Scalable Array SSA and Array Data Flow Analysis

Silvius Rus, Guobin He, and Lawrence Rauchwerger

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

Abstract. Static Single Assignment (SSA) has been widely accepted as the intermediate program representation of choice in most modern compilers. It allows for a much more 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 and scalability of previous efforts in array SSA through the addition of a symbolic memory access descriptor for summarizing, in an aggregated and thus scalable manner, the accesses to the elements of an array over large, interprocedural program contexts. Scalar SSA functionality is extended using compact memory descriptors and thus provides a unified, scalable SSA representation. We then show the power of our new representation by using it to implement a basic data flow algorithm, reaching definitions. Finally we use this analysis to propagate array constants and obtain performance improvement (speedups) for real 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) [9] 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.

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 [33]. The first practical 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 [17, 29] to match definitions and uses of partial array regions. However, their approach lacks a symbolic representation for array regions. This led to limited applicability for compile-time analysis and potentially high overhead for derived run-time analysis. Section 5 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 *RT_LMAD* symbolic representation of array regions, which can represent uniformly memory location sets in a compact way for both static and dynamic analysis techniques. We present a compact 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, for which we present whole application improvement. 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). We implemented a program restructuring algorithm that brings programs to structured form (the only control flow constructs are *If* and *Do*). [1]

We believe this paper makes the following original contributions:

1. Array SSA form providing accurate, control-sensitive *use-def* information at array region level.
2. A control sensitive, precise array region *Reaching Definitions* algorithm based on Array SSA.
3. An array constant propagation algorithm based on Array SSA. We obtained up to 20% whole application speedup for four benchmark codes.

2 Array SSA Form

Array SSA is harder to define then its scalar counterpart because array definitions usually kill only a subregion, but not the whole array. This section introduces a general array region representation, presents our proposed Array SSA design and illustrates its use in a *Reaching Definition* algorithm.

2.1 Array Region Representation

Consider the code in Fig. 1(a). The first loop defines array A from 1 to 10, while the second loop uses the values from 1 to 5. It is easy to see that the definition covers the use, which means constant 0 can be propagated from line 2 to line 5.

We chose the Run-time Linear Memory Access Descriptor (*RT_LMAD*) [27] as the representation for memory reference sets. The RT_LMAD can represent uniformly any memory reference pattern in a structured Fortran program. It consists of a triplet-based linear representation, the LMAD [16, 24], and of symbolic operators that describe operations that are not closed on LMADs. The

[1] Our compiler has successfully restructured most of the *PERFECT* and *SPEC* benchmark suites with less than 40% average code expansion.

```
1 Do i = 1, 10
2   A(i) = 0
3 EndDo
4 Do i = 1, 5
5   ... = A(i)
6 EndDo
```

Def = [1 : 10]
Use = [1 : 5]

(a)

```
 1 Do i = 1, 10
 2   If (C(i).GT.0)
 3     A(i) = 0
 4   EndIf
 5 EndDo
 6 Do i = 1, 5
 7   If (C(i).GT.0)
 8     ... = A(i)
 9   EndIf
10 EndDo
```

Def = $\bigcup_{i=1}^{10}(C(i).GT.0)\#[i]$
Use = $\bigcup_{i=1}^{5}(C(i).GT.0)\#[i]$

(b)

```
1 Do i = 1, 10
2   If (C(i).GT.0)
3     A(i) = 0
4   EndIf
5 EndDo
6 Do i = 1, 5
7   ... = A(i)
8 EndDo
```

Def = $\bigcup_{i=1}^{10}(C(i).GT.0)\#[i]$
Use = [1 : 5]

(c)

Fig. 1. Constant propagation scenarios: (a) statically determined and linear reference pattern, (b) statically determined but nonlinear reference pattern, and (c) dynamic and nonlinear reference pattern.

LMAD describes strided, multidimensional reference patterns using the following notation: $start + [stride_1 : span_1, stride_2 : span_2, ...]$ [2]. The definition and use array sections in Fig. 1(a) are expressed as LMADs.

The RT_LMAD can be viewed as a tree having LMADs as leaves, and operands corresponding to nonlinear operations as internal nodes. For the code in Fig. 1(b), the section of the array defined by the first loop cannot be described using an LMAD due to the fact that the *write* to A is controlled by a nonlinear condition, *C(i).GT.0*. Using RT_LMADs, we can express it using two operators: *predication (#)*, and *expansion* ($\bigcup_{i=1}^{n}$). Using symbolic RT_LMAD operations, we can prove that the use in the second loop is covered by the definition in the first loop and can propagate constant 0 from line 3 to line 8. Nonlinear reference patterns are most often due to indirect indexing (index arrays). They cannot generally be represented using triplet based [13, 24] or linear constraint set representations [3, 11, 10, 22, 8, 25]. They can be represented, and operated upon, using RT_LMADs.

Let us consider the case in Fig. 1(c). If we do not know anything about the values in array C, we cannot propagate the values in A from line 3 to line 7. However, using RT_LMADs we can easily generate a run-time test to verify that $Use - Def = \emptyset$. Its result can be used as a guard around a region that is aggressively optimized based on the assumption of constant propagation.

Unlike triplet-based or linear constraint set representations, the RT_LMAD is closed to all the operations required by our analysis: set intersection, difference, union, expansion by an iteration space, translation across subprogram boundaries. This allows us to think in terms of set operations and to hide implementation details in RT_LMAD internals.

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

                 (a)
```

```
     A_0 :   [A_0, ∅] = Undefined
 1   A_1 :   A_1(1)=0
     A_2 :   [A_2, {1}] = δ(A_0, [A_1, {1}])
 2           If (x>0)
     A_3 :      [A_3, ∅] = π([A_2, (x > 0)])
 3   A_4 :      A_4[2] = 0
     A_5 :      [A_5, {2}] = δ(A_3, [A_4, {2}])
 4           EndIf
     A_6 :   [A_6, {1} ∪ (x > 0)#{2}] =
                   γ(A_0, [A_2, {1}], [A_5, (x > 0)#{2})
 5           Do i = 3, 10
     A_7 :      [A_7, [3 : i + 2] ∪ [11 : i + 8]] =
                     μ(A_6, (i = 3, 10), [A_9, [3 : i − 1]], [A_11, [11 : i + 7]])
 6   A_8 :      A_8(i) = 1
     A_9 :      [A_9, {i}] = δ(A_7, [A_8, {i}])
 7                 ···=A_9(···)
 8   A_10 :     A_10(i + 8) = 1
     A_11 :     [A_11, {i, i + 8}] = δ(A_7, [A_9, {i}], [A_10, {i + 8}])
 9           EndDo
     A_12 :  [A_12, {1} ∪ (x > 0)#{2} ∪ [3 : 18]] =
                   η(A_0, [A_6, {1} ∪ (x > 0)#{2}], [A_7, [3 : 18]]
10           ···=A_12(1)
11           ···=A_12(5)
12           If (x>0)
      A_13 :    [A_13, ∅] = π(A_12, (x > 0))
13              ···=A_13(2)
14           Endif

                              (b)
```

Fig. 2. (a) Sample code and (b) Array SSA form

2.2 Array SSA Definition and Construction

The goal of an SSA representation is to expose the basic dataflow relations within a program. This information can then be exploited by analyses that *search* for more complex dataflow relations such as reaching definitions or liveness analysis.

Array SSA Node Placement

In scalar SSA, pseudo statements ϕ are inserted at control flow merge points. These pseudo statements precisely represent how 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 [29] 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 because they do not represent array indices. *In order to provide a useful form of Array SSA, it is necessary to incorporate array region information into the representation.*

Our node placement scheme is essentially the same as in [29]. 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

[2] For 1-dimensional LMADs with a stride of 1, we use notation [*start* : *end*].

definition with that of all other previous definitions. Each node corresponds to a structured block of code. In the example in Fig. 2, 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.

Accounting for Partial Kills: δ Nodes

In the example in Fig. 2, 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 an SSA node that can summarize the array definitions in these two blocks. Such a summary node could tell us that the range of locations defined from statement 2 to statement 9 does not include A(1).

$$[A_n, @A_n] = \delta(A_0, [A_1, @A_1^n], [A_2, @A_2^n], \ldots, [A_m, @A_m^n]) \tag{1}$$

$$where\ @A_n = \bigcup_{k=1}^{m} @A_k^n\ and\ @A_i^n \cap @A_j^n = \emptyset,\ \forall\ 1 \le i, j \le m, i \ne j \tag{2}$$

The function of a δ node is to aggregate the effect of two or more disjoint structured blocks of code. Equations 1 and 2 present its syntax and properties. [3] SSA name A_n merges the array subregions defined by $A_k, k = \overline{1,m}$. The locations defined by A_k which are *not killed before* A_n are represented as $@A_k^n$. They allow us to narrow down the source of the definition of the data within a specific set of memory locations. Given a set $Use(A_n)$ of memory locations read right after A_n, equation 1 tells us that $Use(A_n) \cap @A_k^n$ was defined by A_k. The free term A_0 is used to report locations undefined within the block corresponding to A_n. Fig. 3(a) shows the way we build δ gates for straight line code. Since the RT_LMAD 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. *Essentially, a δ gate translates the problem of disproving the existence of a use-def edge into the problem of proving that an RT_LMAD is empty.*

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 RT_LMAD intersection, by following trace $\{A_{12}, A_0\}$.

Definitions in 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. 2(a), Array A is used at statement 7 in a loop. In case we are

[3] 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 a subroutine on the array.

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.

$$[A_n, @A_n] = \mu(A_0, (i = 1, p), [A_1, @A_1^n], [A_2, @A_2^n], \ldots, [A_m, @A_m^n]) \quad (3)$$

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. 3(c)), and in the order in which they appear in the loop body. Sets $@A_k^n$ are functions of the loop index i. They represent the sets of memory 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. There are two kinds of definitions that will kill it: definitions ($Kill_s$) that will kill it within the same iteration j and definitions ($Kill_a$) that will kill it at iterations from $j + 1$ to $i - 1$.

$$@A_k^n(i) = \bigcup_{j=1}^{i-1} \left[@A_k(j) - \left(Kill_s(j) \cup \bigcup_{l=j+1}^{i-1} Kill_a(l) \right) \right] \quad (4)$$
$$where\ Kill_s = \bigcup_{h=k+1}^{m} @A_h,\ and\ Kill_a = \bigcup_{h=1}^{m} @A_h$$

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 RT_LMAD operations. The RT_LMAD simplification process will attempt to reduce these expressions to LMADs. However, even when that is not possible, the RT_LMADs can be used in compile-time symbolic comparisons (as in Fig. 1(b)), or to generate efficient run-time assertions (as in Fig. 1(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. 2(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 $@A_9^7(i = 11) = [3 : 10]$, and $@A_{11}^7(i = 11) = [11 : 18]$. We substituted $i \leftarrow 11$, because the *use* happens after the last iteration. The use-def chain is $\{A_{12}, A_7, A_9\}$.

Representation of Control: π Nodes

Array element $A_{13}(2)$ is conditionally used 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 $\{A_{13}, A_{12}, A_6, A_5, A_4\}$.

```
1  $A(R_1) = \ldots$
2  $A(R_2) = \ldots$
...
n  $A(R_n) = \ldots$
```

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

(a) Straight line code.

```
1  $A(R_x) = \ldots$
2  If ( cond ) Then
3    $A(R_y) = \ldots$
4  EndIf
```

```
$[A_0, \emptyset] = Undefined$
$A_1(R_x) = \ldots$
$[A_2, R_x] = \delta(A_0, [A_1, R_x])$
If ( cond ) Then
  $[A_3, \emptyset] = \pi(A_2, cond)$
  $A_4(R_y) = \ldots$
  $[A_5, R_y] = \delta(A_3, [A_4, R_y])$
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.

```
1  Do  i =1,n
2    $A(R_x(i)) = \ldots$
3    $A(R_y(i)) = \ldots$
4  EndDo
```

```
$[A_0, \emptyset] = Undefined$
Do  i =1,n
  $[A_5, @A_2^5(i) \cup @A_4^5(i)] = \mu(A_0, (i = 1, n), [A_2, @A_2^5(i)], [A_4, @A_4^5(i)])$
  $A_1(R_x(i)) = \ldots$
  $[A_2, R_x(i)] = \delta(A_5, [A_1, R_x])$
  $A_3(R_y(i)) = \ldots$
  $[A_4, R_x(i) \cup R_y(i)] = \delta(A_5, [A_2, R_x(i) - R_y(i)], [A_3, R_y(i)])$
EndDo
$[A_6, \bigcup_{i=1}^{n} @A_2^5(i) \cup @A_4^5(i)] = \eta([A_0, \emptyset], [A_5, \bigcup_{i=1}^{n} @A_2^5(i) \cup @A_4^5(i)])$
```

(c) Do block. $@A_k^5(i)$ = definitions from A_k not killed just before iteration i (Equation 4).

Fig. 3. Array SSA transformation: original code on the left, Array SSA code on the right. A_k represents an SSA name (and implicitly its definition statement). R represents an array region (possibly conditional) expressed as an RT_LMAD.

Array SSA Construction

Fig. 3 presents the way we create δ, η, γ, μ, and π gates for various program constructs. The associated array regions 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 postdominating order.

```
Algorithm SearchRD($A_n$ , $Use$ , $GivenBlock$)
If $A_n \in GivenBlock$ And $Use \neq \emptyset$ Then
  − If ($DefinitionSite(A_n)$ = $Original\ Statement$) Then
1|      ReachDef($A_n$) = ReachDef($A_n$) $\cup$ ($@A_n \cap Use$)
  − Else
        // $DefinitionSite(A_n)$: $[A_n, @A_n] = \phi(A_{before}, \ldots, [A_1, @A^n_1], [A_2, @A^n_2], \ldots)$
  −     If $\phi = \pi(A_{before}, cond)$ Then
  |         Call SearchRD($A_{before}$ , $cond\#Use$ , $GivenBlock$)
  |     Else
2|          If $\phi = \mu(A_{before}, (i = 1, p), \ldots)$ Then
  |             Call SearchRD($A_{before}$ , $\bigcup_{i=1}^{p}(Use(i) - @A_n(i))$ , $GivenBlock$)
  −             ForEach $[A_k, @A^n_k]$
3|                  Call SearchRD($A_k$ , $Use \cap @A^n_k$ , $Block(A_k)$)
  −             EndForEach
  |         Else
2|              Call SearchRD($A_{before}$ , $Use - @A_n$ , $Block(A_{before})$)
  −             ForEach $[A_k, @A^n_k]$
3|                  Call SearchRD($A_k$ , $Use \cap @A^n_k$ , $GivenBlock$)
  −             EndForEach
            EndIf
        EndIf
    EndIf
EndIf
```

Fig. 4. Recursive algorithm to find reaching definitions. A_n is an SSA name and Use is an array subregion. Region 1 reports final answers, region 2 continues the search for undefined locations, and region 3 narrows the block of a reaching definition.

2.3 Reaching Definitions

For each array use Use of an SSA name A_n, and for a given block, we want to compute its reaching definition set, $\{[A_1, R_1], [A_2, R_1], \ldots, [A_n, R_n], [\perp, R_0]\}$, in which R_k=ReachDef(A_k) specifies that region R_k of this use is defined by A_k, and not killed by any other definition before it reaches A_n. $[\perp, R_0]$ means region R_0 is not defined 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. 4 presents the algorithm for computing reaching definitions. The algorithm is called as *Call SearchRD(A_u, Use, GivenBlock)* 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: $@Use - (ReachDef(A_1) \cup ReachDef(A_2) \cup \ldots \cup ReachDef(A_n))$. The search paths presented in Section2.2 were obtained using this algorithm.

3 Array Constant Propagation

We present an *Array Constant Propagation* optimization technique based on our 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 dependence analysis.

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 [36]). For instance, for the code in Fig. 5, we can propagate the values defined at line 2 to their uses at lines 7 and 9.

```
 1   Do i=1, num
 2    IA(i)=(i*(i-1))/2
 3   EndDo
 4   ...
 5   Do i=1, num
 6     Do j=1, i
 7       ij=IA(i)+j
 8       Do k=1, i
 9         kl=IA(k)+1
10         ...
11       EndDo
12     EndDo
13   EndDo
```

Fig. 5. Example: TRFD.

```
$[A_0, \emptyset] = Undefined$
$A_1(1) = 0$
$[A_2, \{1\}] = \delta(A_0, [A_1, \{1\}])$
If (x>0)
  $A_3 = \pi(A_1, x > 0)$
  $A_4(1) = 1$
  $[A_5, \{1\}] = \delta(A_3, [A_4, \{1\}])$
Else
  $A_6 = \pi(A_2, x \leq 0)$
  $A_7(1) = 1$
  $[A_8, \{1\}] = \delta(A_6, [A_7, \{1\}]$
EndIf
$[A_9, \{1\}] = \gamma(A_0, [A_5, (x > 0)\#\{1\}], [A_8, (x \leq 0)\#\{1\}])$
```

Fig. 6. Array constant propagation example.

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

```
Subroutine jacld(A)
Do k=2, nz-1, 1
 Do j=2, ny-1, 1
  Do i=2, nx-1, 1
   ...
   A(1,2,i,j,k)=0.0
   A(1,4,i,j,k)=0.0
   ...
  EndDo
 EndDo
EndDo
...
End
```

```
Subroutine blts(A)
Do k=2, nz-1, 1
 Do j=2, ny-1, 1
  Do i=2, nx-1, 1
   Do m=1, 5, 1
    Do l=1, 5, 1
     V(m,i,j,k)=V(m,i,j,k)+
     A(m,l,i,j,k)*V(l,i,j,k)
     ...
    EndDo
  ...
EndDo
End
```

Fig. 7. Example extracted from benchmark code Applu (SPEC)

3.1 Array Constant Collection

In Array SSA, the reaching definitions of an array use can be computed by calling algorithm *SearchRD* (Fig. 4). Based on reaching definition set of the use, the constant regions can be computed by simply uniting the regions of the reaching definitions corresponding to assignments of the same constant. To do interprocedural constant propagation, we (1) propagate constant regions into routines at call sites, and (2) compute constant regions for routines and propagate them out at call sites. We iterate over the call graph until there are no changes.

We define a *value tuple* $[Reg, Val]$ as the array subregion Reg where each element stores a copy of Val. Reg is expressed as an RT_LMAD and Val is an array constant. A *value set* is a set of value tuples. We define the following operations on value sets. *Filter* (Equation 5) restricts the value tuple subregions to a given array region. *Intersection* (Equation 6) and *union* (Equation 7) intersect and unite, respectively, subregions across tuples with the same value.

$$Filter(VS, R) = \bigcup_{VT_i \in VS} [Reg(VT_i) \cap R, Val(VT_i)] \tag{5}$$

$$\begin{aligned} VS_1 \cap VS_2 = \{VT \mid \exists\, VT_i \in VS_1, VT_j \in VS_2, s.t. \\ Val(VT) = Val(VT_i) = Val(VT_j) \; and \; Reg(VT) = Reg(VT_i) \cap Reg(VT_j)\} \end{aligned} \tag{6}$$

$$\begin{aligned} VS_1 \cup VS_2 = \{VT \mid \exists\, VT_i \in VS_1, VT_j \in VS_2, s.t. \\ Val(VT) = Val(VT_i) = Val(VT_j) \; and \; Reg(VT) = Reg(VT_i) \cup Reg(VT_j)\} \end{aligned} \tag{7}$$

Fig. 8 shows the algorithm that collects array constants reaching the definition point of SSA name A_k. The algorithm collects constants either directly from the right hand side of assignment statements, or by merging constant value sets corresponding to δ arguments. For loops, constant value sets collected within an iteration are expanded across the whole iteration space. In order to collect all the constants from a routine (needed for interprocedural propagation), we invoke this algorithm with the last SSA name in the routine and its body.

For example in Fig 6, to collect the constants for this code, we call Collect(A_9, 1-13) to compute VS(A_9). 1-13 is the block of statements 1 to 13. Recursively, Collect(A_5, 5-7) will be called to compute VS(A_5) and Collect(A_8, 9-11) called to compute VS(A_8). The first Collect returns $\{[\{1\}, 1]\}$ and the second returns $\{[\{1\}, 1]\}$. Then VS(A_9)= $\{[(x > 0)\#\{1\}, 1]\} \cup \{[(x \leq 0)\#\{1\}, 1]\}$=$\{[\{1\}, 1]\}$.

3.2 Propagating and Substituting Constants

A subroutine may have multiple value sets for an array at its entry. Suppose these value sets are VS_1, $\cdots$, VS_m, then $VS_1 \cap \cdots \cap VS_m$ is the *incoming value set* for the whole subroutine. The incoming value set can be increased by subroutine cloning. Let us assume that for an array use A_u, its reaching definitions are $\{[A_0, R_0], [A_1, R_1], [A_2, R_2], \ldots [A_n, R_n]\}$. Its value sets for this use are $Filter(VS(A_i), R_i)$, where $VS(A_0)$ is the incoming value set for A_u's subroutine. In general, $VS(A_u) = \bigcup_{i=0}^{n} Filter(VS(A_i), R_i)$.

The whole program is traversed in topological order of its call graph. Within a subroutine, statements are visited in lexicographic order. We compute the value

```
Algorithm $Collect(A_n) \rightarrow VS(A_n)$
 VS($A_n$)=$\emptyset$
 Switch ($DefinitionSite(A_n)$)
  Case assignment statement: $A_n(index) = value$
   $VS(A_n) = [\{index\}, value]$
  Case $\mu$ or $\delta$ gate: $[A_n, @A_n] = \phi(A_{before}, \ldots, [A_1, @A_1^n], [A_2, @A_2^n], \ldots, [A_2, @A_m^n])$
   $VS(A_n) = \bigcup_{k=1}^{n} Filter(Collect(A_k), @A_k^n)$
   If ($DefinitionSite(A_n) = \mu(i = 1, p)$ gate ) Then
    VS($A_n$)=$\bigcup_{i=1}^{p} VS(A_n)(i)$
   EndIf
 EndSwitch
 Return VS($A_n$)
End
```

Fig. 8. Array Constant Collection Algorithm.

set for each *use* encountered. Interprocedural translation of *constant regions* and *expression constants* is performed at routine boundaries as needed. For example, in Fig 7, 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*.

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 Implementation and Experimental Results

We implemented (1) Array SSA construction, (2) the reaching definition algorithm and (3) array constant collection in the Polaris research compiler [2]. 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 the native compiler of each machine 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 inner-most loop body for real computation. ¿From the value set, we get to know the use is all defined with constant 0, 1 and -1.

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. (a) Experimental setup and (b) Speedup after array constant propagation.

We unroll loop OBSERV_do2, substitute the array elements with its corresponding values, eliminate *If* branches and dead assignments. More than 30% of the floating-point multiplications in OBSERV_do2 are removed. 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, the loop is fully unrolled and several *If* branches are eliminated.

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. 7), 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 some of its elements are reassigned with constant -2 and -4 before it is 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 [37, 29]. 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.

5 Related Work

Scalar SSA Forms. Introduced by [9], scalar SSA was extended several times by adding new numbering schemes and/or gate information for more accuracy: control information [1, 5], concurrency [19], pointer dereference [18], and interprocedural issues [20].

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 [32, 12, 11, 3, 34, 22, 25, 21, 26, 15, 14, 8, 23, 6, 37, 31, 27, 28], and triplet based [13, 16]. Most of these approaches approximate nonlinear references with linear ones [21, 8, 16].

Nonlinear references are handled as uninterpreted function symbols in [26], using symbolic aggregation operators in [27] and based on nonlinear recurrence analysis in [14, 35, 38, 28]. [7] 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 detailed discussions are in [34, 21, 13, 16, 23, 27]). Extraction of run-time tests for dynamic optimization based on data flow analysis is presented in [23, 27].

Array SSA and its use in constant propagation and parallelization. In the Array SSA form introduced by [17, 29], 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 is 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.

By using summaries of memory reference sets represented as *RT_LMADs* [27], our Array SSA form has larger applicability, it is more scalable, and can produce cheaper run-time evaluation of data flow edges, that can be used to implement run-time optimizations efficiently.

[6] defines an Array SSA form 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.

Array constant propagation can be done without using Array SSA [37, 30]. However, we believe that our Array SSA form makes it easier to formulate and solve data flow problems in a uniform way.

6 Conclusions and Future Work

We introduced a powerful form of Array SSA providing accurate, interprocedural, control-sensitive *use-def* information at array region level. We used Array 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 shows that our representation is powerful and easy to use.

We plan to design other optimization techniques based on Array SSA, such as *array privatization*, *array dependence analysis*, and *array liveness analysis*.

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 *Proc. of the Conf. on Prog. Language Design and Impl.*, pp. 257–271, White Plains, N.Y., June 1990.
2. W. Blume *et al.* Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.
3. M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *Trans. on Program. Lang. Syst.*, 12(3):341–395, 1990.
4. D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Supercomputing*, pp. 138–171, Athens, Greece, 1987.
5. L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 245, Washington, DC, 1999.
6. J.-F. Collard. Array SSA for explicitly parallel programs. In *Proc. of the Int. Euro-Par Conf.*, pp. 383–390, 1999.
7. J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Symp. on Principles and practice of parallel prog.*, pp. 92–101, New York, 1995.
8. B. Creusillet and F. Irigoin. Exact vs. approximate array region analyses. In *Workshop on Lang. and Compilers for Parallel Computing*, pp. 86–100, 1996.
9. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Symp. on Principles of Prog. Lang.*, pp. 25–35, Austin, TX., Jan. 1989.
10. E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *Conf. on Prog. Language Design and Impl.*, pp. 68–77, Albuquerque, N.M., June 1993.
11. P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. of Parallel Prog.*, 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 & Exp.*, 20(2):133–155, Feb. 1990.
13. J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Proc. of Supercomputing* , page 47, 1995.
14. M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Prog. Lang. and Systems*, 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 *Proc. of Supercomputing*, pp. 49, 1995.
16. J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis.* PhD thesis, Univ. of Illinois, Urbana-Champaign, Aug., 1998.
17. K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *Symp. on Principles of Prog. Lang.*, pp. 107–120, 1998.
18. C. Lapkowski and L. J. Hendren. Extended SSA numbering: Introducing SSA properties to language with multi-level pointers. In *Int. Conf. on Compiler Construction*, pp. 128–143, 1998.
19. J. Lee, S. P. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Int. Workshop on Lang. and Compilers for Parallel Computing*, pp. 114–130, London, UK, 1998.
20. S.-W. Liao, A. Diwan, R. P. B. Jr., A. M. Ghuloum, and M. S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Principles Practice of Parallel Prog.*, pp. 37–48, 1999.

21. V. Maslov. Lazy array data-flow dependence analysis. In *Symp. on Principles of Prog. Lang.*, pp. 311–325, Portland, OR, Jan. 1994.
22. D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array data-flow analysis and its use in array privatization. In *Symp. on Principles of Prog. Lang.*, pp. 2–15, Charleston, S.C., Jan. 1993.
23. S. Moon, M. W. Hall, and B. R. Murphy. Predicated array data-flow analysis for run-time parallelization. In *Proc. of the Int. Conf. on Supercomp.*, July 1988.
24. Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Trans. of Prog. Lang. and Systems*, 24(1):65–109, 2002.
25. W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *1993 Workshop on Lang. and Compilers for Parallel Computing*, pp. 546–566, Portland, OR, Aug. 1993.
26. W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. Technical Report UMIACS-TR-94-123, Univ. of Maryland Institute for Advanced Computer Studies, College Park, MD, 1994.
27. S. Rus, J. Hoeflinger, and L. Rauchwerger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. of Parallel Prog.*, 31(3):251–283, 2003.
28. S. Rus, D. Zhang, and L. Rauchwerger. The value evolution graph and its use in memory reference analysis. In *Conf. on Parallel Architecture and Compilation Techniques*, pp. 243–254, Sept. 2004.
29. V. Sarkar and K. Knobe. Enabling sparse constant propagation of array elements via array ssa form. In *Static Analysis Symp.*, pp. 33–56, 1998.
30. N. Schwartz. Sparse constant propagation via memory classification analysis. Technical Report TR1999-782, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, March, 1999.
31. R. Seater and D. Wonnacott. Polynomial time array dataflow analysis. In *Workshop on Lang. and Compilers for Parallel Computing*, pp. 411–426, 2001.
32. R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of Call statements. *ACM Symp. on Compiler Construction*, pp. 175–185, Palo Alto, CA, June 1986.
33. P. Tu and D. Padua. Gated SSA–based demand-driven symbolic analysis for parallelizing compilers. In *Proc. of the Int. Conf. on Supercomputing*, Barcelona, Spain, pp. 414–423, Jan. 1995.
34. P. Tu and D. A. Padua. Automatic array privatization. In *Workshop on Lang. and Compilers for Parallel Computing*, pp. 500–521, Portland, OR., Aug. 1993.
35. R. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *Int. Conf. on Supercomputing*, pp. 106–115, 2004.
36. P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. Advanced copy propagation for arrays. In *the Conf. on Language, compiler, and tool for embedded systems*, pp. 24–33, New York, NY, USA, 2003.
37. D. Wonnacott. Extending scalar optimizations for arrays. In *Int. Workshop on Lang. and Compilers for Parallel Computing*, pp. 97–111, London, UK, 2001.
38. P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *Int. Workshop on Lang. and Compilers for Parallel Computing*, pp. 427–441, Cumberland Falls, KY, 2001.