Region Array SSA

Silvius Rus, Guobin He, Christophe Alias, Lawrence Rauchwerger

Region Array SSA *

Silvius Rus†
Texas A&M University

Guobin He†
Texas A&M University

Christophe Alias‡
ENS Lyon

Lawrence Rauchwerger†
Texas A&M University

ABSTRACT

Static Single Assignment (SSA) has become the intermediate program representation of choice in most modern compilers because it enables efficient data flow analysis of scalars and thus leads to better scalar optimizations. Unfortunately not much progress has been achieved in applying the same techniques to array data flow analysis, a very important and potentially powerful technology. In this paper we propose to improve the applicability of previous efforts in array SSA through the use of a symbolic memory access descriptor that can aggregate the accesses to the elements of an array over large, interprocedural program contexts. We then show the power of our new representation by using it to implement a basic data flow algorithm, reaching definitions. Finally we apply this analysis to array constant propagation and array privatization and show performance improvement (speedups) for benchmark codes.

1. INTRODUCTION

Important compiler optimization or enabling transformations such as constant propagation, loop invariant motion, expansion/privatization depend on the power of data flow analysis. The Static Single Assignment (SSA) [10] program representation has been widely used to explicitly represent the flow between definitions and uses in a program.

SSA relies on assigning each definition a unique name and ensuring that any use may be reached by a single definition. The corresponding unique name appears at the use site and offers a direct link from the use to its corresponding and unique definition. When multiple control flow edges carrying different definitions meet before a use, a special ϕ node is inserted at the merge point. Merge nodes are the only statements allowed to be reached directly by multiple definitions.

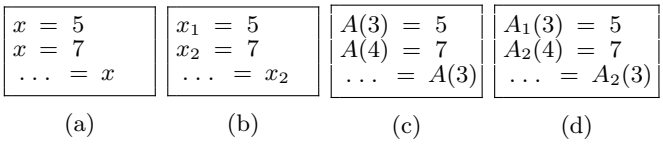

Figure 1: (a) Scalar code, (b) scalar SSA form, (c) array code and (d) improper use of scalar SSA form for arrays.

Classic SSA is limited to scalar variables and ignores control dependence relations. Gated SSA [1] introduced control dependence information in the ϕ nodes. This helps selecting, for a conditional use, its precise definition point when the condition of the definition is implied by that of the use [27]. The first extensions to array variables ignored array indices and treated each array definition as possibly killing all previous definitions. This approach was very limited in functionality. Array SSA was proposed by [16, 24] to match definitions and uses of partial array regions. However, their approach of representing data flow relations between individual array elements makes it difficult to complete the data flow analysis at compile time and requires potentially high overhead run-time evaluation. Section 6 presents a detailed comparison of our approach against previous related work.

We propose an *Array SSA* representation of the program that accurately represents the *use-def* relations between array regions and accounts for control dependence relations. We use the *USR* symbolic representation of array regions [23] which can represent uniformly memory location sets in a compact way for both static and dynamic analysis techniques. We present a reaching definition algorithm based on Array SSA that distinguishes between array subregions and is control accurate. The algorithm is used to implement array constant propagation and array privatization for automatic parallelization, for which we present whole application improvement results. Although the Array SSA form that we present in this paper only applies to structured programs that contain no recursive calls, it can be generalized to any programs with an acyclic Control Dependence Graph (except for self-loops).

2. REGION ARRAY SSA FORM

Static Single Assignment (SSA) is a program representation that presents the flow of values explicitly. In Fig. 1(a), the compiler must analyze the control flow graph in order to

*This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, ACI-0326350, and by the DOE.

†Parasol Lab, Department of Computer Science, Texas A&M University, {silviusr,guobinh,rwerger}@cs.tamu.edu.

‡Laboratoire de l'Informatique du Parallélism, ENS Lyon, Christophe.Alias@ens-lyon.fr

Author version
Published at PACT 2006

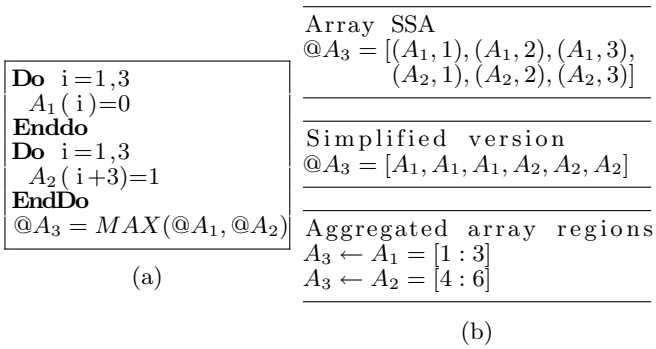

Figure 2: (a) Sample code in Array SSA form (not all gates shown for simplicity). (b) Array SSA forms: (top) as proposed by [16], (center) with reduced accuracy and (bottom) using aggregated array regions.

decide which of the two values, 5 or 7, will be used in the last statement. By numbering each static definition and matching them with the corresponding uses, the use-def chains become explicit. In Fig. 1(b) it is clear that the value used is x_2 (7) and not x_1 (5).

Unfortunately, such a simple construction cannot be built for arrays the same way as for scalars. Fig. 1(d) shows a failed attempt to apply the same reasoning to the code in Fig. 1(c). Based on SSA numbers, we would draw the conclusion that the value used in the last statement is that defined by A_2, which would be wrong. The fundamental reason why we cannot extend scalar SSA form to arrays directly is that an array definition generally does not kill all previous definitions to the same array variable, unlike in the case of scalar variables. In Fig. 1(c), the second definition does not kill the first one. In order to represent the flow of values stored in arrays, the SSA representation must account for individual array elements rather than treating the whole array as a scalar.

Element-wise Array SSA was proposed as a solution by [24]. Essentially, for every array there is a corresponding *@ array*, which stores, at every program point and for every array element, the location of the corresponding reaching definition under the form of an iteration vector. The computation of *@ arrays* consists of lexicografic *MAX* operations on iteration vectors. Although there are methods to reduce the number of *MAX* operation for certain cases, in general they cannot be eliminated. This led to limited applicability for compile-time analysis and potentially high overhead for derived run-time analysis, because the *MAX* operation must be performed for each array element.

We propose a new Region Array SSA (RA SSA) representation. Rather than storing the exact iteration vector of the reaching definition for each array location, we just store the SSA name of the reaching definition. Although our representation is not as precise as [24], that did not affect the success of our associated optimization techniques. This simplification allowed us to employ a different representation of *@ arrays* as aggregated array regions. Fig. 2 depicts the relation between element-wise Array SSA and our Region Array SSA. Rather than storing for each array element its reaching definition, we store, for each use-def relation such as $A_3 \leftarrow A_1$, the whole array region on which values defined at A_1 reach A_3.

We use the USR [23] representation for array regions, which can represent uniformly arbitrarily complex regions. Moreover, when an analysis based on USRs cannot reach a static decision, the analysis can be continued at run time with minimal necessary overhead. For instance, in the example in Fig. 2, let us assume that the loop bounds were not known at compile time. In that case the *MAX* operation could not be performed statically. Its run time as proposed by [24] would require $O(n)$ time, where n is the dimension of the array. Using Region Array SSA, the region corresponding to $A_3 \leftarrow A_1$ can be computed at run time in $O(1)$ time, thus independent of the array size. Our resulting Region Array SSA representation has two main advantages over [16]:

• We can analyze many complex patterns at compile time using symbolic array region analysis (essentially symbolic set operations), whereas the previous Array SSA representation often fails to compute element-wise MAX operations symbolically (for the complex cases).

• When a static optimization decision cannot be reached, we can extract significantly less expensive run time tests based on partial aggregation of array regions.

We will now present the USR representation for array regions, describe the structure of Region Array SSA, and then illustrate its use in an algorithm that computes reaching definitions for arrays.

2.1 Array Region Representation: the USR

In the example in Fig. 3(a), we can safely propagate constant value 0 from the definition at site 2 to the use at site 5 because the array region *used*, [1:5], is included in the array region *defined* above, [1:10]. In the example in Fig. 3(b), we cannot represent the array regions as intervals because the memory references are guarded by an array of conditionals. However, we can represent the array regions as *expressions on intervals*, in which the operators represent predication (#) and expansion across an iteration space ($\otimes^{\cup}$). This symbolic representation allows us to compare the *defined* and *used* regions even though their shapes are not linear. In the example in Fig. 3(c), a static decision cannot be made. The needed values of the predicate array C(:) may only be known at run time. We can still perform constant propagation on array A optimistically and validate the transformation dynamically, in the presence of the actual values of the predicate array. Although the profitability of such a transformation in this particular example is debatable due to the possibly high cost of checking the values of C(:) at run time, in many cases such costs can be reduced by partial aggregation and amortized through hoisting and memoization.

The *Uniform Set of References* (**USR**) previously introduced in [23][1] formalizes the *expressions on intervals* shown in Fig. 3. It is a general, symbolic and analytical representation for memory reference sets in a program. It can represent the aggregation of scalar and array memory references at any hierarchical level (on the loop and subprogram call graph) in a program. It can represent the control flow (predicates), inter-procedural issues (call sites, array reshaping, type overlaps) and recurrences. The simplest form of a USR is the Linear Memory Access Descriptor (LMAD) [20], a symbolic representation of memory reference sets accessed through linear index functions. It may have multiple dimen-

[1]USRs were presented there under the name of RT_LMAD because they were used mostly to produce run time tests.

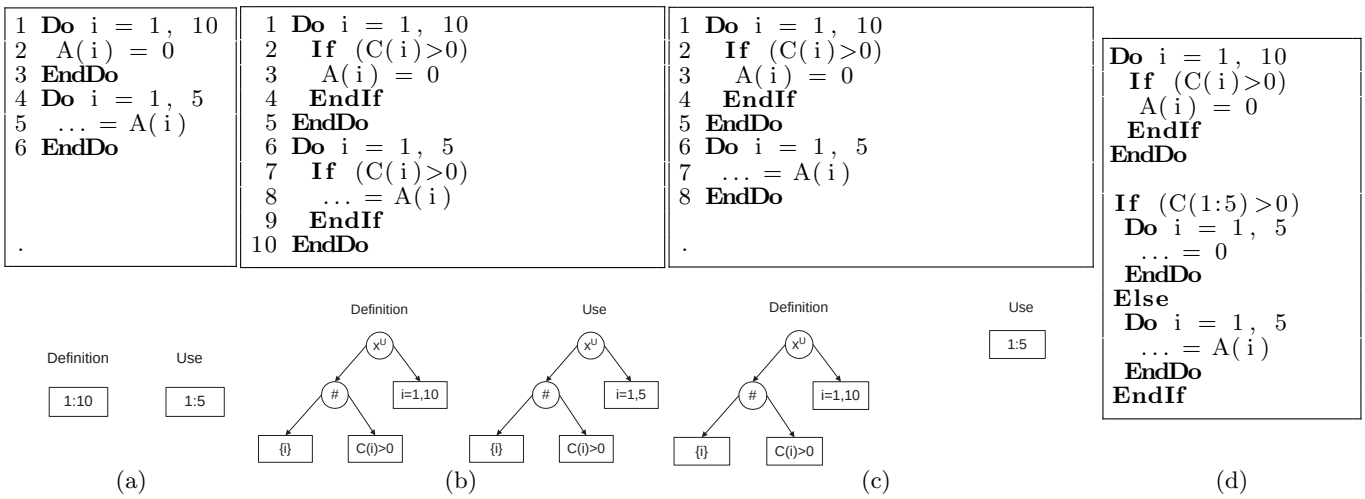

Figure 3: Constant propagation scenarios: (a) symbolically comparable linear reference pattern, (b) symbolically comparable nonlinear reference pattern, (c) nonlinear reference pattern that require a run time test and (d) dynamic constant propagation code for case (c).

sions, and all its components may be symbolic expressions. Throughout this paper we will use the simpler interval notation for unit-stride single dimensional LMADs. For the loop in Fig. 3(a), the array subregion *defined* by the first loop can be represented as an LMAD, [1:10], and the array subregion *used* in the second loop can also be represented as another LMAD, [1:5].

The USR is stored as an abstract syntax tree with respect to the language presented in Fig. 4 and can be thought of as symbolic expressions on sets of memory locations. When memory references are expressed as linear functions, USRs consist of a single leaf, i.e., a list of LMADs. When the analysis process encounters a nonlinear reference pattern or when it performs an operation (such as set difference) whose result cannot be represented as a list of LMADs, we add internal nodes that record accurately the operations that could not be performed.

In the examples in Fig. 3(b,c), memory references are predicated by an array of conditions. This nonlinear reference pattern cannot be represented as an LMAD, so it is expressed as a nontrivial USR. Although nothing is known about the predicates, the USR representation allows us to compare the *definition* and *use* sets symbolically in case (b). In case (c) a static decision cannot be made. However, using USRs we can formulate efficient run time tests that will guarantee the legality of the constant propagation transformation at run time. [23] showed how to extract efficient run time tests from identities of type $S = \emptyset$, where S is a USR. By setting $S = used - defined$, we can extract conditions that guarantee the safety of the optimistic constant propagation (Fig. 3(d)). Additionally, constant propagation may enable other more profitable transformations such as automatic parallelization by simplifying the control flow and the memory reference pattern.

Most examples in this paper only present single-indexed arrays accessed using a unit stride *solely for the simplicity of the presentation*. The USRs can represent *any memory reference pattern* produced by multidimensional arrays accessed over arbitrary large loop nests spanning multiple subroutines. Strides can be arbitrary symbolic expressions.

$$
\begin{aligned}
&\Sigma = \{\cap, \cup, -, (,), \#, \otimes^{\cup}, \otimes^{\cap}, \bowtie, \\
&\qquad LMADs, Gate, Recurrence, CallSite\} \\
&N = \{USR\}, \ S = USR \\
&P = \{USR \rightarrow LMADs | (USR) \\
&\qquad USR \rightarrow USR \cap USR \\
&\qquad USR \rightarrow USR \cup USR \\
&\qquad USR \rightarrow USR - USR \\
&\qquad USR \rightarrow Gate \# USR \\
&\qquad USR \rightarrow \otimes^{\cup}_{Recurrence} USR \\
&\qquad USR \rightarrow \otimes^{\cap}_{Recurrence} USR \\
&\qquad USR \rightarrow USR \bowtie CallSite\}
\end{aligned}
$$

Figure 4: USR formal definition. $\cap$, $\cup$, $-$ **are elementary set operations: intersection, union, difference.** $Gate\#USR$ **represents reference set** USR **predicated by condition** $Gate$**.** $\otimes^{\cup}_{i=1,n} USR(i)$ **represents the union of reference sets** $USR(i)$ **across the iteration space** $i = 1, n$**.** $USR(formals) \bowtie Call\ Site$ **represents the image of the generic reference set** $USR(formals)$ **instantiated at a particular call site.**

USRs also contain control dependence information. The only restriction is that the program must be structured.

2.2 Array SSA Definition and Construction

2.2.1 Region Array SSA Nodes

In scalar SSA, pseudo statements ϕ are inserted at control flow merge points. These pseudo statements show which scalar definitions are combined. [1] refines the SSA pseudo statements in three categories, depending on the type of merge point: γ for merging two forward control flow edges, μ for merging a loop-back arc with the incoming edge at the loop header, and η to account for the possibility of zero-trip loops. The array SSA form proposed in [24] presents the need for additional ϕ nodes after each assignment that does not kill the whole array. These extensions, while necessary, are not sufficient to represent array data flow efficiently because they do not represent array indices.

In order to provide a useful form of Array SSA, it is nec-

```
1   A(1) = 0                5   Do i = 3,10          10  ··· = A(1)
2   If (x > 0)              6     A(i) = 3           11  ··· = A(5)
3     A(2) = 1              7     ... = A(...)       12  If (x > 0)
4   EndIf                   8     A(i + 8) = 4       13    ··· = A(2)
                            9   EndDo                14  EndIf
```

(a)

```
    A0 :  [A0, ∅] = Undefined
 1  A1 :  A1(1)=0
    A2 :  [A2, {1}] = δ(A0, [A1, {1}])
 2        If (x>0)
    A3 :  [A3, ∅] = π([A2, (x > 0)])
 3  A4 :  A4[2] = 0
    A5 :  [A5, {2}] = δ(A3, [A4, {2}])
 4        EndIf
    A6 :  [A6, {1} ∪ (x > 0)#{2}] =
                γ(A0, [A2, {1}], [A5, (x > 0)#{2})
 5        Do i = 3, 10
    A7 :  [A7, [3 : i + 2] ∪ [11 : i + 8]] =
          μ(A6, (i = 3, 10), [A9, [3 : i − 1]], [A11, [11 : i + 7]])
 6  A8 :  A8(i) = 3
    A9 :  [A9, {i}] = δ(A7, [A8, {i}])
 7              ···=A9(···)
 8  A10 : A10(i + 8) = 4
    A11 : [A11, {i, i + 8}] = δ(A7, [A9, {i}], [A10, {i + 8}])
 9        EndDo
    A12 : [A12, {1} ∪ (x > 0)#{2} ∪ [3 : 18]] =
                η(A0, [A6, {1} ∪ (x > 0)#{2}], [A7, [3 : 18]]
10        ···=A12(1)
11        ···=A12(5)
12        If (x>0)
     A13 : [A13, ∅] = π(A12, (x > 0))
13          ···=A13(2)
14        Endif
```

(b)

Figure 5: (a) Sample code and (b) Array SSA form

essary to incorporate array region information into the representation. Region Array SSA gates differ from those in scalar SSA in that they represent, at each merge point, the array subregion (as a USR) corresponding to every ϕ function argument.

$$[A_n, \Re_n] = \phi(A_0, [A_1, \Re_1^n], [A_2, \Re_2^n], \ldots, [A_m, \Re_m^n]) \quad (1)$$

$$where\ \Re_n = \bigcup_{k=1}^{m} \Re_k^n\ and\ \Re_i^n \cap \Re_j^n = \emptyset, \quad \forall\, 1 \le i, j \le m, i \ne j \quad (2)$$

Equation 1 shows the general form of a ϕ node in Region Array SSA. $\Re_k^n$ is the array region (as USR) that carries values from definition A_k to the site of the ϕ node. Since $\Re_k^n$ are mutually disjoint, they provide a basic way to find the definition site for the values stored within a specific array region at a particular program context. Given a set $\Re_{Use(A_n)}$ of memory locations read right after A_n, equation 1 tells us that $\Re_{Use(A_n)} \cap \Re_k^n$ was defined by A_k. The free term A_0 is used to report locations undefined within the program block that contains the ϕ node. Let us note that two array regions can be disjoint because they represent different locations but also because they are controlled by contradictory predicates.

Essentially, our ϕ nodes translate basic data flow relations to USR comparisons. These USR comparisons can
• be performed symbolically at compile time in most practical cases, and
• be solved at run time with minimal necessary overhead, based on USR partial aggregation capabilities.

Our node placement scheme is essentially the same as in [24]. In addition to ϕ nodes at control flow merge points, we add a ϕ node after each array definition. These new nodes are named δ. They merge the effect of the immediately previous definition with that of all other previous definitions. Each node corresponds to a structured block of code. In the example in Fig. 5, A_2 corresponds to statement 1, A_6 to statements 1 to 4, A_{11} to statements 6 to 8, and A_{12} to statements 1 to 9. In general, a δ node corresponds to the maximal structured block that ends with the previous statement.

2.2.2 Abstraction of Partial Kills: δ Nodes

In the example in Fig. 5, the array use A(1) at statement 10 could only have been defined at statement 1. Between statement 1 and statement 10 there are two blocks, an *If* and a *Do*. We would like to have a mechanism that could quickly tell us not to search for a reaching definition in any of those blocks. We need SSA nodes that can summarize the array definitions in these two blocks. Such summary nodes could tell us that the range of locations defined from statement 2 to statement 9 does not include A(1).

The function of a δ node is to aggregate the effect of disjoint structured blocks of code. [2] Fig. 6(a) shows the way we build δ gates for straight line code. Since the USR representation contains built-in predication, expansion by a recurrence space and translation across subprogram boundaries, the δ functions become a powerful mechanism for computing accurate use-def relations.

Returning to our example, the exact reaching definition of the use at line 10 can be found by following the use-def chain $\{A_{12}, A_6, A_2, A_1\}$. A use of $A_{12}(20)$ can be classified as undefined using a single USR intersection, $\{A_{12}, A_0\}$.

2.2.3 Abstraction of Loops: μ Nodes

The semantics of μ for Array SSA is different than those for scalar SSA. Any scalar assignment kills all previous ones (from a different statement or previous iteration). In Array SSA, different locations in an array may be defined by various statements in various iterations, and still be visible at the end of the loop. In the code in Fig. 5(a), Array A is used at statement 7 in a loop. In case we are only interested in its reaching definitions from within the same iteration of the loop (as is the case in array privatization), we can apply the same reasoning as above, and use the δ gates in the loop body. However, if we are interested in all the reaching definitions from previous iterations as well as from before the loop, we need additional information. The μ node serves this purpose.

$$[A_n, \Re_n] = \mu(A_0, (i = 1, p), [A_1, \Re_1^n], \ldots, [A_m, \Re_m^n]) \quad (3)$$

The arguments in the μ statement at each loop header are all the δ definitions within the loop that are at the immediately inner block nesting level (Fig. 6(c)), and in the order in which they appear in the loop body. Sets $\Re_k^n$ are functions of the loop index i. They represent the sets of memory

[2] A δ function at the end of a *Do* block is written as η, and at the end of an *If* block as γ to preserve the syntax of the conventional GSA form. A δ function after a subroutine call is marked as θ, and summarizes the effect of the subroutine call on the array.

locations defined in some iteration $j < i$ by definition A_k and not killed before reaching the beginning of iteration i. For any array element defined by A_k in some iteration $j < i$, in order to reach iteration i, it must not be killed by other definitions to the same element, which occur from that point on until the beginning of iteration i. We must thus subtract the regions defined in iteration j after definition A_k: $Kill_s(j)$, as well as all the regions defined in the subsequent iterations $j+1$, ..., $i-1$: $Kill_a(l)$.

$$\Re_k^n(i) = \bigotimes_{j=1,i-1}^{\cup} \left[\Re_k(j) - \left(Kill_s(j) \cup \bigotimes_{l=j+1,i-1}^{\cup} Kill_a(l) \right) \right] \quad (4)$$

$$where\ Kill_s = \bigcup_{h=k+1}^{m} \Re_h,\ and\ Kill_a = \bigcup_{h=1}^{m} \Re_h$$

This representation gives us powerful closed forms for array region definitions across the loop. We avoid fixed point iteration methods by hiding the complexity of computing closed forms in USR operations. The USR simplification process will attempt to reduce these expressions to LMADs. However, even when that is not possible, the USR can be used in symbolic comparisons (as in Fig. 3(b)), or to generate efficient run-time assertions (as in Fig. 3(c)) that can be used for run-time optimization and speculative execution.

The reaching definition for the array use $A_{12}(5)$ at statement 11 (Fig. 5(b)) is found inside the loop using δ gates. We use the μ gate to narrow down the block that defined A(5). We intersect the use region $\{5\}$ with $\Re_9^7(i = 11) = [3 : 10]$, and $\Re_{11}^7(i = 11) = [11 : 18]$. We substituted $i \leftarrow 11$, because the *use* happens after the last iteration. The use-def chain is $\{A_{12}, A_7, A_9\}$.

2.2.4 Abstraction of Control: π Nodes

The control dependence predicates corresponding to array *definitions* are embedded in USRs as seen in the γ gate at the definition site of A_6 in Fig. 5. The remainder of this section presents an extension to classic SSA which represents explicitly the control predicates of array *uses*.

Array element $A_{13}(2)$ is used conditionally at statement 13. Based on its range, it could have been defined only by statement 3. In order to prove that it *was* defined at statement 3, we need to have a way to associate the predicate of the use with the predicate of the definition. We create fake definition nodes π to guard the entry to control dependence regions associated with *Then* and *Else* branches: $[A_n, \emptyset] = \pi(A_0, cond)$. This type of gate does not have a correspondent in classic scalar SSA, but in the Program Dependence Web [1]. Their advantage is that they lead to more accurate use-def chains. Their disadvantage is that they create a new SSA name in a context that may contain no array definitions. Such a fake definition A_{13} placed between statement 12 and 13 will force the reaching definition search to collect the conditional $x > 1$ on its way to the possible reaching definition at line 2. This conditional is crucial when the search reaches the γ statement that defines A_6, which contains the same condition. The use-def chain is $\{A_{13}, A_{12}, A_6, A_5, A_4\}$.

2.2.5 Array SSA Construction

Fig. 6 presents the way we create δ, η, γ, μ, and π gates for various program constructs. The associated array regions

	$[A_0, \emptyset] = Undefined$
	$A_1(R_1) = \ldots$
	$[A_2, R_1] = \delta(A_0, [A_1, R_1])$
1 $A(R_1) = \ldots$	$A_3(R_2) = \ldots$
2 $A(R_2) = \ldots$	$[A_4, R_1 \cup R_2] = \delta(A_0, [A_2, R_1 - R_2], [A_3, R_2])$
n $A(R_n) = \ldots$	$A_{2n-1}(R_n) = \ldots$
	$[A_{2n}, \bigcup_{i=1}^{n} R_i] =$
	$\delta(A_0, [A_{2n-2}, \bigcup_{i=1}^{n-1} R_i - R_n], [A_{2n-1}, R_n])$

(a) Straight line code.

	$[A_0, \emptyset] = Undefined$
	$A_1(R_x) = \ldots$
	$[A_2, R_x] = \delta(A_0, [A_1, R_x])$
1 $A(R_x) = \ldots$	**If** (cond)
2 **If** (cond)	$[A_3, \emptyset] = \pi(A_2, cond)$
3 $A(R_y) = \ldots$	$A_4(R_y) = \ldots$
4 **EndIf**	$[A_5, R_y] = \delta(A_3, [A_4, R_y])$
	EndIf
	$[A_6, R_x \cup cond\#R_y] =$
	$\gamma(A_0, [A_2, R_x - cond\#R_y], [A_5, cond\#R_y])$

(b) If block.

	$[A_0, \emptyset] = Undefined$
	Do i=1,n
	$[A_5, \Re_2^5(i) \cup \Re_4^5(i)] =$
	$\mu(A_0, (i = 1, n), [A_2, \Re_2^5(i)], [A_4, \Re_4^5(i)])$
1 **Do** i=1,n	$A_1(R_x(i)) = \ldots$
2 $A(R_x(i)) = \ldots$	$[A_2, R_x(i)] = \delta(A_5, [A_1, R_x])$
	$A_3(R_y(i)) = \ldots$
3 $A(R_y(i)) = \ldots$	$[A_4, R_x(i) \cup R_y(i)] =$
4 **EndDo**	$\delta(A_5, [A_2, R_x(i) - R_y(i)], [A_3, R_y(i)])$
	EndDo
	$[A_6, \otimes_{i=1,n}^{\cup}(\Re_2^5(i) \cup \Re_4^5(i))] =$
	$\eta([A_0, \emptyset], [A_5, \otimes_{i=1,n}^{\cup}(\Re_2^5(i) \cup \Re_4^5(i))])$

(c) Do block. $\Re_k^5(i)$ = definitions from A_k not killed upon entry to iteration i (Equation 4).

Figure 6: Region Array SSA transformation: original code on the left, Region Array SSA code on the right.

are built in a bottom-up traversal of the Control Dependence Graph intraprocedurally, and the Call Graph interprocedurally. At each block level (loop body, *then* branch, *else* branch, subprogram body), we process sub-blocks in program order.

2.2.6 Complexity

The number of ϕ nodes is $O(|E(CFG)| * Variable\ Count$ because every statement (CFG node) could modify all the variables. [23] showed that the number of USR nodes added by each operation (union, intersection, etc) is $O(1)$, and that each USR node consumes $O(1)$ memory. Since the number of USR operations to build any ϕ node is also $O(1)$, the space complexity is thus $O(|E(CFG)| * Variable\ count$.

USR construction optimizations such as symbolic aggregation push the time of each operation to $O(|V(CFG)|)$ in the worst case, though we have observed in practice an amortized $O(1)$. The total compilation time ranges from seconds on a 200 lines code to minutes on a 5000 lines code, using a Pentium IV 2.8GHz PC.

2.3 Reaching Definitions

Algorithm Search(A_u, $\Re_{use}$, *GivenBlock*)
If $A_u \notin$ *GivenBlock* or $\Re_{use} = \emptyset$ **Then Return**
Switch *definition site*(A_u)
Case *original statement*:
$\Re_u^{RD} = \Re_u \cap \Re_{use}$
Case δ, γ, η, θ: $[A_u, \Re_u] = \phi(A_0, [A_1, \Re_1^u], \ldots)$
ForEach $[A_k, \Re_k^u]$
Call Search(A_k, $\Re_{use} \cap \Re_k^u$, *GivenBlock*)
Call Search(A_0, $\Re_{use} - \Re_n$, *GivenBlock*)
Case μ: $[A_u, \Re_u(i)] = \mu(A_0, (i = 1, p), [A_1, \Re_1^u(i)], \ldots)$
ForEach $[A_k, \Re_k^u(i)]$
Call Search(A_k, $\Re_{use}(i) \cap \Re_k^u(i)$, *Block*($A_k$))
Call Search(A_0, $\otimes_{i=1,p}^{\cup}(\Re_{use}(i) - \Re_u(i))$, *GivenBlock*)
Case $\pi(A_0, cond)$
Call Search(A_0, $cond\#\Re_{use}$, *GivenBlock*)
EndIf

Figure 7: Recursive algorithm to find reaching definitions. A_u is an SSA name and $\Re_{use}$ is an array region. Array regions $\Re$ are represented as USRs. They are built using USR operations such as $\cap$, $-$, $\#$, $\otimes^{\cup}$.

Finding the reaching definitions for a given use is required to implement a number of optimizations: constant propagation, array privatization etc. We present here a general algorithm based on Array SSA that finds, for a given SSA name and array region, all the reaching definitions and the corresponding subregions. These subregions can then be used to implement particular optimizations such as constant propagation. Any such optimization can be performed either at compile time, when associated USR comparison can be solved symbolically, or at run-time, when USR comparisons depend on input values.

For each array use $\Re_{Use(A_u)}$ of an SSA name A_u, and for a given block, we want to compute its reaching definition set, $\{[A_1, \Re_1^{RD}], [A_2, \Re_2^{RD}], \ldots, [A_n, \Re_n^{RD}], [A_0, \Re_0^{RD}]\}$, in which $\Re_k^{RD}$ specifies the region of this use defined by A_k and not killed by any other definition before it reaches A_u. $\Re_0^{RD}$ is the region undefined within the given block. Restricting the search to different blocks produces different reaching definition sets. For instance, for a use within a loop, we may be interested in reaching definitions from the same iteration of the loop (block = loop body) as is the case in array privatization. We can also be interested in definitions from all previous iterations of the loop (block = whole loop) or for a whole subroutine (block = routine body). Fig. 7 presents the algorithm for computing reaching definitions. The algorithm is invoked as *Search(A_u, $\Re_{Use(A_u)}$, GivenBlock)*. $\Re_{use}$ is the region whose definition sites we are searching for, A_u is the SSA name of array A at the point at which it is used, and *GivenBlock* is the block that the search is restricted to. The set of memory locations containing undefined data is computed as: $\Re_{use} - \bigcup_{i=1}^{n} \Re_i^{RD}$.

In case the SSA name given as input corresponds to an original statement, the reaching definition set is computed directly by intersecting the region of the definition with the region of the use. If the definition is a δ, γ, η, θ, we perform two operations. First, we find the reaching definitions corresponding to each argument of the ϕ function. Second, we continue the search outside the current block for the region containing undefined values. As shown, the algorithm would make repeated calls with the same arguments to search for undefined memory locations. The actual implementation avoids repetitious work, but we omitted the details here for clarity.

When A_u is inside a loop within the given block, the search will eventually reach the μ node at the loop header. At this point, we first compare $\Re_{use}$ to the arguments of the μ function to find reaching definition from previous iterations of the loop. Second, we continue the search before the loop for the region undefined within the loop.

When the definition site of A_u is a π node, we simply predicate $\Re_{use}$ and continue the search.

The search paths presented in Section 2.2 were obtained using this algorithm.

3. APPLICATION: ARRAY CONSTANT PROPAGATION

We present an *Array Constant Propagation* optimization technique based on our Region Array SSA form. Often programmers encode constants in array variables to set invariant or initial arguments to an algorithm. Analogous to scalar constant propagation, if these constants get propagated, the code may be simplified which may result in (1) speedup or (2) simplification of control and data flow which enable other optimizing transformations, such as automatic parallelization.

We define a *constant region* as the array subregion that contains constant values at a particular use point. We define *array constants* are either (1) integer constants, (2) literal floating point constants, or (3) an expression f(v) which is assigned to an array variable in a loop nest. We name this last class of constants *expression constants*. They are parameterized by the iteration vector of their definition loop nest. Presently, our framework can only propagate expression constants when (1) their definition indexing formula is a linear combination of the iteration vector described by a nonsingular matrix with constant terms and (2) they are used in another loop nest based on linear subscripts (similar to [29]).

3.1 Array Constant Collection

3.1.1 Intraprocedural Collection

The constant regions for SSA name A_u in a subprogram are computed (Fig. 8) by invoking algorithm *Search(A_u, $\top$, WholeSubprogram)*, where $\top$ is a symbolic name for the whole subscript space of array A. This call results in a set of tuples $\{[A_1, \Re_1], [A_2, \Re_2], \ldots, [A_n, \Re_n], [A_0, \Re_0]\}$, such that region $\Re_j$ contains all the subscripts at which the value defined by A_j is available to any use of A_u. The constant regions are those descriptors $\Re_j$ for which the definition site of A_j is an assignment of a constant value. All $\Re_j$ are guaranteed to be mutually disjoint by the logic of the *Search* algorithm and the fact that all the array regions on the right hand side of a δ gate are mutually disjoint. When propagating expression constants, special care is taken at μ nodes to update the iteration vector corresponding to the constant.

Region $\Re_0$ contains all the locations that hold data undefined in the subprogram. The conservative approach is to consider them nonconstant. However, if an interprocedural analysis can infer that certain regions are constant upon entry to the subprogram, the algorithm will also report the corresponding subregions that reach A_u (last loop in Fig. 8).

```
Algorithm CollectIntraProcedural
  Input: A_u as SSA name,
         IncomingConstants as [ℜ_0^1, c_1], ..., [ℜ_0^1, c_m]
  Output: AvailableConstants as [ℜ, c], ...

Call Search(A_u, ⊤, WholeSubprogram)
                 —>  {[A_1, ℜ_1], ..., [A_n, ℜ_n], [A_0, ℜ_0]}

For i = 1, n   // Constants from this subprogram
 If (IsArrayConstant(RightHandSide(Definition(A_i))))
   Report[ℜ_i, RightHandSide(Definition(A_i))]
 EndIf
EndFor

For i = 1, m   // Incoming constants
  Report[ℜ_0 ∩ ℜ_0^i, c_i]
EndFor
```

Figure 8: Algorithm to collect array constant regions available to any use of SSA name A_u in a given subprogram.

```
Algorithm CollectInterProcedural
  Input: Program
  Output: AvailableConstants
change = true
While change Do
 change = false
 ForEach subprogram caller
  ForEach SSA name A_i in caller
   Call CollectIntraProcedural(A_i, Incoming(caller))
  EndForEach
  ForEach callsite Call callee(...)
   OldIncoming(callee) = Incoming(callee)
   Call Adjust(Incoming(callee))
   If (Incoming(callee) ≠ OldIncoming(callee))
    change = true
   EndIf
 EndForEach
EndWhile
```

Figure 9: Algorithm to collect array constant regions across the whole program.

In the example in Fig. 5, the constant regions for SSA name A_{12} are $(\{1\}, 0)$, $((x > 0)\#\{2\}, 1)$, $([3 : 10], 3)$ and $([11 : 18], 4)$. It is interesting to note that the algorithm is control sensitive since the USRs embed control dependence information. We can thus know that location $A_{12}(2)$ holds value 1 *when* $x > 0$ holds true.

3.1.2 Interprocedural Collection

When collecting constants in a single subprogram, we have assumed conservatively that the set of undefined elements $\Re_0$ is not constant. However, arrays containing constant regions may be passed as actual arguments or as global names from a calling context into another subprogram. Even though their corresponding formal names may appear undefined locally, they will contain constant values upon entry to the subprogram. For example, in Fig. 10, during the first traversal of the program, the outcoming set of subroutine *jacld* is collected and translated into subroutine *ssor* at call site *call jacld*. In the next traversal, the value set of A at callsite *call blts* is computed and translated into the incoming value set of subroutine *blts*.

```
Sub ssor
...
Call jacld(A)
Call blts(A)
...
End

Sub jacld(A)
Do i = 1, n
  A(1, i) = 0
EndDo
End

Sub blts(A)
Do i = 1, n
 Do j = 1, 5
   V(1, i) = V(1, i)+
   A(j, i) * V(1 + j, i)
 EndDo
EndDo
End
```

Figure 10: Example from benchmark code Applu (SPEC)

Our interprocedural constant propagation algorithm (Fig. 9) starts by invoking its intraprocedural counterpart on each subprogram. The incoming constant regions are considered empty. This phase computes array constants available at each subprogram call site, which are used to compute actual incoming constant regions for the corresponding callees. Conversely, this may lead to the callee producing more constant values which will be taken into account when re-analyzing the call site. The procedure is repeated until the incoming constants do not change globally.

When a subprogram is called at a single site, its incoming constants are simply a copy of the constants available at the call site. The USRs that describe the associated regions are translated symbolically, as are the *expression constants*.

In general, subprograms are called at several call sites, from different contexts and with different arguments, which could result in conflicting incoming constants. The conservative solution is to consider only the constants that are available at *all* call sites to the given subprogram. Alternatively, we can create several versions of the subprogram (by cloning). In the worst case this could lead to the same code increase as if inlining every subprogram. There are also situations where, although the number of call sites is large, the available constants may be grouped in a much smaller number of equivalence (identity) classes.

3.2 Propagating and Substituting Constants

After the available value sets for array uses are computed, we substitute the uses with constants. Since an array use is often enclosed in a nested loop and it may take different constants at different iterations, loop unrolling may become necessary in order to substitute the use with constants. For an array use, if its value set only has one array constant and its access region is a subset of the constant region, then this use can be substituted with the constant. Otherwise, loop unrolling is applied to expose this array use when the iteration count is a small constant. Constant propagation is followed by aggressive dead code elimination based on simplified control and data dependences.

4. APPLICATION: ARRAY PRIVATIZATION

Privatization is a crucial transformation which removes memory related dependences (anti-, and output dependences) and thus allows the parallelization of loops (among other optimizations). This is achieved by allocating *private* storage for each iteration[3] instead of reusing it across the iterations

[3]In practice, private storage is allocated per thread, and not per iteration.

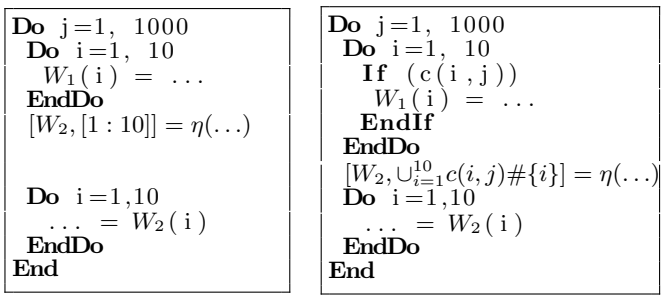

Figure 11: Loop parallelization example. Array *W* must be privatized. Privatization can be proved at compile-time in (a) and only at run-time in (b).

Algorithm IsCovered
Input: $[A_u, \Re_u]$, *Loop*
Output: truth value
Call $Search(A_u, \Re_u, LoopBody)$
—> $\{[A_1, \Re_1], \ldots, [A_n, \Re_n], [A_0, \Re_0]\}$
Return $isEmpty(\Re_0)$

Figure 12: Algorithm to decide whether a *read* is covered by a previous *write* within every iteration of a given loop.

of a loop. To validate such a transformation, the compiler needs to prove that all *read* references within some iteration are covered by previous *write* references to the same memory locations and in the same iteration.

In the example in Fig. 11(a), the parallelization of the outer loop requires the privatization of array *W*. We must prove that all the *reads* in the second inner loop are covered by the *writes* in the first inner loop. Using Array SSA, we can solve this problem by invoking algorithm Search(W_2, [1 : 10], *LoopBody*), which returns $\{[W_1, [1:10]], [W_0, \emptyset]\}$. Since the reference set corresponding to W_0 (undefined) is empty, we conclude that all uses of W_2 are defined within the same iteration (*LoopBody*). Therefore privatization of *W* will remove cross iteration dependences.

In general (Fig. 12), given an array *A* and a loop *Loop*, we invoke algorithm Search(A_u, $\Re_{Use(A_u)}$, *LoopBody*) for each use of SSA name A_u within the loop with footprint $\Re_{Use(A_u)}$. From the result, $\{[A_1, \Re_1], [A_2, \Re_2], \ldots, [A_n, \Re_n], [A_0, \Re_0]\}$, we are only interested in the value of $\Re_0$, which is the set of *reads* not covered by *writes*, or *exposed reads*. The privatization problem can be formulated as *there are no exposed reads*, or $\Re_0 = \emptyset$. In the example in Fig. 11(a), this identity could be proved using symbolic static analysis. However, in the example in Fig. 11(b), the definitions in the first inner loop are controlled by an array of predicates. Depending on their values, there may or may not exist exposed reads. In this case a compile time decision cannot be made. [23] shows how to extract efficient run time tests that prove identities such as $\Re_0 = \emptyset$ at run time. These run-time tests can extract simple conditions based on partial aggregation and invariant hoisting and generally have lower overhead than the element-by-element run time computation of @ arrays proposed by [16].

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

Our goal is not to prove the profitability of constant propagation and array privatization, but to show that they can be implemented easily using Region Array SSA. The purpose of this section is to show that, in the cases where constant propagation and array privatization can improve the performance of an application, the accurate array dataflow information produced by Region Array SSA makes a significant difference.

We have showed how to use the generic *Search* algorithm to implement both constant propagation and array privatization. However, in some cases it is not necessary to perform a full search. When performing array privatization, we only need to compute $\Re_0$, which may be much cheaper than computing the whole list of reaching definitions. When propagating constants, it is not necessary to compute the reaching definition sets that cannot originate from a constant assignment. Our implementation takes into account these optimizations.

We implemented (1) Region Array SSA construction, (2) the reaching definition algorithm and (3) array constant collection in the Polaris research compiler [2], while constant substitution was done by hand because our compiler framework lacked some of the necessary infrastructure. We applied constant propagation to four benchmark codes 173.applu, 048.ora, 107.mgrid (from SPEC) and QCD2 (from PERFECT). The speedups were measured on four different machines (Table 1). The codes were compiled using native compilers at *O3* optimization level (*O4* on the Regatta). 107.mgrid and QCD2 were compiled with *O2* on SGI because the codes compiled with *O3* did not validate).

In subroutine OBSERV in **QCD2**, which takes around 22% execution time, the whole array *epsilo* is initialized with 0 and then six of its elements are reassigned with 1 and -1. The array is used in loop nest OBSERV_do2, where much of the loop body is executed only when *epsilo* takes value 1 or -1. Moreover, the values of *epsilo* are used in the innermost loop body for real computation. From the value set, we discover that the use is all defined with constant 0, 1 and -1. We manually unrolled the loop OBSERV_do2, substituted the array elements with their corresponding values, eliminated *If* branches and dead assignments and succeeded in removing more than 30% of the floating-point multiplications. Additionally, array *ptr* is used in loops HIT_do1 and HIT_do2 after it is initialized with constants in a DATA statement. In subroutine SYSLOP, called from within these two loops, the iteration count of a *While* loop is determined by the values in *ptr*. After propagation, we can fully unroll the loop and eliminate several *If* branches.

In **173.applu**, a portion of arrays *a, b, c, d* is assigned with constant 0.0 in loop JACLD_do1 and JACU_do1. These arrays are only used in BLTS_do1 and BUTS_do1 (Fig. 10), which account for 40% of the execution time. We find that the uses in BLTS_do1 and BUTS_do1 are defined as constant 0.0 in JACLD_do1 and JACU_do1. Loops BLTS_do1111 to BLTS_do1114 and BUTS_do1111 to BUTS_do1114 are unrolled. After unrolling and substitution, 35% of the multiplications are eliminated.

In **048.ora**, array *i1* is initialized with value 6 and then

Machine	Processor	Speed
Intel PC	Pentium 4	2.8 GHz
HP9000/R390	PA-8200	200 MHz
SGI Origin 3800	MIPS R14000	500 MHz
IBM Regatta P690	PowerR4	1.3 GHz

(a)

Program	Intel	HP	IBM	SGI
QCD2	14.0%	17.4%	12.8%	15.5%
173.applu	20.0%	4.6%	16.4%	10.5%
048.ora	1.5%	22.8%	11.9%	20.6%
107.mgrid	12.5%	8.9%	6.4%	12.8%

(b)

Table 1: Constant propagation results. (a) Experimental setup and (b) Speedup.

Program	Coverage		1 proc	2 procs	4 procs
	S	D			
ADM	46%	44%	-3.5%	78%	216%
BDNA	32%	0%	-11.9%	71%	225%
DYFESM	0%	9%	-4.5%	75%	177%
MDG	4%	91%	-2.0%	91%	263%

Table 2: Array privatization results. Speedup after automatic parallelization on 1, 2 and 4 processors on an SGI Altix machine. The coverage column shows the percentage of the execution time that could be parallelized only after array privatization. S = at compile time using static analysis, D = at run time using dynamic analysis.

some of its elements are reassigned with constants -2 and -4 before being used in subroutine ABC, which takes 95% of the execution time. The subroutine body is a *While* loop. The iteration count of the *While* loop is determined by *i1* (there are premature exits). Array *a1* is used in ABC after a portion of it is assigned with floating-point constant values. After array constant propagation, the *While* loop is unrolled and many *If* branches are eliminated.

107.mgrid was used as a motivating example by previous papers on array constant propagation [30, 24]. Array elements A(1) and C(3) are assigned with constant 0.0 at the beginning of the program. They are used in subroutines *RESID* and *PSINV*, which account for 80% of the execution time. After constant propagation, the uses of A(1) and C(3) in multiplications are eliminated.

Table 2 shows the impact of array privatization on the automatic parallelization of major loops in all the applications. In ADM, DYFESM and MDG the privatization problems could be solved only at run time. However, the cost of the run time tests was greatly reduced through partial aggregation of USRs, which led to significant speedups on 2 and 4 processors. The slowdowns on 1 processor are due to the overhead of parallelization and that of run time tests.

6. RELATED WORK

Array Data Flow. There has been extensive research on array dataflow, most of it based on reference set summaries: regular sections (rows, columns or points) [4] linear constraint sets [26, 12, 11, 3, 28, 18, 21, 17, 22, 15, 14, 9, 19, 7, 30, 23], and triplet based [13]. Most of these approaches approximate nonlinear references with linear ones [17, 9].

Nonlinear references are handled as uninterpreted function symbols in [22], using symbolic aggregation operators in [23] and based on nonlinear recurrence analysis in [14]. [8] presents a generic way to find approximative solutions to dataflow problems involving unknowns such as the iteration count of a while statement, but limited to intraprocedural contexts. Conditionals are handled only by some approaches (most relevant are [28, 17, 13, 19, 23]). Extraction of run-time tests for dynamic optimization based on data flow analysis is presented in [19, 23].

Array SSA and its use in constant propagation and parallelization. In the Array SSA form introduced by [16, 24], each array assignment is associated a reference descriptor that stores, for each array element, the iteration in which the reaching definition was executed. Since an array definition may not kill all its old values, a merge function ϕ is inserted after each array definition to distinguish between newly defined and old values. This Array SSA form extends data flow analysis to array element level and treats each array element as a scalar. However, their representation lacks an aggregated descriptor for memory location sets. This makes it in generally impossible to to do array data flow analysis when arrays are defined and used collectively in loops. Constant propagation based on this Array SSA can only propagate constants from array definitions to uses when their subscripts are all constant. [7, 6] independently introduced Array SSA forms for explicitly parallel programs. Their focus is on concurrent execution semantics, e.g. they introduce π gates to account for the out-of-order execution of parallel sections in the same parallel block. Although [6] mentions the benefits of using reference aggregation they do not implement it.

Array constant propagation can be done without using Array SSA [30, 25]. However, we believe that our Array SSA form makes it easier to formulate and solve data flow problems in a uniform way.

Table 3 presents a comparison of some of the most relevant related work to Region Array SSA. The table shows that RA SSA is the only representation of data flow that is explicit (uses SSA numbering), is aggregated, and can be computed efficiently at both compile-time and run-time even in the presence of nonlinear memory reference patterns. The precision of RA SSA is not as good as that of the other two SSA representations because we lack iteration vector information. However, iteration vectors would become very complex in interprocedural contexts (they must include call stack information), whereas USRs represent arbitrarily large interprocedural program contexts in a scalable way.

7. CONCLUSIONS AND FUTURE WORK

We introduced a region based Array SSA providing accurate, interprocedural, control-sensitive *use-def* information at array region level. Furthermore, when the data flow problems cannot be completely solved statically we can continue the process dynamically with minimal overhead. We used RA SSA to write a compact *Reaching Definitions* algorithm that breaks up an array use region into subregions corresponding to the actual definitions that reach it. The implementation of array constant propagation and array privatization shows that our representation is powerful and easy to use.

	RA SSA	Array SSA [24]	Dist. Array SSA [6]	Fuzzy Dataflow [8]	Predicated Dataflow [19]
SSA Form	Yes	Yes	Yes	No	No
Aggregated	Yes	No	No	Yes	Yes
Static/Dynamic	CT/RT	CT/RT	CT/RT	CT	CT/RT
Interprocedural	Yes	No	No	No	Yes
Accuracy	Statement	Operation	Operation x Thread	Operation	Statement
CT Nonlinear	Yes	No	No	Yes	No
RT Nonlinear	Yes	Yes	Yes	No	No

Table 3: Related work on array dataflow and array SSA. CT/RT Nonlinear = able to solve problems involving nonlinear reference patterns at compile time / run time.

8. REFERENCES

[1] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *ACM PLDI*, White Plains, NY, 1990.

[2] W. Blume, *et. al.* Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[3] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM TOPLAS.*, 12(3):341–395, 1990.

[4] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Supercomputing: 1st Int. Conf.*, LNCS **297**, pp. 138–171, Athens, Greece, 1987.

[5] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *IEEE PACT '99*, pp. 245, Washington, DC, 1999.

[6] D. R. Chakrabarti and P. Banerjee. Static single assignment form for message-passing programs. *Int. J. of Parallel Programming*, 29(2):139–184, 2001.

[7] J.-F. Collard. Array SSA for explicitly parallel programs. In *Euro-Par*, 1999.

[8] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *PPOPP '95*, pp. 92–101, New York, NY, USA, 1995. ACM Press.

[9] B. Creusillet and F. Irigoin. Exact vs. approximate array region analyses. In *LCPC*, LNCS **1239**, pp. 86–100, San Jose, CA, 1996.

[10] R. Cytron, *et al* An efficient method of computing static single assignment form. In *16th ACM POPL*, pp. 25–35, Austin, TX., Jan. 1989.

[11] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. of Parallel Programming*, 20(1):23–54, 1991.

[12] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compilers. *Software: Practice & Experience*, 20(2):133–155, 1990.

[13] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing '95*, pp. 47. ACM Press, 1995.

[14] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM TOPLAS*, 18(4):477–518, 1996.

[15] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing '95*, pp. 49, 1995.

[16] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *ACM POPL*, pp. 107–120, 1998.

[17] V. Maslov. Lazy array data-flow dependence analysis. In *ACM POPL*, pp. 311–325, Portland, OR, Jan. 1994.

[18] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array data-flow analysis and its use in array privatization. In *ACM POPL*, pp. 2–15, Charleston, SC, Jan. 1993.

[19] S. Moon, M. W. Hall, and B. R. Murphy. Predicated array data-flow analysis for run-time parallelization. *ACM ICS*, pp. 204–211, Melbourne, Australia, 1988.

[20] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM TOPLAS*, 24(1):65–109, 2002.

[21] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *LCPC 1993*, LNCS **768**, pp. 546–566, Portland, OR.

[22] W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. UMIACS-TR-94-123, Univ. of Maryland, College Park, MD, USA, 1994.

[23] S. Rus, J. Hoeflinger, and L. Rauchwerger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. of Parallel Programming*, 31(3):251–283, 2003.

[24] V. Sarkar and K. Knobe. Enabling sparse constant propagation of array elements via array ssa form. In *SAS*, pp. 33–56, 1998.

[25] N. Schwartz. Sparse constant propagation via memory classification analysis. TR1999-782, Dept. of Compute Science, Courant Institute, NYU, March, 1999.

[26] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of Call statements. In *ACM Symp. on Comp. Constr.*, pp. 175–185, Palo Alto, CA, June 1986.

[27] P. Tu and D. Padua. Gated SSA–based demand-driven symbolic analysis for parallelizing compilers. In *9th ACM ICS*, Barcelona, Spain, pp. 414–423, July 1995.

[28] P. Tu and D. A. Padua. Automatic array privatization. In *LCPC*, LNCS **768** Portland, OR, 1993.

[29] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. Advanced copy propagation for arrays. In *LCTES '03*, pp. 24–33, New York, 2003.

[30] D. Wonnacott. Extending scalar optimizations for arrays. In *LCPC '00*, LNCS **2017**, pp. 97–111.