Compiler Analysis of Irregular Memory Accesses

Yuan Lin
Department of Computer Science
University of Illinois at Urbana-Champaign
yuanlin@uiuc.edu

David Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
padua@uiuc.edu

ABSTRACT

Irregular array accesses are array accesses whose array subscripts do not have closed-form expressions in terms of loop indices. Traditional array analysis and loop transformation techniques cannot handle irregular array accesses. In this paper, we study two kinds of simple and common cases of irregular array accesses: single-indexed access and indirect array access. We present techniques to analyze these two cases at compile-time, and we provide experimental results showing the effectiveness of these techniques in finding more implicit loop parallelism at compile-time and improved speedups.

1. INTRODUCTION

High-level language analysis and optimization techniques have been used to detect parallelism, privatize data, enhance locality, and reduce communication costs. Most traditional methods operate on `do` loops and require that array subscript expressions inside the loops contain only loop indices and loop invariants. In addition, most methods require the subscript expressions to be affine. However, many important scientific programs contain *irregular* array accesses. We define an array access as *irregular* if no closed-form expression, in terms of the loop indices, for the subscript of the accessed array is available at compile-time. Because current analysis techniques cannot handle irregular array accesses, many codes are left unoptimized.

Consider, for example, array privatization [12, 19, 23, 30], an important technique for loop parallelization. An array can be privatized if, within a given iteration, its elements are always assigned before they are read. Thus, in each iteration of the outermost loop `do k` in Fig. 1(a), any element of $x()$ read at statement (3) in loop `do j` is defined at statements (1) and (2) in the *while* loop. Therefore, array $x()$ can be privatized for loop `do k`. Because there is no dependence, loop `do k` can be parallelized. However, because current privatization tests require a closed-form expression in order to compute the section of array elements read or written

```
do k = 1, n
  p = 0
  i = link(1,k)
  while (i != 0) do
    p = p+1
    x(p) = y(i)          (1)
    i = link(i,k)
    if (cond(k,i)) then
      p = p+1
      x(p) = y(i)        (2)
    end if
  end do
  do j = 1,p
    dz(k,j) = x(j)       (3)
  end do
end do
                (a)
```

```
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 loop
                (b)
```

```
do i = 1, n
  do j = 1, m
    x(j) = ..
  end do
  do k = 1, p
    y(i,k) = x(pos(k))
  end do
end do
                (c)
```

Figure 1: Examples of Irregular Array Accesses

in a loop and because, in this example, there is no such expression for p, these techniques can only determine that section $[1 : p]$ of $x()$ is read in loop `do j`. They cannot determine that the same section also is written in the *while* loop and, therefore, fail to privatize $x()$.

A second example is presented in Fig. 1(c). In loop `do k`, array $x()$ is indirectly accessed via another array $pos()$. If it could be detected at compile-time that the values in $pos[1 : p]$ are inside the interval $[1, m]$, then the compiler would be able to privatize $x()$ for loop `do i` and parallelize the loop.

User assertions [24, 18] and run-time tests [32, 27, 26] have been proposed as alternatives to static analysis for irregular memory accesses. However, information useful for a compiler may not be of interest to a programmer. Inserting assertions is a tedious job and can lead to errors that are difficult to detect. A possible strategy is to use run-time analysis methods. However, these methods introduce overhead that is not always negligible and also increase the code size, since the unoptimized version must also be available in case the tests fail. While user assertions and run-time tests are useful and important, compile-time analysis is clearly preferable and therefore should be used whenever possible.

In this paper, we present techniques to analyze irregular array accesses and show how to use the results of the analysis to enhance compiler optimizations. We study two classes of irregular array accesses.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI 2000, Vancouver, British Columbia, Canada
Copyright 2000 ACM 1-58113-199-2/00/0006 ..$5.00

1. *Irregular single-indexed access:* The accesses of an array in a given loop are *single-indexed* if the array is always subscripted by a single variable and that variable is the same throughout the loop. Array $x()$ is single-indexed in the *while* loop in Fig. 1(a) and in the loop `do i` in Fig. 1(b).

2. *Simple indirect array access:* An array is *indirectly-accessed* if it is subscripted by an array element. We call the array itself the "host array", and the array in the subscript the "index array". An indirect array access is *simple* if the innermost enclosing loop is a `do` loop, say L, and the subscript of the index array is the loop index of loop L. The reference to array $x()$ in loop `do k` in Fig. 1(c) is a simple indirect array access.

These two classes of irregular array accesses were chosen because they arise frequently in scientific codes.

This paper makes the following contributions:

1. It presents examples showing that compile-time analysis of irregular array accesses can enhance other analyses and optimizations, such as data dependence tests, privatization tests, and loop parallelization.

2. It presents two compile-time techniques to analyze the two common cases of irregular array accesses just discussed. These two techniques are simple and effective. They take advantage of the fact that, in real programs, irregular array accesses often follow a few fixed patterns and have detectable properties. We also demonstrate the importance of using these two techniques together in analyzing irregular array accesses. For irregular single-indexed access, a *bounded depth-first search* method is used to trace the evolution of index variables between two array accesses. Two classes of index evolutions can be identified: *consecutively-written* and *stack access*. For simple indirect array access, an *array property analysis* method is used. It performs interprocedural array dataflow analysis following a demand-driven approach.

3. It describes the implementation of the techniques in a research compiler and presents experimental results showing the effectiveness of the techniques in detecting loop parallelism at compile-time. Thanks to the new techniques, nine loops in five real programs that could not be handled by the traditional methods were found parallel. And, because of the parallelization of these loops, the performance of four of the programs improved significantly.

The rest of this paper is organized as follows. Sections 2 and 3 present the techniques to analyze single-indexed array accesses and indirect array accesses, respectively. Section 4 describes how to use the two techniques together. Section 5 discusses the implementation and experimental results. Section 6 concludes the paper. Related work is mentioned throughout the paper.

```
   bDFS(u)
1  visited[u] := true ;
2  f_proc(u) ;
3  if (not f_bound(u)) {
4      for each adjacent node v of u {
5          if (f_failed(v))
6              return failed ;
7          if ((not visited[v]) and (bDFS(v) == failed))
8              return failed ;
9      }
10 }
11 return succeeded ;
```

Before the search starts, $visited[]$ is set to $false$ for all nodes.

Figure 2: Bounded depth-first search

2. IRREGULAR SINGLE-INDEXED ARRAY ACCESSES

This section presents a brief description of a method to analyze irregular single-indexed array accesses. A detailed description can be found in [22].

In irregular single-indexed array accesses, the index variable may have different values at different points of the body of a loop. For example, in Fig. 1(a), the index variable p has different values at statements (1) and (2) which are within the body of the *while* loop. It is important to examine how the value of an index variable changes between consecutive accesses to an array reference subscripted by the variable.

Within a loop, the changes can be monotonic or non-monotonic. In the monotonic case, such as p in the *while* loop in Fig. 1(a), the location of array elements accessed increase or decrease during execution. No such order exists in the non-monotonic case, such as p in the body of loop `do i` in Fig. 1(b). For the monotonic case, we want to determine whether this array is consecutively written. In the non-monotonic case, we want to determine whether this array is used as a stack.

2.1 Bounded Depth-first Search

We first describe a *bounded depth-first search* (bDFS) algorithm that will be used later used to build the analysis algorithm for irregular single-indexed arrays. The bDFS algorithm, shown in Fig.2, does a depth-first search on a control flow graph (V, E), where V is the set of vertices and E is the set of edges in the graph. It uses two functions to control the search. These two functions, $f_{bound}()$ and $f_{failed}()$, are defined before the search starts. $f_{bound}()$ maps V onto $(true, false)$. 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 n_0. The nodes whose $f_{bound}()$ values are $true$ are the boundaries of the search. $f_{failed}()$ also maps V onto $(true, false)$. 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.

2.2 Consecutively Written Array Accesses

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 increasing or decreasing order[1]. For example, in the *while* loop in Fig. 1(a), 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.

The knowledge that an array is consecutively written in a loop can be used to enhance many analyses and optimizations, such as array privatization tests and data dependence tests [22]. For the array privatization example in Fig. 1(a), the "consecutively written" property is very important because it guarantees that the writes cover the reads in the same iteration.

To determine whether a single-indexed array $x()$ with index variable p is consecutively written in a loop, we first check whether p is ever defined in any way other than being increased by 1 in the loop. If so, we assume the array is not consecutively written. Then we check, by performing bDFSs starting from each of the "$p = p + 1$" statements on the control flow graph of the loop, whether there exists a path from one "$p = p+1$" statement to another "$p = p+1$" statement 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. To accomplish this, the bDFS algorithm can be used with $f_{bound}(n)$ set to *true* only when n refers to x and $f_{failed}(n)$ set to *true* only when n is of the form "$p = p + 1$".

Wolfe et al [31, 13] have presented an algorithm to recognize and classify *sequence variables* in a loop. R. Gupta and M. Spezialetti [28] have extended the traditional data-flow approach to detect "monotonic" statements. While both of these methods can recognize the index variable p in the *while* loop in Fig. 1(a) as a monotonic variable, none of them can determine that the array $x()$ is consecutively written in the *while* loop. The reason is that these methods are not designed to study the effect of index variables on array accesses. Both methods can recognize a wider class of scalar variables than our method, but so far we have not found any case where this extra power improves array access analysis.

2.3 Stack Access

Many programs implement stacks using arrays because it is both simple and efficient. We call stacks implemented in arrays *array stacks.* Figure 1(b) illustrates an array stack. In the body of loop `do i`, array $t()$ is used as a stack and p is the index of the top of the stack.

The knowledge that an array is an array stack can be used to improve many optimizations. Again, consider array privatization. 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 loop. Different iterations of the loop will reuse the same array elements, but the value of the array elements never flows from one iteration to the other if the stack pointer is always reset to the same position at the beginning of each iteration. Therefore, array stacks in a loop body can be privatized. For example, in

[1]To be concise, we discuss only the increasing case in this paper.

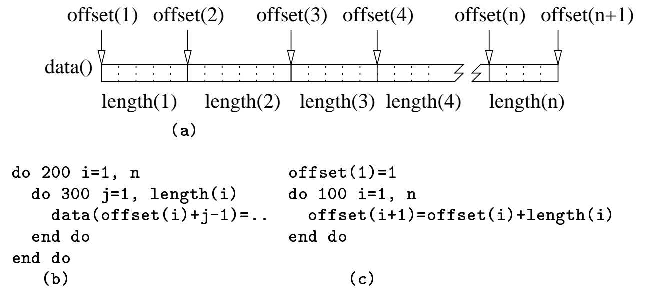

Figure 3: Example of closed-form distance

Fig. 1(b), array $t()$ can be privatized in loop `do i`. In [22], we also describe how to use the stack access information to improve run-time array bounds-checking elimination and loop interchanging.

To identify stack accesses, the algorithm looks for program regions in which the single index variable p is defined in only one of the following three ways: 1) $p := p + 1$; 2) $p := p - 1$; 3) $p := C_{bottom}$, where C_{bottom} is a constant in the program region. The algorithm checks whether a single-indexed array is used as a stack in the region by checking whether any path originating from statements in the left column of Table 1 reaches a statement in the set $S_{bound}(n)$ before it reaches a statement in the set $S_{failed}(n)$. This can be done by performing bDFSs on the control flow graph of the program region. Intuitively, the order in Table 1 ensures that for an array stack $x()$ with index p, p is first set to C_{bottom} before it is modified or used in the subscript of $x()$, the value of p never goes below C_{bottom}, and that the access of elements of $x()$ follows the "last-written-first-read" pattern.

3. INDIRECT ARRAY ACCESSES

In their study of the Perfect Benchmarks, Blume and Eigenmann found that index arrays often have some properties [7] that can be detected statically and that enable program parallelization. Similar results also were obtained by the authors in a study of several sparse and irregular programs [21]. By identifying these properties, compilers can avoid making conservative assumptions in the analyses of indirect array accesses.

For example, in the Compressed Column Storage (CCS) format for sparse matrices, the non-zero elements of a sparse matrix are stored in a one-dimensional host array. The host array is divided into several segments, as illustrated in Fig. 3(a). Each segment corresponds to a column. The starting position and the length of each segment are given by index arrays `offset()` and `length()`, respectively. Figure 3(b) shows a common loop pattern accessing the elements in the matrix. The loop traverses the host array segment by segment. Figure 3(c) shows a common pattern used to define `offset()`. Loop `do 200` does not carry any dependences if $length(i) \geq 0$ because

$$offset(i) + length(i) - 1 < offset(i + 1), \text{where } 1 \leq i < n,$$

which can be derived from the loop in Fig. 3(c).

n	$S_{bound}(n)$	$S_{failed}(n)$
$p = p + 1$	$\{x(p) = .., p = C_{bottom}\}$	$\{p = p + 1, p = p - 1, .. = x(p)\}$
$p = p - 1$	$\{p = p + 1$, 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 1: Order for Array Stacks

Given an array, if the difference of the values of any two consecutive elements can always be represented by a closed-form expression, we say the array has a *closed-form distance* [5]. The other properties of index arrays that can be used by compilers are *injectivity*, *monotonicity*, *having a closed-form value*, and *having a closed-form bound*. An array $a()$ is injective if $a(i) \neq a(j)$ when $i \neq j$. An array $a()$ is monotonically non-decreasing (non-increasing) if $a(i) \leq a(j)$ $(a(i) \geq a(j))$, $i < j$. An array has a closed-form value if all the values of the array elements can be represented by a closed-form expression in terms of array indices and constants at compile-time. An array has a closed-form bound if a closed-form expression is available at compile-time for either the lower bound or the upper bound of the array elements' values.

To determine whether an index array has one of the properties listed above, it is necessary to check all the definitions of the index array reaching an indirectly accessed array reference. If the program patterns at all the definition sites imply that the index array has the property being studied, and none of the statements in between the definition sites and the use site redefines any variables that are used to express the property, then we say that the property is *available* at the use site. Otherwise, it is assumed that the property is not available.

Our strategy is to perform a demand-driven interprocedural array property analysis to solve the available property problem. The analysis is done interprocedurally because, in most real programs, index arrays often are defined in one procedure and used in other procedures. The analysis is demand-driven because the cost of interprocedural array reaching definition analysis and property checking is high. The compiler performs the analysis only when it meets an index array and it checks only the property that the use site suggests. For example, in Fig. 3, the compiler would check only the property of having a closed-form distance for the use site `s3`.

3.1 Dataflow Model for Query Propagation

We model our demand-driven analysis of the available property as a query propagation problem [11]. A query is a tuple $(st, section)$, where st is a statement or a basic block and $section$ is an array section[2] for the index array. Given an index array and a property to be checked, a query $(st, section)$ raises the question whether the elements of the index array in $section$ always have the desired property when the control reaches the point after st.

[2] An array section can be represented as a convex region [29], an abstract data access [2, 25], or a regular section [17]. Our method is orthogonal to the representation of the array section. Any representation can be used as long as the aggregation operation used in Sect. 3.2.5 is defined.

A query is propagated in the reverse direction of the control flow until it can be verified to be $true$ or $false$. Let $OUT(S)$ be the section of index array elements to be examined at statement or basic block S, $GEN(S)$ be the section of index array elements possessing the desired property because of the execution of S, $KILL(S)$ be the section of the index array elements verified not to have the property because of the execution of S, and $IN(S)$ be the section of the index array elements that cannot be verified to possess the property by examining S and, thus, should be checked again at the predecessors of S. The general dataflow equations for the reverse query propagation are

$$\begin{aligned} OUT(S) &= \bigcup_{T \text{ is a successor of } S} IN(T), \\ IN(S) &= OUT(S) - GEN(S) \end{aligned}$$

For a query $(st, section)$, initially, $OUT(st) = section$ and $OUT(s) = \emptyset$ for any statement or basic block s other than st. If, after the propagation finishes, we have $IN(entry) \neq \emptyset$, where $entry$ is the entry node of the program control flow graph, or there exists a statement or basic block s such that $OUT(s) \cap KILL(s) \neq \emptyset$, then the answer to the original query is $false$. Otherwise the answer is $true$.

Our approach of modeling a demand for property checking as a set of queries was inspired by the work of E. Duesterwald et al. [11]. They proposed a general framework for developing demand-driven interprocedural data flow analyzers. They use an iterative method to propagate the queries and can handle only scalars. We use a structural analysis and work on arrays, which is possible because available property is a specific distributive dataflow problem.

3.2 Demand-driven Interprocedural Array Property Analysis

3.2.1 Overview

We represent the program in a hierarchical control graph (HCG), which is similar to the *hierarchical supergraph* [15]. Each statement, loop, and procedure is represented by a node, respectively. There also is a section node for each loop body and each procedure body. Each section node has a single entry node and a single exit node. We assume that the only loops in the program are `do` loops, and we deliberately delete the back edges in the control flow graph. Hence, the HCG is directed acyclic. We also assume no parameter passing, values are passed by global variables only, and if constant numbers are passed from one procedure to another, the callee is cloned. Techniques to handle parameter bounding, array reshaping, and variable aliasing are well known and can be found in [10, 16, 9, 25].

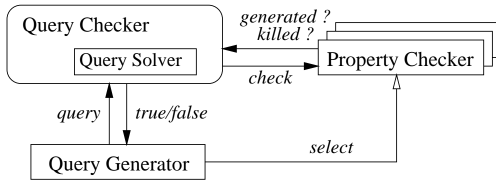

Figure 4: The components of array property analysis

Method: $QuerySolver(query, n_{root})$
Input: 1) a query $query = (n_{init}, sect_{init})$
2) a root node n_{root} that dominates n_{init}
Output: $(anykilled, sect_{remain})$
Begin:

```
1    worklist := ∅ ;
2    add_∪((n_init, sect_init), worklist) ;
3    anykilled := false ;
4    while worklist ≠ ∅ do
5      remove an element (n, sect) from the worklist ;
6      if ( n is n_root ) then
7        sect_remain := sect ;
8        break ;
9      end if
10     (anykilled, sect_remain) := QueryProp(n, sect) ;
11     if ( anykilled ) then break ;
12     if ( sect_remain ≠ ∅ ) then
13       for each node m ∈ pred(n)
14         add_∪((m, sect_remain), worklist) ;
15       end for
16     end if
17   end while
18   return (anykilled, sect_remain) ;
```

End

Figure 5: QuerySolver

Our demand-driven interprocedural analysis method consists of three parts, as shown in Fig. 4. The *DemandGenerator*, which is incorporated in the analysis methods, issues a query when the analysis needs to verify whether an index array has a certain property at a certain point. The *QueryChecker* accepts the query and then uses *QuerySolver* to traverse the program in the reverse direction of the control flow to verify the query. It uses the *PropertyChecker* to get the *GEN* and *KILL* information.

3.2.2 QuerySolver

The major component of *QueryChecker* is *QuerySolver*, which returns a tuple $(anykilled, section_{remain})$ when given a query $(n_{query}, section_{query})$ and a root node n_{root} that dominates n_{query}. The *anykilled*, which is a boolean, is $true$ if the property of some element in $section_{query}$ might be killed when the program is executed from n_{root} to n_{query}. When *anykilled* is $false$, $section_{remain}$ gives the array elements whose properties are neither generated nor killed from n_{root} to n_{query}. In order to check if the index array elements in $section_{query}$ at node n_{query} have the desired property, *QueryChecker* invokes *QuerySolver* with the n_{root} being the entry node of the whole program. If *anykilled* is $true$ or if *anykilled* is $false$ but $section_{remain}$ is not empty, then we assume that the index array does not have the desired property. Otherwise, it has the desired property.

Method: $QueryProp(n, section)$
Input: A query $(n, section)$
Output: $(anykilled, section_{remain})$
Begin:

```
1    (Kill, Gen) := Summarize(n) ;
2    section_remain := section − Gen ;
3    anykilled := ((Kill ∩ section) ≠ ∅) ;
4    return (anykilled, section_remain) ;
```

End

Figure 6: A general framework of reverse query propagation $QueryProp$

Figure 5 shows the algorithm for $QuerySolver$. A worklist whose elements are queries is used. The algorithm takes a query $(n, sect)$ out of the worklist. The query $(n, sect)$ asks whether any array element in $sect$ can have the desired property immediately after the execution of n. This is checked by reverse query propagation $QueryProp$. $QueryProp$ returns a tuple $(anykilled, sect_{remain})$, which has a meaning that is similar to the tuple returned by $QuerySolver$. The *anykilled* is set to $true$ if the property of one or more elements in $sect$ is killed when n is executed. In this case, the answer to the original query is $false$; thus, no further analysis is needed and the algorithm returns. This is an early-termination. When *anykilled* is false, new queries are constructed from the $sect_{remain}$ and the predecessors of n and are inserted into the worklist. This process repeats until the worklist becomes empty or the root node is met. The use of a worklist makes early-termination possible.

The worklist is a priority queue. All the elements are sorted in reverse topological order($rTop$) of its node in the control flow graph (which, as stated above, is a DAG). Therefore, a node is not checked until all its successors have been checked. This ensures that the query presented to a node is composed of the queries propagated by its successors.

Queries are inserted into the list by using $add_{\cup}()$. The operation $add_{op}()$, where op can be either $\cap$ or $\cup$, is defined as follows: if there exists a query $(n, section')$ in the worklist, then replace $(n, section')$ with $(n, section\ op\ section')$; otherwise, insert $(n, section)$ into the $worklist$ according to the $rTop$ order.

3.2.3 Reverse Query Propagation

Conceptually, reverse query propagation $QueryProp$ computes IN $(section_{remain})$ from OUT $(section)$, GEN and $KILL$. Figure 6 shows a general framework of $QueryProp$.

Conceptually, reverse query propagation $QueryProp$ computes IN from OUT, GEN and $KILL$. Figure 6 shows a general framework of $QueryProp$. The $QueryProp$ uses $Summarize$ to summarize the effect of executing a node. The effect of executing a statement can be represented by the $(Kill, Gen)$ tuple.

The $Kill$ and Gen evaluated by the summarization method are often approximate values. There are two reasons for this. First, the index array may be assigned variables whose values or relationships with other variables cannot be determined by the compiler; therefore, the section of the array elements being accessed cannot be represented precisely.

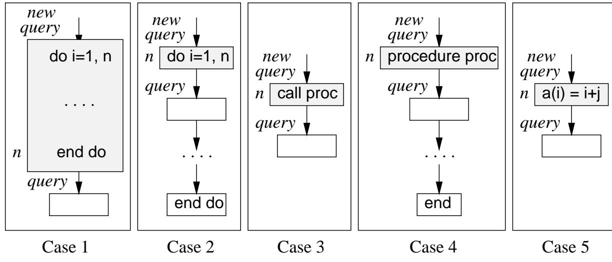

Figure 7: The five cases: 1) n is a *do* node, 2) n is a `do` statement, 3) n is a `call` statement, 4) n is a procedure head, and 5) otherwise.

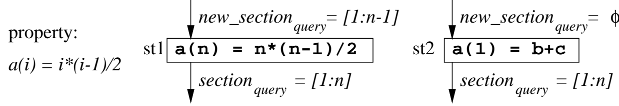

Figure 8: An example of simple reverse query propagation

Second, the summarization method works on array sections, but the set operations being used usually are not closed on the most popular section representations. Hence, the results can be only approximated. In order not to cause incorrect transformations, the approximation must be conservative. $Kill$ is a MAY approximation and Gen is a $MUST$ approximation. In the worst case, $Kill$ can be the universal section $[-\infty, \infty]$ and Gen can be $\emptyset$.

The $Summary$ method can only be applied to statements or basic blocks. In fact, there are five different classes of node, as illustrated in Figure 7. Each case has to be handled by a different reverse query propagation method as described in the next three Subsections.

3.2.4 Simple Reverse Query Propagation

In case 5, the node n is a statement other than a `do` statement, a *do* node, a `call` statement, or a procedure head. In this case, the effect of executing node n can be derived by examining n alone. $QueryProp_{simple}$ uses the same framework as $QueryProp$ in Figure 6 with $Summarize$ being replaced by $SummarizeSimpleNode$, which also is the interface between the $QuerySolver$ and the $PropertyChecker$ (see Sect.3.2.8).

EXAMPLE 1. *In Figure 8, statements $st1$ and $st2$ are simple assignments of array $a()$. The property to be checked is $a(i) = i * (i-1)/2$. Hence,*

$$SummarizeSimpleNode(st1) = (Kill = \emptyset, Gen = [n:n])$$
$$SummarizeSimpleNode(st2) = (Kill = [1:1], Gen = \emptyset)$$

Thus, after the propagation, for statement $st1$ we have:

$$anykilled = false, section_{remain} = [1:n-1].$$

And, for statement $st2$ we have:

$$anykilled = true, section_{remain} = \emptyset.$$

Method: $SummarizeProgSection(n)$
Input: A section node n
Output: $(Kill, Gen)$
Begin:

1 Let n_{entry} be the entry node of section n, and let n_{exit} be the exit node of section n.
2 $Gen := \emptyset$;
3 $Kill := \emptyset$;
4 $WorkList := \emptyset$;
5 $add_{\cap}((n_{exit}, \emptyset), Worklist)$;
6 while $WorkList \neq \emptyset$ do
7   take an element (n, gen') out of the $Worklist$;
8   if $(n = n_{entry})$ then
9     $Gen := gen'$;
10   break ;
11   end if
12   begin case
13   case n is a `call` statement:
14       $(kill, gen) := SummarizeProcedure(n)$;
15   case n is a do node:
16       $(kill, gen) := SummarizeLoop(n)$;
17   otherwise:
18       $(kill, gen) := SummarizeSimpleNode(n)$;
19   end case
20   if (n dominates n_{exit}) $Gen := gen'$;
21   if ($kill = [-\infty, \infty]$) then
22     $Kill := kill$;
23   break ;
24   end if
25   $Kill := Kill \cup (kill - gen')$;
26   for each $m \in pred(n)$
27     $add_{\cap}((m, gen' \cup gen), WorkList)$;
28   end for
29 end while
30 return $(Kill, Gen)$;
End

Figure 9: $SummarizeProgSection$

3.2.5 Loop Analysis

Cases 1 and 2 deal with loops. Array dataflow analysis is different from scalar analysis because different array elements might be accessed in different iterations of a loop, while the same set of scalars are usually accessed in all iterations. To summarize the effect of the loops, aggregation methods such as those proposed by Gross and Steenkiste [14] (for one dimensional arrays) and by Gu et al. [15] (for multiple dimensional arrays) can be used to aggregate the array access.

Given an array section, $section_i$, expressed in terms of index i, $aggregate_{low \le i \le up}(section_i)$ computes the section obtained by aggregating $section_i$ with i ranging from low to up.

In case 1, the initial query comes from outside the loop. Like the simple node case, the framework in Figure 6 can be used. The only difference is that we summarize the effect of executing the whole loop rather than a single statement. Let $(Kill_i, Gen_i)$ be the effect of executing the loop body, up be the upper bound of loop m, and low be the lower bound of loop m. Then,

$$Kill = Aggregate_{low \le i \le up}(Kill_i)$$
$$Gen = Aggregate_{low \le i \le up}(Gen_i - Aggregate_{i+1 \le j \le up}(Kill_j))$$

Method: $QueryProp_{do_header}(m, sect)$
Input: A query $(m, sect)$, where m is a node of `do` statement node
Output: $(anykilled, sect_{remain})$
Begin:

1 Let n be the section node of the loop body, and assume n represents the ith iteration of the loop ;
2 $(Kill_i, Gen_i) := SummarizeProgSection(n)$;
3 Let up be the upper bound of loop, and low be the lower bound of loop;
4 if ($sect \cap Aggregate_{low \le j \le (i-1)}(Kill_j) \ne \emptyset$) then
5 return $(true, \emptyset)$;
6 end if
7 $sect_{remain_i} := sect - Aggregate_{low \le j \le (i-1)}(Gen_j)$;
8 $sect_{remain} := Aggregate_{low \le i \le up}(sect_{remain_i})$;
9 return $(false, sect_{remain})$;

End

Figure 10: $QueryProp_{do_header}$

When $Kill$ and Gen are computed, the dataflow equations can be applied.

$(Kill_i, Gen_i)$ is computed by using the $SummarizeProgSection$ method shown in Fig. 9. It computes the effect of executing a section by reverse propagation of the $Kill$ and Gen set from the exit node to the entry node. It also uses a worklist similar to the one used in $QuerySolver$. The elements (n, gen) in the worklist, however, are not queries here. The gen is the section of array elements that have been generated because of the execution of the program from the exit of node n to the exit of the section. Another difference is that elements are inserted into the worklist by using $add_\cap$ instead of $add_\cup$. The effect of a section also could be computed in the forward direction. We use a backward method here because it is more efficient. It can early-terminate once the kill information is over-approximated to be the universal section (lines 21-24).

SummarizeProgSection uses *SummarizeSimpleNode*, *SummarizeProcedure*, and *SummarizeLoop* recursively depending on the type of statements used in the loop body [20]. *SummarizeProcedure* summarizes the effect of calling a procedure. Without considering parameter boundings, *SummarizeProcedure* can be computed by using *SummarizeProgSection* on the body of the procedure being called.

In case 2, the initial query comes from one iteration of the loop, say iteration I. The method is somewhat more complicated than the framework of Figure 6. When the loop header is found, a summarization is made of iterations preceding iteration I. It is used to compute the new query section corresponding to one iteration, which should then be aggregated in order to get the query section for the predecessors of the loop. The method $QueryProp_{loop_header}$ is shown in Figure 10.

3.2.6 Interprocedural Analysis

The interprocedural reverse query propagation also involves two cases (i.e., cases 3 and 4 in Figure 7).

In case 3, the node n is a `call` statement. We construct a new query problem with the initial query node being the exit node of the callee and the root node being the entry node

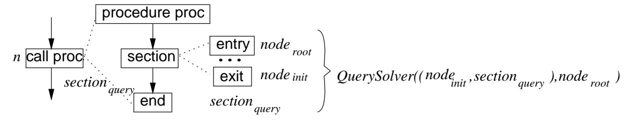

Figure 11: Reusing $QuerySolver$ **for** $QueryProp_{proc_call}$

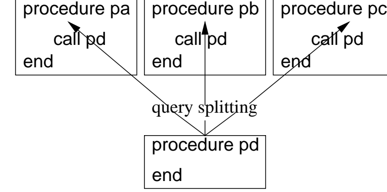

Figure 12: Query splitting

of the caller, as illustrated in Figure 11. A $QuerySolver$ is used to propagate the query. The $QuerySolver$ will early-terminate when any array element in query section is found not to have the desired property or all of them are found to have the property. Note in this case, the $QuerySolver$ always terminates when the header of the callee is reached.

In case 4, the node n is the header of a procedure. If n is not the program entry, then the query will be propagated into the callers of this procedure. We use a query splitting method illustrated in Figure 12.

Suppose the property query at the entry node n of a procedure $proc$ is $(n, sect_{query})$, and the call sites of $proc$ are $n_1, n_2, n_3, ...,$ and n_m. If n is the program entry, then the array elements in $sect_{query}$ are not generated in this program and. As a result, if $sect_{query}$ is not $\emptyset$, the property analysis terminate with the answer being $false$. If n is not the program entry, the query is split into m sub-queries, each of which has a set of initial queries as $\{(n', sect_{query}) | n' \in pred(n_i)\}$. The original query has a $true$ result when all the sub-queries terminate with a $true$ result. Otherwise, the initial query has a $false$ result.

Of the three major components of array property analysis, the *QueryChecker* is a generic method, while the *QueryGenerator* and the *PropertyChecker* are specific to the base problem, such as dependence tests or privatization tests, and specific to the potential property the array is likely to hold. In the next two sub-sections, we use a data dependence test problem to illustrate how to construct a *QueryGenerator* and a *PropertyChecker*.

3.2.7 Generating Demands

A new dependence test, called the *offset-length* test [20], has been designed to disprove data dependences in loops where indirectly accessed arrays are present and the index arrays are used as offset arrays and length arrays, such as the `offset()` and the `length()` in Fig. 3. The *offset-length* test needs array property analysis to verify the relationship between the *offset* arrays and the *length* arrays. Therefore, the *offset-length* test serves as a query generator.

We test whether a data dependence exists between two array accesses $a(f())$ and $a(g())$ with a dependence direction vec-

```
s0: do i=1, n
       do j=2, iblen(i)
          do k=1, j-1
s1:          x(pptr(i)+k-1) = ...
          end do
       end do
       do j=1, iblen(i)-1
          do k=1, j
s2:          ... = x(iblen(i)+pptr(i)+k-j-1)
          end do
       end do
    end do
```

Figure 13: A loop form DYFESM

tor $(=_1, =_2, ..., =_{t-1}, \neq_t, *, ..., *))$ in a loop nest. We assume that the index arrays in the loop nest (including arrays in the loop bounds) are arrays whose subscripts are expressions of the indices of the t outermost loop.

We first compute the ranges of values of $f(i_1, ..., i_t, *, ..., *)$ and $g(i_1, ..., i_t, *, ..., *)$ when $i_1, i_2, ..., i_t$ are kept fixed. Because the index arrays are arrays of only the outermost t loops, the loop indices $i_1, i_2, ..., i_t$ and the index arrays can be treated as symbolic terms in the range computation. If, except for the index arrays, $f()$ or $g()$ is an affine function of the loop indices, the range can be calculated by substituting the loop indices with their appropriate loop bounds, as with *Banerjee's test* [3]. Otherwise, the ranges can be calculated with the method used in some nonlinear data dependence tests, such as the *range test* [8].

If the ranges of $f(i_1, ..., i_t, *, ..., *)$ and $g(i_1, ..., i_t, *, ..., *)$ can be represented as $[x(i_t) + f_{low}\ , x(i_t) + y(i_t) + f_{up}]$ and $[x(i_t) + g_{low}\ , x(i_t) + y(i_t) + g_{up}]$, respectively, where $x()$ and $y()$ are two index arrays, and

$f_{low} = e(i_1, ..., i_{t-1}) + c_1,\ f_{up} = e(i_1, ..., i_{t-1}) - d_1,$

$g_{low} = e(i_1, ..., i_{t-1}) + c_2,\ g_{up} = e(i_1, ..., i_{t-1}) - d_2,$

$e(i_1, ..., i_{t-1})$ is an expression of indices i_1, i_2,...,i_{t-1} and index arrays of the outermost $t-1$ loops, and c_1 and c_2 are some non-negative integers, d_1 and d_2 are some positive integers, then there is no loop carried dependence between the two array accesses if index array $x()$ has a closed-form distance $y()$ and the values of $y()$ are non-negative. Intuitively, there is no dependence because the range of $f(i_1, ..., i_t, *, ..., *)$ does not overlap with the ranges of $f(i_1, ..., i_t \pm k, *, ..., *)$ and $g(i_1, ..., i_t \pm k, *, ..., *)$ for $k > 0$.

EXAMPLE 2. *Figure 13 shows a loop nest excerpted from the subroutine SOLXDD of Perfect Benchmark code DYFESM.*

We want to check if there is any loop-carried dependence between statement `st1` *and statement* `st2` *for the outermost loop* `do i`.

Here, $f(i, j, k) = pptr(i) + k - 1$ *and* $g(i, j, k) = iblen(i) + pptr(i) + k - j - 1$. *By substituting the loop indices with the loop bounds, we can compute the ranges of* $f()$ *and* $g()$ *when* i *is fixed, which are* $[pptr(i), pptr(i) + iblen(i) - 2]$ *and* $[pptr(i) + 1, pptr(i) + iblen(i) - 1]$, *respectively. If* $pptr()$ *has a closed-form distance of* $iblen()$, *which is non-negative, then for the outermost loop* `s0` *there is no flow-dependence from* `s1` *to* `s2`, *no anti-dependence from* `s2` *to* `s1`, *and no output-dependence from* `s1` *to* `s1`.

3.2.8 Checking Properties

Given a property to be verified and an assignment statement, the property checker *PropertyChecker* checks whether the assignment will cause any array elements to be generated or killed. In this subsection, we show how to use a simple pattern matching technique to check the *closed-form distance.*

Suppose the given property to be verified is

$x(i + 1) = x(i) + y(i)$, for $1 \leq i \leq n - 1$.

The *PropertyChecker* can take the following steps to inspect an assignment statement.

1. If the left-hand side (LHS) of the assignment is neither the array $x()$ nor the array $y()$, then nothing is generated or killed.
2. If the LHS is an array element $x(i)$, then the assignment and the other statements in the surrounding loops are checked to see if they match any of the following two patterns shown below. If not, then all elements of $x()$ are killed. Otherwise, $x(i)$ is generated.

```
                              t = ...
x(1)=...                      do i=1, n
do i=2, n                        x(i) = t
   x(i)=x(i-1)+y(i-1)            t=t+y(i)
end do                        end do

       (a)                          (b)
```

3. Otherwise (this includes the case when the LHS is an array element of $y()$ and the case when the LHS is an array element of $x()$ but the subscript is not a simple index), all elements of $x()$ are killed.

In general, the *closed-form distance* can be detected by using abstract interpretation, such as the recurrence recognition method proposed by Z. Ammarguellat and W. Harrison [1]. Compared with abstract interpretation, our pattern matching method is simpler and, thus, conservative. However, we found our method to be very effective in practice.

4. USING IRREGULAR SINGLE-INDEXED ARRAY ACCESS ANALYSIS TO CHECK PROPERTIES

Array property analysis uses *PropChecker* to examine the reaching definition sites and determine whether an array has the properties in the query. Constructing the *PropChecker* is facilitated by the fact that, in real programs, index arrays that have compiler-detectable properties usually have a few fixed definition patterns. Oftentimes, simple pattern matching methods like the ones presented in the previous section suffice. On the other hand, the analysis is complicated by the fact that the accesses of index arrays in the definition loops are usually irregular and, therefore, traditional array analysis methods cannot be used. The irregular single-indexed array access analysis method discussed in Sect. 2 can be used to deal with this case.

Two useful key properties of index arrays are *injectivity* and *having closed-form bounds.* Detecting whether an array section has any of the two properties is difficult in general.

```
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
```

Figure 14: An example of a loop with an inner index gathering loop

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. 14. In this example, 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 p.

With this information available at compile-time, the compiler now is able to determine that there is no data dependence in loop `do j` and 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 on which the code runs.

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 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.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

Compile-time analysis of irregular memory accesses can enable deeper analysis in many parts of an optimizing compiler. To evaluate its effectiveness, we measured its impact in finding more implicit parallelism. In this section, we describe the implementation of the irregular single-indexed array analysis and the implementation of the demand-driven interprocedural array property analysis in the Polaris parallelizing compiler [6]. We also show the impact of our techniques on the parallelization of five benchmark programs.

5.1 Implementation

5.1.1 Reorganization of the Phases in Polaris

```
scanner ;
inlining ;
interprocedural constant
propagation ;
for each program unit do
    program normalization ;
    induction variable sub-
    stitution ;
    constant propagation ;
    forward substitution ;
    dead code elimination ;
    privatization ;
    reduction recognition ;
    data dependence test ;
end do
postpass ;
```

(a) before

```
scanner ;
inlining ;
interprocedural constant
propagation ;
for each program unit do
    program normalization ;
    induction variable sub-
    stitution ;
    constant propagation ;
    forward substitution ;
    dead code elimination ;
end do
for each program unit do
    privatization ;
end do
for each program unit do
    reduction recognition ;
end do
for each program unit do
    data dependence test ;
end do
postpass
```

(b) after

Figure 15: Reorganization of the phases in Polaris

The high-level structure of the original Polaris compiler is shown in Figure 15(a). Except for the inlining and interprocedural constant propagation, all other phases are intraprocedural. For each program unit, Polaris performs a sequence of analyses and transformations in order. This structure is good for data locality and, therefore, good for the efficiency of Polaris. This organization is not appropriate for interprocedural array property analysis. It is better to apply the same set of transformations to all program units before the analysis starts. We, therefore, reorganize the phases in Polaris as shown in Figure 15(b). This "loop distribution" reorganization is possible because of the modularity of the phases implemented in Polaris.

We did not remove the inlining phase because most analyses in Polaris were not interprocedural and relied on inlining to produce precise results. We used the default auto inlining function in Polaris, which inlines procedures that contain no I/O statements and contain less than fifty lines of code. Because not all procedures are inlined, the interprocedural part of our array property analysis is still required and proved to be useful.

5.1.2 Single-indexed Array Access Analysis

Single-indexed array access analysis was implemented in two different places. One was in the array privatization phase to find consecutively-written arrays and the ranges of the array elements being written. The other was in the $PropetyChecker$ part to detect the properties of $injectivity$ and *having closed-form bounds*.

5.1.3 Array Property Analysis as a Demand-driven Tool

Array property analysis is not a stand-alone phase. It is implemented as an independent tool that can be invoked on demand.

The array property analyses that check different properties are implemented as different subclasses of a common $PropertySolver$ class which realizes the property independent $QuerySolver$ discussed in Section 3.2.2. The subclasses

implement the property dependent parts, such as the *PropertyChecker*. When an array property is to be checked, an object of one of the subclasses is created, array sections in the query are passed to the object, and then the analysis is invoked.

In Polaris, the array property analysis is used in the privatization phase and in the data dependence test phase.

5.1.4 Array Privatization

The privatization phase in Polaris privatizes arrays whose upward exposed read sets in each iteration are empty [30]. To compute the upward exposed read set, the sets of array elements that are read or written by each statement are calculated. A set of array elements is represented as an array section. To approximate in the safe direction, a read section can be a superset of its corresponding real read set, and a write section can be a subset of its corresponding real write set.

In the original design, the array subscripts must be linear expressions and the surrounding loops must be `do` loops. The array accesses cannot be irregular; otherwise, the read set has be to approximated to $[-\infty, \infty]$, and the write set to $\emptyset$.

We extended the computation method for the read/write sections so that it can handle the consecutively-accessed arrays and simple indirectly-accessed arrays. The methods described in [22] are used to get the ranges of the index variables in consecutively-accessed arrays. For the simple indirectly-accessed arrays, array property analysis is used to verify the bounds of the index arrays. A set of indirectly-read array elements now can be approximately represented in array sections. For example, $\{a(p(i)) | 1 \leq i \leq n\}$ is approximated to $a[low : high]$, where $low = min(p(i))$ and $high = max(p(i))$ for $(1 \leq i \leq n)$. Although this approximation works for read sets only, it has proven to be useful in our experiments.

5.1.5 Data Dependence Test

An important data dependence test used in Polaris is the *range test* [8], which is a symbolic data dependence test that can identify parallel loops in the presence of certain nonlinear array subscripts and loop bounds.

We extended the range test so that it could function like the *offset-length* test (discussed in Section 3.2.7) when the index arrays were used as offsets and length. We found the range test a natural place to incorporate the offset-length test because it also computed the symbolic range of subscript expressions which were used in the offset-length test. We also implemented a stand-alone *simple offset-length* test that tested the subscripts of the form "a(ptr(i)+j)". It could be used when the user wanted to avoid the overhead of the extended range test, though it was less general. An *injective* test was also added for the case when the subscript was a simple index array like "a(p(i))". All the extended and newly added tests need property analysis of index arrays.

5.2 Experimental Results

Table 2 shows the five programs used in our experiments. TRFD, BDNA and DYFESM are from the Perfect Benchmark suite. P3M is a particle-mesh program from NCSA. TREE is a Barnes-Hut N-body program from the University of Hawaii [4]. The compilation time of the programs using Polaris is listed in column four. The array property analysis increases the compilation time by 4.5% to 10.9%[3]. These data were measured on a Sun Enterprise 4250 Server with four 248MHz UltraSPARC-II processors. The sequential execution time of these programs (measured on an SGI Origin2000 with fifty-six 195MHz R10k processors) is listed in column three.

Table 3 shows the analysis results. Column two shows the loops with irregular array accesses that can be analyzed by Polaris now. The loops with a "*" are the newly parallelized loops. The loops without a "*" are not parallelized, but their analysis results are used to help parallelize the loops with a "*". The properties of the irregular array accesses are listed in columns five and eight. Column nine shows the tests that were used as the query generators in the array property analysis. Column ten shows the percentage of total sequential program execution time (on the Origin2000) accountable to the loops in column two. And, column eleven shows the percentage of total parallel program execution time accountable to these loops if the loops were not parallelized (the number after the % sign is the number of processors used). One to thirty-two processors were used.

Figure 16 shows the speedups of these programs. We compare the speedups of the programs parallelized by Polaris, with and without irregular array access analysis, and the programs compiled using the automatic parallelizer provided by SGI. DYFESM used a tiny input data set and suffered from the overhead introduced by parallelization. The performance of all three versions worsened when multiple processors were used (Fig. 16(e)). We also measured speedups on a slower SGI Challenge machine (four 200MHz R4400 Processors), and got a speedup of 1.6 (four processors) when the extra loops were parallelized (Fig. 16(f)). Loop INTGRL/do_140 in TRFD accounted for only 5% of the total sequential execution time. However, parallelizing it still increased the speedups from five to six when 16 processors were used (Fig. 16(a)). For BDNA, P3M and TREE, the speedups improved significantly.

6. CONCLUSION

Irregular array accesses are array accesses that do not have closed-form expressions of array subscripts. Traditional loop and array analysis methods cannot handle irregular array accesses, and many codes are left unoptimized. In this paper, we presented simple and effective techniques to analyze two common cases of irregular accesses: irregular single-indexed accesses and simple indirect array accesses. We also showed how to use the results of these analyses to enhance other analyses and optimizations. We demonstrated their effectiveness by measuring their impact in finding more implicit loop parallelism. Nine more loops in five programs were found parallel, and the speedups of four programs increased considerably after these loops were parallelized.

The techniques we presented are not silver bullets. They are based on the observation that, in real programs, irregular

[3]The data of P3M is for subroutine PP only.

	Lines of Codes	Sequential Program Execution Time (SGI Origin2000)	Polaris Execution Time (Sun 4250)		
			Whole Program	Array Property Analysis	%
TRFD	380	4.4s	181.3s	8.1s	4.5%
DYFESM	7650	3.2s	302.3s	19.4s	6.4%
BDNA	4896	9.7s	465.7s	31.2s	6.7%
P3M*	2414	355.8s	73.1s	8.0s	10.9%
TREE	1553	8.3s	25.7	1.71	6.7%

Table 2: Compilation time using Polaris. The fourth column shows the whole program compilation time. The fifth column is the time spent in array property analysis.

Program	Loops	Single-indexed Access			Indirectly Array Access				$\%_{seq}$	$\%_{par}$
		Array	Index	Property	Host	Index	Property	Test		
TRFD	INTGRL/do_140*	-	-	-	x	ia	CFV	DD	5%	$24\%_{32}$
DYFESM	SOLXDD/do_4* SOLXDD/do_10* SOLXDD/do_30* SOLXDD/do_50* HOP/do_20*	-	-	-	xdd, z r, y z xdd xdplus, xplus, xd	pptr, iblen	CFD	DD	20%	$7\%_{8}$
BDNA	ACTFOR/do_240*	-	-	-	xdt	ind	CFB	PRIV	32%	$63\%_{32}$
	ACTFOR/do_236	ind	l	CW	-	-	-	-	-	-
P3M	PP/do_100*	-	-	-	x0, ind0 r2, ind	jpr	CFB	PRIV	74%	$76\%_{8}$
	PP/goto_10 PP/do_50 PP/do_57	ind0, x0 ind0, x0 jpr	np0 np0 npr	CW CW CW	-	-	-	-	-	-
TREE	ACCEL/do_10*	stack	sptr	STACK	-	-	-	-	90%	$90\%_{32}$

Table 3: Programs used in our experiment. CW - consecutively written, STACK - stack access, CFV - closed-form value, CFB - closed-form bound, CFD - closed-form distance, PRIV - privatization test, DD - data dependence test.

array access often follows a few fixed patterns and have good properties. These techniques can be used together with user assertions and run-time tests to provide complete support to optimize codes with irregular memory accesses.

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.

7. REFERENCES

[1] Z. Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *PLDI '90*, pages 283–295, NY, June 1990.

[2] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism-enhancing transformations. In *PLDI'89*, pages 41–53, Portland, OR, June 1989.

[3] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976. Report No. 76-837.

[4] J. Barnes. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/. Technical report, Institute for Astronomy, University of Hawaii, 1994.

[5] W. Blume. *Symbolic analysis techniques for effective automatic parallelization.* PhD thesis, University of Illinois at Urbana-Champaign, June 1995.

[6] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[7] W. Blume and R. Eigenmann. An overview of symbolic analysis techniques needed for the effective parallelization of the perfect benchmarks. In *Proceedings of the 23rd International Conference on Parallel Processing. Volume 2: Software*, pages 233–238, Boca Raton, FL, August 1994. CRC Press.

[8] W. Blume and R. Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *Proceedings of the Conference on Supercomputing*, pages 528–537, Los Alamitos, November 1994. IEEE Computer Society Press.

[9] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, October 1988.

[10] B. Creusillet and F. Irigoin. Interprocedural array region analysis. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing (LCPC'95)*, volume 103 of *Lecture Notes in Computer Science*, pages 46–60. Ohio State University, Colombus (Ohio), August 1996.

[11] E. Duesterwald, R. Gupta, and M. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[12] P. Feautrier. Array expansion. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[13] M. Gerlek, E. Stoltz, and M. 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.

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

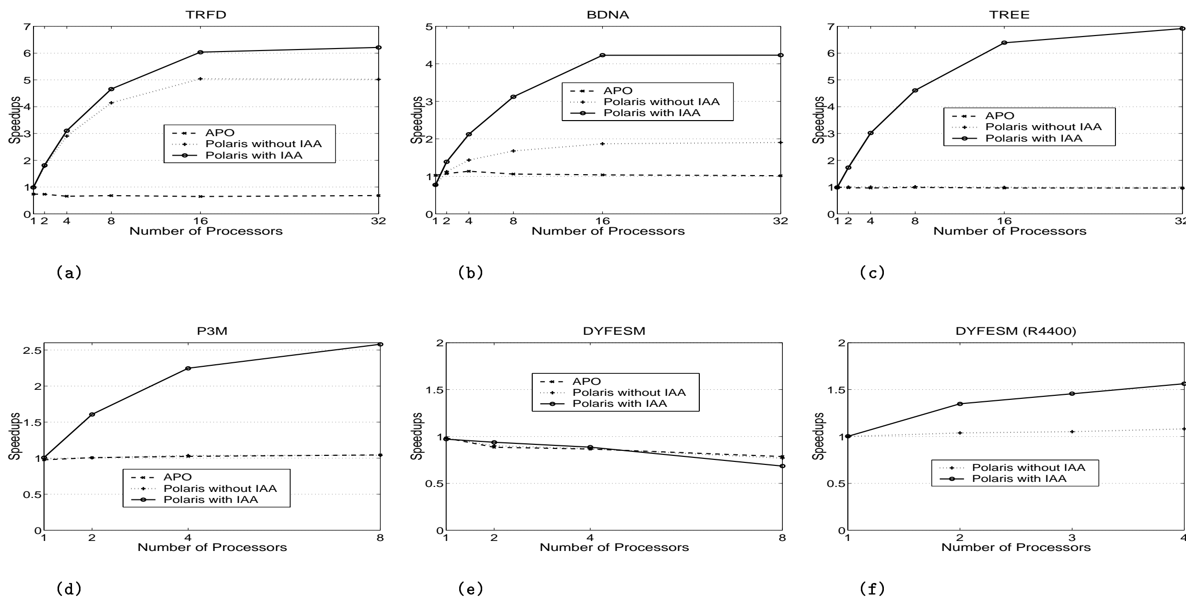

Figure 16: Speedups: IAA - irregular array access analysis, APO - using the automatic parallelization option in the SGI F77 compiler

[15] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Proceedings of the 1995 Supercomputing Conference, San Diego, CA*, 1995.

[16] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of the 1995 Supercomputing Conference, San Diego, CA*, 1995.

[17] P. Havlak. *Interprocedural Symbolic analysis*. PhD thesis, Rice University, May 1994.

[18] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 433–447, Cologne, Germany, June 1991.

[19] Z. Li. Array privatization for parallel execution of loops. In *Proceedings of 1992 International Conference on Supercomputing, DC*, pages 313–322, 1992.

[20] Y. Lin. *Compiler analysis of sparse and irregular computations*. PhD thesis, University of Illinois at Urbana-Champaign, May 2000.

[21] 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 (LCR'98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 41–56. Springer-Verlag, Pittsburgh, PA, 1998.

[22] Y. Lin and D. Padua. Analysis of irregular single-indexed array accesses and its applications in compiler optimizations. In *Proceedings of the 9th International Conference on Compiler Construction*, Berlin, March 2000.

[23] D. Maydan, S. Amarasinghe, and M. 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*, pages 2–15, January 1993.

[24] K. McKinley. Dependence analysis of arrays subscripted by index arrays. Technical Report TR91-162, Dept. of Computer Science, Rice University, June 1991.

[25] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimizations. In *PLDI'98*, pages 60–71, Montreal, Canada, June 1998.

[26] L. Rauchwerger. *Run-time parallelization: a framework for parallel computation*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[27] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[28] M. Spezialetti and R. Gupta. Loop monotonic statements. *IEEE Transactions on Software Engineering*, 21(6):497–505, June 1995.

[29] 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.

[30] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 500–521. Springer-Verlag, August 12–14, 1993.

[31] M. Wolfe. Beyond induction variables. In *PLDI'92*, pages 162–174, July 1992.

[32] C.-Q. Zhu and P.-C. Yew. A synchronization scheme and its applications for large multiprocessor systems. In *International Conference on Distributed Computing Systems*, pages 486–493, 1985.