Analysis of Irregular Single-indexed Arrays 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 usually 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 even require the subscript expressions 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. More powerful methods to analyze array subscripts are desired. Arrays with no closed-form expressions available are called irregular arrays. In real programs, many irregular arrays are single-indexed (i.e. the arrays are always subscripted by a single index variable). In this paper, we presented a technique to analyze the irregular single-indexed arrays. We showed that single-indexed arrays often have very good properties that can be used in compiler analysis. We discussed how to use these properties to enhance loop parallelization, loop interchanging, and array bounds-checking elimination. We also demonstrated the application of these techniques in three real-life programs to exploit more implicit parallelism.

1 Introduction

Many compiler techniques, such as parallelization, loop interchanging and array bounds-checking elimination, usually 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 even 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 desired.

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 the loop is always first defined in the same iteration. For example, in the outermost loop *loop_do_k* in Fig.1, array $x()$ is first defined in the repeat-until loop, and then is read in *loop_do_j*. It is easy to see that any element of $x()$ read in statement (2) is first defined in statement (1) in the same iteration of *loop_do_k*. Therefore, array $x()$ can be privatized for *loop_do_k*. Because there is no dependence, *loop_do_k* can be parallelized. In this example, no closed-form

```
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 irreular single-indexed array

expression for index variable p can be derived. Current privatization tests can handle only do loops and require a closed-form expression of the array subscripts in order to compute the section of array elements read or written in the loop. In this example, these techniques can determine that section $[1 : p]$ of array $x()$ is read in the *loop_do_j*, but they cannot determine that the same section is also written in the repeat-until loop. Therefore, they would fail to privatize $x()$.

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

We chose to investigate irregular single-indexed arrays for several reasons. First, the use of single-indexed arrays often follow a few patterns. Single-index arrays that follow these patterns exhibit very good properties that can be used in compiler optimizations. Second, many irregular arrays are single-indexed. Developing analysis methods for irregular single-indexed arrays is a practical approach toward the analysis of general irregular arrays, which is believed to be difficult. Third, it is easy to check whether an array is single-indexed. Efficient algorithms can be developed to "filter" single-indexed arrays out of general irregular arrays.

In this paper, we present two important access patterns of irregular single-indexed arrays: *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 arrays in these patterns have to enhance compiler optimizations.

Throughout the test of this paper, "single-indexed array" means "irregular single-indexed array".

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 one by one in an increasing or 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. That is, $x()$ is consecutively written in the 1,2,3,... order.

We describe how to detect consecutively written arrays in Sect.2.1, and we show how to use the properties of consecutively written arrays in compiler optimizations in Sect.2.2. To be concise, in this paper, we consider only arrays that are consecutively written in increasing order. It is trivial to extend the techniques we present to handle decreasing 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, and if so, gives the section where the array elements are written.

Since we are dealing with irregular arrays, 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 nature 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. Like the standard DFS, 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. It uses three help functions to change its behavior during the search. These three functions (i.e., $f_{bound}()$, $f_{failed}()$, and $f_{proc}()$) are defined before the search starts. $f_{bound}()$ is a $V \rightarrow (true, false)$ function. Suppose the current node is n_0 during the search. If $f_{bound}(n_0)$ is $true$, then bDFS does not search the nodes adjacent to the current node n_0. The nodes whose $f_{bound}()$ values are $true$ are the boundaries of the search. $f_{failed}()$ also is a $V \rightarrow (true, false)$ function. 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 to detect 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. And if the answer is *YES*, the section where $x()$ is written in L.
- **Steps:**

```
    bDFS(u)
1       visited[u] := true ;
2       f_proc(u) ;
3       if (f_bound(u) = false) then
4           for each adjacent node v of u
5               if (f_failed(v) = true) then return failed ;
6               if (visited[v] = false) then
7                   result := bDFS(v) ;
8                   if (result = failed) then return failed ;
9       return succeeded ;
    Before the search starts, visited[] is set to false for all nodes.
```

Fig. 2. Bounded depth-first search

1. Find all the definition statements of p in the loop. If any are not of the form "$p = p + 1$", then return NO. Otherwise, put the definition statements in a list lst.
2. For each statement n in lst, do a bDFS on the control flow graph from n using the following help functions:

$$f_{bound}(n) = \begin{cases} true & \text{if } n \text{ is an assignment statement for } x() \\ false & \text{otherwise} \end{cases}$$

$$f_{failed}(n) = \begin{cases} true & \text{if } n \text{ is "} p = p + 1\text{"} \\ false & \text{otherwise} \end{cases}$$

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

   If any of the bDFSs returns a $failed$, then return NO.
3. Using the following help functions, do a bDFS on the control flow graph from the loop header h, where the value of $tag1$ is initially set to 0:

$$f_{bound}(n) = \begin{cases} true & \text{if } n \text{ is an assignment statement for } x() \text{ and } tag1 \text{ is } 1 \\ true & \text{if } n \text{ is "} p = p + 1\text{" and } tag1 \text{ is } 2 \\ false & \text{otherwise} \end{cases}$$

$$f_{failed}(n) = \begin{cases} true & \text{if } n \text{ is an assignment statement for } x() \text{ and } tag1 \text{ is } 2 \\ true & \text{if } n \text{ is "} p = p + 1\text{" and } tag1 \text{ is } 1 \\ false & \text{otherwise} \end{cases}$$

$$f_{proc}(n) = \begin{cases} \text{set } tag1 \text{ to } 1 \text{ if } n \text{ is an assignment statement for } x() \text{ and } tag1 \text{ is } 0 \\ \text{set } tag1 \text{ to } 2 \text{ if } n \text{ is "} p = p + 1\text{" and } tag1 \text{ is } 0 \\ \text{do nothing, otherwise} \end{cases}$$

   If the bDFS returns a $failed$, then return NO.
4. Using the same help functions as in the previous step, do a bDFS on the reversed control flow graph from each of the exit nodes, with $tag1$ being replaced with $tag2$. If any of the bDFSs returns a $failed$, then set $tag2$ to 0, and go to step 5.

5. Now, we know $x()$ is consecutively written in the loop. The lower bound of the region is p_0 if $tag1$ is 1, or p_0+1 if $tag1$ is 2, where p_0 is the value of p before entering the loop. The upper bound of the region is p if $tag2$ is 1, or $p-1$ if $tag2$ is 2, or $unknown$ if $tag2$ is 0. Return YES.

The algorithm starts by checking whether the index variable is ever defined in any way other than being increased by 1. If so, 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. Note that the algorithm allows an array element to be written multiple times in one loop iteration before the index variable is increased by 1. Steps 3, 4 and 5 compute the section of array elements being written in the loop. For example, in Fig.1, the section where $x()$ is written after the repeat-until loop is $[1, p]$; and, in Fig.3 (a), the section where $x()$ is written after $loop\_do\_i$ is $[1, p-1]$. The section in Fig.3 (b) is $[1, unknown]$. Step 3 also ensures that an array element is not written in two different iterations of the loop. For example, the algorithm accepts the array $x()$ in Fig.3 (a) and (b) as consecutively written, but rejects the array $y()$ in Fig.3 (c).

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

(a)

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

(b)

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

(c)

Fig. 3. Consecutively written or not?

The algorithm is conservative in the sense that it may fail to report a consecutively written array, but never report one that is not in fact.

2.2 Applications

Dependence Test and Parallelization If a single-indexed array is consecutively written in a loop and is write-only in this loop, then the array does

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

not cause any data dependences. Flow data dependence does exist for the index variable. However, if the index variable is not used in 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 eliminate the dependence in order to parallelize the enclosing loop.

Array splitting and merging has 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 length of its private copy of the array; hence, the starting position in the original array for each processor can be easily calculated. Finally, the private copies are copied back (merged) to the original array. Figure 4 shows an example when two processors are used.

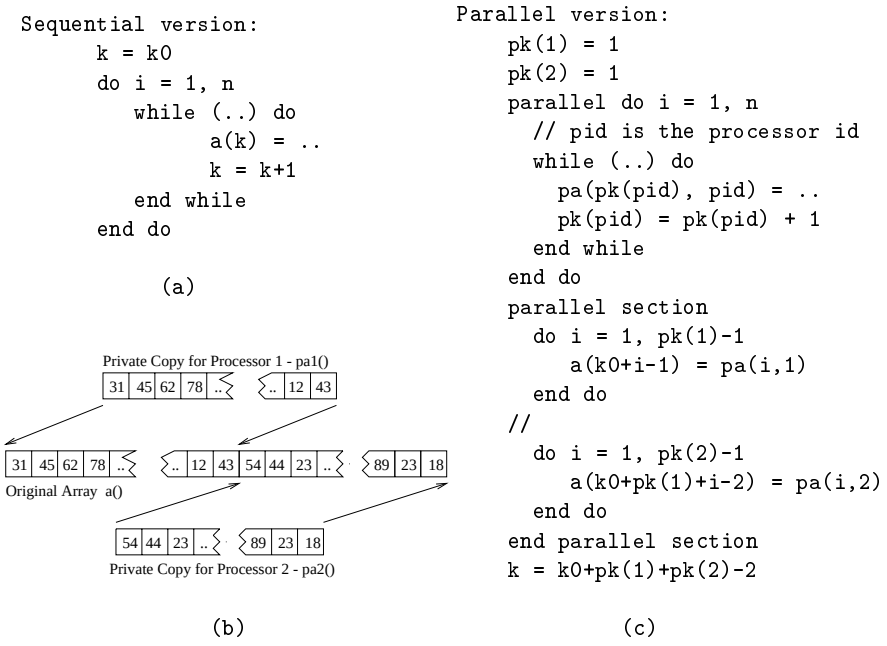

Fig. 4. An Example of Array Splitting and Merging

Privatization Test Consecutively written array analysis can find the array element section where a consecutively written array is written in a natural loop. Traditional array privatization techniques can handle only do loops and require a closed-form expression of the array subscripts in order to compute the section of array elements read or written in the loop. As we have illustrated at the beginning of this paper, with consecutively-written array analysis, we can easily

extend the privatization test to process irregular single-indexed arrays and more general loops.

Index Array Property Analysis The *indirectly accessed array* is another kind of irregular array. An array is indirectly accessed in a statement if the subscript of the array is another array, such as the array $x()$ in statement "$x(ind(i)) = y(i)$". $x()$ is called the *host array*, and $ind()$ is called the *index array*. It is easy to see that traditional techniques cannot handle indirectly accessed arrays. However, recent studies[4, 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 help find the properties an array can have in the array property analysis. For example, two of the key properties that index arrays may have are *injectivity* and *closed-form bounds*. An array section is injective if none of any two different array elements in the section 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 a closed-form expression. 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 *loop_do_i* in Fig.5.

```
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. 5. An example of a loop with an inner index gathering loop

In Fig.5, 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 $x[1:q]$ are defined,
- the values of the array elements in array section $x[1:q]$ are injective, and
- the lower bound of the values of the array elements in section $x[1:q]$ is 1, and the upper bound is q.

With this information available at compile-time, the compiler is now able to determine that:

1. there is no data dependence in *loop_do_j*, and
2. array $ind()$ can be privatized in *loop_do_k*.

Thus, the compiler can choose either to parallelize *loop_do_k* only, parallelize *loop_do_j* only, parallelize both, or parallelize *loop_do_k* and vectorize *loop_do_j*, depending upon the architecture for which the code is generated.

An index gathering loop for an index array can be characterized as:

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 of the index array is the loop index, and
5. one assignment of the index array cannot reach another assignment of the index array without first reaching the do loop header.

The fourth 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 by 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 6 illustrates an array stack. In the body of *loop_do_i*, 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. In Sect.3.1, we show how to detect array stacks, and in Sect.3.2 we demonstrate how to enhance compiler optimizations in programs where array stacks are detected.

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:

```
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. 6. An example of an array stack

1. $p := p + 1$,
2. $p := p - 1$, or
3. $p := C_{bottom}$, where C_{bottom} is a constant 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 1.

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

Table 1. Order for Access of Array Stacks

The left column and the top row in Table 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 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 of kind of statement must be on the path. The 'G' represents an if statement that is "$if\ (p \geq C_{bottom})\ then$".

Intuitively, the order in Table 1 ensures 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 1 can be simplified to Table 2. Any path originating from a node n of the forms in the left column of Table 2 must first reach any node of the forms in $S_{bound}(n)$ before it reaches any node of the forms in $S_{failed}(n)$.

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

Table 2. Simplified Order for Array Stacks

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 minimun 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 the forms 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), then return NO. Otherwise, put the definition statements in a list lst.
  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 help 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 Bounds-checking Elimination Run-time array bounds-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 to 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 bounds-checking is also desirable for programs written in other languages since it helps with program tests and debugging. 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 bounds-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 the 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. Different iterations of the loop will reuse the same array elements, but 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.6 can be privatized in the outermost loop *loop_do_i*.

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, among many other optimizations. Loop interchanging changes the nest 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 just 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, 10], M. Gerlek and E. Stoltz [10] have 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[6, 5]. 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 their work 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 statements, then none of them can detect whether the array itself is consecutively written. For example, Wolfe et al's method can find that the two instances of variable k in statements (1) and (2) in Fig.7 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 Wolfe et al'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.

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

The authors believe it is often important to consider both index variables and arrays in loop analysis. 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 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 the 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 3. Column three in Table 3 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 four. Figure 8 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 the array stack analysis in our Polaris compiler, so for TREE, we show the result of manual parallelization. For all of the 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.

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%

Table 3. Three Real-world Programs

5.1 BDNA

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

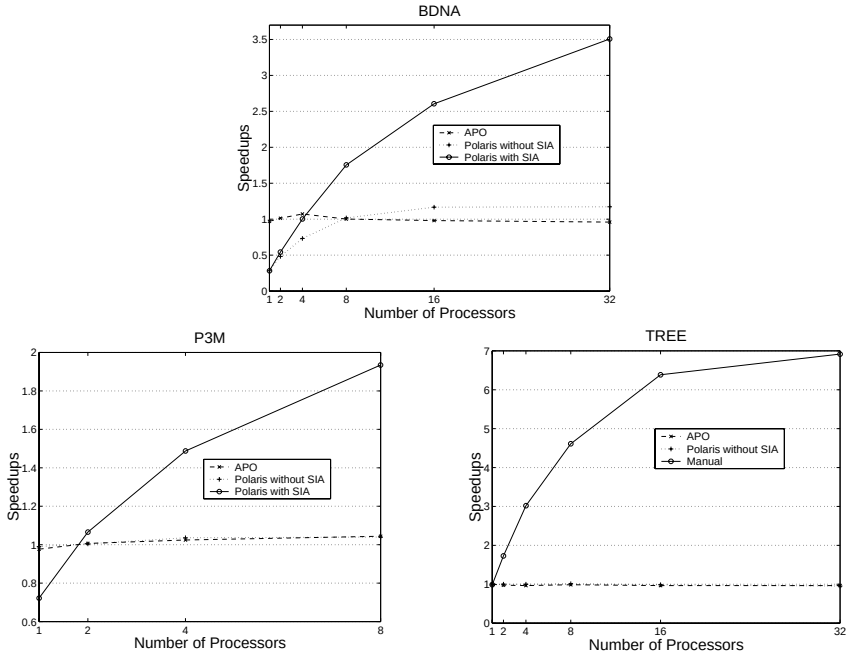

Fig. 8. Comparison of Speedups

Loop *loop_do_*240 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.9

Consecutively written array analysis is used in *loop_do_j* to find that elements in $[1, l]$ of $ind()$ are written in this loop. Furthermore, this loop is recognized as an index gathering loop; thus, the values of the elements in $ind[1, l]$ defined in this loop are bounded by $[1, i-1]$. This information is used to privatize array $ind()$ and $xdt()$ in *loop_do_i*, 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 libraray) is spent in subroutine *pp* and *subpp*. The structure of *pp* and *subpp* are very similar. The major part is a three-perfect-loop nest, which can be parallelized. However, before parallelization, several single-indexed arrays in the loop must be privatized. These single-indexed arrays are used in a way that is a mix of the ways the arrays are used in the loop body in Fig.1 and Fig.5.

```
do i = 2, nsp
   do j = 1, i - 1
      xdt(j) = ..                    (1)
      ind(j) = ..                    (2)
   end do
   l = 0
   do j = 1, i - 1
      if (ind(j) .eq. 0) then
         l = l + 1
         ind(l) = j                  (3)
      end if
   end do
   do k = 1, l
      .. = xdt(ind(k))               (4)
   end do
end do
```

Fig. 9. Outline of *actfor_loop_do_*240 in BDNA

5.3 Barnes & Hut TREE code

The **TREE** code[7] is a program that implements the hierarchical N-body method for simulating the evolution of collisionless systems[3].

The major body 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.10.

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 arrays. We described two common access patterns of irregular single-indexed arrays (i.e., consecutively written and stack access) and presented simple and intuitive algorithms to detect these two patterns. More importantly, we showed that arrays following these two access 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 bounds-checking. In the case study, we showed that, for three real-life programs, the speedups of the

```
sptr = 1
stack(sptr) = root
while (sptr .gt. 0) do
   q = stack(sptr)
   if ( q is a body ) then
      process body-body interaction
   else if ( 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. 10. Treewalk kernel[3] in the TREE code

parallelized versions generated by the Polaris compiler with single-index array analysis are much better than those of other versions.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
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.
3. J. Barnes and P. Hut. A hierarchical o(nlogn) force calculation algorithm. *Nature*, 324(4):446–449, 1986.
4. 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.
5. 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.
6. 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. *ACS Transaction on Programming Languages and Systems*, 13(4):451–490, October 1991.
7. Joshua Edward. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/. Technical report.

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.
9. P. Feautrier. Array expansion. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.
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.
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.
12. Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. *ACM SIGPLAN Notices PLDI 1995*, 30(6):270–278, June 1995.
13. Z. Li. Array privatization for parallel execution of loops. In ACM, editor, *Proceedings of 1992 International Conference on Supercomputing, July 19–23, 1992, Washington, DC*, pages 313–322, New York, NY 10036, USA, 1992. ACM Press.
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 Runtime Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 41–56. Springer-Verlag, Pittsburgh, PA, 1998.
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.
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.
17. Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In ACM, editor, *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.
18. Madalene Spezialetti and Rajiv Gupta. Loop monotonic statements. *IEEE Transactions on Software Engineering*, 21(6):497–505, June 1995.
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.
20. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.
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.