A Simple Framework to Calculate the Reaching Definition of Array References and Its Use in Subscript Array Analysis*

Yuan Lin and David Padua

Department of Computer Science, University of Illinois at Urbana-Champaign
{yuanlin,padua}@uiuc.edu

Abstract. The property analysis of subscript arrays can be used to facilitate the automatic detection of parallelism in sparse/irregular programs that use indirectly accessed arrays. In order for property analysis to work, array reaching definition information is needed. In this paper, we present a framework to efficiently calculate the array reaching definition. This method is designed to handle the common program patterns in real programs. We use some available techniques as the building components, such as data dependence tests and array summary set representations and operations. Our method is more efficient as well as more flexible than the existing techniques.

1 Introduction:

Program restructuring, including automatic parallelization, has been a useful alternative to manual program optimization for regular, dense computations [2]. However, program restructuring techniques for sparse, irregular problems are not well understood. Compilers usually rely on data dependence tests to enable transformations. The effectiveness of the data dependence test is determined by its ability to accurately analyze array subscripts in loops. However, in sparse/irregular programs, indirect addressing via indices stored in auxiliary arrays[1] is often used. The array subscript can be complex and sometimes its value is difficult or impossible to know at compile time. For this reason, automatic program restructuring is usually believed to be less effective at exploiting implicit parallelism in sparse codes than in their dense counterparts. However, a recent study of a collection of sparse and irregular programs [8] has shown that the use of subscript arrays often follows a few common patterns. Based on these patterns, program restructuring techniques can be extended to handle sparse and irregular programs.

* This work is supported in part by Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; NSF contract MIP-9619351; and a Partnership Award from IBM. This work is not necessarily representative of the positions or policies of the Army or Government.

[1] In this paper, we call the array that appears in the subscript of other arrays the *subscript array* and the indirectly accessed array the *host array*.

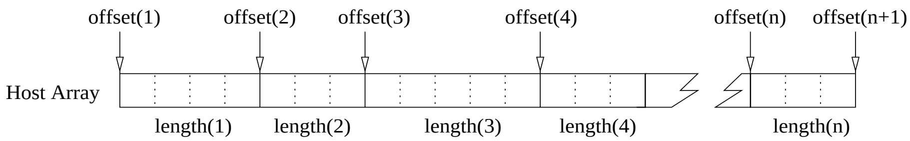
For example, in the sparse matrix computations based on the Compressed Column Storage(CCS) or Compressed Row Storage(CRS) format, the host array is divided into several segments, and subscript arrays are used to store the offset pointer and the length for each segment. Figure 1.(a) shows an example, where `offset()` points to the starting position of each segment whose length is given in `length()`. Figure 1.(b) shows a common loop pattern using the `offset()` and `length()` arrays. The loop traverses the host array segment by segment. Figure 1.(c) shows a common pattern used to define `offset()`. For the code in this example, the compiler first can perform *array property analysis*[8] on the code in Fig.1.(c) to find that `offset()` has a *regular distance* of `length()` and then use this information in the data dependence analysis for array `data()` in Fig.1.(b) to find that loop `do_200` is parallel. Similar examples can be presented for other important compiler transformations, including those for locality enhancement and communication optimizations.

Besides regular distance, the properties of subscript arrays, such as monotonicity and injectivity, as well as having maximum/minimum values and constant values, have been found useful [3]. We can derive the properties and infer the max/min/constant values from where the subscript array is defined and then use the information to analyze the loops that access the host array. We call this process *property analysis of subscript arrays,* or simply *subscript array analysis.*

For the subscript array analysis to be correct, we must make sure that the definitions can reach the use. For the example in Fig.1, we want to know whether the values of all the elements of `offset()` read in statement `s3` are provided by statement `s1` and `s2`. In order to get this information, we need *array reaching definition* analysis.

The array reaching definition problem can be described as, "Given an array read occurrence, find all the assignment statements that may provide the value accessed by the read occurrence." Since different executions of a statement may reference different elements of the same array, the array reaching definition problem may need to identify the instance use/def pairs. The original problem is called the *statement-wise array reaching definition problem* and the latter is called the *instance-wise array reaching definition problem.* Since, in real programs, array regions defined by a single statement tend to have the same property, the statement-wise array reaching definition information is precise enough for our purposes.

In this paper, we present a simple framework to calculate the array reaching definitions at the statement level. Such a simple framework is useful because, in real programs, most array accesses are simple and, in most cases, there is an order between the array accesses. We can use traditional data dependence tests to find this order. The reaching definitions are then computed by using set operations. As a result, our method performs the set operation only once for each occurrence rather than repeatedly on each loop level as done by other methods [5, 6]. Our method, in most cases, is more efficient and more flexible.



(a)

```
   do 200 i = 1, n
      do 300 j = 1, length(i)
s3:      data(offset(i)+j-1) = ..
      end do
   end do
```

(b)

```
s1: offset(1) = 1
    do 100 i = 1, n
s2:   offset(i+1) = offset(i)+length(i)
    end do
```

(c)

Fig. 1. Examples of Offset and Length Subscript Array

2 A Brief Discussion of The New Method

Any framework that computes the array reaching definition has to solve, either explicitly or implicitly, two problems. The first one is Access Order. Suppose two statements S_1 and S_2 write some array elements that are read by another statement S_3. It is necessary to know which statement writes the array elements last. If some writes of S_2 come after some writes of S_1, then those definitions generated by S_1 may be killed by S_2. The second problem is Coverage. Suppose that, in the program region being analyzed, there are several statements writing the array elements read by another statement S_r. If they do not write all the array elements read by S_r, then a definition may come from a statement outside the program region. In our method, we call each read or write of an array in the program text an *occurrence* of that array. We treat as a unit all the array elements that can be accessed by each array occurrence. For example, in the codes in Fig.2.(a), there are four array occurrences: O_1, O_2, O_3 and O_4. The first three are read occurrences and the last one is a write occurrence. When executed, they access $\{a(10:90)\}$, $\{a(1:50)\}$, $\{a(51:100)\}$ and $\{a(1,100)\}$, respectively.

The coverage problem can be solved by using the set subtract operation. For instance, in the previous example, because $(\{[1:100]\} - \{[51:100]\}) - \{[1:50]\} = \phi$, $s2$ and $s3$ provide all the definitions used in $s4$.

The set operation has to be applied by taking into account the access order. The previous example illustrates one of the two cases: the *straight line coverage* case. The other case is the *cross iteration coverage*. In the code in Fig.2(b), an array element written by $s1$ in one iteration will be killed by $s2$ in the following iteration. Since the element is used by $s3$ two iterations later, the definitions in $s1$ do not reach $s3$. In other words, the coverage is performed across the iterations. The access order in this code also is clear. All the elements that can be accessed

```
   do l=10,90
s1:   a(l)=... - O1
   end do
   do i=1, 50
s2:   a(i)=... - O2
   end do
   do j=51, 100
s3:   a(j)=... - O3
   end do
   do k=1, 100
s4:   ...=a(k) - O4
   end do
        (a)
```

```
    do i=1, 100
s1:  a(i)=...     - O1
s2:  a(i-1)=...   - O2
s3:  ...=a(i-2)   - O3
    end do
        (b)
```

```
      do i=1, m
        do j=1, n
s1:       a(1,j-2)=...      - O1
s2:       a(i,j)=...        - O2
s3:       a(i-1,j+1)=...    - O3
s4:       ...=a(i-3,6*i+j)  - O4
        end do
      end do
           (c)
```

Fig. 2. Some code samples

by both $s1$ and $s2$ first are accessed by $s1$ and then by $s2$. Thus, $a[1:98]$ is defined first by $s1$ and then by $s2$.

The basic idea of our method is to (1) identify all the array elements that an occurrence can access, (2) determine the execution order of the occurrences, and (3) use set operations to derive the reaching definitions. A simplified high-level algorithm of our method is shown in Fig.3. Given an array occurrence O of an array $a()$, we use $\mathcal{R}(O)$ to denote the set of the elements of $a()$ that are accessed by O during the execution of the program. The $<_R$ order is explained in the next section.

Input: A read occurrence R, and m write occurrences W_1, W_2, ..., W_m that $W_i <_R R$, $1 \leq i \leq m$. The m write occurrences are sorted such that $W_i <_R W_j$ if $i < j$.

Output: A subset S of the m write occurrences that provide reaching definitions for the read occurrence R, and a set T of the array elements read at R, but whose definitions do not come from any of the m write occurrences.

Procedure:

$S = \phi$, $T = \mathcal{R}(R)$;
for i=m downto 1 do
  if $T \cap \mathcal{R}(W_i) \neq \phi$ then
    $S = S \cup \{W_i\}$;
    $T = T - \mathcal{R}(W_i)$;
  end if
  if $T == \phi$ then return (S, T) ;
end for
return (S, T)

Fig. 3. A simplified high-level algorithm of our method

3 Access Order

3.1 Access Order

The algorithm in Fig.3 requires an order between the write occurrences and the read occurrence, as well as an order between the write occurrences. These orders guarantee that we can obtain the reaching definitions by performing the set operations (intersection, union, and subtraction) on the set of elements these occurrences can access.

Given a read occurrence R of an array and a write occurrence W of the same array, we say that $W <_R R$ if any element in $\mathcal{R}(R) \cap \mathcal{R}(W)$ is written at least once by W before it is first read by R. When $W <_R R$, $\mathcal{R}(R) - \mathcal{R}(W)$ gives all the array elements that are read at R but are not previously defined at W.

Given two write occurrences W_1 and W_2 and one read occurrence R of an array, we say $W_1 <_R W_2$ if

1. $W_1 <_R R$, and
2. $W_2 <_R R$, and
3. for each element in $\mathcal{R}(R) \cap \mathcal{R}(W_1) \cap \mathcal{R}(W_2)$, there is a write of this element by W_1 after any write of this element by W_2 and before any read by R.

When $W_1 <_R W_2$, $(\mathcal{R}(R) - \mathcal{R}(W_1)) \cap \mathcal{R}(W_2)$ gives the set of elements that are defined by W_2 and read by R but not killed by W_1.

3.2 Access Distance

We define the *access time* of an array element by an occurrence O to be the time during the program execution when the statement containing O is executed and the array element is accessed. The access time can be represented by a set of tuples $(iter, pos)$, where $iter$ is the iteration vector of the enclosing loops and pos is the text position of the statement in the program. It is a set because an array element may be accessed multiple times by an occurrence.

Suppose occurrence O_1 and O_2 can access some common array elements. Then, the *access distance* between O_1 and O_2 is the difference between the access times of the common array elements by these two occurrences. The access distance is represented by $(< d_1, d_2, ..., d_n >, (pos1, pos2))$, where d_i represents the difference of the corresponding elements in the iteration vectors. The value of d_i can be either a constant number or a $*$. It is a constant number when the corresponding elements in all the differences of iteration vectors are equal; otherwise, it is a $*$.

Given three occurrence O_1, O_2, and O_3 of an array, we say $O_1 \ll_{O_3} O_2$, if

1. there is a path from statement S_1 which contains O_1 to statement S_3 which contains O_3, and there is a path from statement S_2 which contains O_2 to statement S_3 in the control flow graph, and
2. S_1 dominates S_2, and
3. S_1 and S_2 are not enclosed in any common loop that does not enclose S_3.

Given two write occurrences W_1 and W_2 and one read occurrence R of an array, suppose the access distance between W_1 and R is $D_1 = (diter_1, (pos_{W_1}, pos_R))$ and the access distance between W_2 and R is $D_2 = (diter_2, (pos_{W_2}, pos_R))$. We say that $D_1 \prec_R D_2$ if

1. $diter_1$ is lexicographically less than $diter_2$, or
2. $diter_1 = diter_2$, and they contain no $*$ element, and $W_1 \ll_R W_2$.

We say $D_1 \asymp_R D_2$ if $diter_1 = diter_2$ and they contain no $*$ element, and pos_{W_1} is the same as pos_{W_2}. We say D_1 and D_2 is not comparable when neither $D_1 \prec_R D_2$, $D_2 \prec_R D_1$, nor $D_1 \asymp_R D_2$ holds.

Suppose $d = (diter, (pos_{O_1}, pos_{O_2}))$ is the access distance between O_1 and O_2. We say $0 \prec d$ if $diter$ is lexicographically bigger than vector 0, or $diter$ equals vector 0 and pos_{O_1} precedes pos_{O_2}.

Proposition 1. *Given a write occurrence W and a read occurrence R that can access some common array elements, let d be the access distance between W and R. If $0 < d$, then $W <_R R$.*

Proposition 2. *Given two write occurrences W_1 and W_2 and a read occurrence R that can access some common array elements, let D_1 be the access distance between W_1 and R and D_2 be the access distance between W_2 and R. If $D_1 \prec_R D_2$, then $W_1 <_R W_2$.*

3.3 Sorting the Write Occurrences

Now, given m write occurrences and a read occurrence, we can calculate the access distance from each write occurrence to the read occurrences and then sort the write occurrences according to the access distance. For example, the four occurrences in Fig.2.(c) can be ordered as O_2, O_1, O_3, and O_4.

Because two access distances may not be comparable, the result of the sorting may be a DAG.

3.4 Calculating the Access Distance

The access distance can be calculated in a way similar to that of the dependence distance. We will not present any detailed method here, but rather point out a way to extend existing methods.

Most methods that can calculate the dependence distance require that the subscript expression have only one loop index, such as the occurence O_1 in Fig.4. In real programs, however, many subscript expressions do not fit this constraint, especially when arrays are *reshaped* across procedure boundaries. Fortunately, as some researchers [9, 11] have indicated, the affine subscript expressions often have interleaved access patterns and can be *delinearized* into several independent subscripts with each subscript having the form $c_1(i + c_2) + c_3$. In this case, a dependence test such as [9] can be used to calculate the dependence distance. For example, occurrences O_2, O_3, and O_4 in Fig.4 can be delinearized to $b'(i+8, j)$, $b'(i, j)$, and $b'(i+1, j-1)$, respectively, with b' declared as $b'(20, *)$. Hence, it is easy to see that O_2 is executed before O_4.

```
    do i=1, 10                                      do i=1, 10
       do j=1, 100                                     do j=1, 100
O1:       a(i,j) =                                        a(i,j) = ...
O2:       b(i+20*j+8) = ...                               b'(i+8,j) =
O3:                   = b(i+20*j)    ===>                           = b'(i,j)
O4:       b(i+20*j-19) = ...                              b'(i+1,j-1) = ...
       end do                                           end do
    end do                                           end do
```

Fig. 4. Subscript Delinearization

4 Coverage

4.1 Summary Set Representation

Once the execution order of the write occurrences is clear, the compiler can check the coverage to eliminate from the final reaching definitions the occurrences whose definitions are killed by other occurrences. Our method represents the array regions by set. Several schemes for set representations have been proposed, including convex regions [13], data access descriptors [1, 11] and regular sections [7]. Any one of these can be used in our framework.

Here we use regular sections [6] to illustrate the basic idea. A regular section of an array $A()$ is denoted by $(A(s_1, s_2, ..., s_m), accuracy)$, where m is the dimension of $A()$, $s_i(i = 1, ..., m)$ is a section in the form of $(l : u)$, and l, u are symbolic expressions. $(l : u)$ represents all integer values between l and u, including l and u. The value of *accuracy* can either be MAY or $MUST$. MAY means $A(s_1, s_2, ..., s_m)$ is a superset of the real section, while $MUST$ means it is a subset. The regular section, which represents the array region an occurrence can access in a loop, can be calculated by *expanding* the loop index across the range of the loop [6, 11]. For example, in the loop in Fig.2.(c), the region of array $a()$ written by O_3 is $(a(0 : m-1, 2 : n+1), MUST)$, and it is $(a(4 : m+3, 7 : 6m+n), MAY)$ by O_4.

4.2 Coverage Check

The major operation on the summary set is the *coverage check*, which checks if the set of elements written by an occurrence contains those written by other occurrences and computes difference as the uncovered region. The basic operation in coverage check is the set subtraction operation.

The result of the set subtraction often is an approximation of the real result because the subtraction operation usually is not closed in the set representation. We, therefore, want the uncovered region to be the superset of the real one. To be conservative, we compute the MAY regions for read occurrences and $MUST$ regions for write occurrences.

5 Putting It All Together

The overall algorithm is shown in Fig.5.

Input: A read occurrence R in a program section.
Output: 1. The set S of the write occurrences in the program section that provide the reaching definitions.
2. The array region $RegOut$ that contains elements read in R but may not be defined in the program section.
Procedure:
(1) Find in the program section all the write occurrences on which R is flow dependent. Assume they are W_1, W_2,...,W_n;
(2) Summarize the MAY region $\mathcal{R}(R)$ for the read occurrence R and the $MUST$ region $\mathcal{R}(W_i)$ for each write occurrence W_i, $1 \leq i \leq n$;
(3) Calculate the access distance from each write occurrence to the read occurrence;
(4) Sort the write occurrences according to the access distance and store in T the resulting DAG ;
(5) Let $S = \{W_i | W_i$ is executed before R, and *not* $(W_i <_R R)$ $\}$;
(6) Do the coverage check by using T and R as the input and store the result in $(S', RegOut)$;
(7) $S = S \cup S'$;
(8) return $(S, RegOut)$.

Fig. 5. Putting It All Together

6 Related Work

The existing array dataflow analysis techniques can be categorized according to the granularity of array information they can get, instance-wise or statement-wise.

In order to get the instance-wise reaching definition, the techniques in the first category usually use expensive techniques. Feautrier [4] uses a parametric integer programming method. Maydan provides a faster technique for many common situations. However, when handling multiple writes, both methods can grow exponentially. Pugh and Wonnacott [12] model the problem as verifying Presburger formulas. They extend the Omega test to answer questions in a subclass of Presburger arithmetic. Another method that also uses the Omega test is proposed by Maslov [10]. Although these methods can derive very precise reaching definition information, we have found that, for analysis of the programs we have studied, coarse grain statement-wise information suffices.

Statement-wise techniques [5, 6, 14] in the second category do not analyze array elements individually. Instead, these techniques work on sets of array elements accessed by each statement in the program. These sets usually have a regular shape. Simple set operations, such as union, intersection, and difference, also are performed on such units.

The method discussed in this paper belongs to the statement-wise category. One difference between our method and the others is that we check the coverage directly, while the other methods follow the general procedure proposed in [5]. This procedure calculates the upward exposed use set and downward exposed write set repeatedly while program structures, such as intervals, are being built. Ideally, this approach should be able to get a more precise result than our simple technique. However, on the forms used to represent sets in practice, the set operations usually are not closed and the representation itself often is an approximation of the real array region. Consequently, the advantage of their method is lost in practice. In the cases where the set operation is exact, the occurrences usually have single dependence distance. Our framework is based on this fact and uses a simple method to achieve the same result as the method in [5] in most real cases we have found. Another difference is that most of the other methods were designed to solve a specific array data flow problem, such as array privatization, whereas our method was designed to compute the array reaching definition in general. The array reaching definition not only can be used to test array privatization, but also can be used for many other purposes, such as the subscript array analysis discussed early in this paper.

7 Conclusion

Indirectly accessed arrays are used intensively in sparse and irregular programs and make the programs difficult to parallelize by parallelizing compilers. Our previous study [8] has shown that property analysis of subscript arrays can be used to facilitate the automatic detection of parallelism in this case. In order for array property analysis to work, we need array data flow analysis to derive the array reaching definition information.

In this paper, we presented a simple framework to calculate the statement-wise array reaching definitions. We found that, in real programs, most array occurrences have single or partial single dependence distance. We use this distance to establish an order between these occurrences. Based on this order, the array reaching definition can be computed by using set operations in a simple way. This technique and subscript array analysis will allow many automatic parallelization methods to be extended to handle sparse and irregular programs.

References

1. V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism-enhancing transformations. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, Portland, OR, June 1989.
2. William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng

Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 141–154, Ithaca, New York, August 8–10, 1994. Springer-Verlag.

3. Rudolf Eigenmann and William Blume. An effectiveness study of parallelizing compiler techniques. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, Software, pages II–17–II–25, Boca Raton, FL, August 1991. CRC Press.
4. P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–52, February 1991.
5. Thomas Gross and Peter Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software Practice and Experience*, 20(2):133–155, February 1990.
6. Junjie Gu, Zhiyuan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In Sidney Karin, editor, *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, New York, NY 10036, USA, 1995. ACM Press and IEEE Computer Society Press.
7. P. Havlak. *Interprocedural Symbolic analysis.* PhD thesis, Rice University, May 1994.
8. Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular fortran programs. In *Proc. of 4th Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 41–56. Springer-Verlag, Pittsburgh, PA, 1998.
9. Vadim Maslov. Delinearization: An efficient way to break multiloop dependence equations. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 152–161, New York, NY, July 1992. ACM Press.
10. Vadim Maslov. Lazy array data-flow dependence analysis. In ACM, editor, *Proceedings of 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 311–325, New York, NY, USA, ???? 1994. ACM Press.
11. Yunheung Paek, Jay Hoeflinger, and David Padua. Simplification of array access patterns for compiler optimizations. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 60–71, Montreal, Canada, 17–19 June 1998.
12. William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the Sixth Annual Workshop on rogramming Languages and Compilers for Parallel Computing*, December 93.
13. R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 176–185, Palo Alto, CA, July 1986.
14. Peng Tu and David Padua. Automatic array privatization. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 500–521, Portland, Oregon, August 12–14, 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag.