The Advantages of Instance-wise Reaching Definition Analyses in Array (S)SA

Jean-François Collard

CNRS & PR*i*SM-University of Versailles,
45 avenue des États-Unis, 78035 Versailles, France
Jean-Francois.Collard@prism.uvsq.fr

Abstract. Several parallelizing or general-purpose compilers have been using intermediate representations based on some form of single-assignment. Extending these representations to arrays has been done in two ways: Array SA, and Array SSA, the extension of the widely-used Static Single Assignment (SSA). We formally define and compare Array SA and Array SSA, and show that (1) They both need instance-wise reaching-definition analyses to (a) be streamlined and (b) allow a parallelizing compiler to choose the most appropriate form, (2) The "same name - same value" property holds for Array SSA in a restricted way only.

1 Introduction

Single assignment is an important concept to reason about, and perhaps modify, imperative programs. Why? Because it allows to exhibit a fundamental property of the algorithm coded in a program: flows of values. In single assignment (SA), each scalar is assigned at most once. I.e., the dynamic, run-time instances of assignments in a program with loops assign to distinct variables. *Static Single Assignment* (SSA) is a slightly different form, defined as follows: There is only one assignment *statement* for each variable. Rosen et al. stress this difference from usual single assignment: In *static* SA, the unique statement assigning to a variable may have several instances writing successive values to the same variable. SSA has been widely used as an *intermediate representation* in compilers, because of its "same name - same value" property (to use an expression coined in [20]). Applications of SSA include constant propagation, global value numbering [21, 2], register promotion [22], etc. Moreover, some researchers have been using SSA as a *concrete form in the generated code* because it eliminates all output- and anti-dependences (e.g., cf [7] page 160).

Several methods have been crafted to extend these intermediate representations to arrays. Array SSA, which extends SSA to arrays, has been introduced lately thanks to Knobe and Sarkar [18]. In Array SSA, each *array* is assigned to by one *statement*. On the other hand, Array SA is the extension of SA [15, 16]: Each array *element* is written at most once, by a single statement *instance*. In both frameworks, ϕ-functions may be inserted to restore the flow of data when nondeterministic confluence points occur in the control flow.

However, several issues in Array SSA are still unclear.

1. Whereas formal definition of scalar SSA is well known, no formal definition exists for Array SSA. This in turn hampers the comparison with other related frameworks, such as its "sibling" Array SA.
2. Because of the lack of formal definition, it is unclear when and where properties on control flow should be taken into account in Array SSA, and what its relationship with reaching-definition analysis precisely is.
3. When parallelizing programs automatically, single-assignment is interesting not only as an intermediate representation but also as a concrete form for generated code. However, Array SSA does not eliminate all output- and anti-dependences, which in turn hampers from extracting all the parallelism from a program. For this reason, Knobe and Sarkar do not always parallelize using Array SSA in [18] (The reader may check that the loop in Figure 12 is different from the one in Figure 6 in [18]). As explained later, they use a special form of Array SA instead.
4. The *"same name - same value" property of SSA breaks down*, on most cases, for Array SSA. More precisely, two arrays with the same name store the same *set of* values. The "same name - same value" property, therefore, holds for the entire arrays ("Same array name - same array value"). Most programs, however, use arrays on a per-element basis, and the value equality of two elements is often what we are looking for. We show that Array SA solves this issue since it has the "same element name - same element value" property.

Issues 1, 2 and 3 are directly related to parallelizing compilers, and will be the core of this paper. Issue 4, hinting to extensions of classical algorithms based on (scalar) SSA, is addressed shortly in Section 8 and is left for future work. To sum things up, contributions of this paper include:

- Formal definitions of Array SSA and Array SA.
- An algorithm based on reaching definition analysis to translate to Array SA any program with reducible flow graph and with any array subscripts.
- Evidence that Array SSA fails to extract all the parallelism from programs.
- Evidence that Array SSA and Array SA without *instance-wise reaching-definition analyses* (IRDA, to be formally defined later) introduce useless ϕ-functions and the associated run-time overhead. Thanks to an instance-wise reaching-definition analysis, ϕ functions need to be introduced only when the control flow is nondeterministic or when array subscripts cannot be analyzed at compile-time.
- Criteria, based on the output of the IRDA, to guide the compiler when choosing among the various expansion schemes. We compare the respective benefits of Array SA and Array SSA for parallelization and give preliminary experimental results on an SGI Origin 2000.

Section 2 motivates this work. Section 3 iteration-wise reaching definition analyses. Section 4 then defines Array SA and Array SSA in a unified way. The algorithm for Array SA conversion appears in Section 5. Related work are discussed in Section 6, before we report some preliminary experimental results.

2 Motivations

In the context of automatic parallelization, it is well known that expansion of data structures allow to eliminate some (if not all) output- and anti-dependences and some (if not all) spurious true dependences. Actually, the only dependences to be preserved are those carrying the flow of values.

However, when arrays come into play, this property is not preserved by the extension of reaching definition analyses and of SSA to arrays:

Consider example `std` in Figure 1. Its Array SSA in Figure 2 does not expand array `a`, and therefore fails to extract all possible parallelism. On the contrary, Array SA form in Figure 3 eliminates all anti- and output-dependences. (Array SA uses φ-functions. Intuitively, these functions are similar to ϕ-functions in Array SSA.) This does not prove, however, that parallel programs with Array SA perform better: the overhead due to φ-functions and to the management of bigger arrays may not pay off. This point is discussed in Section 7 and in the conclusion.

On the other hand, we now show that, whichever single-assignment form is chosen, an instance-wise reaching definition analysis is needed to avoid useless ϕ- or φ-functions.

```
   a[..] = ...
   for i = 1 to n
     if( P(i) ) then
       for j = 1 to n
S        a[i+j] = 1+a[i+j-1]
       end for
     end if
   end for
```

Fig. 1. Example `std`.

Consider the examples in Figure 4. In both cases, an Array SSA without IRDA has spurious ϕ-nodes or ϕ-nodes with spurious arguments. (The corresponding codes translated to Array SSA appear in Figures 5 and 6.) For program `isv`, the value stored by S_2 is killed either by S_1 in the next iteration or, in the final iteration, by S_3. Therefore, whatever the value of `foo`, the only definitions reaching R are those of S_1 and S_3, array `A2` is not used by R and does not need to appear as an argument to ϕ in R'.

Now consider program `sjs`; it has only affine loops and affine subscripts, but it needs an IRDA to see that S_2 cannot be the definition reaching any of its right-hand side expressions or, more precisely, that the definition reaching

```
a[..] = ...
for i = 1 to n
   if( P(i) ) then
     for j = 1 to n
       AS[i+j] = 1+( @A_S[ i + j − 1 ]== −
         ?a[i+j-1]
         :AS[i+j-1])
       @A_S[ i + j ]= max≪ (@A_S[ i + j ],(i, j))
     end for
   end if
end for
```

Fig. 2. Array SSA form for example `std`. `@A` (introduced by Knobe and Sarkar) serves to restore the flow of data (which cannot be predicted at compile-time because of P). $-$ conventionally denotes the undefined value.

`A3[i+j-1]`, for given i and j, is:

$$\begin{array}{l} \textbf{if } j \geq 2 \\ \textbf{then } S_1 \text{ at iteration}(i, j-1) \\ \textbf{else } \textbf{if } i \geq 2 \\ \qquad \textbf{then } S_1 \text{ at iteration}(i-1, j) \\ \qquad \textbf{else } S_0 \end{array} \tag{1}$$

Moreover, no run-time decision has to be taken about the identity of the reaching definition: this identity is exactly given by (1), and, thanks to IRDA, the Array SSA version of `sjs` is simply the code in Figure 6.(b).

This yields to a very important remark. In the code in Figure 6.(b), some run-time computation does occur to compute the array element that stores the correct value. However, there is only one such array element, i.e., a *single* reaching definition for a given read, and thus no run-time decision has to be made among several possible array elements. In other words, the IRDA has been able to take into account the static behavior of control to eliminate non-reaching definitions.

3 Reaching Definition Analyses

Classical reaching-definition analyses compute, for every statement R with a reading reference to `r`, the set of all statements S defining `r` and such that there is a program path from S to R being free of any modification of `r`.

Let S be a statement in the program. Because of the surrounding control structures, S may execute several times. Our aim is to distinguish between these successive instances. The set of all the instances of S is denoted by $D(S)$. Therefore, a specific instance of S when its iteration vector is equal to w, $w \in D(S)$, is denoted by $\langle S, w \rangle$.

Let I be the set of all statements, and Ω and $\mathcal{W} \subset \Omega$ be, respectively, the set of all instances of all statements and the set of all instances of writes

```
for i = 1 to n
  if( P(i) ) then
    for j = 1 to n
      Last[i+j]= max≪(Last[i+j],(i,j))
      A[i,j] = 1 + if j ≥ 2
                   then A[i,j-1]
                   else if i ≥ 2
                        then φ(i,j)
                        else a[i+j-1]
    end for
  end if
end for

int φ (i,j) {  (j has to be equal to 1)
  return A[Last[i]] }
```

Fig. 3. Array SA form for example `std`. φ-functions in Array SA are similar to ϕ-functions in Array SSA, see Section 4. In the example, φ picks the last executed instance in $\{\langle S, i', j'\rangle : 1 \leq i' < i, 1 \leq j' \leq n, i' + j' = i + j - 1 = i\}$, where $\langle S, i', j'\rangle$ denotes the instance of S for iteration vector (i', j').

```
   for i = 1 to n do
S1   a[i] = ...
S2   if(..) a[i+1] = ...
   end for
S3 a[n+1] = ..
R  .. = a[foo]
```

(a) Example `isv`

```
S0 a[1] = 0
   for i = 1 to n
     for j = 1 to n
S1     a[i+j] = ...
S2     a[i] = ... a[i+j-1]
     end for
   end for
```

(b) Example `sjs`

Fig. 4. Two example programs `isv` and `sjs`.

(assignments). Moreover, the execution order on instances is a simple extension of the lexicographic order $\ll$ on iteration vectors [16], so $\ll$ is overloaded to mean both in the rest of this paper.

The goal of Instance-wise reaching definition analyses (IRDAs) is, for every *instance* of a statement R containing a reading reference `r`, to compute the set of instances of all assignments S, such that there is a program path from an instance of S to the instance of R being free of any modification of `r`. These analyses are sometimes iterative [14], but more often based on integer linear programming [12, 25, 4]. A more comprehensive study is presented in [13]. Notice that IRDAs handle any program, but a previous phase of the compiler is supposed to have eliminated `gotos` [3] (perhaps at the cost of some code duplication in the rare cases where the control graph is not reducible [1]). Arrays with *any* subscripts are handled, but not pointers.

```
   for i = 1 to n do
S1   A1[i] = ...
S2   if(..) A2[i+1] = ...
   end for
S3 A3[n+1] = ..
R' A4= φ(A1,A2,A3) (A2 spurious)
R  .. = A4[foo]
```

Fig. 5. Example `isv` from Figure 4 translated to Array SSA without IRDA, illustrating that spurious arguments may appear in ϕ-nodes.

```
a[1] = 0
for i = 1 to n
  for j = 1 to n
    A1[i+j] = ...
    A3 = φ(a,A1,A2) (φ spurious)
    A2[i] = ... A3[i+j-1]
  end for
end for
```

(a) Array SSA without IRDA.

```
a[1] = 0
for i = 1 to n
  for j = 1 to n
    A1[i+j] = ...
    A2[i] = ... if j ≥ 2
                then A1[i+j-1]
                else if i ≥ 2
                     then A1[i+j-1]
                     else a[1]
  end for
end for
```

(b) Array SSA with IRDA.

Fig. 6. Array SSA without IRDA (a) and with IRDA (b) for Program `sjs`. Thanks to IRDA, we know the exact single array element to be read and, therefore, no run-time desambiguation (ϕ) is needed.

The set of (instance-wise) definitions reaching reference r in $\langle R, r\rangle$ is denoted with $\mathbf{RD}(\mathsf{r}, \langle R, r\rangle)$. When the reference r is clear from the context, $\mathbf{RD}(\mathsf{r}, \langle R, r\rangle)$ is simply written $\mathbf{RD}(\langle R, r\rangle)$. As an example, let us consider an *instance*, parameterized with i and j, of the read reference `a[i+j-1]` in example `std` in Figure 1. Then, the definition $\mathbf{RD}(\langle S, i, j\rangle)$ reaching $\langle S, i, j\rangle$ is given by:

$$\begin{array}{l} \textbf{if } j \geq 2 \\ \textbf{then } \{\langle S, i, j-1\rangle\} \\ \textbf{else if } i \geq 2 \\ \qquad \textbf{then } \{\langle S, i', j'\rangle \; 1 \leq i' < i \;\wedge\; 1 \leq j \leq n \\ \qquad\qquad\qquad\qquad \wedge\; i' + j' = i + j - 1\} \\ \qquad \textbf{else } \{-\} \end{array} \tag{2}$$

where $-$ conventionally denotes the undefined value.

4 Array SA and Array SSA: Definitions and Comparison

4.1 Definitions of (S)SA Forms

The formal definition of (scalar) SSA is quite simple because, in the scalar case, all run-time instances of a statement write in the same memory cell: the cell indicated by the l-value. Array SSA gets intricate because we need to work element-wise: Two different instances of the same statement do access the same variable (the same array), but they may access two distinct elements of this array.

Let M (resp. M') denote the map from writes to memory locations in the original program (resp., in the transformed program). Expanding (the data structure associated with) two writes w_1 and w_2 is denoted with $M(w_1) = M(w_2) \ \wedge \ M'(w_1) \neq M'(w_2)$: the memory locations written by the two references were equal, but are not the same any longer in the transformed program.

Array SSA form is defined as:

$$\begin{array}{l} \forall S_p \in I, S_q \in I, \ w_p \in D(S_p), w_q \in D(S_q) : \\ (S_p \neq S_q) \ \Rightarrow \ (M'(\langle S_p, w_p \rangle) \neq M'(\langle S_q, w_q \rangle)) \end{array}$$

The Array SA form is defined as follows:

$$\begin{array}{l} \forall S_p \in I, S_q \in I, w_p \in D(S_p), w_q \in D(S_q) : \\ (S_p \neq S_q \ \vee w_p \neq w_q) \ \Rightarrow \ (M'(\langle S_p, w_p \rangle) \neq M'(\langle S_q, w_q \rangle)) \end{array}$$

For the sake of comparison, we also define Array Privatization and Maximal Static Expansion [5] in the same framework. Array Privatization is defined as:

$$\begin{array}{l} \forall S_p \in I : \\ \quad (\forall w_p \in D(S_p), S_q \in I, w_q \in D(S_q), \ \langle S_p, w_p \rangle \in \mathbf{RD}(\langle S_q, w_q \rangle) \Rightarrow w_q = w_p) \\ \quad \Rightarrow \big(\forall w_p, w'_p \in D(S_p) \ : \ M'(\langle S_p, w_p \rangle) \neq M'(\langle S_p, w'_p \rangle)\big) \end{array}$$

stating that the data structure written by S_p are expanded if all reached uses belong to the same iteration.

A Static Expansion is defined as:

$$\begin{array}{l} \forall \langle S_p, w_p \rangle, \langle S_q, w_q \rangle \in \mathcal{W} : \\ \quad (\exists z \in \Omega, \ \langle S_p, w_p \rangle \in \mathbf{RD}(z) \wedge \langle S_q, w_q \rangle \in \mathbf{RD}(z) \wedge M(\langle S_p, w_p \rangle) = M(\langle S_q, w_q \rangle) \\ \quad \Rightarrow (M'(\langle S_p, w_p \rangle) \neq M'(\langle S_q, w_q \rangle)) \end{array}$$

A static expansion M' is maximal if, for any static expansion M'', $\forall u, v \in \mathcal{W} \ : \ M'(u) = M'(v) \implies M''(u) = M''(v)$.

4.2 Construction of Array SSA and Array SA Forms

For expository reasons, we make the following restrictions: The input program has only one array `A` and only one statement R having one single reference `r` reading `A`. These assumptions can be removed easily: programs with several statements, each having possibly several read references to array `A`, are handled by subscripting ϕ- and φ-functions with the reference identity `r`. Multiple arrays are handled separately.

Array SSA Let us consider a statement R with iteration vector r reading an array `A`. Statement R is of the form:

$$\ldots \texttt{ := } \ldots \texttt{ A[} g(r) \texttt{] } \ldots$$

Let statements $S_1..S_n$ be the definitions reaching R. Each statement $S_p, 1 \leq p \leq n$, with iteration vector w_p has the form:

$$\texttt{A[} f_p(w_p) \texttt{] := } \ldots .$$

By definition of reaching definitions, we have: $\forall p, 1 \leq p \leq n \ : \ \exists w_p, \exists r$ s.t. $\langle S_p, w_p \rangle \in \mathbf{RD}(\texttt{A[} g(r) \texttt{]}, \langle R, r \rangle)$.

Conversion to Array SSA intermediate representation proceeds as follows. Statements $S_1..S_n$ become:

$$\texttt{A}_p \texttt{ [} f_p(w_p) \texttt{] := } \ldots$$

for all p, $1 \leq p \leq n$. Arrays $\texttt{A}_p$ match `A` in type, size and dimensions.

To each statement S_p writing in new array $\texttt{A}_p$, we associate function @ $\texttt{A}_p$ mapping an element $\texttt{A}_p\texttt{[}x\texttt{]}$ to its last definition by S_p. That is:

$$@\,\texttt{A}_p\texttt{[} x \texttt{]} = \max_{\ll} \{w_p \ : \ x = f_p(w_p) \wedge \texttt{exec}(\langle S_p, w_p \rangle) = \textbf{true}\}, \tag{3}$$

where $\texttt{exec}(\langle S_p, w_p \rangle)$ denotes that $\langle S_p, w_p \rangle$ actually executes. Obviously, this predicate is not known at compile-time, and this is one reason a reaching definition analysis yields sets. However, (3) defines a single value since $@\,\texttt{A}_p\texttt{[} x \texttt{]}$ is evaluated at run-time and $\texttt{exec}(\langle S_p, w_p \rangle)$ is then known.

On the other hand, due to non-deterministic branching structure of the control-flow graph, preserving the data flow requires some run-time mechanism. This mechanism is called a *ϕ-function* (which, admittedly, is not a real function according to the strict mathematical definition). A ϕ-function returns an array consistent with the data flow. ϕ returns an array $\texttt{A}_0\texttt{[1..n]}$ such that, for all $\texttt{A}_0\texttt{[} x \texttt{]}$, $1 \leq x \leq n$, there is an instance $\langle S_k, w_k \rangle$, such that $\texttt{A}_0\texttt{[}x\texttt{]}=\texttt{A}_k\texttt{[}x\texttt{]}$, and:

$$\langle S_k, w_k \rangle = \max_{\ll} \{\langle S_p, w_p \rangle \ : \ w_p = @\,\texttt{A}_p\texttt{[} x \texttt{]} \wedge \langle S_p, w_p \rangle \ll \langle R, r \rangle\} \tag{4}$$

(Recall that $\ll$ is overloaded to statement instances.) We see that predicate $\texttt{exec}(\langle S_p, w_p \rangle)$ does not need to be stored. Only @ `A` is stored and updated on the fly. The read reference $\texttt{A[}g(r)\texttt{]}$ in Statement R is replaced by a reference to the ϕ-function:

R: $\quad \ldots \texttt{ := } \ldots \phi(\texttt{A}_1, .., \texttt{A}_n) \texttt{ [} g(r) \texttt{] } \ldots$

Since the same function ϕ may be use at different places, it is more efficient to create a new array and to initialize this array by function ϕ at merge point in the control flow graph. Traditional SSA then replace statement R above with the following two statements:

R_1: $\mathtt{A}_0 \ \mathtt{:=} \ \phi(\mathtt{A}_1, .., \mathtt{A}_n)$
R_2: $\ldots \ \mathtt{:=} \ \ldots \ \mathtt{A}_0\mathtt{[}\, g(r) \,\mathtt{]} \ \ldots$

thus making explicit the array $\mathtt{A}_0$ defined above.

Array SA Array SA considers, for a given R, the same statements $S_1..S_n$. Transformation to Array SA, however, proceeds in a different way.

Each statement $S_p, 1 \leq p \leq n$, becomes:

$$\mathtt{A}_p\mathtt{[}\, w_p \,\mathtt{]} \ \mathtt{:=} \ \ldots$$

That is, for each S_p, there is a one-to-one correspondence between the elements of the new array $\mathtt{A}_p$ and the iteration domain of S_p.

For reasons explained above, some run-time restoration of data flows has to be performed using auxiliary functions. These functions are very similar to ϕ-function, but to stress the difference, we'll use the alternative Greek letter φ. Thus, the right-hand side of R could become:

$$\ldots := \varphi\left(\{\mathtt{A}_p\mathtt{[}\, w_p \,\mathtt{]} \ : \ \langle S_p, w_p\rangle \in \mathbf{RD}(\langle R, r\rangle)\}\right).$$

However, since the argument of φ just depends on R and r, we construct φ with $\langle R, r\rangle$ as the argument.

Formally, the semantics of φ is given by

$$\varphi(\langle R, r\rangle) = \mathtt{A}_k\mathtt{[}\, w_k \,\mathtt{]},$$

where:

$$\langle S_k, w_k\rangle = \max_{\ll} \{\langle S_p, w_p\rangle \ : \ \langle S_p, w_p\rangle \in \mathbf{RD}(\langle R, r\rangle) \wedge f_p(w_p) = g(r)\} \tag{5}$$

I.e., φ returns the array element defined by the last executed reaching definition instance. It is easy to see that ϕ and φ compute the same values: For a given $\langle R, r\rangle$, just take $x = g(r)$ in (4) and inline (3) in (4). φ is computed on the fly too, as explained in the next section.

5 Conversion to Array SA Form

We present here an algorithm to perform conversion into Array SA form. Let us first define $\mathsf{Stmt}(\langle S, w\rangle) = S$ and $\mathsf{Ind}(\langle S, w\rangle) = w$.

1. $\mathsf{isAffine} =$ **yes** if all array subscripts are affine, **no** otherwise.
2. For each assignment S (whose iteration vector is w and the left-hand side is $\mathtt{A}\mathtt{[}\, f(w) \,\mathtt{]}$):
   (a) Create an array $\mathtt{D}_S$ whose shape is (the rectangular hull of) $D(S)$.
   (b) Let w be an index in $D(S)$. The control structures surrounding S, such as conditionals or loops sweeping over w, are left unchanged.

(c) If there is a read u such that $\mathbf{RD}(u)$ is not a singleton and $\langle S, w\rangle \in \mathbf{RD}(u)$. Then, just after S, insert assignment
   i. $\mathsf{Last}\,[\,u\,] = \max_{\ll}(\mathsf{Last}\,[\,u\,], \langle S, w\rangle)$, if $\mathsf{isAffine} = \mathbf{yes}$.
   ii. $\mathsf{Last}\,[\,u, f(w)\,] = \max_{\ll}(\mathsf{Last}\,[\,u, f(w)\,], \langle S, w\rangle)$, if $\mathsf{isAffine} = \mathbf{no}$.
   (Last computes the result of the φ function on the fly.)

(d) Replace the left-hand side with $\mathtt{D}_S\,[\,w\,]$.

3. For each statement S, replace each read reference r to $\mathtt{A}[\ g(r)\ ]$ with $\mathsf{Convert}(\mathsf{r})$, where:
   - If $\mathbf{RD}(\langle S, w\rangle) = \{u\}$, then $\mathsf{Convert}(\mathsf{r}) = \mathtt{D}_{\mathsf{Stmt}(u)}\,[\,\mathsf{Ind}(u)\,]$.
   - If $\mathbf{RD}(\langle S, w\rangle) = \{-\}$, then $\mathsf{Convert}(\mathsf{r}) = \mathsf{r}$ (the initial reference expression).
   - If $\mathbf{RD}(\langle S, w\rangle)$ is a non-singleton set, then $\mathsf{Convert}(\mathsf{r}) = \varphi_{\mathsf{r}}(\langle S, w\rangle)$.
   - If $\mathbf{RD}(\langle S, w\rangle) =$ **if** p **then** r_1 **else** r_2, then $\mathsf{Convert}(\mathsf{r}) =$ `if` p `then` $\mathsf{Convert}(\mathsf{r}_1)$ `else` $\mathsf{Convert}(\mathsf{r}_2)$.

4. For each φ_{r}, output: if $\mathsf{isAffine} = \mathbf{yes}$:

   appropriate_type $\varphi(w)$ {
   `return` $\mathtt{D}_{\mathsf{Stmt}(\mathsf{Last}[w])}\,[\,\mathsf{Ind}(\mathsf{Last}[w])\,]$}

   if $\mathsf{isAffine} = \mathbf{no}$:

   appropriate_type $\varphi(w)$ {
   `return` $\mathtt{D}_{\mathsf{Stmt}(\mathsf{Last}[w, g(r)])}\,[\,\mathsf{Ind}(\mathsf{Last}[w, g(r)])\,]$}

   In both cases, output use the initial reference if the value stored in Last is $-$.

Let us apply this algorithm to Example `std` in Figure 1. The instance-wise reaching definitions are given by (2). We first create an array `A[1..n,1..n]`. The left-hand side is turned into `A[i,j]`, a 2-D array since the iteration domain is two-dimensional. The read reference in the right-hand side is changed according to (2).

Let us now generate the function φ. All instances of $\langle S, i', 1\rangle$, $2 \leq i' \leq n$ have a non-singleton set of reaching definitions. We thus define an array Last `[2..n,1]`. If $\langle S, i, j\rangle \in \mathbf{RD}(\langle S, i', 1\rangle)$, then $i' = i + j$. Therefore, we insert the assignment:

$$\mathsf{Last}\,[\,i + j, 1\,] = \mathtt{max}(\,\mathsf{Last}\,[\,i + j, 1\,], (i, j)\,)\ .$$

For a read $\langle S, i, 1\rangle$, $2 \leq i \leq n$, function φ is then:

$$\mathtt{integer}\ \ \varphi(i, 1)\ \{\ \mathtt{return}\ \mathtt{A}[\,\mathsf{Last}[i, 1]\,]\ ;\ \}$$

The resulting code appear in Figure 1.(c) (The second dimension of `Last` has been dropped since it is defined for 1 only).

Notice that our technique does not include any "pseudo-assignments" of φ-functions to intermediary arrays. The benefit is that placing φ-nodes is simple. The drawback is that, when the same φ-function is used several times, our scheme generates several instances of the same function. Notice also that, when iteration domains are not bounded at compile-time, the data structures $\mathtt{D}_S$ we allocate are not bounded either; we thus have to allocate them dynamically, or to tile the iteration space.

6 Related Work

Other Work on Array (S)SA Our method to convert programs with arrays to single-assignment form, adapted from [16,17], uses the result of a reaching definition analysis on arrays to get the SA intermediate representation. Recent work by Knobe and Sarkar [18], on the other hand, does not make this separation. So, is this an important issue?

We believe the answer is yes. Cutting the conversion to (S)SA into two phases (IRDA *then* transformation of the internal representation) has several benefits:

- A ϕ- or φ-function is needed only when the (compile-time) analysis fails to find the unique instance-wise reaching definition. In [18], ϕ-functions are inserted even for affine programs, whereas well-known analyses such as [16] can easily prove that ϕ-functions are not necessary. This has practical applications for compiler writers: When a useless ϕ-function has been inserted in a program, we know what to blame: The reaching definition analysis.
- The program may not be converted to single-assignment, perhaps because ϕ- and φ-functions are considered too expensive. A maximal static extension [5] may be preferred instead, on top of the reaching definition analysis.

Notice that the work by Knobe and Sarkar is, in our mind, complementary, since they prefer a robust "safety net" to more elaborate methods. How to use an instance-wise reaching definition analysis to improve array SSA has been described in this paper.

Other Work on SSA and Privatization Chow et al. [9] proposed an algorithm to derive (scalar) SSA without involving iterative dataflow analysis nor bit vectors. Interestingly enough, the same holds for our algorithm for Array SA. The Array SA we presented takes benefit of structured control-flow, and is therefore related to the work by Brandis and Mössenböck [8]. However, and on the contrary to what is often stated, the techniques presented here are not limited to structured counted loops nor to arrays with affine subscripts. Array privatization by Tu and Padua [23,24] also expose parallelism from programs and is based on data-flow analysis too. They, too, handle non-affine subscripts and detect private arrays whose last value assignments can be determined statically (which is related to avoiding spurious ϕ or φ functions). Their data-flow analysis, however, is not instance-wise. Moreover, their array expansion is limited to privatization, whereas our framework allows to use a wider set of expansions ranging from MSE [5] to Array SA.

Extending SSA to other data structures, such as languages with pointers, has been addressed in [10,20].

7 Preliminary Experimental Results

Preliminary experiments were made for Program `std` in Figure 1 on an SGI Origin 2000. Code in Array SA has been generated according to the algorithm

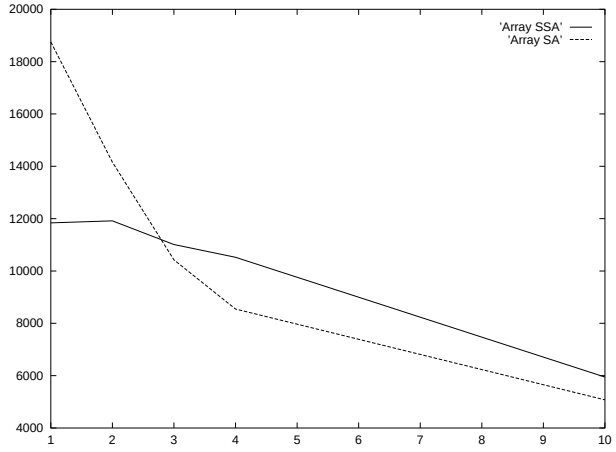

Fig. 7. Performance measures for Example `std` on an Origin 2000. The X-axis displays the number of processors. The Y-axis gives execution times, in milliseconds.

in Section 5. This form of code suits PCA, the automatic parallelizer for C. However, to be sure not to measure possible overhead or improvement due to PCA and not to Array SA, the generated codes were augmented by hand with directives from the `mp` library. Because of the lack of space, the reader is refered to the tech report [11] for details about the generated code.

Execution times appear in Figure 7. The experiment was done with $n = 9600$, from 1 to 10 processors, and the chosen predicate P was $P(i) = odd(i)$. These results are not surprising: with few processors, the parallelism in the Array SA version does not compensate for the overhead due to the management of bigger arrays and to the φ function. On the other hand, the Array SA version scales better. Comparing performances on a more meaningful set of programs is left for future work, so as to analyse which architecture-dependent parameters the choice of expansion should be based upon.

8 Extending Algorithms Based on SSA Form

SSA form has been very successful, as demonstrated by the large number of algorithms based on it. This success is due in part to the ability of SSA to express the semantic equality of expressions. So how can this success carry over to arrays?

An instance-wise reaching definition analysis gives accurate use-def relationships at the array-element- and statement-instance- level. Therefore, element-wise reaching definition analysis directly allows some optimizations, without the need for SA or SSA form. For instance, element-wise dead code elimination is simply given by eliminating the set of operations $\{u : \nexists v, u \in \mathbf{RD}(v)\}$.

The main benefit of instance-wise reaching definition analyses, however, is due to the following observation: If two references have the same single instance-wise definition, then their values are equal[1]. (Note that we don't know what this value is, but the value numbers are the same [20].) Array SA exactly captures the information given by an instance-wise reaching definition analysis (which Array SSA does not) and, therefore, allows to syntactically express value equalities.

On the other hand, it has been shown [19] that the value-flow graph is a representation of semantic equivalence superior to (classical scalar) SSA. It would thus be interesting to try and extend the work of Knoop et al. to programs with arrays.

Extending classical optimization algorithms (detection of common subexpression, redundancy elimination, construction of value flows [6], etc) to arrays, and comparing the respective benefits of Array SA and Array SSA, is definitely a very important future work.

9 Conclusion

We formally defined Array SSA and Array SA in a uniform way, and compared the two frameworks. We have shown that both Array SSA and Array SA require an instance-wise reaching definition analyses on arrays to avoid, when possible, the run-time overhead of restoring data flows. Moreover, we cannot express with Array SSA that two array elements having the same name have the same value. In addition, in the context of automatic parallelization, Array SSA fails to extract all the parallelism from programs.

This last point, however, may not be a drawback, since for some programs extracting all the parallelism implies a lot of ϕ- or φ-functions, with the associated overhead. The question is: "When choose Array SSA for my favorite programs? When choose Array SA? When choose maximal static expansion [5]?". The intuitive answer is: Let the compiler make an instance-wise reaching definition analysis first, because it will be needed anyway. Then, the more (and the larger) the reaching definition sets are, the more costly the ϕ- and φ-functions will probably be. Much more experimental studies are needed to assess this intuition and to allow expansion tuning.

Acknowledgment The author is supported by the CNRS (National Center for Scientific Research) and, in addition, by INRIA (Project A3) and a German-French project Procope. Access to the SGI Origin 2000 was provided by the University Louis Pasteur, Strasbourg. Many thanks to G.-R. Perrin and P. Brua for their help. This work benefited from fruitful discussions with D. Barthou, L. Bougé, A. Cohen, P. Feautrier, J. Knoop, F. Irigoin, V. Lefebvre, and L. Vibert.

[1] Therefore, iteration-wise reaching definition analyses allow to detect equal values, in a more general way (because arrays are handled element-wise) and more precise way (because loop iterations are considered separately) than most classical meet-over-paths dataflow analyses.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass, 1986.
2. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 1–11, January 1988.
3. B. S. Baker. An algorithm for structuring programs. *J. of the ACM*, 24:98–120, 1977.
4. D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints*. PhD thesis, Univ. Versailles, February 1998.
5. D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 98–106, San Diego, CA, January 1998.
6. R. Bodík and S. Anik. Path-sensitive value-flow analysis. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 237–251, San Diego, CA, January 1998.
7. R. Bodík and R. Gupta. Partial dead code elimination using slicing transformations. In *ACM SIGPLAN Conf on Prog. Lang. Design and Implem. (PLDI)*, pages 159–170, Las Vegas, Nevada, January 1997.
8. M. M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. on Prog. Languages and Systems*, 16(6):1684–1698, November 1994.
9. F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on ssa form. In *ACM SIGPLAN Conf on Prog. Lang. Design and Implem. (PLDI)*, pages 273–286, Las Vegas, Nevada, June 1997.
10. F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in ssa form. In *Int. Conf on Compiler Construction (CC'96)*, pages 253–267, 1996.
11. J.-F. Collard. Array SSA: Why? how? how much? Technical report, PRISM, U. of Versailles, 1998.
12. J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *ACM SIGPLAN Symp. on Principles and Practive of Parallel Prog. (PPoPP)*, pages 92–102, Santa Barbara, CA, July 1995.
13. J.-F. Collard and J. Knoop. A comparative study of reaching definitions analyses. Technical Report 1998/22, PRISM, U. of Versailles, 1998.
14. E. Duesterwald, R. Gupta, and M.-L. Soffa. A practical data flow framework for array reference analysis and its use in optimization. In *ACM SIGPLAN'93 Conf. on Prog. Lang. Design and Implementation*, pages 68–77, June 1993.
15. P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, pages 429–441, 1988.
16. P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.
17. M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In S. Haridi, K. Ali, and P. Magnusson, editors, *EURO-PAR '95*, Lecture Notes in Computer Science 966, pages 315–326. Springer-Verlag, 1995.
18. K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 107–120, San Diego (CA), January 1998.

19. J. Knoop, O. Rüthing, and B. Steffen. Code motion and code placement: Just synonyms? In *Proceedings of the 7th European Symposium on Programming (ESOP'98)*, volume 1381, pages 154–169, Lisbon, Portugal, May 1998.
20. C. Lapkowski and L. J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. In K. Koskimies, editor, *Compiler Construction CC'98*, volume 1383 of *LNCS*, pages 128–143, Lisbon, Portugal, March 1998. Springer-Verlag.
21. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 12–27, 1988.
22. A.V.S. Sastry and R. D. C. Ju. A new algorithm for scalar register promotion based on SSA form. In *ACM SIGPLAN Conf on Prog. Lang. Design and Implem. (PLDI)*, pages 15–25, Montreal, Canada, June 1998.
23. P. Tu and D. Padua. Automatic array privatization. In *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 500–521, August 1993. Portland, Oregon.
24. P. Tu and D. Padua. Gated SSA-Based demand-driven symbolic analysis for parallelizing compilers. In *ACM Int. Conf. on Supercomputing*, pages 414–423, Barcelona, Spain, July 1995.
25. D. G. Wonnacott. *Constraint-Based Array Dependence Analysis.* PhD thesis, University of Maryland, 1995.