Array SSA form and its use in Parallelization

Kathleen Knobe
Digital Cambridge Research Laboratory
One Kendall Square, Building 700
Cambridge, MA 02139, U.S.A.

Email: knobe@crl.dec.com

Vivek Sarkar
MIT Laboratory for Computer Science
545 Technology Square, NE43-206
Cambridge, MA 02139, U.S.A.

Email: vivek@lcs.mit.edu

Abstract

Static single assignment (SSA) form for scalars has been a significant advance. It has simplified the way we think about scalar variables. It has simplified the design of some optimizations and has made other optimizations more effective. Unfortunately none of this can be said for SSA form for arrays. The current SSA processing of arrays views an array as a single object. But the kinds of analyses that sophisticated compilers need to perform on arrays, for example those that drive loop parallelization, are at the element level. Current (scalar) SSA form does not provide the element-level data flow information required for such analyses.

In this paper, we introduce 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 ϕ 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 code optimization. The paper also discusses the application of Array SSA form to automatic parallelization. The parallelization techniques based on Array SSA form exploit the possibility of computing ϕ functions at runtime in addition to analyzing array indices at compile time, and can thus uncover more general forms of loop-level and region-level parallelism than past techniques.

1 Introduction

Static single assignment (SSA) form for scalar variables has been a significant advance. It has simplified the design of some optimizations and has made other optimizations more effective. Some of the earliest applications of SSA form were in the design of new algorithms for global constant propagation [17] and global value numbering [2, 12]. The popularity of SSA form surged after an efficient algorithm for computing SSA form was made available [5]. SSA form is now a standard representation used in modern optimizing compilers in both industry and academia.

To appear in Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98), January, 1998, San Diego, California. **(NOTE: this preprint contains typographical corrections that are not in the conference version. See appendix for the list of changes.)**

The salient properties of SSA form are as follows:

1. Each definition is assigned a unique name.
2. At certain points in the program, new names are generated which combine the results from several definitions.
3. Each use refers to exactly one name generated from either of the two rules above.

In the scalar case, combining (rule 2) is via a ϕ function which determines which of several values to use based on the flow path traversed. For example, SSA form converts the code in Figure 1 to that in Figure 2. The combining function, ϕ depends on the path taken through the `if` statement. Notice that this path is unknown until runtime and may vary with each dynamic execution of this code.

Given the code in Figure 3, we might consider dealing with arrays in a similar way by recognizing this special case in which all elements of X are written in either the then branch or the else branch. We could use a combining ϕ function that chooses the first or the second definition of the whole array depending on the path taken at runtime. However, the situation becomes complicated if the nesting of the condition and the loop are reversed as in Figure 4. The combining ϕ function for this example must merge the two definitions on an element-by-element basis based on the value of the conditional which is now an array of booleans. This complication arises for arrays because an assignment to an array element does not necessarily kill a previous assignment to the same array variable. (The same problem arises for pointers and aliases as well.)

Thus, the analysis required to generate a useful SSA form for array variables is more complex than for scalar variables. In this paper, we introduce an Array SSA form that captures precise element-level data flow information for array variables in all cases. It is general and simple, and coincides with standard SSA form when applied to scalar variables. It can also be used for structures and other variable types that can be modeled as arrays. The three rules in standard SSA form also apply to Array SSA form. However, combining in Array SSA form is via a more powerful ϕ function that can merge values from distinct definitions on an element by element basis.

To further understand the difference between standard SSA and Array SSA forms, consider loop L in Figure 5 which we will use as a running example in the rest of the paper. The use of an element $X[j]$ after the loop may refer to the initial value of X or to the value defined inside the loop. If

```
if (C) then
    S := ...
else
    S := ...
end if
```

Figure 1: Control Flow with Scalar Definitions

```
if (C) then
    S_1 := ...
else
    S_2 := ...
end if
S_3 := φ(S_1, S_2)
```

Figure 2: Traditional Scalar SSA

```
if (C) then
    do i := ...
        X[i] := ...
    end do
else
    do i := ...
        X[i] := ...
    end do
end if
```

Figure 3: Trivial Array SSA Problem

```
do i := ...
    if (C[i]) then
        X[i] := ...
    else
        X[i] := ...
    end if
end do
```

Figure 4: Not so Trivial Array SSA Problem

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

Figure 5: Loop L with Conditional and Indirection

subscript expression $f(i)$ is not a permutation, the value of a given element of X may be written several times by the same static assignment. The ϕ functions in Array SSA form are capable of handling this complexity effectively.

There are several potential uses for Array SSA form in compiler analysis and optimization for uniprocessor and multiprocessor systems. In this paper, we focus on the application of Array SSA form to the area of automatic parallelization. In general, parallel execution in a program may be inhibited by a variety of control and data dependences. Of these dependences, only *true* data flow dependences cannot be broken. Several transformations have been proposed in past work to break other kinds of data dependences (anti and output) by renaming and privatization and to break control dependences by IF conversion and speculation [18]. We show that the renaming and ϕ functions present in Array SSA form can enhance automatic parallelization by breaking anti/output dependences and control dependences in more general ways than has been done in the past for array variables. Specifically, we first show that Array SSA form can enable parallelization of any loop that is free of loop-carried true data dependences. This includes loops with loop-carried anti and output dependences, unanalyzable subscript expressions, and arbitrary control flow within an iteration (such as loop L in Figure 5). We present performance measurements for the parallelization of a computer graphics rasterization application using this technique. We also show how Array SSA form can be used to enable speculative execution so as to parallelize loops with loop-carried true data dependences that are present in cycles that also contain control dependences. Finally, we show how Array SSA form can be used to break loop-independent output and anti data dependences to enable parallel execution of distinct regions in a program.

The rest of the paper is organized as follows. Section 2 introduces Array SSA form and discusses the placement, semantics, and optimization of ϕ functions in Array SSA form. Section 3 shows several different ways that Array SSA form can be used for parallelization. Section 4 discusses related work, and Section 5 contains our conclusions and indicates possible directions for future work.

2 Array SSA Form

Section 2.1 addresses the issue of ϕ function placement in Array SSA form, and section 2.2 discusses the semantics of ϕ functions and how ϕ functions can be implemented at runtime when so desired. Finally, section 2.3 outlines optimizations that can be performed on @ functions and ϕ functions.

2.1 ϕ Function Placement

This section addresses the question of where to place ϕ functions in Array SSA form. As in scalar SSA, the key requirement on ϕ 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 ϕ placement used by scalar SSA; however, this choice is not fundamental and subsequent optimization of ϕ nodes (addressed in Section 2.3) may result in a different but semantically equivalent placement of ϕ functions.

Initial placement of ϕ functions follows two rules:

1. **define ϕ**

   A ϕ function is inserted immediately after each definition of an array variable that does not completely kill the array value. This *define* ϕ merges the values of the element(s) modified in the definition with the values available immediately prior to the definition. Define ϕ's need not be inserted for definitions of scalar variables because such a definition completely kills the old value of the variable and there is no need to merge modified elements and unmodified elements.

For example, consider the def $X[f(i)] :=$ in figure 5, and assume that the def has been renamed to $X_2[f(i)] :=$. Also, let X_1 be the def of variable X that reaches the point just prior to the def of X_2. Then, the ϕ function, $X_3 := \phi(X_2, X_1)$ is inserted immediately after the def of X_2 to represent an element-by-element 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. **merge** ϕ

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

One way to think of any of these ϕ functions is simply as an identity assignment of the form $X := X$ (called pseudo-assignments in [14]). Such an assignment by itself is semantically void, but during conversion to SSA the LHS becomes a newly generated name. The RHS combines elements from other names generated for the same source array (those listed as arguments to the ϕ). As in scalar SSA, after each ϕ is given a unique name and each source definition is given a unique name, any use in the source will simply point to (be renamed as) one of these newly generated names.

As an example, we can see that placement of these define ϕ and merge ϕ functions would convert the code for loop L in Figure 5 to the Array SSA form shown in figure 6. The algorithm for computing the initial Array SSA form for a program uses the ϕ placement algorithm for scalar SSA form [5] to determine the placement of merge ϕ functions. In addition, define ϕ functions are inserted as described above. Details of the placement algorithm for Array SSA form have been omitted for the sake of brevity.

An important property of the numbers assigned to definitions by Array SSA form is that they are *topologically sorted i.e.,* if there exists a *forward path* (a path that does not include a *back edge* [1]) in the control flow graph from def X_i to def X_j, then we require that $i < j$. This topological numbering simplifies the discussion of ϕ function semantics in the next section.

2.2 ϕ Function Semantics and Implementation

Now that we know where the ϕ functions are placed in Array SSA form, this section addresses their semantics and implementation. For some uses of Array SSA, such as static analysis, the ϕ functions are not manifest at runtime and incur no runtime overhead. In such cases, only the semantics are relevant. The implementation of ϕ functions becomes relevant if Array SSA form is used in a way that requires runtime evaluation of ϕ functions (as in our parallelization examples in the section 3). As we describe the semantics of ϕ functions in Array SSA form, it might appear that we are introducing a lot of runtime overhead. Section 2.3 presents optimizations of the ϕ functions to reduce this overhead. The experimental results presented later in Section 3 also

```
/* X0 is the initializing definition from figure 5 */
X0[...] :=
do i := 1, n
   X1 := φ(X4, X0)
   if (C[i]) then
      X2[f(i)] := ...
      X3 := φ(X2, X1)
   end if
   X4 := φ(X3, X1)
end do
X5 := φ(X4, X0)
... := X5[...]
```

Figure 6: Array SSA form for loop L in figure 5

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

X0[...] :=
@X0[...] := ()
do i := 1, n
   X1 := φ(X4, X0)
   @X1 := max(@X4, @X0)
   if (C[i]) then
      X2[f(i)] := ...
      @X2[f(i)] := (i)
      X3 := φ(X2, X1)
      @X3 := max(@X2, @X1)
   end if
   X4 := φ(X3, X1)
   @X4 := max(@X3, @X1)
end do
X5 := φ(X4, X0)
@X5 := max(@X4, @X0)
... := X5[...]
```

Figure 7: Insertion of @ Computation

$$X_1^i[j] = \begin{array}{lll} \textbf{if} & @X_4^{i-1}[j] \succeq @X_0[j] \quad \textbf{then} & X_4^{i-1}[j] \\ \textbf{else} & X_0[j] & \\ \textbf{end if} & & \end{array}$$

$$X_3^i[j] = \begin{array}{lll} \textbf{if} & @X_2^i[j] \succeq @X_1^i[j] \quad \textbf{then} & X_2^i[j] \\ \textbf{else} & X_1^i[j] & \\ \textbf{end if} & & \end{array}$$

$$X_4^i[j] = \begin{array}{lll} \textbf{if} & @X_3^i[j] \succeq @X_1^i[j] \quad \textbf{then} & X_3^i[j] \\ \textbf{else} & X_1^i[j] & \\ \textbf{end if} & & \end{array}$$

$$X_5[j] = \begin{array}{lll} \textbf{if} & @X_4^n[j] \succeq @X_0[j] \quad \textbf{then} & X_4^n[j] \\ \textbf{else} & X_0[j] & \\ \textbf{end if} & & \end{array}$$

Figure 8: Semantics of the ϕ functions in Loop L

provide evidence that the remaining overhead is not significant for a rasterization application that has a high-level structure similar to loop L.

To define the semantics of ϕ functions, we introduce the concept of an *@ array*. For each static definition X_k the @ array $@X_k[j]$ 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 $\perp$, $@X_k[j] = \perp$, which 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. A single point in the iteration space is specified by the iteration vector $\vec{i} = (i_1, \ldots, i_n)$, which is an n-tuple of iteration numbers one for each enclosing loop. The construction of Array SSA form does not require that the surrounding loops be structured counted loops (*i.e.,* like Fortran DO loops) or that the surrounding loops be tightly nested. Our only assumption is that all loops are single-entry, or equivalently, that the control flow graph is *reducible* [8, 1]. For single-entry loops, we know that each def executes at most once in a given iteration of its surrounding loops. All structured loops (e.g., `do`, `while`, `repeat-until`) are single-entry even when they contain multiple exits; also, most unstructured loops (built out of `goto` statements) found in real programs are single-entry as well. A multiple-entry loop can be transformed into multiple single-entry loops by *node splitting* [8, 1].

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-ϕ) 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 ϕ definition also has an associated @ array. Figure 7 shows the result of inserting @ array computations into figure 6 so as to obtain the complete Array SSA form for loop L.

Consider @ array $@X_2$ for static definition $X_2[f(i)] :=$. Recall that nothing is known about subscript function $f(i)$ — it need not be monotonic, and it need not even be a permutation. Nevertheless, setting $@X_2[j] = (i)$ correctly records the "most recent" iteration in which static definition X_2 assigned a value to element $j = f(i)$. Notice that this computation is accurate even if f is not a permutation since $@X_2$ may be overwritten in subsequent iterations but the final value indicates the iteration number of the last modification. It is also accurate in the presence of the conditional since $@X_2$ is updated exactly when the definition would have occurred.

This leads us to the notion of *dynamic definitions*. An array element may be defined multiple times during the execution of the program. Each of these is called a *dynamic definition* of the element. A dynamic definition of an element is specified by

- a specific static definition and
- a point in the iteration space of that definition.

Consider the dynamic definition for element $X[17]$ in Figure 6. The two non-ϕ static definitions are the initialization outside the loop (X_0) and the one assigning through the indirection vector, f, within the loop (X_2). Maybe X_0 defined element $X[17]$ in the initialization and X_2 defined element $X[17]$ on iterations $i = 10$, $i = 39$ and $i = 82$. These are all dynamic definitions. In this case, $@X_0[17]$ will be set to (), and $@X_2[17]$ will be set to (10), (39), (82) in iterations 10, 39, 82, with a final value of $@X_2[17] = (82)$.

For a given array element and execution point in the program, the last dynamic definition to have executed is *visible*. Dynamic definitions executed earlier are *occluded* by subsequent ones. Each element has at most one visible dynamic definition at any time. The role of the ϕ function is to identify the visible dynamic definition for each individual element of an array array.

Thus, the semantics of ϕ functions is specified most naturally by conditional expressions. For example, the define ϕ function, $X_3 = \phi(X_2, X_1)$ in figure 7 represents an element-by-element merge of array values X_2 and X_1 (recall that this ϕ function has no analog in scalar SSA form). The conditional expression for this ϕ function is defined as follows:

$$X_3^i[j] \quad = \quad \begin{array}{lll} \textbf{if} & @X_2^i[j] \succeq @X_1^i[j] & \textbf{then} \quad X_2^i[j] \\ \textbf{else} & X_1^i[j] & \\ \textbf{end if} & & \end{array}$$

where the notation X_k^i denotes the value of the dynamic definition of definition X_k in iteration i of loop L.

$\succeq$ represents the *lexicographic* $\geq$ relationship on iteration vectors (@ array element values). The conditional expression dictates that the value of array element $X_3^i[j]$ comes from either $X_2^i[j]$ or from $X_1^i[j]$ depending on which element was computed more recently. This selection is controlled by the value of @ array elements $@X_2^i[j]$ and $@X_1^i[j]$: if $@X_2^i[j] \succ @X_1^i[j]$ then the conditional expression sets $X_3^i[j] = X_2^i[j]$, and if $@X_1^i[j] \succ @X_2^i[j]$ the conditional expression sets $X_3^i[j] = X_1^i[j]$. In the case that $@X_2^i[j] = @X_1^i[j]$, the conditional expression sets $X_3^i[j] = X_2^i[j]$; static definition X_2 is selected instead of X_1 because for two definitions with the same iteration vector the topological numbering property of Array SSA form guarantees that the definition with the largest number will be more recent.

Since @ arrays $@X_2$ and $@X_1$ are used in the ϕ function, the definition of ϕ as a pure function actually includes these @ arrays as arguments *e.g.,* $X_3 = \phi(X_2, @X_2, X_1, @X_1)$ However, for conciseness, we often omit writing the @ arrays as explicit arguments and instead just write $X_3 = \phi(X_2, X_1)$. Note that in scalar SSA, a ϕ function such as $X_3 = \phi(X_2, X_1)$ is not a pure function of X_2 and X_1 but also depends on implicit information analogous to the @ variables that identifies the control path taken.

Notice in figure 7 that the @ array for each ϕ def $X_k := \phi(X_{h_1}, X_{h_2})$ is computed as $@X_k = \max(@X_{h_1}, @X_{h_2})$. This is because, for each element j, the ϕ function selects as its result the input $X_h[j]$ value with the largest $@X_h[j]$ value.

Analogous to the define ϕ, the value of the merge ϕ $X_4 := \phi(X_3, X_1)$ in figure 7 can be expressed as a conditional expression for element $X_4^i[j]$ as follows:

$$X_4^i[j] \quad = \quad \begin{array}{lll} \textbf{if} & @X_3^i[j] \succeq @X_1^i[j] & \textbf{then} \quad X_3^i[j] \\ \textbf{else} & X_1^i[j] & \\ \textbf{end if} & & \end{array}$$

In the semantics of ϕ functions in scalar SSA form, a conditional expression is built from branch conditions; here, in Array SSA form, the @ arrays capture the merging of elements due to branch conditions as well as the merging of elements assigned in different iterations. Figure 8 shows the conditional expressions for all the ϕ functions from figure 7.

Consider the computation of the merge ϕ value $X_3[17]$, using the conditional expression from figure 8. If $@X_2[17]$ is not $\perp$, $X_3[17]$ takes its value from $X_2[17]$ rather than from $X_0[17]$ because the two static assignments are in no shared loops and X_2 has a larger topological number than X_0 (because there is a forward path from X_0 to X_2).

The Array SSA form presented thus far is defined for array variables and (as a special case) for scalar variables. However, Array SSA form can be easily constructed for structure variables since a structure can be viewed as a fixed-size array. A read/write operation of a structure element can be viewed as a read/write operation of an array element through a subscript that is a compile-time constant. Operations that read/write multiple elements of a structure (*e.g.,* structure copy) can be modeled by read/write operations on sets of structure elements. The ϕ function in Array SSA form can then merge distinct fields modified in distinct parts of the code exactly as it merges distinct elements for array variables. In fact, Array SSA form for structure variables is a simpler representation than for general arrays, because all element accesses can be resolved at compile-time. This technique directly extends to construction of Array SSA form for nested arrays and structures. For example, an array of rank n of structures can be modeled as an array of rank $n+1$.

In concluding this section, we observe that Array SSA form enables a new optimization, "element-level dead code elimination". Notice that the code in our example may compute many instances of the right hand side of the assignment to X_2 that are not used after the loop. If they are not used within the loop and they have no side-effects, then they need not be computed at all. We simply suppress the assignment $X_2[f(i)]$:= $rhs(i)$ within the loop. The ϕ after the loop tells us exactly which computations are needed. If we determine that element j is the one computed on iteration $i = @X_5[j]$, we execute $X_5[j]$:= $rhs(i)$ in the finalization loop instead of computing $rhs(i)$ for each iteration of the execution loop. Notice that classical dead code elimination typically eliminates static instances of code. Although statically, this code is required, we have eliminated its execution dynamically for many elements.

2.3 Optimization of ϕ Functions and @ Arrays

In this section, we outline how @ arrays and ϕ functions can be optimized to reduce runtime overhead (in cases where these functions are made manifest in the output code).

The first observation is that, in unoptimized Array SSA form, a distinct @ array is created for each real (non-ϕ) definition of an array variable. For many assignments, however, the conditions under which the assignment occurs is completely analyzable at compile time. In such cases neither the storage nor the computation of @ arrays are needed. For example, an assignment to $X[i]$ in a loop from $imin$ to $imax$ needs no @ array. If a ϕ function needs to test for ($@X[j] = \perp$) we can instead substitute the test, ($j < imin$ *or* $j > imax$).

In addition, we will show that @ arrays and ϕ functions can be analyzed and optimized just like other variables and instructions/statements in a compiler's intermediate language. The conditional expression semantics of ϕ functions gives it some very useful properties *e.g.,* we will show that ϕ functions are associative. Therefore, traditional optimizations such as copy propagation, loop-invariant code motion, common subexpression elimination, elimination of partial redundancies, dead store elimination, etc. can all be used to eliminate or reposition computations for @ arrays and ϕ functions. We illustrate this by showing how the code structure from figure 7 with naive placement of @ and ϕ computations can be transformed to obtain the optimized code structure in figure 11.

Consider the Array SSA form with @ arrays shown in figure 7. Our goal is to remove as much computation of @ arrays and ϕ functions as possible. As a first step, we can copy propagate the def of X_1 into the two uses of X_1 and the def of $@X_1$ into the two uses[1] of $@X_1$ to obtain the code structure shown in figure 9. Since max is an associative function, a nested max expression such as $\max(@X_2, \max(@X_4, @X_0))$ in figure 9 can be rewritten more simply as $\max(@X_2, @X_4, @X_0)$. Further, we can prove that *ϕ functions are associative* because composing ϕ functions is equivalent to nesting of conditional expressions. For example, we can show that $\phi(X_2, \phi(X_4, X_0)) = \phi(\phi(X_2, X_4), X_0)$ by noting that $X_3 := \phi(X_2, \phi(X_4, X_0))$ and $X_3 := \phi(\phi(X_2, X_4), X_0)$ would result in the same nested conditional expression for X_3 (examples of conditional expressions for ϕ functions can be seen in figure 8):

$$X_3^i[j] \quad = \quad \begin{array}{lll} \textbf{if} & @X_2^i[j] \succeq @X_1^i[j] & \textbf{then} \quad X_2^i[j] \\ \textbf{else if} & @X_4^{i-1}[j] \succeq @X_0[j] & \textbf{then} \quad X_4^{i-1}[j] \\ \textbf{else} & X_0[j] & \\ \textbf{end if} & & \end{array}$$

Since ϕ is associative, we use $\phi(X_2, X_4, X_0)$ as shorthand for $\phi(X_2, \phi(X_4, X_0))$ in our example.

We now continue with figure 9 and copy propagate the defs of X_3 and $@X_3$ to obtain the code structure shown in figure 10 (which also uses the flattened representations of max functions and ϕ functions). It is easy to see that $\max(@X_2, @X_4, @X_0, @X_4, @X_0)$ in figure 10 can be simplified to $\max(@X_2, @X_4, @X_0)$. Further, because of the $\succeq$ relation used in defining the conditional expressions for ϕ functions, we can also simplify $\phi(X_2, X_4, X_0, X_4, X_0)$ to $\phi(X_2, X_4, X_0)$.

The simplified max computation forms a recurrence as follows, $@X_4^i := \max(@X_2^i, @X_4^{i-1}, @X_0)$. 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 $@X_2[j]$). Therefore the recurrence can be solved to obtain $@X_4^i[j] = \max(@X_2^i[j], @X_0[j])$.

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

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

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

[1] For simplicity, we do not show copy propagation of $@X_1$ into the implicit uses of $@X_1$ in the two ϕ functions that take X_1 as input.

```
X_0[...] :=
@X_0[...] := ()
do i := 1, n
   if (C[i]) then
      X_2[f(i)] := ...
      @X_2[f(i)] := (i)
      X_3 := φ(X_2, φ(X_4, X_0))
      @X_3 := max(@X_2, max(@X_4, @X_0))
   end if
   X_4 := φ(X_3, φ(X_4, X_0))
   @X_4 := max(@X_3, max(@X_4, @X_0))
end do
X_5 := φ(X_4, X_0)
@X_5 := max(@X_4, @X_0)
... := X_5[...]
```

Figure 9: Transformation of figure 7 after copy propagation of X_1 and $@X_1$

```
X_0[...] :=
@X_0[...] := ()
do i := 1, n
   if (C[i]) then
      X_2[f(i)] := ...
      @X_2[f(i)] := (i)
   end if
   X_4 := φ(X_2, X_4, X_0, X_4, X_0)
   @X_4 := max(@X_2, @X_4, @X_0, @X_4, @X_0)
end do
X_5 := φ(X_4, X_0)
@X_5 := max(@X_4, @X_0)
... := X_5[...]
```

Figure 10: Transformation of figure 9 after copy propagation of X_3 and $@X_3$

```
X_0[...] :=
@X_0[...] := ()
do i := 1, n
   if (C[i]) then
      X_2[f(i)] := ...
      @X_2[f(i)] := (i)
   end if
end do
X_5 := φ(X_2, X_0)
@X_5 := max(@X_2, @X_0)
... := X_5[...]
```

Figure 11: Transformation of figure 10 after solving recurrences for X_4 and $@X_4$ and copy propagation out of the loop

recurrence:

$$X_4^i[j] \;=\; \begin{array}{lll} \textbf{if} & @X_2^i[j] \succeq @X_0[j] & \textbf{then} \quad X_2^i[j] \\ \textbf{else} & X_0[j] & \\ \textbf{end if} & & \end{array}$$

$$\Rightarrow X_4^i \;=\; \phi(X_2^i, X_0)$$

Thus, the recursive ϕ definition for X_4, $X_4^i := \phi(X_2^i, X_4^{i-1}, X_0)$, is equivalent to the non-recursive ϕ definition $X_4^i := \phi(X_2^i, X_0)$, assuming that X_2 and $@X_2$ contain all the values that were written during iterations $1 \ldots i$ 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 $X_5 := \phi(\phi(X_2, X_0), X_0)$ and $@X_5 := \max(\max(@X_2, @X_0), @X_0)$, which can be further simplified to $X_5 := \phi(X_2, X_0)$ and $@X_5 := \max(@X_2, @X_5)$ resulting in the optimized code structure shown in figure 11. Because the ϕ definition for X_5 includes the final values computed by the loop, we also refer to this ϕ function as a *finalization* ϕ. Note that the code structure in figure 11 is very efficient because it does not have any ϕ function computations inside the loop.

Note that the loop in figure 11 still contains an @ array computation, which is unavoidable for this example because there is only a single real def to the array variable and it contains an unanalyzable subscript expression. As we will see in section 3, the overhead of the @ computation in this example is small enough to not be a significant performance impediment. However, the overhead of @ array computations could become significant for a loop that contains multiple defs if each real def is accompanied by an @ array computation. So we conclude this section by briefly mentioning some optimizations that can be used to reduce the computation and storage overhead of maintaining @ arrays.

Consider the following loop after insertion of @ array computations. Assume there is no control flow within the loop.

```
do i = 1, n
   X_1[f(i)] := ...
   @X_1[f(i)] := (i)
   ...
   Y_1[g(f(i))] := ...
   @Y_1[g(f(i))] := (i)
   ...
   Z_1[f(g(i))] := ...
   @Z_1[f(g(i))] := (i)
   ...
enddo
```

(Since this discussion is focused on @ array computations, we do not show the ϕ function details in the above loop.)

As before, we allow f to be an arbitrary function. Let us consider how we might derive $@Y_1$ array values from $@X_1$ array values. We assume that function g is known to have a unique inverse *i.e.,* $x1 \neq x2 \Leftrightarrow g(x1) \neq g(x2)$. Then

$$\begin{aligned} @Y_1[j] &= \max\left(\{i | 1 \leq i \leq n \text{ and } g(f(i)) = j\}\right) \\ &= \max\left(\{i | 1 \leq i \leq n \text{ and } f(i) = g^{-1}(j)\}\right) \\ &= @X_1[g^{-1}(j)] \end{aligned}$$

Therefore, we can avoid computing array $@Y_1$ by using the identity, $@Y_1[j] = @X_1[g^{-1}(j)]$, to look up its value from $@X_1$. If $g^{-1}(j)$ falls outside the range of array $@X_1$ then we just set $@Y_1[j] = \perp$ (which includes the case when $g^{-1}(j)$

is not an integer). For example, if $g(f(i)) = 2f(i)$, we have $@Y_1[j] = @X_1[j/2]$. This means that $@Y_1[j] = \perp$ for all odd values of j, which reflects the fact that an odd-numbered element of array Y_1 could not have been set by the assignment statement $Y_1[2f(i)] = \ldots$. The fact that iteration i modifies element $Y_1[2f(i)]$ is captured by examining $@X_1[(2f(i))/2] = @X_1[f(i)]$, which would have been set to $@X_1[f(i)] = (i)$. The extra overhead in looking up $@X_1$ lies in computing $g^{-1}(j)$ so this optimization is best suited to cases in which g^{-1} is a simple function *e.g.,* when $g(f(i)) = constant_1 \times f(i) + constant_2$ (note that making g^{-1} a simple function does not place any constraint on function $f(i)$). This reuse of the @ array is valid if both assignments are identically control dependent.

Now consider how we might derive $@Z_1$ array values from $@X_1$ array values. We make the additional assumption that g is a monotonically increasing function *i.e.,* $x1 < x2 \Leftrightarrow g(x1) < g(x2)$. Then

$$
\begin{aligned}
@Z_1[j] &= \max\left(\{i | 1 \le i \le n \;\text{ and }\; f(g(i)) = j\}\right) \\
&= \max\left(\{g^{-1}(i') | 1 \le g^{-1}(i') \le n \;\text{ and }\; f(i') = j\}\right) \\
&= g^{-1}(@X_1[j])
\end{aligned}
$$

Therefore, we can also avoid computing array $@Z_1$ by using the identity, $@Z_1[j] = g^{-1}(@X_1[j])$, to look up its value from $@X_1$.

These optimizations for eliminating @ array computation are quite general in nature, and can easily catch common cases such as $g(f(i)) = f(i) \pm constant$. For example, we can just use one @ array for the four defs $X_1[f(i)]$, $Y_1[f(i)]$, $X_2[f(i) + 1]$ and $X_3[f(i + 1)]$.

We have shown some of the optimization techniques used to reduce the potential overhead of the ϕ and @ computations. These include compile time analysis of @ arrays, how classical optimizations such as reassociation and copy propagation can be applied and how a single @ function can be used for distinct assignments.

3 Using Array SSA form for Automatic Parallelization

As mentioned earlier, there are several potential uses for Array SSA form in compiler analysis and optimization. In this section, we focus on the application of Array SSA form to the area of automatic parallelization. Other possible uses of Array SSA form are mentioned briefly in section 5 as subjects for future work.

Due its renaming of array variables and the ability to compute @ arrays and ϕ functions at runtime, Array SSA form enables parallelization of a wider range of loops than past techniques. In the following discussion, we assume that parallelization-enhancing transformations such as loop distribution [18] have been performed as a pre-pass, so that a loop-carried true data dependence only inhibits parallelization if it belongs to a cycle of data dependences. Serial execution is fundamentally necessary for a cycle of true data dependences, in which a value in one iteration is computed using a value from the previous iteration (and the two statements cannot be distributed into separate loops). However, programs often rely on serial execution for other reasons. If an array element may be modified in multiple iterations (*i.e.,* there is a dependence cycle containing an output dependence), serial execution ensures that the correct final values are visible when the loop terminates. If an array element computed in one iteration is used to determine the direction of a conditional branch in a subsequent iteration that may modify an element of the same array (*i.e.,* there is a dependence cycle containing a loop-carried true data dependence and a loop-independent control dependence), again serial execution ensures the correct final values are visible when the loop terminates. However, the ϕ function in Array SSA form is an alternative way of ensuring that the correct values are visible of ensuring that the correct values are visible on loop termination without requiring serial execution.

The rest of this section is organized as follows. Section 3.1 describes a new loop parallelization technique based on Array SSA form that can enable parallelization of any loop that is free of loop-carried true data dependences. This includes loops with loop-carried anti and output dependences, unanalyzable subscript expressions, and arbitrary control flow within an iteration. Section 3.2 shows Array SSA form can be used to enable speculative execution so as to parallelize loops with loop-carried true data dependences that are present in cycles that also contain control dependences. Section 3.3 shows how Array SSA form can be used to break loop-independent output and anti data dependences to enable parallel execution of distinct regions in a program.

3.1 Parallelization Across Loop Iterations

In this section, we show how Array SSA form can be used to parallelize loops containing loop-carried anti and output data dependences but no loop-carried true data dependences. This parallelization technique works for loops containing arbitrary array subscript expressions and is thus much more general in scope than loop parallelization techniques proposed in the past that are based on array data flow analysis and array privatization for loops containing affine array subscript expressions.

We illustrate loop parallelization via Array SSA form with our running example from the previous section, loop L in figure 5. This loop contains a conditional write to array element $X[f(i)]$ where $f(i)$ is an arbitrary subscript expression that may depend on program input. The subscript expression may also be many-to-one *i.e.,* $X[f(i_1)]$ and $X[f(i_2)]$ may map to the same element for two distinct iterations $i_1 \neq i_2$. To the best of our knowledge, loop L cannot be parallelized by any compiler today, though it could be amenable to *speculative* parallel execution combined with renaming using the software approach described in [11] or building on the hardware approaches described in [15, 16, 10]. In contrast, the loop parallelization transformation described in this section is not speculative; after renaming arrays and inserting ϕ function computations, the compiler knows at compile-time that the loop can be safely executed in parallel at run-time.

The rest of this section is organized as follows. Section 3.1.1 describes an abstract parallelization of loop L that follows directly from the optimized Array SSA form. Abstract parallelization reveals the potential parallelism in the loop without paying attention to overhead issues on real machines. Section 3.1.2 then describes a concrete parallelization of loop L for a small-scale multiprocessor. Concrete parallelization directs the parallelism so as to obtain efficient code for a given machine.

3.1.1 Abstract Parallelization

We first present an abstract parallelization that reveals the potential parallelism in the program without commiting to any specific computation mapping or data distribution. Our goal is to transform the stylized serial code produced by Array SSA analysis to parallel form. For the code in figure 11, the only dependences that prevent parallel execution are the output dependence between distinct assignments to X_2 and the output dependence between distinct assignments to $@X_2$. Both are due to the fact that f may result in multiple modifications to the same location.

To enable these assignments to execute in parallel, we *expand* [18] both X_2 and $@X_2$ along the iteration axis (adding a dimension of the same extent as the range of i). This expansion permits distinct iterations $i_1 \neq i_2$ of loop L to concurrently write into $X_2[f(i_1), i_1]$ and into $X_2[f(i_2), i_2]$ even when $f(i_1) = f(i_2)$. The loop in figure 11 can now be executed in parallel. However, the computation of the ϕ function, $X_5 := \phi(X_2, X_0)$, outside the loop in figure 11 now needs to be transformed so that it can work with the expanded arrays.

Using the semantics of ϕ functions introduced in section 2, the conditional expression for the *finalization* ϕ function, $X_5 := \phi(X_2, X_0)$, before array expansion of X_2 and $@X_2$ can be expressed as:

$$X_5[j] \quad = \quad \begin{array}{ll} \textbf{if} & @X_2^n[j] \succeq @X_0[j] \\ \textbf{then} & X_2^n[j] \\ \textbf{else} & X_0[j] \\ \textbf{end if} & \end{array}$$

In general, the computation of the finalization ϕ function after array expansion of its inputs needs to be performed in two steps. First, a *single-assignment reduction* is performed for each static assignment. Then a *multi-assignment reduction* is performed to combine the results of the single-assignment reductions on the same array variable.

First consider the single-assignment reduction. The serial version reduced the (potentially multiple) iterations on which $X_2[k]$ might have been modified to a single last iteration simply by overwriting them into a single location $@X_2[k]$ in order. This serial in-order execution ensured that the largest iteration was assigned last. The parallel expanded version instead performs this computation as a reduction operation that locates the largest value in $@X_2[k, 1 : n]$. The multi-assignment reduction combines the results of distinct assignments. We simply modify the conditional expression for the ϕ function, $X_5 := \phi(X_2, X_0)$, in teh above conditional expression to instead retrieve its value from the correct location in the expanded X_2 or from X_0.

The resulting abstract parallel version is found in figure 12. Step 1 in figure 12 performs the *initialization* for the abstract parallelization. It allocates array temporaries X_2, $@X_2$, and X_5, as dictated by the optimized Array SSA form. Step 2 in figure 12 performs the *execution* of the modified original loop. Step 3 in figure 12 performs the *finalization* recreating the view of the arrays as in the source for use by the remainder of the program. The single-assignment reduction for X_2 is performed by the MAX function. The multi-assignment reduction for X (i.e., combining X_0 and X_2) is performed by the `if` construct.

The total $O(m \times n)$ time spent in the MAX computation is the largest amount of extra work introduced by abstract parallelization, where m is the size of the original array X and n is the number of iterations in the loop. This can be a significant source of overhead even though this work can be done in $O(log\ n)$ parallel time. As we will see, the total amount of work required for the MAX operation gets reduced to $O(m \times P)$ in the concrete parallelization, where P is the number of processors in the target machine.

3.1.2 Concrete Parallelization

Figure 13 shows a concrete parallelization of the abstract parallelism in figure 12. Concrete parallelization requires that the data and computation be mapped to a limited number of processors in the target machine. We assume that each processor executes its iterations in the same relative order as the original loop. The concrete target we assume in this discussion is a small-scale multiprocessor with physically distributed memories and hardware support for a shared address space.

Step 1 in figure 13 performs the *initialization* for concrete parallelization. It only allocates array temporaries X_2 and $@X_2$ but not X_5. More importantly, the size of the array temporaries is $m \times P$ in the concrete parallelization compared to $m \times n$ in the abstract parallelization, which is a significant reduction in the amount of temporary storage required. The data distributions in step 1 are provided as an optimization; the correctness of our parallelization transformation does not depend on the data distributions.

Step 2 in figure 13 performs the *execution* of the concrete parallelization. It is legal to execute this transformed loop in parallel because the expansion of arrays X_2 and $@X_2$ ensure that no data races occur (the assumption that each processor executes its iterations in the same relative order as the original loop is also necessary for ensuring correctness). Note that the data distributions ensure that all writes to arrays X_2 and $@X_2$ are local in this concrete parallelization. However, due to the unpredictable nature of the $f(i)$ array subscript expression, the uniprocessor spatial locality of the writes performed by the original loop and by the parallelized loop may be poor.

Step 3 in figure 13 performs the *finalization* for the concrete parallelization. As in the abstract parallelization, the finalization loop in step 3 is a parallel loop, but its extent equals m, the number of elements in array X rather than n, the number of iterations in loop L. It involves the same two step process described earlier; single-assignment reduction followed by multi-assignment reduction. Recall that the single-assignment reduction was achieved via overwriting in the serial case and via an actual reduction across an expanded dimension in the abstract case. The concrete case accomplishes this reduction via overwriting within each processor and a reduction across processors.

The single-assignment reduction for X_2 computes $temp :=$ MAXLOC($@X_2[j, P]$), the index number of the processor that assigned to $X[j]$ with the largest iteration number. The total amount of time spent in the MAXLOC computation is $O(m \times P)$, which can be done in $O(m)$ parallel time on P processors. The finalization loop makes a significant number of remote memory references, but these references have good spatial locality (unlike the memory references in the execution loop) and hence should be amenable to latency hiding techniques such as prefetching. Though we show the MAXLOC computation as accessing shared non-local data, it is also well understood how to perform a MAXLOC reduction in a distributed-memory execution model with no hardware support for a shared address space.

The multi-assignment reduction determines the final value of $X[j]$ by adapting the conditional expression for X_5's ϕ function derived in section 2. If $@X_2[j, temp] > 0$, we set $X[j] := X_2[j, temp]$, the last value of element j computed on processor # $temp$. Otherwise (when $temp = 0$), element j was not written in loop L and we leave $X[j]$ unchanged.

We can expect this concrete parallelization to scale linearly so long as the total work in loop L is at least $O(m \times P)$. This will be true if $n \geq m \times P$ or if $n = m$ and

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
        X2[f(i), i] := ...
        @X2[f(i), i] := i
    end if
end doall
```

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

```
doall j := 1, m
    temp := MAX(@X2[j, 1 : n])
    if (temp > 0) then
        X5[j] := X2[j, temp]
    else
        X5[j] := X0[j]
    end if
end doall
```

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

Figure 12: Abstract parallelization of loop L

each iteration of loop L does at least $O(P)$ work. Note that the distribution of the $f(i)$ subscript function values does not affect the scalability of concrete parallelization. The concrete parallelization will scale linearly even if $f(i)$ happens to map to the same element in all iterations (assuming that the total work in loop L is at least $O(m \times P)$). This is in contrast to schemes proposed in hardware for dynamic address resolution (*e.g.,* [15]) in which a single location can become a performance bottleneck if it receives a disproportionately large number of memory operations.

3.1.3 Experimental Results

In this section, we present some performance results for a rasterization [6] example from computer graphics using the concrete parallelization technique from the previous section. Rasterization (also known as "scan conversion") is a simple example of a real application that includes a loop containing loop-carried output data dependences but no loop-carried true data dependences. Polygons are "painted" onto a display buffer from back to front *i.e.,* in decreasing order of their z values.

For the serial program in our experimental results, we used the code for generic convex polygon scan conversion written by Paul Heckbert that is distributed with the "Graphics Gems" book [7]. This scan conversion code was called on n randomly generated triangles, for a target display buffer that contains $m = 1024 \times 1024 \approx 10^6$ pixels. Each triangle was generated as follows. First, the same (randomly generated) z value was assigned to all three of its vertices. Next, a randomly generated (x, y) pair was selected as the first vertex. Finally, the x and y offsets for the second and third vertices, with respect to the first vertex, were randomly selected from the range $-50 \ldots +50$. Measurements were

1. /* INITIALIZATION. Allocate array temporaries. We assume a multiprocessor target with distributed shared memory, and a data distribution for arrays X_2 and $@X_2$ that places each column on a separate processor. Compared to the abstract parallelization, note that we only need arrays X_2 and $@X_2$ and that their expansion factor is P (= # processors) instead of n (= # iterations). */

   `allocate` $X_2[1:m, 1:P]$, $@X_2[1:m, 1:P]$
   `distribute` $X_2[*, \text{BLOCK}]$, $@X_2[*, \text{BLOCK}]$
   Also initialize $X_2[*, *] := 0$

2. /* EXECUTION. Execute loop in parallel using array temporaries X_2 and $@X_2$ */

   /* At the concrete level, the computation model is that a processor may execute many iteration but it executes its iterations in the same relative order as the original loop. */

```
doall i := 1, n
    q := processor number for iteration i
    if (C[i]) then
        X2[f(i), q] := ...
        @X2[f(i), q] := i
    end if
end doall
```

3. /* FINALIZATION. Update array X with final value. Execute loop j with a BLOCK computation mapping to exploit spatial locality in arrays $X, X_2, @X_2$. */

```
doall j := 1, m
    /* Due to the shrinking of arrays X2 and @X2
    from m × n to m × P, we need to use
    MAXLOC instead of MAX.  MAXLOC(@X2[j, 1 : n])
    returns (the smallest) z such that
    ∀1≤k≤n @X2[j, z] ≥ @X2[j, k].  */
    temp := MAXLOC(@X2[j, 1 : P])
    if (@X2[j, temp] > 0) then
        X[j] := X2[j, temp]
    end if
end doall
```

4. `free` $X_2[1:m, 1:P]$, $@X_2[1:m, 1:P]$

Figure 13: Concrete parallelization of loop L

n	Serial Version	Parallel Version ($P=1$)	Parallel Version ($P=4$)	Speedup
10,000	3.6 s	3.8 s	1.4 s	2.6 ×
50,000	17.1s	17.4 s	4.8 s	3.6 ×
100,000	34.5 s	34.0 s	9.1 s	3.8 ×

Figure 14: Execution time measurements (in seconds) for rasterization of n polygons on a 4 processor SMP using the concrete parallelization transformation from section 3.1.2

made separately for n = 10,000, 50,000, and 100,000 to cover a reasonable range for the number of polygons typically encountered in rasterization [6].

The parallel version of this code was obtained by renaming and expanding the display buffer array variable by hand, according to the concrete parallelization technique described in section 3.1.2. In the execution phase on P processors, each processor performed rasterization for $1/P$ of the randomly generated polygons and stored the output in its local copy of the expanded display buffer while also updating a local @ array. The finalization phase then performed the reduction described in section 3.1.2 using the local @ arrays to obtain the final value of the global display buffer. In section 3.1.2, we said that we expected the concrete parallelization to scale linearly so long as the total work in the loop is at least $O(m \times P)$. If T is the average time to rasterize one triangle, this means that we expect linear speedup so long as T is at least $O(m \times P/n)$. Note that the array size, $m \approx 10^6$, is much larger than the values used for n, the number of iterations (polygons), which means that it will be harder to obtain linear speedup as n decreases.

For each value of n, figure 14 shows the wallclock execution times measured for the rasterization phase (after generation of n random triangles) in three cases:

1. Serial version — the sequential code for generic convex polygon scan conversion written by Paul Heckbert that is distributed with the "Graphics Gems" book [7]. This code was executed on a single processor.

2. Parallel version ($P = 1$) — the code obtained by concrete parallelization, but executed on a single processor.

3. Parallel version ($P = 4$) — the code obtained by concrete parallelization, and executed on four processors.

The execution times were measured on a Digital AlphaServer 4100 SMP containing four Alpha 21164 400 MHz pocessors. The speedup column reports the ratio of the execution time of the parallel version on four processors to the execution time of the serial version. The speedups for $n = 100,000$ and $n = 50,000$ are close to linear (3.8× and 3.6× respectively), whereas the speedup for $n = 10,000$ drops to 2.6×. This shows that the concrete parallelization transformation from section 3.1.2 can be effective in delivering speedup for a real application.

3.2 Parallelization with Speculative Execution

In this section, we show how Array SSA form can be used to enable speculative execution so as to parallelize loops with loop-carried true data dependences, so long as each cycle containing a loop-carried true data dependence also includes a loop-independent control dependence. As in section 3.1, this parallelization transformation works for loops with loop-carried anti and output dependences, unanalyzable subscript expressions, and arbitrary control flow within an iteration.

As an example, consider loop P shown in figure 15. The main difference from loop L in section 3.1 is that there is now a loop-carried data dependence from statement s3 to statement s2 on variable X. The key dependence cycle in this code is formed by this loop-carried data dependence and by a loop-independent control dependence from s2 to s3 due to the if construct. The Array SSA form for loop P with full insertion of @ arrays and ϕ functions is shown in

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

Figure 15: Example loop P

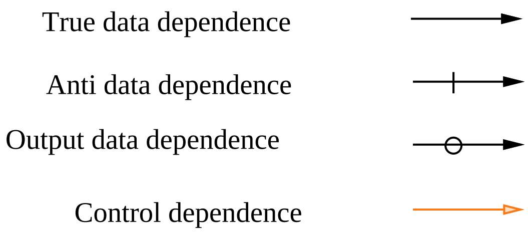
```
X0[...] :=
@X0[...] := ()
do i := imin, imax
   X1  := φ(X4, X0)
   @X1 := max(@X4, @X0)
   if ( f(X1[g(i)]) ) then
      X2[h(i)] := rhs(i)
      @X2[h(i)] := (i)
      X3  := φ(X2, X1)
      @X3 := max(@X2, @X1)
   endif
   X4  := φ(X3, X1)
   @X4 := max(@X3, @X1)
enddo
X5  := φ(X4, X0)
@X5 := max(@X4, @X0)
... := X5[...]
```

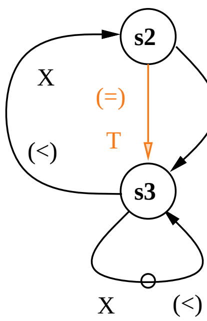
Figure 16: Array SSA form for loop P

figure 16. Figure 18 shows the optimized Array SSA form for loop P obtained by using copy propagation as described in section 2.3 (we show $@X_2$ as an explicit argument to the ϕ functions to make it easier to see the dependence structure of the loop).

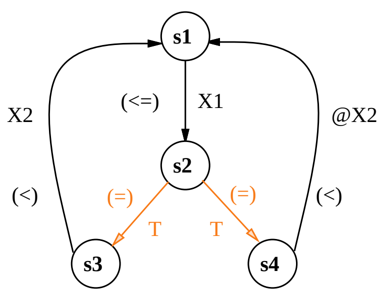
Consider statements s3 and s4 in figure 18. Note that statement s4: $@X_2[h(i)] := (i)$ cannot be executed speculatively because it is an @ array computation and the correctness of Array SSA form depends on element $@X_2[h(i)]$ being set $= (i)$ only for those iterations in which the if condition in s2 evaluates to true. However, Array SSA form gives us the option of computing statement s3: $X_2[h(i)] := rhs(i)$ speculatively for each iteration i, assuming that $rhs(i)$ has no side effects that inhibit speculative execution. There is no problem if statement s3 initializes extra elements of array X_2, because $@X_2$ will be used by the ϕ functions to select only those elements of X_2 that would have been computed by statement s3 in the original program (figure 15).

Figure 17 shows the dependences graphs for different versions of loop P. The dependence graph for the original loop P from figure 15 is shown in figure 17(a). We use the standard direction vector notation (=) and (<) to identify loop-independent and loop-carried dependences [18]. In addition to the true, output, and anti data dependences on variable X, there is a loop-independent control dependence from s2 to s3 due to the if construct. Figure 17(b) shows the dependence graph for the optimized Array SSA form in figure 18. Two new statements have been introduced, s1 for the ϕ function and s4 for the @ array computation. This dependence graph can be computed using standard techniques once Array SSA form has been constructed. Notice that there are no anti or output dependences in this dependence graph. Finally, figure 17(c) shows the dependence graph

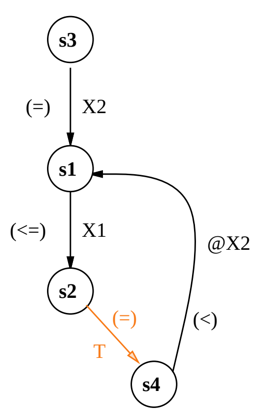




(a) Dependence graph for original loop P in figure 15.



(b) Dependence graph for optimized Array SSA form in figure 18.



(c) Dependence graph after selecting statement `s3` for speculative execution.

Figure 17: Dependence graphs for loop P

$X_0[\ldots] :=$
$@X_0[\ldots] := ()$
do $i := imin, imax$
`s1:` $X_1 := \phi(X_2, @X_2, X_0)$
`s2:` **if** ($f(X_1[g(i)])$) **then**
`s3:` $X_2[h(i)] := rhs(i)$
`s4:` $@X_2[h(i)] := (i)$
endif
enddo
$X_5 := \phi(X_2, @X_2, X_0)$
$@X_5 := \max(@X_2, @X_0)$
$\ldots := X_5[\ldots]$

Figure 18: Optimized SSA for loop P

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 19: Abstract parallelization for loop P

that is obtained when we decide to execute statement s3 speculatively. Statement s3 is no longer control dependent on statement s1. This reduces the dependence cycle for loop P to only contain statements s1, s2 and s4. Assuming that there was significant work in computing $rhs(i)$ in statement s3, this makes the critical cycle significantly shorter than before. In fact, array X_2 can be speculatively precomputed in parallel as we will now see.

Figure 19 shows the abstract parallelization of the optimized Array SSA form in figure 18, using speculative execution of statement s3. Analogous to the abstract parallelization transformation in section 3.1.1, the transformed code in figure 19 consists of an initialization phase, an execution phase, and a finalization phase. The initialization phase allocates arrays X_2, $@X_2$, and X_5, of which only array X_2 is expanded. Since this an abstract parallelization, the expansion factor for array X_2 equals the number of iterations in the loop (as in section 3.1.1). As before, a concrete parallelization will instead limit the expansion factor to be $\leq P$, the number of processors. The execution phase in figure 19 consists of a parallel loop and a sequential loop. The parallel loop speculatively precomputes expanded array X_2, and the sequential loop executes the dependence cycle consisting of statements s1, s2 and s4. No expansion is required for $@X_2$ because it is computed in the sequential loop. The following conditional expression for $\phi(X_2, X_0)$ is used as the basis for computing X_1 and X_5 at different points in figure 19:

$$X_1[j] = X_5[j] \quad = \quad \begin{array}{l} \textbf{if} \ \ @X_2[j] > 0 \ \ \textbf{then} \ \ X_2[j, @X_2[j]] \\ \textbf{else} \ \ X_0[j] \\ \textbf{end if} \end{array}$$

For X_1, statement s1 in the sequential loop in the execution phase simply computes $temp = X_1[g(i)]$ to obtain the value of the single element of array X_1 that is used by statement s2 in iteration i. Array X_5 is computed in the finalization phase, analogous to the finalization phase in section 3.1.1.

Though the parallel loop in figure 19 is written as a doall, a full barrier is not required between the parallel loop and the sequential loop in the execution phase. The parallel loop has to simply stay ahead of the sequential loop so that iteration i of the parallel loop is completed before iteration i of the sequential loop begins. How this is best accomplished depends on the target architecture. One possible concrete parallelization is shown in figure 20. It is based on pipelining chunks of iterations of the parallel and sequential loops. For a chunk size of n iterations, a single pipeline cycle consists of executing chunk $k+1$ of the the parallel loop in conjunction with chunk k of the sequential loop. Thus, the degree of parallelism in a single pipeline cycle is $(n+1)$. The array expansion factor for X_2 only needs to be n for this concrete parallelization. For convenience, figure 20 only shows the concrete parallelization for a single pipeline cycle of the execution phase, and excludes the prolog and epilog for this software pipeline. The chunk size n establishes the granularity of synchronization. Figure 20 does not address the mapping of the work in a single pipeline cycle to processors; the granularity of the processor mapping will depend on the amount of work in the parallel and sequential loops and on the target machine.

3.3 Parallelism Across Regions

The array renaming provided by Array SSA form can also be used to break loop-independent output and anti data dependences to enable parallel execution of distinct regions in a program.

```
/* Execute the (k+1)th chunk of the parallel loop. */

doall i := n*(k+1),  n*(k+2) - 1
    X2[h(i), i mod n] := rhs(i)
enddo

/* Execute the kth chunk of the sequential loop. */

do i := n*k,  n*(k+1) - 1
s1:    /* Set temp := X1[g(i)] */
       j := g(i)
       if (@X2[j] > 0) then
          temp := X2[j, @X2[j] mod n]
       else
          temp := X0[j]
       end if
s2:    if f(temp) then
s4:       @X2[h(i)] := i
       endif
enddo
```

Figure 20: Concrete parallelization for loop P: one pipeline cycle

```
{X initialized here.}

/* Region 1 */
do i := ...
   X[f(i)] := ...
enddo

/* Region 2 */
do i := ...
   if (cond(i)) then
      X[g(i)] := ...
   endif
enddo
```

Figure 21: Example code fragment with regions

As an example, consider the code fragment in figure 21. Its Array SSA form after optimizing the ϕ functions is shown in figure 22. We will consider three regions in figure 22: the two original loops that compute X_1 and X_2 respectively, and the finalization ϕ that computes X_3. If region 2 does not use values computed by region 1, regions 1 and 2 can be executed concurrently. Region 3 uses the ϕ function,

$$X_3 := \phi(X_2, @X_2, X_1, @X_1, X_0)$$

to combine the results of regions 1 and 2 as follows:

$$X_3[j] \quad = \quad \begin{array}{lll} \textbf{if} & @X_2[j] \neq \bot & \textbf{then} \ \ X_2[j] \\ \textbf{else if} & @X_1[j] \neq \bot & \textbf{then} \ \ X_1[j] \\ \textbf{else} & X_0[j] & \\ \textbf{end if} & & \end{array}$$

In this example, renaming removed an output dependence which enabled parallel execution. If the reference to array X in region 1 in figure 21 were a use rather than a definition, renaming would have broken an anti-dependence and would also have enabled parallel execution. In that case, the final ϕ would only need to combine values X_2 and X_0.

```
X_0[...] :=
@X_0[...] := ()

/* Region 1 */
do i := ...
    X_1[f(i)] := ...
    @X_1[f(i)] := (i)
enddo

/* Region 2 */
do i := ...
    if cond(i) then
        X_2[g(i)] := ...
        @X_2[g(i)] := (i)
    endif
enddo

/* Region 3 */
X_3 := φ(X_2, @X_2, X_1, @X_1, X_0)
```

Figure 22: SSA form for region parallelism

We only discussed abstract parallelization in this section. The appropriate strategy for concrete parallelization will depend on the level of parallelism to be exploited *e.g.,* task parallelism, multi-threading, parallel sections, or instruction level parallelism.

4 Related Work

There are currently several important approaches to program analysis. We discuss three below, scalar SSA, data dependence analysis and array data flow analysis. They attack the problem in three quite distinct ways and have distinct strengths and weaknesses. In brief, scalar SSA captures control flow and does renaming but lacks array index information and so is not useful for arrays. 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.

Static single assignment (SSA) form for scalar variables has been a significant advance. It has simplified the design of some optimizations and has made other optimizations more effective. Some of the earliest applications of SSA form were in the design of new algorithms for global constant propagation [17] and global value numbering [2, 12]. The popularity of SSA form surged after an efficient algorithm for computing SSA form was made available [5]. SSA form is now a standard representation used in modern optimizing compilers in both industry and academia.

However, it has been widely recognized that SSA form is much less effective for array variables than for scalar variables. The approach recommended in [5] is to treat an entire array like a single scalar variable in SSA form. For example, in this approach an assignment to a single array element $A[j] := v$ gets translated to an operation on the entire array $A := Update(A, j, v)$ which after SSA renaming would become $A_2 := Update(A_1, j, v)$. The most serious limitation of this approach is that it lacks precise data flow information on a per-element basis. Array SSA form addresses this limitation by providing ϕ functions that can combine array values on a per-element basis.

Data dependence analysis [18] has historically been the analysis of choice in the parallel 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 [9, 4, 3]). Of the approaches suggested in past work, the last write tree (LWT) in [9] 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 ϕ 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 ϕ functions identify which def to follow in each case. Even for the special cases in which LWT's can be created, the ϕ 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 classical SSA form for array variables because it is an element level approach. It 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 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 [13]. 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 ϕ functions in Array SSA form so as to increase the scope of compiler analysis and transformation.

5 Conclusions and Future Work

In this paper, 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 standard SSA form when applied to scalar variables. Its power lies in a very simple mechanism that combines control flow analysis, array index analysis and renaming.

One important application of our Array SSA form is in automatic parallelization. We showed that Array SSA form can enable parallelization of any loop that is free of loop-carried true data dependences. This includes loops with loop-carried anti and output dependences, arbitrary conditionals, unanalyzable loop bounds and strides, and unanalyzable subscript expressions. We validated this use of Array SSA form on a rasterization code which showed good speedups indicating that the overhead of the technique is minimal.

For future work, we plan to pursue other applications of Array SSA form including its use in representing data flow information for pointer accesses and other forms of aliasing. In this way, Array SSA form can be extended to serve as a universal SSA form for all classes of variables. We also plan to study the improvements that can be obtained in register allocation and instruction scheduling by using Array SSA information. We plan on investigating its use in supporting compilation problems requiring summaries of array accesses using systems of linear inequalities.

Acknowledgments

We would like to thank Rishiyur Nikhil, Bert Halstead, Matt Frank, Norm Rubin, Leonidas Kontothanassis, Roy Ju and especially Carl Offner and Saman Amarasinghe for technical discussions contributing to the ideas presented here. In addition, we would like to thank Bert Halstead for suggesting the graphics algorithm, Satyan Coorg for help in accessing and understanding the graphics code.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting Equality of Variables in Programs. *Fifteenth ACM Principles of Programming Languages Symposium*, pages 1–11, January 1988. San Diego, CA.

[3] S. P. Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities.* PhD thesis, Computer Systems Laboratory, Stanford University, January 1997.

[4] R. Bodik and R. Gupta. Array Data-Flow Analysis for Load-Store Optimizations in Superscalar Architectures. *Lecture Notes in Computer Science*, (1033):1–15. Proceedings of Eighth Annual Workshop on Languages and Compilers for Parallel Computing, Columbus, Ohio, August 1995.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth 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.

[6] James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley Publishing Company, 1990.

[7] Andrew S. Glassner. *Graphics Gems.* Academic Press, 1993.

[8] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, Inc., 1977.

[9] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array Data-Flow Analysis and its Use in Array Privatization. *Conf. Rec. Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[10] Jeffery Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University Computer Systems Lab, February 1997.

[11] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[12] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global Value Numbers and Redundant Computations. *Fifteenth ACM Principles of Programming Languages Symposium*, pages 12–27, January 1988. San Diego, CA.

[13] 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.

[14] R. M. Shapiro and H. Saint. The Representation of Algorithms. Technical report, Massachusetts Computer Associates, February 1970. Technical Report CA-7002-1432.

[15] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[16] J. Gregory Steffan and Todd C. Mowry. The Potential for Thread-Level Data Speculation in Tightly-Coupled Multiprocessors. Technical Report CSRI-TR-350, Department of Electrical and Computer Engineering, University of Toronto, February 1997.

[17] Mark Wegman and Ken Zadeck. Constant Propagation with Conditional Branches. *Conf. Rec. Twelfth ACM Symposium on Principles of Programming Languages*, pages 291–299, January 1985.

[18] Michael 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.

A Corrections

This preprint contains the following typographical corrections and modifications that are not in the conference version (the page boundaries remain unchanged):

- Page 1, column 1 — abstract has been rewritten
- Page 1, column 2 — "loops" replaced by "loop"
- Page 4, column 2 — ".." replaced by "."
- Page 4, column 2 — "." inserted after "$\max(@X_{h_1}, @X_{h_2})$"
- Page 4, column 2 — "Analagous" replaced by "Analogous"
- Page 5, column 1 — "from" replaced by "form"
- Page 5, column 2 — ":" replaced by "."
- Page 5, column 2 — space removed before "."
- Page 5, column 2 — extra ")" removed
- Page 5, column 2 — replaced "$\max(@X_2^i[j], X_0[j])$" by "$\max(@X_2^i[j], @X_0[j])$"
- Page 6, column 2 — inserted "iterations" before "$1 \ldots i$"
- Page 7, column 1 — replaced "Due" by "Due to"
- Page 9, column 1 — replaced "disproprotionately" by "disproportionately"
- Page 12, column 1 — replaced "$X0$" by "X_0"
- Page 12, column 1 — inserted "." after "$(n + 1)$"

If you find any other typographical errors, please send email to *knobe@crl.dec.com* and/or *vivek@lcs.mit.edu*.