Analysis of Irregular Single-Indexed Array Accesses and Its Applications in Compiler Optimizations

Yuan Lin and David Padua

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

Abstract. Many compiler techniques require analysis of array subscripts to determine whether a transformation is legal. Traditional methods require the array subscript expressions to be expressed as closed-form expressions of loop indices. Most methods further require the subscript expressions to be linear. However, in sparse/ irregular programs, closed-form expressions of array subscripts are not available. More powerful methods to analyze array subscripts are needed. Array accesses with no closed-form expressions available are called irregular array accesses. In real programs, many irregular array accesses are single-indexed. In this paper, we present techniques to analyze irregular single-indexed array accesses. We show that single-indexed array accesses often have properties that are useful in compiler analysis. We discuss how to use these properties to enhance compiler optimizations. We also demonstrate the application of these techniques in three real-life programs to exploit more implicit parallelism.

1 Introduction

Many compiler techniques, such as loop parallelization and optimizations, need analysis of array subscripts to determine whether a transformation is legal. Traditional methods require the array subscript expressions to be expressed as closed-form expressions of loop indices. Furthermore, most methods require the subscript expression to be linear. However, in many programs, especially sparse/irregular programs, closed-form expressions of array subscripts are not available, and many codes are left unoptimized. Clearly, more powerful methods to analyze array subscripts are needed.

For example, array privatization [9,13,17,19] is an important technique in loop parallelization. An array can be privatized if any array element that is read in one iteration of a loop is always first defined in the same iteration. For example, in the outermost `do k` loop in Fig.1, array $x()$ is first defined in the repeat-until loop, and then is read in the `do j` loop. Any element of $x()$ read in statement (2) is first defined in statement (1) in the same iteration of the `do k` loop. Therefore, array $x()$ can be privatized for the `do k` loop, and the `do k` loop can be parallelized. Current techniques can determine that section $[1 : p]$

A. Watt (Ed.): CC/ETAPS 2000, LNCS 1781, pp. 202–218, 2000.

```
i = 1
do k=1, n
   p = 0
   i = link(i,k)
   repeat
     p = p + 1
     x(p) = y(i)             (1)
     i = link(i,k)
   until ( i == 0 )
   do j=1, p
      z(k,j) = x(j)          (2)
   end do
end do
```

Fig. 1. An example of a loop with an irregular single-indexed array

of array $x()$ is read in the `do j` loop, but they cannot determine that the same section also is written in the repeat-until loop because no closed-form expression for index variable p can be derived. Therefore, they fail to privatize $x()$.

In this paper, we introduce the notion of *irregular single-indexed array access.* An array access is *irregular* in a loop if no closed-form expression for the subscript of the array access in terms of loop indices is available. An array access is *single-indexed* in a loop if the array is always subscripted by the same index variable in the loop. An array access is *irregular single-indexed* in a loop if the array access is both irregular and single-indexed in the loop . For example, the access of array $x()$ in the repeat-until loop in Fig.1 is an irregular single-indexed access.

We chose to investigate irregular single-indexed array accesses for several reasons. First, in the programs we have studied, the single-indexed array accesses often follow a few patterns. These array accesses exhibit properties that are useful in compiler optimizations. Second, many irregular array accesses are single-indexed. Developing analysis methods for irregular single-indexed array accesses is a practical approach toward the analysis of general irregular array accesses, which is believed to be difficult. Third, it is easy to check whether an array access is single-indexed. Efficient algorithms can be developed to "filter" single-indexed array accesses out of general irregular array accesses.

In this paper, we present two important patterns of irregular single-indexed array accesses: *consecutively-written* and *stack-access.* We present the techniques to detect these two patterns and show how to use the properties that irregular single-indexed array accesses have to enhance compiler optimizations.

Throughout the rest of this paper, we will use "single-indexed array access" and "irregular single-indexed array access" interchangeably.

2 Consecutively Written Arrays

An array is *consecutively written* in a loop if, during the execution of the loop, all the elements in a contiguous section of the array are written in a non-increasing or a non-decreasing order. For example, in the repeat-until loop in Fig.1, array element $x(2)$ is not written until $x(1)$ is written, $x(3)$ is not written until $x(2)$ is written, and so on. Array $x()$ is written consecutively in the $1, 2, 3, \ldots$ order in the loop.

To be concise, in this paper, we consider only arrays that are consecutively written in the non-decreasing order. It is trivial to extend the techniques to handle the non-increasing cases as well.

2.1 Algorithm for Detecting Consecutively Written Arrays

In this section, we present an algorithm that tests whether a single-indexed array is consecutively written in a loop.

Since we are dealing with irregular array accesses, we must consider not only do loops, but also other kinds of loops, such as while loops and repeat until loops. In general, we consider natural loops [1]. A natural loop has a single entry node, called the *header*. The header dominates all nodes in the loop. A natural loop can have multiple exits, which are the nodes that lead the control flow to nodes not belonging to the loop.

Before we present the algorithm, we first describe a *bounded depth-first search* (bDFS) method, which is used several times in this paper.

The bDFS is shown in Fig.2. A bDFS does a depth-first search on a graph (V, E), where V is the set of vertices and E is the set of edges in the graph. bDFS uses three auxiliary functions ($f_{bound}()$, $f_{failed}()$, and $f_{proc}()$) to change its behavior during the search. The auxiliary functions are defined before the search starts. $f_{bound}()$ maps V to $(true, false)$. Suppose the current node is n_0, if $f_{bound}(n_0)$ is $true$, then bDFS does not search the nodes adjacent to n_0. The nodes whose $f_{bound}()$ values are $true$ are the boundaries of the search. $f_{failed}()$ also maps V to $(true, false)$. If, for the current node n_0, $f_{failed}(n_0)$ is $true$, then the whole bDFS terminates with a return value of $failed$. The nodes whose $f_{failed}()$ values are $true$ cause an early termination of the bDFS. $f_{proc}()$ does not have a return value; it does predefined computations for the current node.

Now we can show the algorithm that detects consecutively written arrays.

– **Input:** a loop L with header h and a set of exit nodes $(t_1, t_2, ..., t_n)$, a single-indexed array $x()$ in the loop, and the index variable p of $x()$.
– **Output:** answer to the question whether $x()$ is consecutively written in L.
– **Steps:**
  1. Find all the definition statements of p in the loop. If any of them are not of the form "$p = p + 1$", then return NO. Otherwise, put the definition statements in a list lst.

```
$bDFS(u)$
    $visited[u] := true$ ;
    $f_{proc}(u)$ ;
    if ($not\ f_{bound}(u)$) {
        for each adjacent node $v$ of $u$ {
            if ($f_{failed}(v)$)
                return $failed$ ;
            if (($not\ visited[v]$) $and$ ($bDFS(v) == failed$))
                return $failed$ ;
        }
    }
    return $succeeded$ ;
Before the search starts, $visited[]$ is set to $false$ for all nodes.
```

Fig. 2. Bounded depth-first search

2. For each statement n in lst, do a bDFS on the control flow graph from n using the following auxiliary functions:

$$f_{bound}(n) = \begin{cases} true, \text{ if } n \text{ is ``} x() = .. \text{''} \\ false, \text{ otherwise} \end{cases}, f_{failed}(n) = \begin{cases} true, \text{ if } n \text{ is ``} p = p + 1 \text{''} \\ false, \text{ otherwise} \end{cases}$$

$$f_{proc}(n) = NULL$$

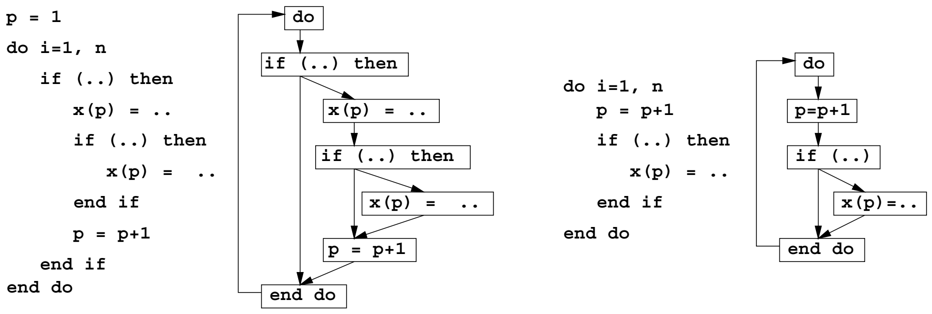
If any of the bDFSs returns $failed$, then return NO. Otherwise, return YES.

The algorithm starts by checking whether the index variable is ever defined in any way other than being increased by 1. If it is, we assume the array is not consecutively written. Step 2 checks whether in the control flow graph there exists a path from one "$p = p + 1$" statement to another "$p = p + 1$" statement[1] and the array $x()$ is not written on the path. If such a path exists, then there may be "holes" in the section where the array is defined and, therefore, the array is not consecutively written in the section. For example, the array $x()$ is consecutively written in Fig. 3.(a), but is not in Fig. 3.(b). The algorithm allows an array element to be written multiple times before the index variable is increased by 1.

2.2 Applications

Dependence Test and Parallelization Suppose a single-indexed write-only array $x()$ with index variable p is consecutively written in a loop, where the assignments of p are of the form "$p = p + 1$". If there does not exist a path from one "$x() = ..$" assignment to another "$x() = ..$" assignment such that the loop header is on the path, but there is no "$p = p + 1$" statement on the path,

[1] These two statements can be the same statement, in which case the path is a circle.



```
p = 1
do i=1, n
   if (..) then
      x(p) = ..
      if (..) then
         x(p) =  ..
      end if
      p = p+1
   end if
end do
```

(a) YES

```
do i=1, n
   p = p+1
   if (..) then
      x(p) = ..
   end if
end do
```

(b) NO

Fig. 3. Consecutively written or not?

```
do i=1, n
   x(p) = ..
   p = p + 1
end do
```

```
do j=1, n
   x(p) = ..     (1)
   p = p + 1
   x(p) = ..     (2)
end do
```

Fig. 4. Data dependence for consecutively written arrays

then $x()$ does not cause any loop-carried dependence in the loop. For example, although array $x()$ is consecutively written in both loops in Fig. 4, there is no dependence between different instances of the access of $x()$ in the loop `do i`, but there is a loop-carried output dependence between statement (1) and (2) in loop `do j`.

This kind of dependence can be detected by using the following method. Here, we assume $x()$ is write-only and found consecutively written with the method described in the previous section.

1. Using the following auxiliary functions, do a bDFS on the control flow graph from the loop header, where the value of $tag1$ is initially set to $null$,

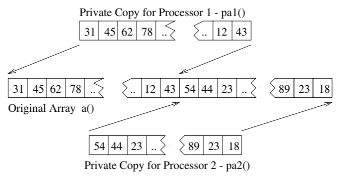
$$f_{bound}(n) = \begin{cases} true, & \text{if } n \text{ is "}x()\text{=.."} \\ & \text{or "}p = p+1\text{"} \\ false, & \text{otherwise} \end{cases}, f_{failed}(n) = \begin{cases} true, & \text{if } tag1 \text{ is } asgn \\ false, & \text{otherwise} \end{cases}$$

$$f_{proc}(n) = \begin{cases} \text{set } tag1 \text{ to } asgn, \text{ if } n \text{ is "}x() = ..\text{"} \\ \text{set } tag1 \text{ to } incr, \text{ if } n \text{ is "}p = p+1\text{" and } tag1 \text{ is } null \\ \text{do nothing, otherwise} \end{cases}$$

   If $tag1$ is $incr$ after the bDFS, then there is no dependence; otherwise, goto step 2.
2. Using the same auxiliary functions as in the previous step, do a bDFS on the reversed control flow graph from the loop header, with $tag1$ being replaced with $tag2$. If, after the bDFS, both $tag1$ and $tag2$ are $asgn$, then there is loop-carried output dependence for $x()$; otherwise, there is no such dependence.

```
Sequential version:
      k = k0
      do i = 1, n
         while (..) do
               a(k) = ..
               k = k+1
         end while
      end do
```

(a)



(b)

```
Parallel version:
    pk(1) = 1
    pk(2) = 1
    parallel do i = 1, n
      // pid is the processor id
      while (..) do
        pa(pk(pid), pid) = ..
        pk(pid) = pk(pid) + 1
      end while
    end do
    parallel section
      do i = 1, pk(1)-1
         a(k0+i-1) = pa(i,1)
      end do
    //
      do i = 1, pk(2)-1
         a(k0+pk(1)+i-2) = pa(i,2)
      end do
    end parallel section
    k = k0+pk(1)+pk(2)-2
```

(c)

Fig. 5. An example of *array splitting and merging*

In order to parallelize the loop with single-indexed and consecutively written arrays, we also need to eliminate the flow dependence caused by the index variable. If the index variable is not used anywhere other than in the array subscript and the increment-by-1 statements, then the *array splitting-and-merging* method [14] can be used to parallelize the enclosing loop.

Array splitting and merging consists of three phases. First, a private copy of the consecutively written array is allocated on each processor. Then, all the processors work on their private copies from position 1 in parallel. After the computation, each processor knows the number of array elements of its private copy that are written in the loop; hence, the starting position in the original array for each processor can be calculated by using the parallel prefix method. Finally, the private copies are copied back (merged) to the original array. Figure 5 shows an example when two processors are used.

Privatization Test As we have illustrated at the beginning of this paper, with consecutively written array analysis, we can extend the privatization test to process irregular single-indexed arrays and more general loops.

Suppose a single-indexed array $x()$ with index variable p is found consecutively written in a loop by using the method described in the previous section, we can use the following two steps to calculate the section of $x()$ written in the loop.

1. Using the following auxiliary functions, do a bDFS on the control flow graph from the loop header h, where the value of $tag1$ is initially set to $null$:

$$f_{bound}(n) = \begin{cases} true, & \text{if } n \text{ is ``}x() = ..\text{'' and } tag1 \text{ is } asgn \\ true, & \text{if } n \text{ is ``}p = p+1\text{'' and } tag1 \text{ is } incr \\ false, & \text{otherwise} \end{cases}$$

$$f_{failed}(n) = \begin{cases} true, & \text{if } n \text{ is ``}x() = ..\text{'' and } tag1 \text{ is } incr \\ true, & \text{if } n \text{ is ``}p = p+1\text{'' and } tag1 \text{ is } asgn \\ false, & \text{otherwise} \end{cases}$$

$$f_{proc}(n) = \begin{cases} \text{set } tag1 \text{ to } asgn, \text{ if } n \text{ is ``}x() = ..\text{'' and } tag1 \text{ is } null \\ \text{set } tag1 \text{ to } incr, \text{ if } n \text{ is ``}p = p+1\text{'' and } tag1 \text{ is } null \\ \text{do nothing, otherwise} \end{cases}$$

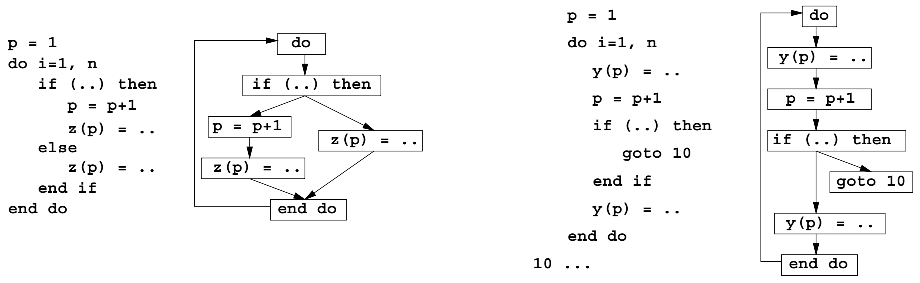
   If the bDFS returns a $failed$, then set $tag1$ to $null$.
2. Using the same auxiliary functions as in the previous step, do a bDFS on the reversed control flow graph from each of the exit nodes (including the loop header), with $tag1$ being replaced with $tag2$. If any of the bDFSs returns a $failed$, then set $tag2$ to $null$.
3. The section where $x()$ is written in the loop is $[lower, upper]$, where

$$lower = \begin{cases} p_0, & \text{if } tag1 \text{ is } asn \\ p_0 + 1, & \text{if } tag1 \text{ is } incr \\ unknown, & \text{otherwise,} \end{cases}, upper = \begin{cases} p, & \text{if } tag2 \text{ is } asn \\ p - 1, & \text{if } tag2 \text{ is } incr \\ unknown, & \text{otherwise.} \end{cases}$$

   and p_0 is the value of p before entering the loop.

For example, the section of $x()$ written in the loop in Fig. 3.(a) is $[1, p-1]$. The section of $z()$ written in the loop in Fig. 6.(a) is $[unknown, p]$, and that of $y()$ in Fig. reffig:section.(b) is $[1, unknown]$.

Index Array Property Analysis The *indirectly accessed array* is another kind of irregular array. An array is indirectly accessed if its subscript is another array, such as $x()$ in statement "$x(ind(i)) = ..$". $x()$ is called the *host array*, and $ind()$ is


```
p = 1
do i=1, n
   if (..) then
      p = p+1
      z(p) = ..
   else
      z(p) = ..
   end if
end do
```

(a) [unknown, p]

```
  p = 1
  do i=1, n
     y(p) = ..
     p = p+1
     if (..) then
        goto 10
     end if
     y(p) = ..
  end do
10 ...
```

(b) [1, unknown]

Fig. 6. The section of consecutively written array

```
do k = 1, n
   q = 0
   do i = 1, p
      if ( x(i) > 0 ) then
         q = q + 1
         ind(q) = i
      end if
   end do
   do j = 1, q
      jj = ind(j)
      z(k,jj) = x(jj) * y(jj)
   end do
end do
```

Fig. 7. An example of a loop with an inner index gathering loop

```
do i = 1,  n
   p = 1
   t(p) = ...
   loop
      p = p + 1
      t(p) = ...
      if (...) then
         loop
         if (p>=1) then
            ... = t(p)
            p = p - 1
         end if
         end loop
      end if
   end loop
end do
```

Fig. 8. An example of an array stack

called the *index array*. Traditional techniques cannot handle indirectly accessed arrays. However, recent studies [5,14] have shown that index arrays often have simple properties, which can be used to produce more accurate analysis of host arrays. An *array property analysis* method has been developed to check whether an index array has any of these key properties [15].

Consecutively written array analysis can be used to find the properties an index array has in the array property analysis. For example, two of the key properties are *injectivity* and *closed-form bounds*. An array section is injective if any two different array elements in the section do not have the same value. An array section has closed-form bounds if the lower bound and upper bound of the values of array elements in the section can be expressed by closed-form expressions. Detecting whether an array section has any of the two properties is difficult, in general. However, in many cases, we only need to check whether the array section is defined in an *index gathering loop*, such as the `do i` loop in Fig.7.

In Fig.7, the indices of the positive elements of array $x()$ are gathered in array $ind()$. After the gathering loop is executed, all the array elements in section $ind[1:q]$ are defined, the values of the array elements in array section $ind[1:q]$ are injective, the lower bound of the values of the array elements in section $ind[1:q]$ is 1, and the upper bound is q.

With this information available at compile-time, the compiler is now able to determine that there is no data dependence in the `do j` loop, and array $ind()$ can be privatized in the `do k` loop. Thus, the compiler can choose either to parallelize the `do k` loop only, parallelize the `do j` loop only, parallelize both, or parallelize the `do k` loop and vectorize the `do j` loop, depending upon the architecture for which the code is generated.

An index gathering loop for an index array has the following characteristics:

1. the loop is a do loop,
2. the index array is single-indexed in the loop,
3. the index array is consecutively written in the loop,
4. the right-hand side of any assignment to the index array is the loop index
5. one assignment to the index array cannot reach another assignment to the index array without first reaching the do loop header.

The fifth condition above ensures that the same loop index value is not assigned twice to the elements of the index array. This condition can be verified using a bDFS. After an index gathering loop, the values assigned to the index array in the loop are injective, and the range of the values assigned is bounded by the range of the do loop bound.

3 Array Stack

The stack is a very basic data structure. Many programs implement stacks using arrays because it is both simple and efficient. We call stacks implemented in arrays *array stacks*. Figure 8 illustrates an array stack. In the body of the `do i` loop, array $t()$ is used as a stack, and variable p is used as the stack pointer which always points to the top of the stack.

3.1 Algorithm for Detecting Array Stacks

In this section, we present an algorithm that checks whether a single-indexed array is used as a stack in a program region. A region [1] is a subset of the control flow graph that includes a header, which dominates all the other nodes in the region.

To be concise, we consider program regions in which the single index variable p is defined only in one of the following three ways:

1. $p := p + 1$,
2. $p := p - 1$, or
3. $p := C_{bottom}$, where C_{bottom} is an invariant in the program region.

We check whether a single-indexed array is used as a stack in a region by checking whether the statements involved in the array operations appear in some particular orders. These orders are shown in Table 3.1.

The left column and the top row in Table 3.1 give the statements to be checked. If there is a path in the control flow graph from a statement of the form shown in the left column of the table to a statement of the form shown in the top row, then the statement in the corresponding central entry of the table must be on the path. For example, if there is a path from a statement "$x(p) = ..$" to another statement "$x(p) = ..$", then before the control flow researches the

Table 1. Order for access of array stacks

	$p = p + 1$	$p = p - 1$	$x(p) = ..$	$.. = x(p)$	$p = C_{bottom}$
$p = p + 1$	$x(p) = ..$	$.. = x(p)$	-	$x(p) = ..$	-
$p = p - 1$	-	$.. = x(p)$	$p = p{+}1$	G	-
$x(p) = ..$	-	$.. = x(p)$	$p = p + 1$	-	-
$.. = x(p)$	$p = p - 1$	-	$p = p + 1$	$p = p + 1$	-

second "$x(p) = ..$" statement, it must first reach a "$p = p + 1$" statement. A '-' in a table entry means there is no restriction on what kind of statement must be on the path. The 'G' represents an *if* statement that is "*if* $(p \geq C_{bottom})$ *then*".

Intuitively, we want to ensure that for an array stack $x()$ with index p, (1) p is first set to C_{bottom} before it is modified or used in the subscript of $x()$, (2) the value of p never goes below C_{bottom}, and (3) the access of elements of $x()$ follows the "last-written-first-read" pattern.

Table 3.1 can be simplified to Table 3.1. Any path originating from a node n of the forms in the left column of Table 3.1 must first reach any node of the forms in $S_{bound}(n)$ before it reaches any node of the forms in $S_{failed}(n)$.

Next, we present the algorithm to detect array stacks.

- **Input:** a program region R with header h, a single-indexed array $x()$ in the region, and the index variable p of $x()$.
- **Output:** answer to the question whether $x()$ is used as a stack in R. And, if the answer is YES, the minimum value C_{bottom} the index variable p can have in the region.
- **Steps:**
  1. Find all the definition statements of p in R. If any are not of a form in the set $\{p = p{+}1, p = p{-}1, p = C_{bottom}\}$ (if there are multiple "$p = C_{bottom}$" statements, the C_{bottom} must be the same), where C_{bottom} is invariant in R, then return NO. Otherwise, put the definition statements in a list lst. If there are no statements of the form "$p = C_{bottom}$", then find all if statements of the form "*if* $(p \geq C_{if})$ *then*". If all C_{if}'s are the same, set C_{bottom} to C_{if}; otherwise, return NO. If no such if statement is found, set C_{bottom} to $unknown$.

Table 2. Simplified order for array stacks

n	$S_{bound}(n)$	$S_{failed}(n)$
$p = p + 1$	$\{x(p) = .., p = C_{bottom}\}$	$\{p = p + 1, p = p - 1, .. = x(p)\}$
$p = p - 1$	$\{p = p + 1, \text{G}, p = C_{bottom}\}$	$\{p = p - 1, x(p) = .., .. = x(p)\}$
$x(p) = ..$	$\{p = p + 1, .. = x(p), p = C_{bottom}\}$	$\{p = p - 1, x(p) = ..\}$
$.. = x(p)$	$\{p = p - 1, p = C_{bottom}\}$	$\{p = p + 1, x(p) = .., .. = x(p)\}$

2. Find all the "$x(p) = ..$" and "$.. = x(p)$" statements in R, and add them to lst.
3. For each statement m in lst, do a bDFS on the control flow graph from this statement using the following auxiliary functions:

$$f_{bound}(n) = \begin{cases} true & n \in S_{bound}(m) \\ false & \text{otherwise} \end{cases}, f_{failed}(n) = \begin{cases} true & n \in S_{failed}(m) \\ false & \text{otherwise} \end{cases}$$

$$f_{proc}(n) = NULL$$

If any of the bDFSs returns a $failed$, then return NO. Otherwise, return YES and C_{bottom}.

3.2 Applications

Run-Time Array Bound Checking Elimination Run-time array bound checking is used to detect array bound violations. The compiler inserts bound checking codes for array references. At run-time, an error is reported if an array subscript expression equals a value that is not within the declared bounds of the array. Some languages, such as Pascal, Ada and Java, mandate array bound checking. Array bound checking also is useful in testing and debugging programs written in other languages. Since most references in computationally intense loops are to arrays, these checks cause a significant amount of overhead.

When an array is used as a stack in a program region, the amount of array bound checking for the stack array can be reduced by 50%. Only the upper bound checkings are preserved. The lower bound checking is performed only once before the header of the program region. Elimination of unnecessary array bound checking also has been studied by Markstein et al [16], Gupta [11], and Kolte and Wolfe [12]. Gupta and Spezialetti [18] proposed a method to find monotonically increasing/decreasing index variables, which also can be used to eliminate the checking by half. But, their method cannot handle array stacks, which are more irregular.

Privatization Test Array stack analysis also can improve the precision of array privatization tests. Here, we consider the loop body as a program region. When an array is used as a stack in the body of a loop, the array elements are always defined ("pushed") before being used ("popped") in the region. If C_{bottom} is a loop invariant, then different iterations of the loop will reuse the same array elements, and the value of the array elements never flow from one iteration to the other. Therefore, array stacks in a loop body can be privatized. For example, the array stack $t()$ in Fig.8 can be privatized in the outermost the `do i` loop.

Loop Interchanging Loop interchanging [2,20] is the single most important loop restructuring transformation. It has been used to find vectorizable loops, to change the granularity of parallelism, and to improve memory locality. Loop interchanging changes the order of nested loops. It is not always legal to perform loop interchanging since data dependence cannot be violated. Data dependence tests must be performed before loop interchanging.

Traditionally, loop interchanging is not possible when array stacks are present because current data dependence tests cannot handle irregular arrays. However, as in the privatization test, array stacks cause no loop carried dependences. If the index variables of array stacks are not used in any statements other than stack access statements, then the data dependence test can safely assume no dependence between the stack access statements. The loop interchanging test then can ignore the presence of array stacks and use traditional methods to test other arrays. By using array stack analysis, we have extended the application domain of loop interchanging.

4 Related Work

There are two closely related studies done by two groups of researchers. M. Wolfe [21] and M. Gerlek, E. Stoltz, and M. Wolfe [10] presented an algorithm to recognize and classify *sequence variables* in a loop. Different kinds of sequence variables are linear induction variables, periodic, polynomial, geometric, monotonic, and wrap-around variables. Their algorithm is based on a demand-driven representation of the Static Single Assignment form [7,6]. The sequence variables can be detected and classified in a unified way by finding strongly connected components of the associated SSA graph.

R. Gupta and M. Spezialetti [18] have extended the traditional data-flow approach to detect "monotonic" statements. A statement is monotonic in a loop if, during the execution of the loop, the statement assigns a monotonically increasing or decreasing sequence of values to a variable. They also show the application of their analysis in run-time array bound checking, dependence analysis, and run-time detection of access anomalies.

The major difference between both these studies and ours is that we focus on arrays while they focus on index variables. While both of their methods can recognize the index variable for a consecutively written array as a monotonic variable, if the array is defined in more than one statement, then none of them can detect whether the array itself is consecutively written. For example, Gerlek, Stoltz and Wolfe's method can find that the two instances of variable k in statements (1) and (2) in Fig.9 have a strictly increasing sequence of values. Gupta and Spezialetti's method can classify statements (1) and (2) as monotonic. However, neither can determine whether the access pattern of the array $x()$ is consecutively written. As for array stack analysis, as the index variable does not have a distinguishable sequence of values, both Gerlek, Stoltz and Wolfe's method and Gupta and Spezialetti's method treat the index variable as a generally irregular variable. Without taking the arrays into the account in their analysis, they can do little in detecting array stacks.

The authors believe it is often important to consider both index variables and arrays. While both of the two other methods can recognize a wide class of scalar variables beyond the variable used as the subscript of single-indexed

```
do i = 1, n
   if ( .. ) then
      x(k) = ..
      k = k + 1                    (1)
   else if ( .. ) then
      x(k) = ..
      k = k + 1                    (2)
   end if
end do
```

Fig. 9. Both array $x()$ and index k should be analyzed to know that $x()$ is consecutively written.

arrays in our method, they are not necessarily more powerful in analyzing the access pattern of the arrays.

5 Case Studies

In this section, we show how consecutively written array analysis and array stack analysis can be used to enhance the automatic parallelism detection in three real-world programs.

These three programs are summarized in Table 5. Column 3 in Table 5 shows the loops that can be parallelized only after the techniques presented in this paper have been used to analyze the arrays shown in Column 4. Figure 10 shows the difference in speedups when these loops are parallelized. We compare the speedups of the programs generated by our Polaris parallelizing compiler, with and without single-indexed array analysis, and the programs compiled using the automatic parallelizer provided by SGI. The experiments were performed on an SGI Origin2000 machine with 56 195MHz R10000 processors (32KB instruction case, 32KB data cache, 4MB secondary unified level cache) and 14GB memory running IRIX64 6.5. One to thirty-two processors are used for BDNA and TREE. One to eight processors are used for P3M. “APO” means using the “-apo” option when compiling the programs. This option invokes the SGI automatic parallelizer. “Polaris without SIA” means using the Polaris compiler without the single-indexed array analysis. “Polaris with SIA” means using the Polaris compiler with the single-indexed array analysis. As we have not yet implemented array stack analysis in our Polaris compiler, for TREE we show the result of manual parallelization. For all three codes, the speedups of the versions in which the single-index array analysis had been used are much better than those of the other versions.

5.1 BDNA

BDNA is a molecular dynamics simulation code from the PERFECT benchmark suite [8].

Table 3. Three real-life programs

Program Name	Lines of Codes	Major Loops	Single-indexed Arrays	% of Exe. Time
BDNA	4000	$actfor\_do\_240$	$xdt()$	31%
P3M	2500	pp_do_100	$ind0()$, $jpr()$	74%
		$subpp\_do\_100$	$ind0()$, $jpr()$	14%
TREE	1600	$accel\_do10$	$stack()$	70%

The `do 240` loop in subroutine $ACTFOR$ is a loop that computes the interaction of biomolecules in water. It occupies about 31% of total computation time. The main structure of this loop is outlined in Fig.11

Consecutively written array analysis is used in the `do j2` loop to find that elements in $[1, k]$ of $ind()$ are written in this loop. Furthermore, this loop is recognized as an index gathering loop; thus, the values of the elements in $ind[1, k]$ defined in this loop are bounded by $[1, i-1]$. This information is used to privatize array $ind()$ and $xdt()$ in the `do i` loop, which is then determined to be parallel.

5.2 P3M

P3M is an N-body code that uses the particle-mesh method. This code is from NCSA.

Most of the computation time (about 88% after using vendor provided FFT library) is spent in subroutine pp and $subpp$, whose structures are very similar. The core is a three-perfect-loop nest, which can be parallelized. Before parallelization, several single-indexed arrays in the loop must be privatized. The outline of the core loops is shown in Fig. 12. The simplified loop pattern is similar to that in Fig. 11. The difference is that both $x()$ and $ind()$ are consecutively written arrays here.

5.3 Barnes & Hut TREE Code

The **TREE** code [4] is a program that implements the hierarchical N-body method for simulating the evolution of collisionless systems [3].

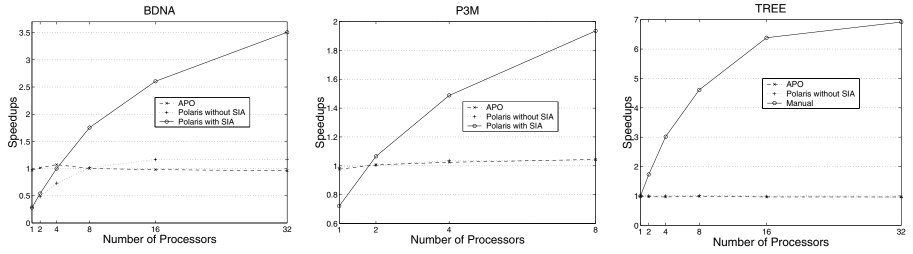


Fig. 10. Comparison of Speedups

```
do i = 2, n
   do j1 = 1, i-1
      xdt(j1) = ..
   end do
   k = 0
   do j2 = 1, i-1
      if (..) then
         k = k+1
         ind(k) = j2
      end if
   end do
   do j3 = 1, k
      .. = xdt(ind(j3))
   end do
end do
```

Fig. 11. BDNA

```
do i1 = 1, n
do i2 = 1, n
do i3 = 1, n
   p = 0
   repeat
     p = p+1
     x(p) = ..
   until (..)
   k = 0
   do j2 = 1, p
      if (..) then
         k = k+1
         ind(k) = j2
      end if
   end do
   do j3 = 1, k
      .. = x(ind(j3))
   end do
end do
end do
end do
```

Fig. 12. P3M

```
sptr = 1
stack(sptr) = root
while (sptr .gt. 0) do
   q = stack(sptr)
   sptr = sptr - 1
   if (q is a body) then
      process body-body interaction
   elseif (q is far enough from p) then
      process body-cell interaction
   else
      do k = 1, nsubc
         if (subp(q,k) .ne. null) then
            sptr = sptr + 1
            stack(sptr) = subp(q,k)
         end if
      end do
   end if
end while
```

Fig. 13. TREE

The core of the program is a time-centered leap-frog loop, which is inherently sequential. At each time step, it computes the force on each body and updates the velocities and positions. About 70% of the program execution time is spent in the force calculation loop. Each iteration of the force calculation loop computes the gravitational force on a single body *p* using a tree walk method that is illustrated in Fig.13.

In the tree walk code, single-indexed array *stack* is used as a stack to store tree nodes yet to be visited. Variable *sptr* is used as the stack pointer. As discussed in Sect.3.2, array *stack* can be privatized for the force calculation loop. As there is no other data dependence in the loop, the loop can be parallelized (i.e., the force calculation of the n bodies can be performed in parallel).

6 Conclusion

In this paper, we introduced the notion of irregular single-indexed array access. We described two common patterns of irregular single-indexed array accesses (i.e., consecutively written and stack access) and presented simple and intuitive algorithms to detect these two patterns. More importantly, we showed that array accesses following these two patterns exhibit very important properties. We demonstrated how to use these properties to enhance a variety of compiler analysis and optimization techniques, such as the dependence test, privatization test, array property analysis, loop interchanging, and array bound checking. In the case study, we showed that, for three real-life programs, the speedups of the parallelized versions generated by the Polaris compiler with single-index array access analysis are much better than those of other versions.

Acknowledgments

We wish to thank the anonymous referees for their many useful comments and suggestions. This work is supported in part by Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; NSF contract NSF ACI98-70687; and a Partnership Award from IBM. This work is not necessarily representative of the positions or policies of the Army or Government. This work also was partially supported by National Computational Science Alliance and utilized the NCSA SGI Origin2000.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986. 204, 210
2. John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 233–246, New York, NY 10036, USA, 1984. ACM Press, ACM Press. 212
3. J. Barnes and P. Hut. A hierarchical o(nlogn) force calculation algorithm. *Nature*, 324(4):446–449, 1986. 215
4. Joshua E. Barnes. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/. Technical report. 215
5. W. Blume and R. Eigenmann. An overview of symbolic analysis techniques needed for the effective parallelization of the perfect benchmarks. In K. C. Tai, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 2: Software*, pages 233–238, Boca Raton, FL, USA, August 1994. CRC Press. 209
6. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of 16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989. 213
7. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transaction on Programming Languages and Systems*, 13(4):451–490, October 1991. 213
8. Mike Berry et.al. The perfect club benchmarks: Effective performance evaluation of supecomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989. 214
9. P. Feautrier. Array expansion. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988. 202
10. Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995. 213
11. Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, March–December 1993. 212
12. Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. *ACM SIGPLAN Notices PLDI 1995*, 30(6):270–278, June 1995. 212
13. Z. Li. Array privatization for parallel execution of loops. In *Proceedings of 1992 International Conference on Supercomputing, July 19–23, 1992, Washington, DC*, pages 313–322, New York, NY 10036, USA, 1992. ACM Press. 202

14. Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular fortran programs. In *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 41–56. Springer-Verlag, Pittsburgh, PA, 1998. 207, 209
15. Y. Lin and D. Padua. Demand-driven interprocedural array property analysis. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999. 209
16. Victoria Markstein, John Cocke, and Peter Markstein. Optimization of range checking. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 114–119. ACM, ACM, 1982. 212
17. Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 2–15, New York, NY, USA, 1993. ACM Press. 202
18. Madalene Spezialetti and Rajiv Gupta. Loop monotonic statements. *IEEE Transactions on Software Engineering*, 21(6):497–505, June 1995. 212, 213
19. 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. 202
20. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982. 212
21. Michael Wolfe. Beyond induction variables. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 162–174, New York, NY, July 1992. ACM Press. 213