Dependence Testing Without Induction Variable Substitution

Albert Cohen [†] Peng Wu [‡]

[†] A3 Project, INRIA Rocquencourt
78153 Le Chesnay, France
Albert.Cohen@inria.fr

[‡] Department of Computer Science, University of Illinois
Urbana, IL 61801, USA
pengwu@cs.uiuc.edu

Abstract

We present a new approach to dependence testing in the presence of induction variables. Instead of looking for closed form expressions, our method computes *monotonic evolution* which captures the direction in which the value of a variable changes. This information is used for dependence testing of array references. Under this scheme, closed form computation and induction variable substitution can be delayed until after the dependence test and be performed on-demand. The technique can be extended to dynamic data structures, using either pointer-based implementations or standard object-oriented containers. To improve efficiency, we also propose an optimized (non-iterative) data-flow algorithm to compute evolution. Experimental results show that dependence tests based on evolution information match the accuracy of that based on closed-form computation (implemented in Polaris), and when no closed form expressions can be calculated, our method is more accurate than that of Polaris.

1 Introduction

Many optimizations are based on induction variable (IV) detection and classification, including loop transformations for parallelization and strength reduction. An induction variable is broadly defined as a scalar variable (or array element) referenced in a cycle of a def-use graph. Although production compilers only implement well-known techniques for *linear* IVs [1], more general IV classes have been proposed for parallelizing compilers [7].

In classical dependence analyses, induction variable occurrences are replaced by closed form expressions in order to break dependences inherent in inductions and enable accurate dependence analysis. Substitution-based approaches, however, have several drawbacks. A dependence test may report false dependences on some closed form expressions. Consider Figure 1.a, the closed form expression of `ijk` is `(j-1)*ns*3 + (k-1)*3 + l`. Since the expression is symbolic (due to the coefficient `ns`), most tests will report loop-carried dependences over the j-loop. Moreover, it is sometimes not possible to represent the value of an induction variable as a closed form expression. This is the case for `nred` in Figure 1.b. However, if the compiler could determine that `nred` is strictly increasing across iterations, then, it could also decide that the loop carries no dependence over array `lisred`. Eventually, closed form expressions can be complex, and testing dependences on them may be expensive.

In this paper, we present a method that exploits IV information in dependence testing without closed form computation. We observe that the values of most induction variables change ("evolve") monotonically. Consider the example in Figure 1.a again, any path in the control-flow graph from statement p back to itself must also traverse statement `ijk = ijk+1`. This means that the value of `ijk` increases between *any* two visits of p. In other words, values of `ijk` at p are strictly increasing. With the information on the monotonicity of `ijk`, the compiler can prove that different executions of p access different array elements, even when a closed form expression can not be captured as is the case in the example of Figure 1.b. Therefore, our analysis tries to identify *monotonic evolution* which captures the monotonicity of variables over paths of a control-flow graph.

There has been some related work on monotonic variables [9, 8] that target sequence properties of a *single* reference. Monotonic evolution is more general since it can be computed between *any* two statements and along *selected paths* between the two statements. The latter feature is especially useful to distinguish between intra-loop and loop-carried dependences.

The rest of the paper is organized as follows. Section 2 defines the concept of monotonic evolution. Section 3 presents the dependence test based on monotonic evolution. To improve precision, we propose a simple extension to

```
— mdg predic do1000 — line 758 —
ijk = 0
do j = 1,ns
   do k = 1,ns
      do l = 1,3
         ijk = ijk+1
p        zc(ijk) = ···
      end do
   end do
end do
```

(a) IV with a closed form expression

```
— qcd setcol do1 — line 2387 —
do l = 1,latt(1)
   if (···) then
      nred = nred+1
      lisred(nred) = ···
   else
      ···
   endif
   ···
end do
```

(b) IV with no closed form expression

Figure 1: Motivating examples from the Perfect Benchmark suite

the lattice of evolution states in Section 4. Section 5 modifies the dependence test to take advantage of this new lattice. Section 6 deals with practical and efficient computation. Section 7 presents experimental results. Section 8 explores some ideas about extending monotonic evolution to dynamic data structures. Section 9 compares our technique with others. Section 10 outlines future work and concludes.

2 Monotonic Evolution

The rest of the paper uses the following notions: a *control-flow path* refers to a path in a control-flow graph; a *statement instance* refers to an execution of a statement; the *value of* `i` *at* p refers to the value of variable `i` immediately before statement instance p is executed.

2.1 Evolution

For dependence testing, it is important to determine whether a variable that appears in subscript expressions may have the same value at different statement instances. Given an execution sequence and a variable `i`, the *monotonic evolution* (evolution) of `i` describes how the value of `i` changes from the beginning to the end of the sequence. We introduce *evolution states* to describe possible values of an evolution. The semi-lattice of evolution states is given in Figure 2, the join operator is $\sqcup$.

$\top$ unknown evolution;
$\unlhd$ monotonically increasing;
$\lhd$ strictly increasing;
$\unrhd$ monotonically decreasing;
$\rhd$ strictly decreasing;
$\diamond$ constant evolution;
$\bot$ no evolution.

Ordering:

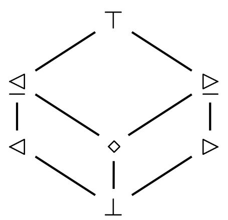


Figure 2: The lattice of evolution states

	$\bot$	$\lhd$	$\unlhd$	$\diamond$	$\unrhd$	$\rhd$	$\top$
Identity	$\bot$	$\lhd$	$\unlhd$	$\diamond$	$\unrhd$	$\rhd$	$\top$
Forward	$\bot$	$\lhd$	$\lhd$	$\lhd$	$\top$	$\top$	$\top$
Backward	$\bot$	$\top$	$\top$	$\rhd$	$\rhd$	$\rhd$	$\top$
Arbitrary	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Figure 3: Transfer functions of evolution values

Any execution sequence corresponds to a control-flow path where each node along the path represents a statement instance. We define evolutions in terms of control-flow paths. Each statement is thus interpreted as a transfer function of evolution values. Given a variable `i`, we classify a statement as:

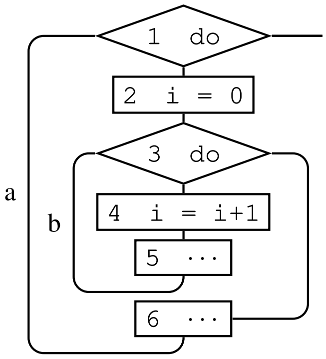
- an *identity statement* (Identity) if it does not change the value of `i`, such as `j = n`;
- a *forward induction* (Forward) if it always increases the value of `i`, such as `i = i+1`;
- a *backward induction* (Backward) if it always decreases the value of `i`, such as `i = i-3`;
- an *arbitrary assignment* (Arbitrary) if it may assign an arbitrary value to `i`, such as `i = n`.

The transfer functions corresponding to each class of statements are given in Table 3.

The notation $p \dashv_{\mathtt{i}}^{N} q$ represents the *join* ($\sqcup$) of the evolution of `i` over all paths that starts from p and ends at q (immediately before the last q is executed), *excluding* those traversing any edges in the set N. A special case is when q can not be reached from p, then $p \dashv_{\mathtt{i}} q$ is $\perp$. Intuitively, $p \dashv_{\mathtt{i}}^{N} q$ tells us how the value of `i` changes when the program executes from an instance of `p` to an instance of `q`. The set N is introduced to be able to represent evolutions for a selected part of a control-flow graph. For instance, we may exclude the back-edge of a loop from an evolution. This can be used to analyze the evolution within a single loop iteration.

For dependence testing, we are also interested in evolutions that must traverse an intermediate node. We use, $p \dashv_{\mathtt{i}}^{N} r \dashv_{\mathtt{i}}^{N} q$, to denote such an evolution of i along all paths from p *via* r before finally reaching q, excluding those that traverse any edge in N.

A simple example is presented below where statement "$\cdots$" does not change the value of `i`. Nodes in the graph are named by statement labels. Evolutions below can be computed following paths in the control-flow graph.



- $3 \dashv_{\mathtt{i}} 3 = \top$, since `i = 0` is traversed along some paths from 3 to 3;
- $5 \dashv_{\mathtt{i}}^{\mathtt{b}} 4 = \perp$, since `b` is excluded, 4 can not be reached from 5;
- $4 \dashv_{\mathtt{i}}^{\mathtt{b}} 5 = \lhd$, after `b` is excluded, there is only one path from 4 to 5, and it traverses `i = i+1`;
- $3 \dashv_{\mathtt{i}}^{\mathtt{a}} 6 = \unlhd$, 3 may directly follow the exit edge of loop 3 to reach 6 without traversing `i = i+1`.

Figure 4: An example

2.2 The Iterative Algorithm

One may check that the lattice of evolution states is finite and that transfer functions are distributive and monotonic over this lattice. Therefore, $p \dashv_{\mathtt{i}}^{N} q$ can be computed as the *least fixed point* of an iterative application of transfer functions [12] over a "pruned" control-flow graph (to exclude edges in N). For each statement s, let f_s be the transfer function of s and pred(s) be the set of predecessors of s in the control-flow graph. Let in_s and out_s hold the values of the evolution that ends right before and immediately after s, respectively. For q, we also keep a special state, $state_q$, that holds the final result of the computation. The following data flow equations have to be solved:

$$in_s = \bigsqcup_{i \in \text{pred}(s)} out_i \qquad\qquad out_s = f_s(in_s) \tag{1}$$

To compute $p \dashv_{\mathtt{i}}^{N} q$, the iterative algorithm starts from node p. Initially, we have $in_p \leftarrow \diamond$, $state_q \leftarrow \perp$, and $out_s \leftarrow \perp$ for all statement s. The algorithm iteratively solves the data-flow equations (1). During this process, every time in_q is updated,[1] the new state is joined to $state_q$, i.e., $state_q \leftarrow state_q \sqcup in_q$. The algorithm terminates when it reaches a fixed point, and $state_q$ yields the result of $p \dashv_{\mathtt{i}}^{N} q$. An early termination is also possible as soon as $state_q$ is set to $\top$. In Section 6, we will present a more efficient non-iterative method to compute evolution.

As opposed to traditional data-flow schemes, this algorithm starts from a node p that may be on a cycle on a control-flow graph. Joining in_q to $state_q$ at each update is thus mandatory because in_q does *not* necessarily increase (in the lattice) across each iteration, as required by the safety and termination proofs in [12]. For instance, applying the algorithm to $5 \dashv_{\mathtt{i}} 3$ in Figure 4 yields $in_3 = \diamond$ at the first iteration after traversing 5, and $in_3 = \lhd$ after traversing $5, 3, 4, 5$, yet $5 \dashv_{\mathtt{i}} 3 = state_3 = \unlhd$.

[1] When $q = p$, the initial value of in_p ($\diamond$) is not joined to $state_p$, since it is not computed (updated) from the data-flow equations.

2.3 Evolution Composition

The composition operator ($\circ$) "chains" two evolutions into a single one where the ending node of the first evolution is also the starting node of the second one. The value of $p \dashv_{\texttt{i}}^{N} r \circ r \dashv_{\texttt{i}}^{M} q$ is $state_q$ of the second evolution which is computed by the iterative algorithm starting with $in_r \leftarrow p \dashv_{\texttt{i}}^{N} r$ (instead of $in_r \leftarrow \diamond$). Composition can be used to compute evolutions that must traverse an intermediate node, i.e., $p \dashv_{\texttt{i}}^{N} r \dashv_{\texttt{i}}^{N} q = p \dashv_{\texttt{i}}^{N} r \circ r \dashv_{\texttt{i}}^{N} q$. In practice, composition of evolution does not need to be computed in order. We may compute each evolution in the composition independently, then, compose their evolution states. Figure 5 defines this operation. Notice that composition is *idempotent*, i.e., $a \circ a = a$; and $\diamond$ is the neutral element, i.e., $\diamond \circ a = a$.

$\circ$	$\bot$	$\lhd$	$\unlhd$	$\diamond$	$\unrhd$	$\rhd$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\lhd$	$\bot$	$\lhd$	$\lhd$	$\lhd$	$\top$	$\top$	$\top$
$\unlhd$	$\bot$	$\lhd$	$\unlhd$	$\unlhd$	$\top$	$\top$	$\top$
$\diamond$	$\bot$	$\lhd$	$\unlhd$	$\diamond$	$\unrhd$	$\rhd$	$\top$
$\rhd$	$\bot$	$\top$	$\top$	$\rhd$	$\rhd$	$\rhd$	$\top$
$\unrhd$	$\bot$	$\top$	$\top$	$\unrhd$	$\unrhd$	$\rhd$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

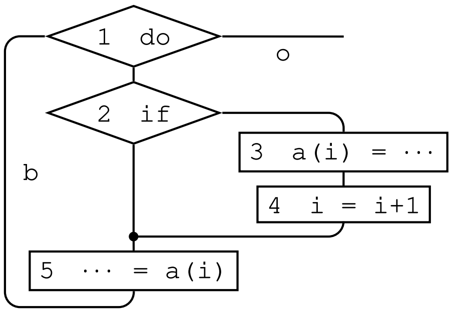
Figure 5: Composition of evolution states

- Identity$(state) = state$
- Forward$(state) = state \circ \lhd$
- Backward$(state) = state \circ \rhd$
- Arbitrary$(state) = state \circ \top$

Figure 6: Transfer funcitons by composition

Composition is also *commutative* and *associative*. In fact, an evolution along a path consisting of two consecutive paths A and B can be computed as the evolution state along A composed with that along B. Consider Figure 4, the evolution of `i` along path $(4, 5)$ is $\lhd$, which can be computed as the composition of the evolutions along 4 and 5 (i.e., $\lhd \circ \diamond$). Thus, we may re-define transfer functions given in Table 3 in terms of composition. The new definitions are given in Figure 6. In fact, transfer functions may be defined for sub-graphs with a single entry and a single exit such as basic blocks. Consider a sub-graph with an entry node, x. Let y be the node that immediately follows the exit node of the subgraph. The transfer function of the sub-graph composes the input state with the evolution that traverses the entire sub-graph from x to y (not including y).

3 Dependence Test



We use evolution to determine dependences between array references subscripted by expressions involving induction variables. Consider statements 3 and 5 in the figure to the right. A dependence test shall determine whether values of i at 3 may overlap with those of i at 5. We observe that the evolution of i from 3 to 5 is strictly increasing ($3 \dashv_{\texttt{i}} 5 = \lhd$), i.e., the value of i at 3 is *less than* the value of `i` at any *later* instance of 5. If the same can be proven for $5 \dashv_{\texttt{i}} 3$, then values of i at 3 and 5 are always distinct, meaning that 3 and 5 are not dependent. In this example, however, $5 \dashv_{\texttt{i}} 3$ is evaluated to $\unlhd$, and therefore it has to be assumed that 3 is anti-dependent on 5.

Consider an expression e and two statements p and q with identical array access `a(`e`)`, at least one of them being on the left-hand side of an assignment statement. The observation is that, if there are dependences between p and q, then $p \dashv_{\texttt{i}} q$ and $q \dashv_{\texttt{i}} p$ cannot both be strict ($\lhd$, $\rhd$) or unreachable ($\bot$). To prove the absence of any dependence, we may check that the evolutions between p and q are either $\lhd$, $\rhd$, or $\bot$. On the other hand, if a dependence may exist, the test needs to further classify them as intra-loop or loop-carried. In our scheme, intra-loop and loop-carried dependences are tested in separate steps for each single loop ℓ of a nest.

- *Intra-loop* dependences are concerned with instances that are spawned by the same iteration of a loop. Thus, evolutions used in the test only need to consider paths between p and q that do not traverse any back-edge of loop ℓ. As discussed below, we assume that loops only exit at the top. Therefore, paths that start within the loop and do not traverse a back-edge never leave the body of the loop.

There is *no* intra-loop dependence between p and q for loop ℓ, if,

$$p \dashv_e^B q \in \{\perp, \lhd, \rhd\} \ \wedge \ q \dashv_e^B p \in \{\perp, \lhd, \rhd\} \tag{2}$$

where B is the set of all back-edges of the loop.

Consider the previous example again, since $3 \dashv_{\mathtt{i}}^{\mathsf{b}} 5 = \lhd$ and $5 \dashv_{\mathtt{i}}^{\mathsf{b}} 3 = \perp$, we conclude that there is no intra-loop dependence between 3 and 5.

- *Loop-carried* dependences are concerned with instances that are spawned by different iterations of a loop, ℓ, but the same iteration of any loop that surrounds ℓ. Thus, evolutions used in the test must traverse a back-edge of ℓ, and must not traverse any exit edge of ℓ. Note that any path traversing a back-edge of a loop also traverses the header of the loop (the statement that checks for the loop's termination). Thus, forcing the traversal of a back-edge is equivalent to forcing the traversal of the loop header.

  There is *no* loop-carried dependence between p and q for loop ℓ, if,

$$p \dashv_e^E h \dashv_e^E q \in \{\perp, \lhd, \rhd\} \wedge \ q \dashv_e^E h \dashv_e^E p \in \{\perp, \lhd, \rhd\} \tag{3}$$

  where E is the set of exit edges of ℓ and h is the header of ℓ.

  In the previous example, since $3 \dashv_{\mathtt{i}}^{\mathsf{o}} 1 \dashv_{\mathtt{i}}^{\mathsf{o}} 5 = \lhd$ and $5 \dashv_{\mathtt{i}}^{\mathsf{o}} 1 \dashv_{\mathtt{i}}^{\mathsf{o}} 3 = \unlhd$, there is a loop-carried dependence between 3 and 5 for loop 1.

4 Lattice Extension

The lattice of evolution states is useful to detect value changes in induction variables but not to *quantify* such changes. Dependence testing for some cases require a little more information. Consider the examples in Figure 7 where array `a` is accessed with subscripts `k` and `k+1`. If the dependence test knows that values of `k` change by a minimum of 2, it will detect no loop-carried dependence for accesses to `a`.

```
do i = 1,100
   a(k) = a(k+1) + ···
   k = k+2
end do
```

```
do i = 1,100
   a(k) = a(k+1) + ···
   if (···) k = k+1
   k = k+2
end do
```

```
do i = 1,100
   k = k+2
   t = a(k)
   a(k) = a(k+1)
   a(k+1) = t
end do
```

Figure 7: Induction variables with offsets

We introduce new lattice elements to discover such information. For all $n \geq 0$, $\lhd_n$ and $\rhd_n$ describe increasing and decreasing evolutions, respectively, where the difference between values of the variable must be *greater than or equal* to n. In other words, n characterizes the *minimal difference* between any two values of the variable at the starting and ending points of the evolution. We call n the *minimal distance* of the evolution. The evolution states $\lhd_1$ and $\rhd_1$ correspond to $\lhd$ and $\rhd$ of the original lattice, respectively, where the only information is that "the value has changed" in some direction; and $\lhd_0$/$\rhd_0$ correspond to $\unlhd$/$\unrhd$, where the value might have been constant but the direction is known in case of a change. Table 1 gives the formal definition of $\sqcup$ using the *minimum* of distances when joining two increasing or two decreasing evolution states.

Operator $\circ$ needs to be updated as well. The new table is given in Table 2. The main difference is that composing $\lhd_p$ with $\lhd_q$ yields $\lhd_{p+q}$. According to this new definition, $\circ$ is still commutative and associative, but not idempotent anymore, e.g., $\lhd_p \circ \lhd_p = \lhd_{2p}$.

Transfer functions have changed to cope with new elements of the lattice, but the algorithm that drives the iterative computation remains the same. The distance after an induction `i = i+`q is the *sum* of current distance, call it p, with q. Notice that q has to be *known at compile-time* (possibly after constant propagation). Consider the previous examples, all evolutions between assignments to `k` across iterations yield $\lhd_2$.

$\sqcup$	$\bot$	$\lhd_p$	$\diamond$	$\rhd_p$	$\top$
$\bot$	$\bot$	$\lhd_p$	$\diamond$	$\rhd_p$	$\top$
$\lhd_q$	$\lhd_q$	$\lhd_{\min(p,q)}$	$\lhd_0$	$\top$	$\top$
$\diamond$	$\diamond$	$\lhd_0$	$\diamond$	$\rhd_0$	$\top$
$\rhd_q$	$\rhd_q$	$\top$	$\rhd_0$	$\rhd_{\min(p,q)}$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Table 1: Distance-extended lattice

$\circ$	$\bot$	$\lhd_p$	$\diamond$	$\rhd_p$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\lhd_q$	$\bot$	$\lhd_{p+q}$	$\lhd_q$	$\top$	$\top$
$\diamond$	$\bot$	$\lhd_p$	$\diamond$	$\rhd_p$	$\top$
$\rhd_q$	$\bot$	$\top$	$\rhd_q$	$\rhd_{p+q}$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

Table 2: Distance-extended composition

	$\bot$	$\lhd_p$	$\diamond$	$\rhd_p$	$\top$
Identity	$\bot$	$\lhd_p$	$\diamond$	$\rhd_p$	$\top$
Forward$_q$ (with step q)	$\bot$	$\lhd_{p+q}$	$\lhd_q$	$\top$	$\top$
Backward$_q$ (with step q)	$\bot$	$\top$	$\rhd_q$	$\rhd_{p+q}$	$\top$
Arbitrary	$\bot$	$\top$	$\top$	$\top$	$\top$

This extension makes the lattice infinite, even of infinite height. However, since $state \circ state \sqsubset state$ holds, we still have an interesting property: $state \sqcup state \circ state = state$. This means that all descending chains (using operator $\sqcup$) are of *finite* length during the fixed point computation. This property guarantees the termination of the fixed point computation.

5 Improved Dependence Test

We devise a new dependence test to exploit the additional minimal distance information. This test shares some similarities with the *range test* used in Polaris [3]. Consider two accesses, `a(i)` at p and `a(i+d)` at q, `i` being an induction variable and `d` being a constant.[2] The value difference between `i` at p and `i`$+d$ at q is used in the dependence test. However, since subscripts of `a` at p and q are different, this information cannot be computed directly from evolutions. Of course, $\mathtt{i} + d \geq \mathtt{i}$ if $d \geq 0$ at q. Therefore, we may compute the evolution of `i` from p to q, then compose it with $\lhd_d$. The resulting evolution captures the difference between the value of `i` at an instance of p and the value of `i+d` at any later instance of q.

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.

- If $d > 0$, there is *no* intra-loop dependence between p and q for loop ℓ when,

$$p \dashv_e^B q \circ \lhd_d \notin \{\top, \diamond, \lhd_0, \rhd_0\} \ \wedge \ q \dashv_e^B p \circ \rhd_d \notin \{\top, \diamond, \lhd_0, \rhd_0\}. \tag{4}$$

  If $d < 0$, $\lhd_d$ (resp. $\rhd_d$) is replaced by $\rhd_{-d}$ (resp. $\lhd_{-d}$) in (4).

- If $d > 0$, there is *no* loop-carried dependence between p and q for loop ℓ when,

$$p \dashv_e^E h \dashv_e^E q \circ \lhd_d \notin \{\top, \diamond, \lhd_0, \rhd_0\} \ \wedge \ q \dashv_e^E h \dashv_e^E p \circ \rhd_d \notin \{\top, \diamond, \lhd_0, \rhd_0\}. \tag{5}$$

  If $d < 0$, $\lhd_d$ (resp. $\rhd_d$) is replaced by $\rhd_{-d}$ (resp. $\lhd_{-d}$) in (5).

Consider the right-most loop in Figure 7, which swaps every pair of elements of array `a`. Since `k` is incremented by 2 per iteration, to test loop-carried dependence between 3 and 5, we have $3 \dashv_{\mathtt{k}} 1 \dashv_{\mathtt{k}} 5 \circ \lhd_1 = \lhd_2 \circ \lhd_1 = \lhd_3$ and $5 \dashv_{\mathtt{k}} 1 \dashv_{\mathtt{k}} 3 \circ \rhd_1 = \lhd_2 \circ \rhd_1 = \lhd_1$. This proves there is no loop-carried dependence between 3 and 5.

6 Practical Computation

We designed a basic (non-iterative) algorithm to compute evolutions which traverse *no back-edges* in the control-flow graph. This algorithm is the core of the non-iterative method. Any general evolution can be decomposed into segments, each of which can be computed by the basic algorithm. This method is described in [16].

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

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

Table 3: Experiments with the Perfect Club benchmark suite

6.1 Caching Intermediate Evolution

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. Note that, the non-iterative algorithm decomposes evolutions into the same segments, over and over again. 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 (on demand) local evolutions that traverse an entire block, and store the results. During later computations, the algorithm may "short-cut" the basic block by composing its (cached) local state with the input state.

6.2 Complexity Analysis

We are interested in an upper bound on the complexity of computing all evolutions for the dependence test of the overall program. Of course, we would like to take the "caching" of intermediate results into account.

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

$$O(ek + m^2k^2). \tag{6}$$

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 scheme, dependency is tested individually for each loop of a nest (as reflected by the occurrence of m in (6)). Therefore, compared to classical dependence tests *without induction variable recognition*, our scheme requires more steps.[3] However, (6) estimates the number of operations (i.e., $\circ$, $\times$, 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, depending on the mathematical tools employed, the cost of individual operations is difficult to estimate.

7 Experimental Results

For our experimental studies, we used Polaris [4], a Fortran source-to-source parallelizing compiler, as the basis for comparison. In Polaris, there is a dedicated idiom recognition pass to identify induction variables and find their closed forms. Each induction variable is then substituted by its closed form expression before the dependence test is performed. In the context of dependence testing for array accesses, we focus on *integer induction variables* (IIVs)

[3] m times, when testing a large number of array accesses, i.e., when k is close to e.

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. Other IIVs may still be used to drive locality optimizations and to prove array properties such as the injectivity of array values [13], but these applications are left for future work.

In the experiment, we used Polaris to find candidate IIVs from the Perfect Club benchmark suite. We applied our dependence test by hand (for dependences involving IIVs). Table 3 presents the experimental results. 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), i.e., loops which are the target of our technique. The next four 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, and by combining Polaris with our technique (Best). 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.

These results allow us to draw an early conclusion: when dealing with induction variables, our technique is efficient and matches the precision of Polaris. Thus, closed form computation can be delayed until after the dependence test and performed only for loops free of other loop-carried dependences. This would not only enable closed form expression computation to be on demand (to remove sequential induction computations), but also avoid expensive dependence testing on complex subscripts due to closed form substitution. Such a scheme only misses three loops that can be parallelized by Polaris using closed form expressions, but it finds 19 more parallel loops (or 13 without considering conditional induction variable updates).

8 Induction Variables with Dynamic Structures

By design, monotonic evolution is not limited to scalar induction variables. Interestingly, many data-intensive algorithms use dynamic structures where inductive traversals are implemented with *pointers*, *references* and/or *iterators*. Indeed, when dealing with non-numerical programs, arrays share their preeminence with pointer-based structures and more general *container* components, such as lists, vectors, hash-tables, sets, and so on. General-purpose containers are often provided by standard libraries, such as C++ Standard Template Library (STL) and Java Standard Development Kit (SDK). Iterators abstract the implementation details and provide an efficient and portable interface to traverse containers.

In this section, we sketch a few ideas about extending the analysis to pointer-based data structures and object-oriented containers. Monotonic evolution is restricted to structures with a *linked-list* backbone (bidirectional or not). It can be extended easily to any *acyclic* structure in defining two evolution directions (e.g., depth in a tree or directed acyclic graph) or extending the lattice with additional directions.

We use a Java-like syntax. By convention, pointer-based structures have a `next` field, and possibly a `prev` field (such that `p.next.prev=p` and `p.prev.next=p`, except on the list's head and tail). We only studied a small set of basic container operations: `first()` and `last()` generates a new iterator attached to the container; `advance()` and `retreat()` move an iterator; `get()`, `put()`, `insert()` and `delete()` update/accesses a container through an iterator. The syntax of these operations is borrowed from the Java Generic Library (JGL).[4] The semantics is slightly simplified, e.g., no implicit iterator move in container accesses.

Most important, monotonic evolution and dependence testing is applied after a static *pointer* analysis of the loop nest. This preliminary phase needs to discover three critical properties that will be assumed in the rest of the section:

1. when dealing with pointer-based structures, *non-circular listness* (or treeness) is a required *shape* property;
2. when dealing with object-oriented containers, *combness* needs to be proven to deduce dependence information: a container is a *comb* when every container element is attached at a single container position, i.e., when the mapping between positions and elements is injective;
3. *aliased* pointers/references induce dependences, independently of monotonic evolution results; such dependences are not taken into account here.

Many shape and alias analyses have been crafted for pointer-based data structures, see [15] for a detailed review; virtually any one can be used, depending on the precision, scalability, and interprocedural requirements. Shape analysis

[4] A project from ObjectSpace, see `http://www.objectspace.com`.

of container structures is a new research topic: we have designed two techniques to analyze combness (and other static pointer properties) in general-purpose containers [5] and nested Java arrays [17]. The reader is referred to these papers for details.

```
static void sparseLU (Iterator k) {
  while (!k.atEnd ()) {
    Iterator j = k.clone ();
    while (!j.atEnd ()) {
      Object b = j.get ();
      List l = j.get ();
      Iterator i = l.first ();
      while (!i.atEnd ()) {
        Object a = i.get ();
        i.put (···);
        i.advance ();
      }
      j.advance ();
    }
    k.advance ();
  }
}
```

(a) Iterator IV

```
static Node bubbleSort (Node p) {
  Node q = NULL;
  while (p!=NULL) {
    r = q;
    while (··· && r.val<p.val)
      r = r.next;
    if (···) {
      r.prev.next = p;
      p.prev.next = p.next;
      p.next = r.next;
    }
  }
  return q;
}
```

(b) Pointer-chasing IV

Figure 8: Examples with dynamic structures

Pointer assignments and iterator updates can be handled by the usual transfer functions:

- effects of `i.advance()` and `p=p.next` are abstracted by Forward;
- effects of `i.retreat()` and `p=p.prev` are abstracted by Backward;
- `i.delete()` and `i.insert(o)` perform structural container updates but leave `i` unchanged, they are abstracted by Identity;
- `i=l.first()`, `i=l.last()` and `p=···` (where ··· is not `p.next`) are considered arbitrary assignments, hence abstracted by Arbitrary; it is conservatively the same for `i = j.clone()`;
- C-like `p = &x`, `p = *q` and pointer arithmetic are abstracted by Arbitrary; it is the same for aliased iterators (`Iterator i = j`) because they may hide further moves;
- all other statements have no effect on IVs and are abstracted by Identity.

Two code fragments are proposed in Figure 8. The first one implements LU factorization using a sparse representation with lists of lists. The two innermost `while` loops carry no dependences over the list elements, because `j` iterate over distinct sub-lists and `i` enumerates elements of each sub-list. However, *structural* dependences remain, i.e., dependences induced by the linear accesses and backbone structure associated with the `List` container. Parallelization techniques can still be applied, see [18] for details.

The second example implements a bubble sort on a doubly-linked list. This example performs structural updates on both lists, and a classical shape analysis [15] can prove that `p` and `q` are unaliased at the beginning of each iteration of the outer loop, and that the returned list is acyclic if `p` points to an acyclic loop. Monotonic evolution only cares with statement `r = r.next` and accesses `r.val` and `p.val` (it does not deal with structural updates). Of course, it detects no dependence on the list's elements, but many *structural* dependences still hamper parallelization.

This short overview of the interplay of monotonic evolution and dynamic structures motivates further research in combining pointer analyses with our dependence test. This is left for a promising future work.

9 Related Work

Most dependence tests handle induction variables by idiom recognition and closed form substitution. Those closed form expressions usually involve only the indices of the surrounding loops and loop invariants. Using patterns proposed by Pottenger and Eigenmann [14], the Polaris compiler recognizes polynomial sequences that are not limited to scalar and integer induction variables. Other closed form computation techniques explore various approaches. Abstract interpretation is used by Ammarguellat and Harrison [2] to compute symbolic expressions and compare it with known templates, but it leads to a rather inefficient algorithm and does not handle irregular nests. Two general classification techniques have been designed. The first one by Gerlek, Stoltz and Wolfe [8] is based on a SSA representation [6] optimized for efficient demand-driven traversals. It relies on Tarjan's algorithm to detect strongly connected components. The second one is designed by Haghighat and Polychronopoulos for the Parafrase 2 compiler[10]. It combines symbolic execution (iterated forward substitution) and recurrence interpolation. Both of them handle a broad scope of closed form expressions, such as linear, arithmetic (polynomial), geometric (with exponential terms), periodic, and wrap-around.

Closed form expression computation has obvious benefits for optimizations. It is also critical for removing dependences due to computation of induction variables themselves. For irregular nests with `while` loops or complex bounds (e.g., with array accesses), and for conditional IV updates, closed form expressions are generally not hoped for. Gupta and Spezialetti extended the linear IV detection framework with arithmetic and geometric sums, as well as *monotonic* sequences [9], for non-nested loops only. Their technique is applied to optimizations such as efficient run-time array bounds checking. Lin and Padua [13] also studied monotonicity for values of index arrays in the context of parallelizing irregular codes. This property can be used later to detect dependences between accesses to sparse matrices through index arrays. Like in our technique, they compute monotonicity on-demand from non-iterative traversals of the control-flow graph, but their technique does not target general induction variables. More general *monotonic* sequences could be detected by Gerlek, Stoltz and Wolfe as a special class of induction variables [8], as soon as a strongly connected component in the SSA graph traverses a ϕ-function. As far as the monotonic class of IVs is concerned, their classification of sequences is less powerful than our evolution in the following ways:

- Monotonicity is estimated for each *sequence of values* associated with a variable. Since SSA gives different names after each definition, references to the same variable separated by induction variable updates can *not* be compared. This may yield spurious dependences.

- It is not clear whether sequences are defined loop-wise or for the whole nest. In the latter case, it may make too conservative assumptions for inner loops. Extending their technique for statement-to-statement evolutions seems difficult: SSA graphs are not well suited for disabling traversal of control-flow edges.

- Stride-extended monotonicity information is not computed, but closed form expressions associated with other IV classes may be suitable for such sequences. This would incur a higher analysis cost and is likely to blur further dependence testing.

It is worth noticing that recursive pointer "chasing" like `p = p->next` may also be interpreted as a form of induction. Dealing with pointer and dependence analysis in the presence of *recursive data structures*, some techniques use abstractions closely related to monotonicity to compare pointer variables. For instance, Hendren, Hummel and Nicolau [11] are able to discover when a tree access is "below" another one or when two accesses target distinct branches. They abstract access paths with regular expressions that might be interpreted as monotonicity and strides generalized to multiple independent dimensions.

10 Conclusion

We use monotonic evolution for dependence testing on array accesses indexed by induction variables. This method requires no closed form expression computation. The experiment showed that our technique matches the precision of Polaris when closed form expressions are available, and when there are no closed form expressions, our technique can detect additional parallel loops. An efficient non-iterative algorithm is devised, achieving incremental computation of evolutions at a very low cost. IV substitution only needs to be performed after the dependence analysis, and it can be performed on demand. This saves unnecessary closed form computation on loops that eventually may not be parallelized.

We plan to extend the algorithm to handle arbitrary assignments, such as `i = j`, more precisely. This may lead us to solving the two patterns yet to be handled. Furthermore, since arbitrary assignments link the values of two variables, they may be used as reference points to compare different variables. From the lattice side, we would like to compute both the maximal and minimal distance of an evolution. Dependence tests may exploit such information [3]. Eventually, we proposed a first attempt to apply monotonic evolution on other forms of induction operations, such as pointer chasing and iterator traversals, where closed form abstractions are impractical. Applications include parallization and improved pointer analyses. Monotonic evolution is well-suited for dynamic structures since traversals are likely to be monotonic, and closed form abstractions are impractical for such accesses.

Acknowledgment

The work reported in this paper was supported in part by NSF contracts ACI 98-70687 and CCR 00-81265 and by a cooperative agreement between CNRS (Centre National de la Recherche Scientifique) in France and University of Illinois at Urbana-Champaign. We are very thankful to David Padua and Jay Hoeflinger for their numerous contributions to monotonic evolution, fruitful comments, and help with Polaris.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] 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, New York, USA, June 1990.

[3] 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., USA, November 1994. IEEE Computer Society Press.

[4] 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.

[5] A. Cohen, P. Wu, and D. Padua. Pointer analysis for monotonic container traversals. Technical Report CSRD TR-1586, U. of Illinois, January 2001.

[6] 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.

[7] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. on Parallel and Distributed Systems*, 9(1):5–23, January 1998.

[8] 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.

[9] 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, 91.

[10] M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 18(4):477–518, July 1996.

[11] J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *ACM Symp. on Programming Language Design and Implementation (PLDI'94)*, pages 218–229, Orlando, Florida, USA, June 1994.

[12] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[13] 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.

[14] 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.

[15] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. on Programming Languages and Systems*, 20(1):1–50, January 1998.

[16] 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 (ICS'01)*, Sorrento, Italy, June 2001. To appear.

[17] P. Wu, P. Feautrier, D. Padua, and Z. Sura. Points-to analysis for java with applications to loop optimizations. Technical Report CSRD TR-1585, U. of Illinois, November 2000.

[18] P. Wu and D. Padua. Containers, on the parallelization of general-purpose java programs. *Int. Journal of Parallel Programming*, 28(6):589–605, dec 2000.