Induction Variable Analysis Without Idiom Recognition: Beyond Monotonicity

Peng Wu, Albert Cohen, David Padua

Induction Variable Analysis without Idiom Recognition: Beyond Monotonicity

Peng Wu[1], Albert Cohen[2], and David Padua[3]

[1] IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
pengwu@us.ibm.com
[2] A3 Project, INRIA Rocquencourt
78153 Le Chesnay, France
Albert.Cohen@inria.fr
[3] Dept. of Computer Science, U. of Illinois
Urbana, IL 61801
padua@cs.uiuc.edu

Abstract. Traditional induction variable (IV) analyses focus on computing the closed form expressions of variables. This paper presents a new IV analysis based on a property called *distance interval*. This property captures the value changes of a variable along a given control-flow path of a program. Based on distance intervals, an efficient algorithm detects dependences for array accesses that involve induction variables. This paper describes how to compute distance intervals and how to compute closed form expressions and test dependences based on distance intervals.

This work is an extension of the previous induction variable analyses based on *monotonic evolution* [11]. With the same computational complexity, the new algorithm improves the monotonic evolution-based analysis in two aspects: more accurate dependence testing and the ability to compute closed form expressions.

The experimental results demonstrate that when dealing with induction variables, dependence tests based on distance intervals are both efficient and effective compared to closed-form based dependence tests.

1 Introduction

Dependence analysis is useful to many parallelization and optimization algorithms. To extract dependence information, array subscripts must be compared across statements and loop iterations. However, array subscripts often include variables whose value at each loop iteration is not easily available. An important class of such variables are *induction variables*.

In classical dependence analyses, occurrences of induction variable are often replaced by their closed form expressions. Since most dependence tests handle affine expressions only, this approach only applies to induction variables with affine closed form expressions. To handle more general induction variables, in our previous work, we proposed a dependence test based on a light-weight IV

```
1     do i = 0,n
2        do j = 0,m
3           k = k+1
4           a(k) = ···
5        end do
6     end do
```

Fig. 1. Non-affine closed form expression

```
1  do i = 1,10
2     ···
3     a(k) = ···
4     ···= a(k+1)
5     k = k+2
6  end do
```

Fig. 2. IV incremented by step of 2

property called *monotonic evolution* [11]. In essence, monotonic evolution captures whether the value of a variable is increasing or decreasing along a given execution path. For example, consider the loop nest in Fig. 1 where m is not a compile time constant. The closed form expression of `k` is not affine. However, knowing that the value of `k` at statement 4 is strictly increasing, one can prove that statement 4 is free of output-dependences.

Nevertheless, there are cases where monotonic evolution is not sufficient for accurate dependence testing. Consider the loop in Fig. 2. Knowing that the value of `k` is strictly increasing is not enough. A dependence test needs to know that the value of `k` increases by a minimum of 2 to determine statement 3 and 4 as dependence-free. To obtain such additional information, this paper extends the concept of monotonic evolution to *distance interval*, which captures the minimal and maximal value changes of a variable along any given execution path. We also extend the algorithms in [11] to compute distance intervals and to perform dependence tests based on distance intervals. In addition, we present a method to compute closed form expressions from distance intervals.

Experimental results show that when dealing with induction variables, dependence tests based on distance intervals are both efficient and effective compared to closed-form based dependence tests (implemented in Polaris). In particular, our technique misses three loops that can be parallelized by Polaris, but finds 74 more parallel loops than can Polaris.

The rest of the paper is organized as follows. Section 2 gives an overview of monotonic evolution. Section 3 defines distance interval. Section 4 and 5 describe how to use distance intervals in dependence testing and closed form computation. Section 6 proposes a technique to handle IVs defined by arbitrary assignments. Section 7 presents the experimental results. Section 8 compares our technique with others, and Section 9 concludes.

2 Overview of Monotonic Evolution

Monotonic evolution of a variable describes the *direction* in which the value of the variable changes along a given execution sequence. Possible values of an evolution are described by the lattice of *evolution states* as shown in Fig. 3. We define two types of evolutions:

- The notation $p \dashv\!\!\!\downarrow_{\mathtt{i}}^{N} q$ represents the *join* ($\sqcup$) of the evolution of `i` over all paths that starts from p and ends at q *excluding* those that traverse any edge

in the set N. Intuitively, $p \dashv_{\mathtt{i}}^{N} q$ captures how the value of `i` changes when the program executes from an instance of `p` to an instance of `q`. When q can not be reached from p, $p \dashv_{\mathtt{i}}^{N} q$ is $\bot$.
- The notation $p \dashv_{\mathtt{i}}^{N} r \dashv_{\mathtt{i}}^{N} q$ represents an evolution that must traverse an intermediate node, i.e., the evolution of i along all paths from p *via* r to q, excluding those that traverse any edge in N.

Lattice elements:

- $\top$ unknown evolution;
- $\trianglelefteq$ monotonically increasing;
- $\triangleleft$ strictly monotonically increasing;
- $\trianglerighteq$ monotonically decreasing;
- $\triangleright$ strictly monotonically decreasing;
- $\diamond$ constant evolution;
- $\bot$ no evolution.

Ordering:

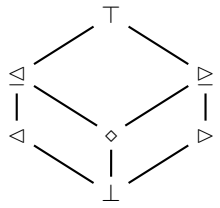


Fig. 3. The lattice of evolution states

To compute the value of an evolution, each statement in the program is interpreted as a transfer function of evolution values. Given a variable `i`, a statement is classified as: *identity statement* if it does not change the value of `i`, such as `j = n`; *forward induction* if it always increases the value of `i`, such as `i = i+1`; *backward induction* if it always decreases the value of `i`, such as `i = i-3`; *arbitrary assignment* if it assigns any value to `i`, such as `i = n`. The corresponding transfer functions are given in Table 1.

	$\bot$	$\triangleleft$	$\trianglelefteq$	$\diamond$	$\trianglerighteq$	$\triangleright$	$\top$
Identity	$\bot$	$\triangleleft$	$\trianglelefteq$	$\diamond$	$\trianglerighteq$	$\triangleright$	$\top$
Forward	$\bot$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\top$	$\top$	$\top$
Backward	$\bot$	$\top$	$\top$	$\triangleright$	$\triangleright$	$\triangleright$	$\top$
Arbitrary	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Table 1. Transfer functions of evolution values

3 Distance-extended Evolution

This section defines distance interval and its operations, and describes the algorithm to compute distance interval.

3.1 Distance Interval

A distance interval captures the minimal and maximal value changes of a variable along a given execution sequence. More precisely, for any a and b such that $-\infty \leq a \leq b \leq +\infty$, $[a, b]$ describes any evolution where the value difference of the variable at the starting and ending nodes of the evolution is *no less* than a and *no greater* than b. When $a = b$, $[a, b]$ is *exact*. In this case, we may use the shorter a for $[a, a]$. The lattice of distance intervals is formally defined in Table 2. $\perp$ describe unreachable evolutions. For example, consider the loop in Figure 2, we have $5 \dashv_{\mathtt{k}} 5 = [2, 18]$.

$\sqcup$	$\perp$	$[a, b]$
$\perp$	$\perp$	$[a, b]$
$[c, d]$	$[c, d]$	$[\min(a, c), \max(b, d)]$

Table 2. Distance-extended lattice

A distance interval can always be mapped to an evolution state according to the signs of the interval's bounds. For instance, $[0, 0]$ corresponds to $\diamond$; $[a, b]$ corresponds to $\trianglelefteq$ when $a \geq 0$, to $\trianglerighteq$ when $b \leq 0$, and to $\top$ when a and b are of opposite signs.

3.2 Distance Intervals of Expressions

Distance intervals can be computed for expressions, i.e., $p \dashv_e^N q$ where e is an arithmetic expression. We define two operations, "$\times$" and "$+$", on distance intervals in Table 3 and Table 4, respectively.

$\times$	$a\ (> 0)$	0	$a\ (< 0)$
$\perp$	$\perp$	$[0, 0]$	$\perp$
$[c, d]$	$[ac, ad]$	$[0, 0]$	$[ad, ac]$

Table 3. The $\times$ operator

$+$	$\perp$	$[a, b]$
$\perp$	$\perp$	$\perp$
$[c, d]$	$\perp$	$[a + c, b + d]$

Table 4. The $+$ operator

The rules to compute evolution of expressions are as follows. When e is a constant expression, $p \dashv_e^N q = [0, 0]$ if q is reachable from p; otherwise, $p \dashv_e^N q = \perp$. When e is of the form ae_1 where a is a constant, $p \dashv_{ae_1}^N q = p \dashv_{e_1}^N q \times a$. Lastly, when e is of the form $e_1 + e_2$, $p \dashv_{e_1 + e_2}^N q = p \dashv_{e_1}^N q + p \dashv_{e_2}^N q$.

For example, suppose that e is `2i-3j+6`,

$$p \dashv_{2i - 3j + 6}^N q = p \dashv_{\mathtt{i}}^N q \times 2 + p \dashv_{\mathtt{j}}^N q \times (-3) + [0, 0].$$

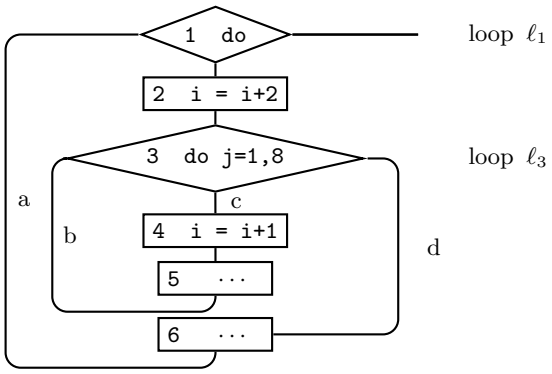


Fig. 4. Stride information

3.3 Stride Information

We define a special evolution, called *stride*, that traverses *at most* one iteration of a loop. Consider a loop ℓ with a header h and a statement p enclosed in ℓ. We define three strides between p and h:

Up$_{\mathtt{i},N}(p,h)$ denotes an evolution from p *up* to the first h reached excluding edges in N. It is called an *up-stride* of p.

Down$_{\mathtt{i},N}(h,p)$ denotes an evolution from h *down* to p *without* traversing h twice, excluding edges in N. It is called a *down-stride* of p.

Stride$_{\mathtt{i},N}(\ell)$ denotes an evolution from h to the next h, excluding edges in N and the exit edge of ℓ. It is called a *stride* of loop ℓ. In fact, $\mathrm{Stride}_{\mathtt{i},N}(\ell)$ is a special case of $\mathrm{Down}_{\mathtt{i},N}(h,p)$ and $\mathrm{Up}_{\mathtt{i},N}(p,h)$ when $p = h$.

Let us illustrate these definitions on the example shown in Fig. 4 where nodes are named by statement labels.

- $\mathrm{Stride}_{\mathtt{i}}(\ell_3) = [1,1]$ since it traverses exactly one iteration of the inner loop;
- $\mathrm{Stride}_{\mathtt{i}}(\ell_1) = [10,10]$ since it traverses statement 2 and the entire inner loop exactly once (knowing ℓ_3 has 8 iterations);
- $\mathrm{Down}_{\mathtt{i}}(3,5) = [1,1]$ since it traverses the single path from statement 3 to the first statement 5 reached;
- $\mathrm{Up}_{\mathtt{i}}(5,1) = [0,7]$ since statement 4 is traversed at most 7 times.

3.4 Computing Distance Interval

The non-iterative algorithm described in [11] can be extended to compute distance intervals of evolutions. Due to space constraints, we only briefly describe the algorithm here. The full algorithm can be found in [12].

The core of the non-iterative algorithm is the basic algorithm, which is based on a depth-first traversal of a non-cyclic control-flow graph. For each statement

traversed, it applies the transfer function as specified below. Given a variable `i`, a statement is classified as: an *identity statement* if it does not change the value of `i`, such as `j = n`; an *induction with a step* c if it always changes the value of `i` by c, such as `i = i+c`, where c could be a runtime constant of any sign; an *arbitrary assignment* if it may assign any value to `i`, such as `i = n`. The corresponding transfer functions for these statements are:

- $\mathsf{Identity}(state) = state$
- $\mathsf{Induction}_c(state) = state + [c, c]$
- $\mathsf{Arbitrary}(state) = [-\infty, +\infty]$

Then, the algorithm decomposes any evolution into segments, each of which can be computed by the basic algorithm. In particular, to compute $p \dashv_{\mathtt{i}}^{N} q$, the evolution is decomposed into an up-stride of p, a down-stride of q, and some strides of the surrounding loops of p and q. Since evolutions are decomposed into the same segments over and over again, we can reuse values of intermediate segments to compute different evolutions. This leads to a very efficient algorithm to compute multiple evolutions.

4 Dependence Test Using Distance Information

Distance intervals can be used for dependence testing. Such a dependence test shares some similarities with the range test [2].

4.1 Dependence Test

Consider two array accesses, `a(i)` at p and `a(i+d)` at q, where `d` is a constant.[1] The dependence test computes the value difference between `i` at p and `i`+d at q, and use this information to decide whether the two accesses are independent. It is obvious that, at any given run-time program point, the value of `i` will differ from that of `i+d` by d. Therefore, the value difference between `i` at p and `i`+d at q can be computed as the evolution of `i` from p to q summed with $[d, d]$.

Consider a loop ℓ, two accesses, `a[`e`]` at p and `a[`e`+`d`]` at q. Let B and E denote the sets of back-edges and exit edges of ℓ, respectively.

1. There is *no* intra-loop dependence between p and q for loop ℓ when

$$p \dashv_e^B q + [d, d] \in \{[a, b] \mid ab > 0\} \\ \wedge\ q \dashv_e^B p + [-d, -d] \in \{[a, b] \mid ab > 0\}. \quad (1)$$

2. There is *no* loop-carried dependence between p and q for loop ℓ with a header h when

$$p \dashv_e^E h \dashv_e^E q + [d, d] \in \{[a, b] \mid ab > 0\} \\ \wedge\ q \dashv_e^E h \dashv_e^E p + [-d, -d] \in \{[a, b] \mid ab > 0\}. \quad (2)$$

```
1    do i = 1,100
2       k = k+2
3       t = a(k)
4       a(k) = a(k+1)
5       a(k+1) = t
6    end do
```

Fig. 5. Example of dependence test

Consider the loop in Fig. 5 that swaps every pair of consecutive elements of array `a`. Since `k` is incremented by 2 per iteration, we have,

$$3 \dashv_{\mathtt{k}} 1 \dashv_{\mathtt{k}} 5 + [1,1] = [2,2] + [1,1] = [3,3]$$
$$5 \dashv_{\mathtt{k}} 1 \dashv_{\mathtt{k}} 3 + [-1,-1] = [2,2] + [-1,-1] = [1,1].$$

According to (2), this proves that there is no loop-carried dependence between 3 and 5.

4.2 Practical Computation

The dependence test computes two evolutions for any pair of accesses and for each surrounding loop to be tested (e.g., from p to q, and from q to p). Obviously, computation will not be efficient without optimizing computations across different evolutions. We propose to cache and reuse intermediate evolutions. We can compute and tabulate the results for each statement and loop to be tested. To further optimize the algorithm, for each basic block, we compute local evolutions that traverse an entire block, and store the results. During later computation, the algorithm may "short-cut" the basic block by summing its cached local state with the input state. The full algorithm is described in [12].

4.3 Complexity Analysis

Since dependence tests are local to individual loop nests, we consider an arbitrary loop nest L and an induction variable `i`. Let e be the number of edges in L, and m be the maximal nesting of L. Suppose that k statements in L are involved in the dependence test. The dependence test computes $p \dashv_{\mathtt{i}}^{N} h \dashv_{\mathtt{i}}^{N} q$ and $p \dashv_{\mathtt{i}}^{N} q$, for all possible p, q and h, where N may only contain back-edges and exit edges. In fact, when computing intermediate evolutions, we can drop N and explicitly compute those evolutions for each loop. We showed that the dependence test complexity is

$$O(ek + m^2k^2). \tag{3}$$

Any flow-sensitive, statement-wise dependence test for k statements in a loop nest of depth m must take at least mk^2 steps, our test is no exception. In our

[1] Accesses of the form `a[i+`d_1`]` and `a[i+`d_2`]` can be handled as `a[j]` and `a[j+(`$d_2 - d_1$`)]`.

scheme, dependency is tested individually for each loop of a nest (as reflected by the occurrence of m in (3)). Therefore, compared to classical dependence tests *without induction variable recognition*, our scheme requires more steps.[2] However, (3) estimates the number of operations (i.e., $\times$, $+$, and $\sqcup$) involved in the dependence test: the formula gives a fairly accurate account of the cost of the test. On the other hand, for classical dependence tests, the cost of individual operations is difficult to estimate. Depending on the mathematical tools employed, some operations may be as expensive as solving a system of linear equations.

5 Closed Form Computation

Although our dependence test requires no closed form computation, closed form expressions are still needed by subsequent loop transformations to break the dependence inherent to inductions. Distance intervals can be used to compute closed form expressions.

Given a variable v and a statement p, the closed form expression of v at p explicitly computes the value of v at any instance of p. Suppose that p is enclosed in a loop nest, $\ell_n, \ldots, \ell_1, \ell_0$, where ℓ_0 immediately encloses p. Assume that all loop indices have an initial value of 1 and a step of 1. Let i_k denote the loop index of any loop ℓ_k, and v_k denote the value of v before entering ℓ_k. The closed form computation is conducted in the following steps:

- First, we compute the closed form expression of v at p for loop ℓ_0. If $\mathrm{Stride}_{\mathrm{v}}(\ell_0)$ and $\mathrm{Down}_{\mathrm{v}}(\ell_0, p)$ are exact,[3] the value of v at p at iteration i_0 of loop ℓ_0 can be expressed as

$$v = v_0 + \mathrm{Stride}_{\mathrm{v}}(\ell_0) \times (i_0 - 1) + \mathrm{Down}_{\mathrm{v}}(\ell_0, p). \tag{4}$$

- Then, consider ℓ_1 that immediately enclose ℓ_0. Applying (4) again, v_0 can be computed as

$$v_0 = v_1 + \mathrm{Stride}_{\mathrm{v}}(\ell_1) \times (i_1 - 1) + \mathrm{Down}_{\mathrm{v},O_0}(h_1, h_0)$$

  where O_0 is the set of outgoing edges of h_0. Basically, $\mathrm{Down}_{\mathrm{v},O_0}(h_1, h_0)$ computes the evolution from h_1 down to the first h_0. Replacing v_0 in (4) by the above equation, the closed form expression of v for ℓ_1 and ℓ_0 is

$$\begin{aligned} v = v_1 + \mathrm{Stride}_{\mathrm{v}}(\ell_1) \times (i_1 - 1) + \mathrm{Down}_{\mathrm{v},O_0}(h_1, h_0) \\ + \mathrm{Stride}_{\mathrm{v}}(\ell_0) \times (i_0 - 1) + \mathrm{Down}_{\mathrm{v}}(h_0, p). \end{aligned} \tag{5}$$

[2] m times, when testing a large number of array accesses, i.e., when k is close to e.
[3] This ensures that (4) indeed computes a singleton interval.

```
1    k = 1
2    do i = 1, 10
3       do j = 1, 10
4          k = k+2
5          a[k] = ···
6       end do
7    end do
```

Fig. 6. Closed form computation

– Finally, generalizing (4) and (5), the closed form expression of v at p for any loop nest $\ell_n, \ldots, \ell_0$ is

$$\begin{aligned} v = v_n &+ \mathrm{Stride}_{\mathtt{v}}(\ell_1) \times (i_1 - 1) + \mathrm{Down}_{\mathtt{v},O_0}(h_1, h_0) \\ &+ \mathrm{Stride}_{\mathtt{v}}(\ell_2) \times (i_2 - 1) + \mathrm{Down}_{\mathtt{v},O_1}(h_2, h_1) \\ &+ \cdots + \mathrm{Stride}_{\mathtt{v}}(\ell_n) \times (i_n - 1) + \mathrm{Down}_{\mathtt{v},O_{n-1}}(h_n, h_{n-1}) \\ &+ \mathrm{Stride}_{\mathtt{v}}(\ell_0) \times (i_0 - 1) + \mathrm{Down}_{\mathtt{v}}(h_0, p) \end{aligned} \quad (6)$$

where O_k is the set of outgoing edges of loop ℓ_k and provided that $\mathrm{Stride}_{\mathtt{v}}(\ell_k)$, $\mathrm{Down}_{\mathtt{v},O_0}(h_k, h_{k-1})$, and $\mathrm{Down}_{\mathtt{v}}(h_0, p)$ are exact.

For example, consider the loop nest in Fig. 6. Let O denote the exit-edge of loop 3. Applying (6), the closed form expression of k at 5 is

$$\begin{aligned} k = 1 &+ \mathrm{Stride}_{\mathtt{k}}(\ell_2) \times (i - 1) + \mathrm{Down}_{\mathtt{k},O}(2, 3) \\ &+ \mathrm{Stride}_{\mathtt{k}}(\ell_3) \times (j - 1) + \mathrm{Down}_{\mathtt{k}}(3, 5). \end{aligned}$$

Hence, $k = 1 + 20(i-1) + 2(j-1) + 2 = 20(i-1) + 2(j-1) + 3$.

6 Handling Arbitrary Assignments

The transfer function of arbitrary assignment given in Section 3.4 conservatively maps any input state to $[-\infty, +\infty]$. We would like to provide a more precise transfer function for arbitrary statements.

Consider an assignment s of the form `i = j`. Suppose that $i < j$ holds at any statement instance of s, then the value of i always increases after an execution of s. This means that the effect of s on i is equivalent to that of an induction statement (with a positive step). Therefore, we define the transfer function of s, denoted as f_s, according to the inequality between i and j at s: if $c \leq i - j \leq d$ at s, then

$$f_s(in) = in + [c, d]. \quad (7)$$

In order to obtain $[c, d]$, we need to estimate the bounds of $i - j$ at s. Obviously, i and j have the same value immediately after s, hence after denoted as

```
1  do i = 1,100
2     k = n
3     ···
4     do j = 1,10
5        a[k] = ···
6        k = k+1
7     end do
8     n = n + 11
9  end do
```

Fig. 7. Example of arbitrary assignment

s^+. Therefore, s^+ can be used as a reference point to compare the values of i and j at s.

Let $[a, b]$ (resp. $[a', b']$) denote the evolution of i (resp. j) from an instance of s^+ to the instance of s from the very next iteration of ℓ_s. We assume that s is executed at every iteration of ℓ_s. This condition can be checked as whether h_s can reach itself without traversing incoming edges of s and exit edges of ℓ_s. Then, $[a, b]$ and $[a', b']$ can be computed as follows:

$$[a, b] = \mathrm{Up}_{\mathtt{i}}(s^+, h_s) + \mathrm{Down}_{\mathtt{i}}(s, h_s) \qquad [a', b'] = \mathrm{Up}_{\mathtt{j}}(s^+, h_s) + \mathrm{Down}_{\mathtt{j}}(s, h_s).$$

Knowing $i = j$ at s^+, the difference between values of i and j, at any instance of s executed after an instance of s^+, is bounded by the "difference" between $[a, b]$ and $[a', b']$:

$$c \leq i - j \leq d \text{ where } [c, d] = [a - b', b - a']. \tag{8}$$

Since s is executed at every iteration of ℓ_s, any instance of s executed after the first iteration of ℓ_s follows some instance of s^+. When computing evolutions, (8) holds at node s only after a back-edge has been traversed along the path.

We now apply the method to compute $5 \dashv_{\mathtt{k}} 5$ in Fig. 7, where statement 2 is an arbitrary assignment. Corresponding distance intervals are computed as

$$[a, b] = \mathrm{Up}_{\mathtt{k}}(3, 1) + \mathrm{Down}_{\mathtt{k}}(1, 2) = [10, 10]$$
$$[a', b'] = \mathrm{Up}_{\mathtt{n}}(3, 1) + \mathrm{Down}_{\mathtt{n}}(1, 2) = [11, 11].$$

Since any path from statement 5 to 2 always traverses the back-edge of loop 1 first, $1 \leq k - n \leq 1$ holds at each traversal of 2 along paths of $5 \dashv_{\mathtt{k}} 5$. Hence, applying (7), the transfer function of statement 2 is $f_2(in) = in + [1, 1]$.

7 Experimental Results

For our experimental studies, we used Polaris [3], a Fortran source-to-source parallelizing compiler, as the basis for comparison. In Polaris, induction variables are substituted by their closed form expressions before the dependence test is performed. In the context of dependence testing for array accesses, we focus on

integer induction variables (IIVs) which are used in *array subscripts*, and we do not deal with IIVs unrelated to any dependences, e.g., IIVs used in subscripts for arrays that only appear in right-hand side.

In the experiment, we used Polaris to find candidate IIVs from the Perfect Club benchmark suite. Applying our dependence test by hand (for dependences involving IIVs) and using the dependence information reported by Polaris (for other dependences), we detected parallel loops involving IIVs. Table 5 presents the experimental results.[4] The first three columns classify loops with IIVs into three sets: loops containing IIVs (Total); loops where IIVs appear as subscripts (Subscript); and loops where the analysis of IIVs is required for parallelization (Targeted), that is, loops that are the target of our technique. The next five columns give the number of loops with IIVs parallelized by different techniques: by Polaris (Polaris), by our dependence analysis with either the original (Monotonic) or the distance-extended (w/ Distance) lattice, combined with the method to handle arbitrary assignments (w/ Assign), combined with a run-time test for stride and loop bounds (w/ Test). Note that, in columns Monotonic and w/Distance, a loop counted as parallel simply means that when disabling IV substitution in Polaris and "plugging in" our analysis, Polaris reports no loop-carried dependence for the loop except for those due to assignments to IVs themselves. Such dependences can be handled either by finding closed form expressions and performing the substitution, or by the techniques described in the next paragraph.

	Loops with IIVs			Parallel Loops with IIVs				
	Total	Subscript	Targeted	Polaris	Monotonic	w/ Distance	w/ Assign	w/ Test
`adm`	17	17	5	3	2	3	3	4
`bdna`	63	62	60	22	34	34	34	34
`dyfesm`	15	11	8	7	8	8	8	8
`flo52`	15	15	15	12	12	12	12	12
`mdg`	29	29	24	14	12	13	13	16
`mg3d`	97	97	89	5	5	5	39	58
`ocean`	11	6	4	4	4	4	4	4
`qcd`	69	69	69	58	64	64	64	64
`spec77`	99	59	54	44	1	44	44	44
`trfd`	13	13	9	7	6	6	7	7

Table 5. Experiments with the Perfect Club benchmark suite

Let us comment on the results. Our dependence test matches or outperforms Polaris on all loops with IIVs but one (in `mdg`). We discovered 74 new loops whose only dependences came from operations on induction variables themselves. Among them, 56 (1 in `adm`, 1 in `mdg`, 53 in `mg3d` and 1 in `qcd`) do have

[4] Three programs have been omitted: `arc2d` and `track` because they contain no loop with IIVs, and `spice` because it could not be handled by Polaris.

closed form expression (but the dependence test in Polaris failed to handle these closed form expressions). Twelve (11 in `bdna` and 1 in `dyfesm`) have no closed form expressions because the loop bounds involve array references; but they can be parallelized without much overhead, using a parallel reduction scheme. The other six (1 in `bdna` and 5 in `qcd`) involve conditional induction variable updates; one may resort to a more general `doacross` technique to parallelize such loops: the loop body is split into a "head" sequential part for induction variable computation and a "tail" part which can be run in parallel with the next iteration.

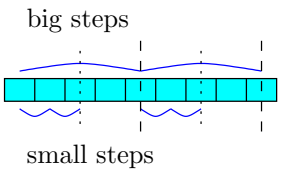
Notice that unknown symbolic constants (for loop bounds and induction variable strides) are sometimes a reason for unsuccessful parallelization by Polaris. Using our technique, a run-time test is inserted to check for inequalities assumed during monotonic evolution and dependence testing.

7.1 Additional Patterns that can be handled

This section describes the patterns that can be handled by our method, in addition to the four patterns already described in [11].

Pattern 5: monotonic small- and big-step. In this case, variable `mrsij` is incremented by a "small" step (1) in every iteration of the inner loop, and is re-assigned to the value of `mrsij0` in every iteration of the outer loop. Variable `mrsij0` itself is an induction variable incremented by a "big" step `nrs` by the outer loop. As opposed to the stride and offset pattern, proving there is no dependence requires comparing the accumulative effect of the "small" step of the *inner* loop—which usually depends on the bounds of the inner loop and the step—with the big step of the *outer* loop.

```
— trfd olda do100 — line 331 —
do mrs = 1,nrs
   ...
   mrsij = mrsij0
   do mi = 1,morb
      ...
      do mj = 1,mi
         mrsij = mrsij+1
         xrsij(mrsij) = xij(mj)
      end do
   end do
   mrsij0 = mrsij0+nrs
end do
```



Polaris detects no dependences carried by any of the loops because the closed form expression of `mrsij` yielded disjoint intervals [2]. This is illustrated on the right-hand side figure: there are no dependences carried by the outer loop because the dotted lines—the last value of `mrsij`—always precede the dashed ones—the next value of `mrsij0`. Our technique may parallelize the two inner loops based on the strict monotonicity of `mrsij`. Using the dedicated technique to handle

arbitrary assignments (`mrsij = mrsij0`) and the distance-extended lattice, the outer loop may also be parallelized. We found 35 loops (1 in `trfd` and 34 in `mg3d`) share this pattern.

7.2 Patterns that Could not be Handled

The following nests illustrate the two most common cases where our technique could not successfully detect parallel loops.

Pattern *a*: complex small- and big-step The following nest is similar to pattern 5, but induction variables appear in loop bounds instead of array accesses. Neither Polaris nor our technique can parallelize the outer loop. Nevertheless, it should not be difficult to extend the arbitrary assignment method to loop counter assignments, and detect that array accesses span disjoint regions across iterations of the outer loop.

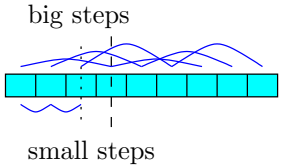
— mdg nrmlkt do300 — line 494 —

```
do j = 1,3
   ...
   kmin = kmax+1
   kmax = kmax+natmo
   do k = kmin,kmax
      var(k) = var(k) * ···
   end do
end do
```

Pattern *b*: interleaved big- and small-step Our distance-extended lattice handles complex combinations of offsets and strides spanning multiple loops, as long as offsets are explicit in every reference. In many benchmarks, a comparison is required between the stride of an *inner* loop and an *outer* loop bound (the opposite of the previous pattern).

— mdg correc do1000 — line 989 —

```
do i = 1,nt
   ...
   jj = i
   do j = 1,nor1
      var(jj) = var(jj) + ···
      jj = jj+nt
   end do
end do
```



To parallelize the outer loop, one has to show that `i`—hence the initial value of `jj`—is always greater than 0 and less than or equal to `nt`. This is illustrated on the right-hand side figure: there are no dependences carried by the outer loop because the dotted line—the greatest possible value of `i`—precedes the dashed one—the stride of `jj`. On this example, our improvement to handle arbitrary assignments is not very helpful: values of `jj` are interleaved across iterations of the outer loop. We found 20 loops sharing this pattern in the perfect benchmarks (1 in `bdna` and 19 in `mg3d`). Polaris cannot handle this pattern either.

8 Related Work

Most induction variable analyses focus on idiom recognition and closed form computation. Using patterns proposed by Pottenger and Eigenmann [10], the Polaris compiler recognizes polynomial sequences that are not limited to scalar and integer induction variables. Abstract interpretation is used by Ammarguellat and Harrison [1] to compute symbolic expressions. Two general classification techniques have been designed. The first one [6] by Gerlek, Stoltz and Wolfe is based on a SSA representation [5] optimized for efficient demand-driven traversals. The second one [8] is designed by Haghighat and Polychronopoulos for the Parafrase 2 compiler. It combines symbolic execution and recurrence interpolation. Both techniques handle a broad scope of closed form expressions, such as linear, arithmetic (polynomial), geometric (with exponential terms), periodic, and wrap-around.

IV properties other than closed form expressions have also been studied. Gupta and Spezialetti [7] extended the linear IV detection framework with arithmetic and geometric sums as well as *monotonic* sequences, but for non-nested loops only. Their technique is applied to efficient run-time array bounds checking. Lin and Padua [9] studied monotonicity for values of index arrays in the context of parallelizing irregular codes. This property is used later to detect dependences between accesses to sparse matrices through index arrays. However, their technique does not target general induction variables. Gerlek, Stoltz and Wolfe [6] also detect *monotonic* sequences as a special class of induction variables. But details were not provided as how to use such information in dependence testing.

9 Conclusion and Future Work

We presented an extension of our previous work [11] on using monotonic evolution to test dependence for array subscripts that involve induction variables. It is a natural step to extend monotonic evolution states with the minimal and maximal distance information. Distance interval enables precise dependence testing in presence of interleaved variable assignments, symbolic constants, evolutions between different variables, non-monotonic evolutions, and closed form computation. In the experiment carried out with the Perfect benchmarks, we showed that our technique matches the precision of Polaris when closed forms are available, and when there are no closed form expressions, we can still detect additional parallel loops.

The immediate future work is to implement this technique in Polaris and validate its use for fast dependence testing. Since arbitrary assignments link the values of two variables, they may be used as reference points to relate (compare) values of different variables. We would also like to apply monotonic evolution on other forms of induction operations, such as pointer chasing in recursive data structures and *container* traversals through *iterators* [4], either for pointer analysis or for parallization. Monotonic evolution is well-suited for dynamic structures since traversals of such structures are likely to be monotonic, and closed form abstractions are impractical for such accesses.

References

1. Z. Ammarguellat and W.L. Harrison. Automatic recognition of induction & recurrence relations by abstract interpretation. In *ACM Symp. on Programming Language Design and Implementation (PLDI'90)*, pages 283–295, Yorkton Heights, NY, June 1990.
2. W. Blume and R. Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *Supercomputing'94*, pages 528–537, Washington D.C., November 1994. IEEE Computer Society Press.
3. W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.
4. A. Cohen, P. Wu, and D. Padua. Pointer analysis for monotonic container traversals. Technical Report CSRD 1586, University of Illinois at Urbana-Champaign, January 2001.
5. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.
6. M.P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. on Programming Languages and Systems*, 17(1):85–122, January 1995.
7. R. Gupta and M. Spezialetti. Loop monotonic computations: An approach for the efficient run-time detection of races. In *ACM Symp. on Testing Analysis and Verification*, pages 98–111, 1991.
8. M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 18(4):477–518, July 1996.
9. Y. Lin and D. Padua. Compiler analysis of irregular memory accesses. In *ACM Symp. on Programming Language Design and Implementation (PLDI'00)*, Vancouver, British Columbia, Canada, June 2000.
10. B. Pottenger and R. Eigenmann. Parallelization in the presence of generalized induction and reduction variables. In *ACM Int. Conf. on Supercomputing (ICS'95)*, June 1995.
11. P. Wu, A. Cohen, D. Padua, and J. Hoeflinger. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *ACM Int. Conf. on Supercomputing*, Sorrento, Italy, June 2001.
12. Peng Wu. Analyses of pointers, induction variables, and container objects for dependence testing. Technical Report UIUCDCS-R-2001-2209, University of Illinois at Urbana-Champaign, May 2001. Ph.D Thesis.