call them  $c1$  and  $c2$ , either they are definitely different (written as the boolean relation  $DD(\downarrow\infty, \downarrow\epsilon)$ ) or they are definitely the same (written as the boolean relation  $DS(\downarrow\infty, \downarrow\epsilon)$ ). For variables it is possible that  $DD(\downarrow\infty, \downarrow\epsilon) = DS(\downarrow\infty, \downarrow\epsilon) = \{\neg\uparrow, f\}$ . But for constants  $DS(\downarrow\infty, \downarrow\epsilon) \neq DD(\downarrow\infty, \downarrow\epsilon)$  and it is clear by inspection which is true and which is false. This simplifies

Source:	Partial Array SSA:	Resolved Partial Array SSA:
	<code>[ a0 is in effect here]</code>	<code>[ a0 is in effect here]</code>
<code>a[5] := ...</code>	<code>a1<sub>5</sub> := ...</code>	<code>a1<sub>5</sub> := ...</code>
<code>if ( ... ) then</code>	<code>a2 := dφ(a1<sub>5</sub>, a0)</code>	<code>a2 := dφ(a1<sub>5</sub>, a0)</code>
<code>a[5] := ...</code>	<code>if ( ... ) then</code>	<code>if ( ... ) then</code>
	<code>a3<sub>5</sub> := ...</code>	<code>a3<sub>5</sub> := ...</code>
	<code>a4 := dφ(a3<sub>5</sub>, a2)</code>	<code>a4 := dφ(a3<sub>5</sub>, a0)</code>
<code>else</code>	<code>else</code>	<code>else</code>
<code>a[5] := ...</code>	<code>a5<sub>5</sub> := ...</code>	<code>a5<sub>10</sub> := ...</code>
	<code>a6 := dφ(a5<sub>5</sub>, a2)</code>	<code>a6 := dφ(a5<sub>10</sub>, a0)</code>
<code>endif</code>	<code>endif</code>	<code>endif</code>
<code>... := a[10]</code>	<code>a7 := φ(a4, a6)</code>	<code>a7 := φ(a4, a6)</code>
<code>... := a[5]</code>	<code>... := a7[10]</code>	<code>... := a0[10]</code>
	<code>... := a7[5]</code>	<code>... := φ(a3<sub>5</sub>, a5<sub>5</sub>)</code>
	<code>[ a7 is in effect here]</code>	<code>[ a7 is in effect here]</code>

**Figure 23. Control Flow - Constant Indices**

several aspects of the processing.

Consider figure 21. Upon the second definition of  $a[5]$  we know we are writing over the first definition of  $a[5]$  because  $DS(5, 5)$ . This enables the elimination of  $a1_5$  from the  $\phi$  structure rooted at  $a4$ . (It does not remove  $a1_5$  from the  $\phi$  structure rooted at  $a2$ .) In this case we replace  $a2$  as an argument in the definition of  $a4$  by  $a0$ . When we reach the definition of  $a[10]$  we know this definition does not interact with the value at index 5 because  $DD(5, 10)$ . Therefore, the uses to both  $a6[5]$  and  $a6[10]$  can be resolved to  $a3_5$  and  $a5_{10}$  respectively.

## 5.2 Straight line Code - Symbolic Indices

Now, consider the code in figure 22. It is identical to the code in figure 21 except that the constants 5 and 10 are replaced by the symbols  $j$  and  $k$ .

If we don't know anything about the relationship between  $k$  and  $j$ ,  $DS(k, j) = DD(k, j) = \text{false}$ . The situation is slightly more complicated. The definition  $a1_j$  is still occluded by the definition of  $a3_j$  since we know that  $j$  is the same in both cases even though it is symbolic. It is the impact of the definition of  $a5_k$  that is more complicated. Since we can not guarantee that  $k$  is definitely different from  $j$ , i.e.  $DD(k, j)$  is not true,  $a[j]$  does not necessarily refer to  $a3_j$ .  $a[j]$  might refer to either  $a3_j$  or to  $a5_k$  depending on whether  $k = j$  or not. In this example,  $a6[k]$  must refer to the value defined by  $a5_k$ .

This example illustrates the two types of resolution:

1. The replacement of  $a4 := d\phi(a3_5, a2)$  by  $a4 := d\phi(a3_5,$

$a0)$  in figure 21 and the replacement of  $a4 := d\phi(a3_j, a2)$  by  $a4 := d\phi(a3_j, a0)$  in figure 22 are examples of forward resolution.

2. The replacement of  $a6[10]$  by  $a5_{10}$  in figure 21 and the replacement of  $a6[k]$  by  $a5_k$  in figure 22 are examples of backward resolution.

A RHS intermediate use, such as  $a6[j]$  in the last line of the example in figure 22, indicates that the value may come from any of the leaves of the  $\phi$  structure corresponding to  $a6$ . Such a use is an *unresolved* reference. A source-level use such as the use of  $a5_k$  in the resolved Array SSA form is a *fully resolved* use. A use that is optimized from the naive use but is still an intermediate use is *partially resolved*. A fully resolved use is typically the result of backward resolution. The use of  $a6[j]$  is partially resolved since it no longer is reached from  $a1_j$ . A partially resolved use may be a result of either forward or backward resolution.

If for all relevant pairs,  $i$  and  $j$ , either  $DS(i, j)$  or  $DD(i, j)$  then for straight line code we can fully resolve all uses. Otherwise some uses may remain partially resolved or unresolved.

## 5.3 Control Flow

Consider the code in figure 23 which uses only constant indices but the control flow is unknown. To determine that a definition,  $d$ , can not reach a use,  $r$ ,  $d$  must not reach  $r$  via any path.

Forward resolution determines that, in the definition of  $a_4$ ,  $a_2$  can be replaced by  $a_0$ . To perform this resolution we must consider control flow. The occluding definition,  $a_{3_5}$ , is executed on all paths from the definition of  $a_{1_5}$  to the definition of  $a_4$ . (Similar arguments apply to  $a_6$ .) Notice that  $a_{1_5}$  is occluded on both paths and so it is still occluded at the control  $\phi$  merging both branches of the `if`. If it were only occluded on one branch then the  $\phi$  at the control merge would make it visible at the control merge.

Backward resolution applies to the use of  $a_{7_5}$  and removes all assignments to the element 10 regardless of the control flow. This results in partial resolution of this use. The same argument applied to the use of  $a_{7_{10}}$  results in total resolution.

## 6 Related Work

The two primary approaches to program analysis for enhanced parallelization in past work are data dependence analysis and array data flow analysis. They attack the problem in different ways and have distinct strengths and weaknesses. Briefly, dependence analysis has historically been very useful for arrays because it performs sophisticated index analysis, but it does not capture control flow or perform any renaming. Array data flow analysis captures both control flow and index analysis but does not include any renaming other than array privatization. The array SSA form presented in this paper incorporates control flow analysis, index analysis and array renaming more generally than in past approaches.

Data dependence analysis [10] has historically been the analysis of choice in the parallel compilation community. It performs detailed analysis of subscripts to determine if two references to the same array within common loops can ever touch the same element. However, as has been observed in the past, dependence analysis is location based and is thus insufficient for array data flow analysis.

Array data-flow analysis has received an increasing amount of attention recently (*e.g.*, see [8, 3, 2]). Of the approaches suggested in past work, the last write tree (LWT) in [8] is the most closely related to Array SSA form. The LWT identifies the instance of the last write operation that provides the array element value for a given instance of a read operation, where instances of read/write operations are defined with respect to common surrounding loops. Several restrictions are placed on a program region to enable construction of LWT's. It is assumed that the only control flow in the region consists of structured counted loops (*i.e.*, like Fortran DO loops). It is also assumed that all array subscripts contain affine functions of the index variables of surrounding loops. Array SSA form is far more general in scope than LWT's. As discussed earlier in the paper, Array SSA form supports general reducible control flow

and places no restrictions whatsoever on array subscript expressions. The network of  $\phi$  functions connecting a use to its defs in Array SSA form can be viewed as a generalization of the LWT. The combining rules for the  $\phi$  functions identify which def to follow in each case. Even for the special cases in which LWT's can be created, the  $\phi$  functions represent the same information more efficiently because a separate LWT structure need not be created for each read operation.

In summary, Array SSA form is more precise than dependence analysis because it takes control flow into account. It is more general in scope than the last write tree representation because it allows arbitrary control flow and arbitrary index expressions and provides inherent support for array renaming.

There has also been some past experience with runtime structures similar to @ arrays. A notable example is the use of the inspector/executor model to enable runtime parallelization of loops containing unanalyzable array references [9]. The focus of that work is on runtime scheduling and communication optimization on codes that operate on irregular grids. In contrast, our focus is on exposing @ arrays and  $\phi$  functions in Array SSA form so as to increase the scope of compiler analysis and transformation.

## 7 Conclusions and Future Work

In this paper, we showed how Array SSA form can be used for enhanced parallelization. The transformations that can be enabled by Array SSA form include loop parallelization, region/task parallelism, pipeline parallelism, speculation, and general forms of code reordering)

This paper extends our previous work in two ways. First, it shows how existing optimizations can be applied to  $\Phi$  functions to set the stage for enhanced parallelization. Second, it introduces a new analysis, Array SSA resolution, which improves the precision of use-def associations in Array SSA form by reducing the number of definitions that reach a use.

## References

- [1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] S. P. Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Computer Systems Laboratory, Stanford University, January 1997.
- [3] R. Bodik and R. Gupta. Array Data-Flow Analysis for Load-Store Optimizations in Superscalar Architectures. *Lecture Notes in Computer Science*, (1033):1–15, 1995. Proceedings of Eighth Annual Workshop on Languages and Compilers for Parallel Computing, Columbus, Ohio, August 1995.
- [4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. 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.

- [5] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc., 1977.
- [6] K. Knobe and V. Sarkar. Array SSA form and its use in Parallelization. In *25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, Jan. 1998.
- [7] K. Knobe and V. Sarkar. Conditional constant propagation of scalar and array references using array SSA form. In G. Levi, editor, *Lecture Notes in Computer Science, 1503*, pages 33–56. Springer-Verlag, 1998. Proceedings from the *5th International Static Analysis Symposium*.
- [8] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array Data-Flow Analysis and its Use in Array Privatization. *Conf. Rec. Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.
- [9] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4), April 1990.
- [10] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.