Monotonic Evolution: an Alternative to Induction Variable Substitution for Dependence Analysis

Peng Wu, Albert Cohen, Jay Hoeflinger, David Padua

▶ **To cite this version:**

Peng Wu, Albert Cohen, Jay Hoeflinger, David Padua. Monotonic Evolution: an Alternative to Induction Variable Substitution for Dependence Analysis. Intl. Conf. on Supercomputing, Jun 2001, Sorrento, Italy. hal-01257312

HAL Id: hal-01257312

https://hal.science/hal-01257312

Submitted on 20 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Monotonic Evolution: an Alternative to Induction Variable Substitution for Dependence Analysis

Peng Wu † Albert Cohen ‡ Jay Hoeflinger § David Padua †

† Department of Computer Science
University of Illinois
Urbana, IL 61801, USA
{pengwu,padua}@cs.uiuc.edu

‡ A3 Project
INRIA Rocquencourt
78153 Le Chesnay, France
Albert.Cohen@inria.fr

§ KAI Software
Intel Americas, Inc.
Champaign, IL 61820, USA
jay.p.hoeflinger@intel.com

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 then used in the dependence test to help determine whether array references are dependence-free. Under this scheme, closed form computation and induction variable substitution can be delayed until after the dependence test and be performed on-demand. To improve computative efficiency, we also propose an optimized (non-iterative) data-flow algorithm to compute evolution. Experimental results show that dependence tests based on evolution information matches 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, the most important ones being loop transformations for parallelization [3] and strength reduction [1]. An induction variable is broadly defined as any scalar variable or array element that is 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 [8].

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. Previous 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 + 1`.

  Since the expression is symbolic (due to the coefficient `ns`), most tests will report loop-carried dependences over the j-loop.
- Sometimes, it is 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`.
- 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 [10, 9] that target sequence properties of a *single* reference. Monotonic evolution is more general since it can be computed between *any* two statements and

```
— 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

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 the lattice of evolution states in Section 4. Section 5 modifies the dependence test to take advantage of this new lattice. Section 6 gives an efficient non-iterative algorithm to compute evolution. Algorithm complexity is studied in Section 7. Section 8 presents experimental results. 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. The values of an evolution can be:

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

These values are arranged as the lattice of evolution states as shown in Figure 2. The join operator is $\sqcup$.

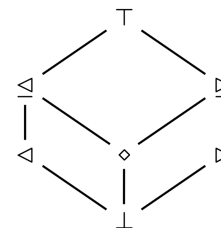


Figure 2: The lattice of evolution states

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* if it does not change the value of i, such as `j = n`;
- a *forward induction* if it always increases the value of i, such as `i = i+1`;
- a *backward induction* if it always decreases the value of i, such as `i = i-3`;
- an *arbitrary assignment* 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 1.

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

Table 1: Transfer functions of evolution values

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

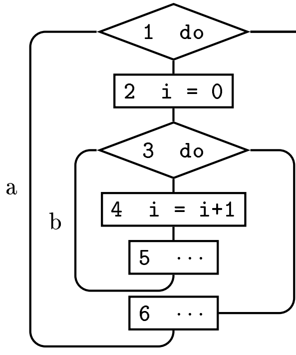


Figure 3: An example

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 $\bot$. 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 in Figure 3 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;
- $5 \dashv_{\mathtt{i}}^{\mathtt{b}} 4 = \bot$, 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`.

We make the following observations from this example. First, evolutions are not symmetric: for instance, $4 \dashv_{\mathtt{i}}^{b} 5 = \lhd$ and $5 \dashv_{\mathtt{i}}^{b} 4 = \bot$. Second, the transfer function for arbitrary assignments is very conservative: $3 \dashv_{\mathtt{i}} 3$ is evaluated to $\top$ because of the traversal of `i = 0`. However, knowing that `i` at 3, immediately following `i = 0`, is always a constant, $3 \dashv_{\mathtt{i}} 3$ should rather be $\diamond$. The precise handling of arbitrary assignments is left for future work.

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 [13] 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 $\mathrm{pred}(s)$ 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:

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

To compute $p \dashv_{\mathtt{i}}^{N} q$, the iterative algorithm starts from node p. Initially, we have $in_p \leftarrow \diamond$, $state_q \leftarrow \bot$, and $out_s \leftarrow \bot$ 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 [13]. For instance, applying the algorithm to $5 \dashv_{\mathtt{i}} 3$ in Figure 3 yields $in_3 = \diamond$ at the first iteration after traversing 5, and $in_3 = \lhd$ after traversing $5, 3, 4, 5$, yet $5 \dashv_{\mathtt{i}} 3 = state_3 = \unlhd$.

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 $p \dashv_{\mathtt{i}}^{N} r \circ r \dashv_{\mathtt{i}}^{M} q$ is $state_q$ of the second evolution which is computed by the iterative algorithm starting with $in_r \leftarrow p \dashv_{\mathtt{i}}^{N} r$ (instead of $in_r \leftarrow \diamond$). Composition can be used to compute evolutions that must traverse an intermediate node. In fact,

$$p \dashv_{\mathtt{i}}^{N} r \dashv_{\mathtt{i}}^{N} q = p \dashv_{\mathtt{i}}^{N} r \circ r \dashv_{\mathtt{i}}^{N} q \qquad (2)$$

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 4 defines this operation. Notice

[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.

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

Figure 4: Composition of evolution states

that composition is *idempotent*, i.e., $a \circ a = a$; and $\diamond$ is the neutral element, i.e., $\diamond \circ a = a$. 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 3, 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 1 in terms of composition as follows:

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

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).

2.4 Evolution of Expressions

We compute $p \dashv_{\mathtt{e}}^{N} q$ where expression e is the sum of several terms. When `e` is a constant expression, $p \dashv_{\mathtt{e}}^{N} q$ is constant ($\diamond$) if q can be reached from p (otherwise, $\bot$). Considering expression `-i`, the state of $p \dashv_{\mathtt{-i}}^{N} q$ is always the "opposite" of the state of $p \dashv_{\mathtt{i}}^{N} q$. Operator $-$ computes the "opposite" of an evolution state:

state	$\bot$	$\lhd$	$\unlhd$	$\diamond$	$\unrhd$	$\rhd$	$\top$
$-state$	$\bot$	$\rhd$	$\unrhd$	$\diamond$	$\unlhd$	$\lhd$	$\top$

In general, when `e` is of the form `a*i`, $p \dashv_{\mathtt{a*i}}^{N} q$ equals $p \dashv_{\mathtt{i}}^{N} q$ when $a > 0$, or $-(p \dashv_{\mathtt{i}}^{N} q)$ when $a < 0$.

Finally, when e of the form $e_1 + e_2$, the evolution of e shall be the *sum* of those of e_1 and e_2. This can be easily computed since the sum of two evolution states can be computed by composition. For instance, suppose that e is `2i-3j+6`,

$$p \dashv_{e}^{N} q = p \dashv_{\mathtt{i}}^{N} q \circ -(p \dashv_{\mathtt{j}}^{N} q) \circ p \dashv_{\mathtt{6}}^{N} q.$$

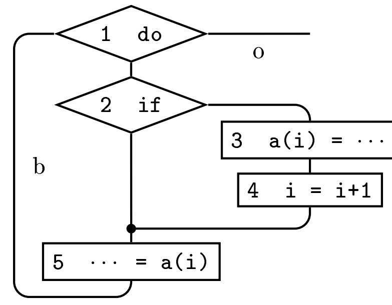


Figure 5: Dependence test example

3. DEPENDENCE TEST

We use evolution to determine dependences between array references subscripted by expressions involving induction variables. Consider statements 3 and 5 in Figure 5. 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 ($3 \dashv_{\mathtt{i}} 5 = \lhd$), 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 $5 \dashv_{\mathtt{i}} 3$, then values of i at 3 and 5 are always distinct, meaning that 3 and 5 are not dependent. In this example, however, $5 \dashv_{\mathtt{i}} 3$ 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 $\mathtt{a}(e)$, 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 $p \dashv_{\mathtt{i}} q$ and $q \dashv_{\mathtt{i}} p$ 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,

  $$p \dashv_{e}^{B} q \in \{\bot, \lhd, \rhd\} \;\wedge\; q \dashv_{e}^{B} p \in \{\bot, \lhd, \rhd\} \quad (3)$$

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

  Consider the previous example again, since $3 \dashv_{\mathtt{i}}^{\mathtt{b}}$

```
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
```

```
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
```

Figure 6: Induction variables with offsets

$5 = \lhd$ and $5 \dashv_{\mathtt{i}}^{\mathtt{b}} 3 = \bot$, 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,

  $$p \dashv_e^E h \dashv_e^E q \in \{\bot, \lhd, \rhd\} \wedge q \dashv_e^E h \dashv_e^E p \in \{\bot, \lhd, \rhd\} \quad (4)$$

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

  In the previous example, since $3 \dashv_{\mathtt{i}}^{\circ} 1 \dashv_{\mathtt{i}}^{\circ} 5 = \lhd$ and $5 \dashv_{\mathtt{i}}^{\circ} 1 \dashv_{\mathtt{i}}^{\circ} 3 = \unlhd$, 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 6 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`.

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 2 gives the formal definition of $\sqcup$ using the *minimum* of distances when joining two increasing or two decreasing evolution states.

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

Table 2: Distance-extended lattice

Operator $\circ$ needs to be updated as well. The new table is given in Table 3. 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., $\lhd_p \circ \lhd_p = \lhd_{2p}$.

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

Table 3: Distance-extended composition

We define a new binary operator $\times$ that multiplies an evolution state by a positive number.

state	$\bot$	$\lhd_a$	$\diamond$	$\rhd_a$	$\top$
state $\times\, n$	$\bot$	$\rhd_{a \times n}$	$\diamond$	$\lhd_{a \times n}$	$\top$

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 $\lhd_2$.

	$\bot$	$\lhd_p$	$\diamond$	$\rhd_p$	$\top$
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 [4].

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,

$$p \dashv_e^B q \circ \lhd_d \notin \{\top, \diamond, \lhd_0, \rhd_0\} \quad \wedge \quad q \dashv_e^B p \circ \rhd_d \notin \{\top, \diamond, \lhd_0, \rhd_0\}. \quad (5)$$

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

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

$$p \dashv_e^E h \dashv_e^E q \circ \lhd_d \notin \{\top, \diamond, \lhd_0, \rhd_0\} \quad \wedge \quad q \dashv_e^E h \dashv_e^E p \circ \rhd_d \notin \{\top, \diamond, \lhd_0, \rhd_0\}. \quad (6)$$

  If $d < 0$, $\lhd_d$ (resp. $\rhd_d$) is replaced by $\rhd_{-d}$ (resp. $\lhd_{-d}$) in (6).

Consider the right-most loop in Figure 6, 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,

$$3 \dashv_{\mathtt{k}} 1 \dashv_{\mathtt{k}} 5 \circ \lhd_1 = \lhd_2 \circ \lhd_1 = \lhd_3$$
$$5 \dashv_{\mathtt{k}} 1 \dashv_{\mathtt{k}} 3 \circ \rhd_1 = \lhd_2 \circ \rhd_1 = \lhd_1$$

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

This proves there is no loop-carried dependence between 3 and 5. In Section 8, we will show that the new dependence test is able to parallelize many more nests.

6. EFFICIENT COMPUTATION

This section presents the core techniques for efficient computation of evolutions. These techniques are then combined into an optimized and non-iterative algorithm. This section uses the following notations:

- h_k denotes the header of any loop ℓ_k.
- ℓ_p denotes the loop that immediately encloses p.[3]

6.1 The Basic Algorithm

We first give 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.

We make some assumptions about the control-flow graph. Loop headers do not change any program variable. One may suppose, without loss of generality, that a loop always exits from its header. An early exit is interpreted as a `continue` statement (i.e., an unconditional branch to the loop header) immediately followed by a normal loop exit. We also assume that each node s of the control-flow graph is assigned a number, d_s, according to its depth-first search ordering [1] (a.k.a. topological ordering [6]).

The basic algorithm takes, as input, an *innermost* loop ℓ with header h, two nodes p and q in ℓ, and a "pruned" control-flow graph that excludes edges in a set N. There are two cases:

- The first case is when $q \neq h$. The algorithm returns the value of $p \dashv_{\mathtt{i}}^{N \cup O} q$ where O is the set of outgoing edges of ℓ. Testing reachability is straightforward: the algorithm returns $\perp$ when $d_p > d_q$. Then, if q is reachable from p, the algorithm starts from p and processes every node s such that $d_p \leq d_s < d_q$ in increasing depth-first search order. Let f_s be the transfer function of s and pred(s) be the predecessors of s inside loop ℓ. Initially, the algorithm sets $in_p \leftarrow \diamond$. Then, in_s and out_s are computed for each node s using equation (1). When in_q is computed, the algorithm returns in_q and terminates.
- The second case is when $q = h$. The algorithm returns $\perp$, if all back-edges of ℓ are excluded. Otherwise, q is reachable from p. Then, the algorithm is similar to the previous case except that it processes every node s such that $d_s \geq d_p$. After every out_s is computed, the final result, in_h, is computed as the join of all out_s from the back-edges.

[3] When p is the header of a loop ℓ, the innermost enclosing loop of p is the loop that immediately encloses ℓ.

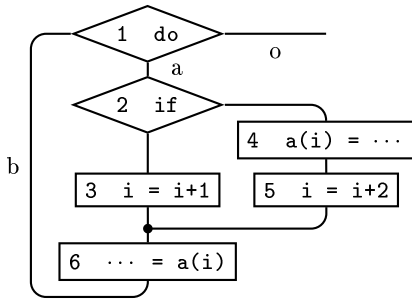


Figure 7: Using the basic algorithm

Consider the example in Figure 7, where the statement label is also the depth-first numbering of the statement. To compute The basic algorithm computes $2 \dashv^{\mathtt{b}}_{\mathtt{i}} 6$ following the depth-first order: $in_2 = \diamond$, $out_2 = \diamond$, $in_3 = \diamond$, $out_3 = \lhd_1$, $in_4 = \diamond$, $out_4 = \diamond$, $in_5 = \diamond$, $out_5 = \lhd_2$, and $in_6 = \lhd_1 \sqcup \lhd_2 = \lhd_1$. The result is $2 \dashv^{\mathtt{b}}_{\mathtt{i}} 6 = in_6 = \lhd_1$. Now, consider an evolution that ends at a loop header, $2 \dashv^{\mathtt{a,o}}_{\mathtt{i}} 1$. The basic algorithm computes the evolution in two steps: it first computes $out_6 = \lhd_1$; then, 6 being the only predecessor of 1 in the loop, the result is $2 \dashv^{\mathtt{b}}_{\mathtt{i}} 1 = in_1 = \lhd_1$ as well.

6.2 Definition of Stride

In a structured program, all cyclic paths arise from loop constructs. We are interested in evolutions that traverse *at most* one iteration of a loop, called *strides*. With stride information, it is possible to compute evolutions that traverse loops without actually iterating on the graph.

Consider a loop ℓ with a header h and a statement p enclosed in ℓ. We define three strides between p and h:

- $\mathrm{Up}_{\mathtt{i},N}(p,h)$ denotes an evolution from p *up* to the first h reached, excluding edges in N. It captures the value difference of i from an instance of p to the first instance of h that follows it. $\mathrm{Up}_{\mathtt{i},N}(p,h)$ is called an *up-stride* of p. When p is immediately enclosed by ℓ, $\mathrm{Up}_{\mathtt{i},N}(p,h)$ is called the *local up-stride* of p.
- $\mathrm{Down}_{\mathtt{i},N}(h,p)$ denotes an evolution from h *down* to p *without* traversing h twice, excluding edges in N. It is also called a *down-stride* of p. It When p is immediately enclosed by ℓ, $\mathrm{Down}_{\mathtt{i},N}(h,p)$ is called the *local down-stride* of p.
- $\mathrm{Stride}_{\mathtt{i},N}(\ell)$ denotes an evolution from h to the next h, excluding edges in N and the exit edge of ℓ. It is a special case of $\mathrm{Down}_{\mathtt{i},N}(h,p)$ and $\mathrm{Up}_{\mathtt{i},N}(p,h)$ when $p = h$. Since $\mathrm{Stride}_{\mathtt{i},N}(\ell)$ captures the effect of iterating *exactly one* iteration of ℓ, it is called the *local stride* of ℓ.

Let us illustrate strides on the example in Figure 8.

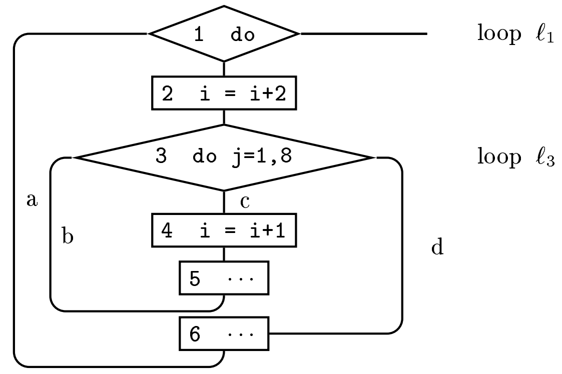


Figure 8: Stride information

- $\mathrm{Down}_{\mathtt{i}}(3,5) = \lhd_1$ is the evolution along the path $3,4,5$;
- $\mathrm{Up}_{\mathtt{i}}(5,3) = \diamond$ is the evolution along the path $5,3$;
- $\mathrm{Stride}_{\mathtt{i}}(\ell_3) = \lhd_1$ is the evolution along the path $3,4,5,3$;
- $\mathrm{Down}_{\mathtt{i}}(1,3) = \lhd_2$ is the evolution along all paths $1,2,3(,4,5,3,\cdots)$. This path traverses 2 once, but may not traverse any node in ℓ_3;
- $\mathrm{Up}_{\mathtt{i}}(3,1) = \lhd_0$ is the evolution along all paths $(3,4,5,\cdots,)3,6,1$. It traverses 6, but may *not* traverse any node in ℓ_3;
- $\mathrm{Down}_{\mathtt{i}}(1,5) = \lhd_3$ is the evolution along paths $1,2,3,(4,5,3,\cdots,)4,5$. It enters the inner loop at least once (reach 5) and traverses statement 2 exactly once;
- $\mathrm{Up}_{\mathtt{i}}(5,1) = \lhd_0$ is the evolution along all paths $5,3,(4,5,3,\cdots,)6,1$.
- $\mathrm{Stride}_{\mathtt{i}}(\ell_1) = \lhd_{10}$ is the evolution that traverses one iteration of loop ℓ_1. It traverses 2 once and the entire inner loop once (knowing ℓ_3 has 8 iterations).

6.3 Computing Stride

To compute evolution efficiently, we break down evolutions into up- and down-strides. This section describes how to compute strides efficiently. We start with local strides, which are computed by the basic algorithm. Then, non-local strides are computed as the composition of local strides.

Local Strides

We first show how to compute $\mathrm{Stride}_{\mathtt{i},N}(\ell)$. The algorithm starts from the innermost loops and proceeds outwards. Consider an innermost loop ℓ_x. $\mathrm{Stride}_{\mathtt{i},N}(\ell_x)$ can be computed by the basic algorithm. Then, ℓ_x is reduced to a single abstract node x, where exit edges of ℓ_x now leave from x instead, and the incoming edges of the loop header (excluding those from within the loop) now point to x. The transfer function for node x is,

$$f_x(in) = in \circ (\mathrm{Stride}_{\mathtt{i},N}(\ell_x) \times it_x),$$

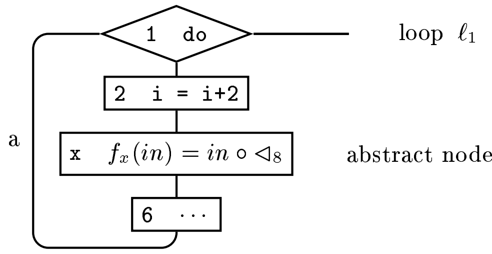


Figure 9: Abstract nodes

where it_x is the minimal number of iterations of ℓ_x. Loops with early exits or `continue` statements have $it_x = 1$. When all the innermost oops are reduced to single abstract nodes, loops immediately enclosing them become "innermost" and the stride can be computed by the basic algorithm.

Considering Figure 8, $\mathrm{Stride}_{\mathtt{i}}(\ell_1)$ is computed in three steps.

1. $\mathrm{Stride}_{\mathtt{i}}(\ell_3) = \lhd_1$ is computed by the basic algorithm.
2. Loop ℓ_3 is abstracted to a single node x as shown in Figure 9; knowing ℓ_3 executes exactly 8 iterations, the transfer function is

$$f_x(in) = in \circ (\lhd_1 \times 8) = in \circ \lhd_8.$$

3. $\mathrm{Stride}_{\mathtt{i}}(\ell_1) = \lhd_{10}$ is computed by the basic algorithm on the transformed graph.

When p is not a loop header, $\mathrm{Down}_{\mathtt{i},N}(h_p, p)$ and $\mathrm{Up}_{\mathtt{i},N}(p, h_p)$ can be computed by the same algorithm using the appropriate starting and ending nodes.

When p is the header of a loop ℓ_0 that is immediately enclosed by ℓ_1, local strides of p (h_0) are computed:

$$\mathrm{Up}_{\mathtt{i},N}(h_0, h_1) = \mathrm{Up}_{\mathtt{i},N\cup O\setminus E}(h_0, h_1) \circ (\mathrm{Stride}_{\mathtt{i},N}(\ell) \times 0) \quad (7)$$

$$\mathrm{Down}_{\mathtt{i},N}(h_1, h_0) = \mathrm{Down}_{\mathtt{i},N\cup O}(h_1, h_0) \circ (\mathrm{Stride}_{\mathtt{i},N}(\ell) \times 0), \quad (8)$$

where O is the set of outgoing edges of h_0, and E is the exit edge of ℓ_0. By excluding edges entering loop ℓ_0, $\mathrm{Down}_{\mathtt{i},N\cup O}(h_1, h_0)$ and $\mathrm{Up}_{\mathtt{i},N\cup O\setminus E}(h_0, h_1)$ can be computed by the basic algorithm.

Consider Figure 8 again, the local strides of 3 can be computed as,

$$\begin{aligned}\mathrm{Up}_{\mathtt{i}}(3,1) &= \mathrm{Up}_{\mathtt{i},c}(3,1) \circ (\mathrm{Stride}_{\mathtt{i}}(\ell_3) \times 0) \\ &= \diamond \circ (\lhd_1 \times 0) = \lhd_0 \\ \mathrm{Down}_{\mathtt{i}}(1,3) &= \mathrm{Down}_{\mathtt{i},\{c,d\}}(1,3) \circ (\mathrm{Stride}_{\mathtt{i}}(\ell_3) \times 0) \\ &= \lhd_2 \circ (\lhd_1 \times 0) = \lhd_2\end{aligned}$$

Non-local Strides

Consider a statement p enclosed in a loop nest $\ell_n, \ldots, \ell_0$ where $\ell_0 = \ell_p$. Let us consider the down-stride of p for an arbitrary header h_{k+1}. Since any path from h_{k+1} to p also traverses h_k, $\mathrm{Down}_{\mathtt{i},N}(h_{k+1}, p)$ can be decomposed into two segments: one is from h_{k+1} to h_k; the other is from h_k to h_p. $\mathrm{Up}_{\mathtt{i},N}(p, h_{k+1})$ can be decomposed in a similar way:

$$\mathrm{Up}_{\mathtt{i},N}(p, h_{k+1}) = \mathrm{Up}_{\mathtt{i},N}(p, h_k) \circ \mathrm{Up}_{\mathtt{i},N}(h_k, h_{k+1}) \quad (9)$$

$$\mathrm{Down}_{\mathtt{i},N}(h_{k+1}, p) = \mathrm{Down}_{\mathtt{i},N}(h_{k+1}, h_k) \circ \mathrm{Down}_{\mathtt{i},N}(h_k, p) \quad (10)$$

Since $\mathrm{Down}_{\mathtt{i},N}(h_{k+1}, h_k)$ and $\mathrm{Up}_{\mathtt{i},N}(h_k, h_{k+1})$ are local strides, this leads to an inductive computation of non-local strides using (9) and (10).

Using previously computed stride information, we apply these equations to the example in Figure 8.

$$\begin{aligned}\mathrm{Up}_{\mathtt{i}}(5,1) &= \mathrm{Up}_{\mathtt{i}}(5,3) \circ \mathrm{Up}_{\mathtt{i}}(3,1) \\ &= \diamond \circ \lhd_0 = \lhd_0 \\ \mathrm{Down}_{\mathtt{i}}(1,5) &= \mathrm{Down}_{\mathtt{i}}(1,3) \circ \mathrm{Down}_{\mathtt{i}}(3,5) \\ &= \lhd_2 \circ \lhd_1 = \lhd_3\end{aligned}$$

6.4 Non-iterative Algorithm

We give a non-iterative algorithm to compute evolutions using the basic algorithm and the stride information. The algorithm is presented in two steps. First, we describe how to compute evolutions that traverse an intermediate loop header. Then, we show how to compute general evolutions.

1. The first step computes evolutions that traverse intermediate loop headers. Such evolutions are needed in the next step to compute general evolutions (in form of $p \dashv_{\mathtt{i}}^{N} q$). They are also used in dependence tests to determine loop-carried dependences.

   Consider such an evolution, $p \dashv_{\mathtt{i}}^{N} h_0 \dashv_{\mathtt{i}}^{N} q$, where loop ℓ_0 encloses both p and q and is enclosed in a loop nest, $\ell_n, \ldots, \ell_1$. The algorithm proceeds as follows. For each loop ℓ_k where $0 \leq k \leq n$, it computes an intermediate result $cycl_k$ starting from ℓ_0: $cycl_k$ is the evolution of `i` from p to q without traveling outside ℓ_k. Finally, $p \dashv_{\mathtt{i}}^{N} h \dashv_{\mathtt{i}}^{N} q$, which is confined to the outermost loop ℓ_n, is computed as $cycl_n$. We first describe how to compute $cycl_0$. Any path from p to q via h_0 within ℓ_0 can be split into three segments:

$$cycl_0 = \mathrm{Up}_{\mathtt{i},N}(p, h_0) \circ (\mathrm{Stride}_{\mathtt{i},N}(\ell_0) \times 0) \circ \mathrm{Down}_{\mathtt{i},N}(h_0, q). \quad (11)$$

   Now, consider any $cycl_{k+1}$. Then, the corrsponding evolution is either confined to ℓ_k (i.e., $cycl_k$) or travels outside ℓ_k but within ℓ_{k+1} (i.e., from p

to q via h_{k+1}). Hence,

$$cycl_{k+1} = cycl_k \sqcup \big(\mathrm{Up}_{\mathtt{i},N}(p, h_{k+1}) \circ (\mathrm{Stride}_{\mathtt{i},N}(\ell_{k+1}) \times 0) \circ \mathrm{Down}_{\mathtt{i},N}(h_{k+1}, p)\big) \quad (12)$$

This leads to an inductive computation of $p \dashv_{\mathtt{i}}^{N} h_0 \dashv_{\mathtt{i}}^{N} q$ as $cycl_n$ using (11) and (12).

2. The second step computes general evolutions in form of $p \dashv_{\mathtt{i}}^{N} q$. Let $\ell_{p,q}$ be the loop that immediately encloses p and q. Paths traversed by $p \dashv_{\mathtt{i}}^{N} q$ can be separated in two sets, depending on whether they traverse $h_{p,q}$ or not:

$$p \dashv_{\mathtt{i}}^{N} q = p \dashv_{\mathtt{i}}^{N \cup B} q \sqcup p \dashv_{\mathtt{i}}^{N} h_{p,q} \dashv_{\mathtt{i}}^{N} q \quad (13)$$

where B is the set of back-edges of $\ell_{p,q}$. Since the second term in (13) is computed in the first step, we focus on describing the computation of $p \dashv_{\mathtt{i}}^{N \cup B} q$.

Let ℓ_1 (resp. ℓ_2) be the *outermost* enclosing loop of p (resp. q) *within* $\ell_{p,q}$. We would like to split $p \dashv_{\mathtt{i}}^{N \cup B} q$ around h_1 and h_2 into the following segments,

$$\begin{aligned} p \dashv_{\mathtt{i}}^{N \cup B} q \;=\; & \mathrm{Up}_{\mathtt{i},N}(p, h_1) \\ & \circ (\mathrm{Stride}_{\mathtt{i},N}(\ell_1) \times 0) \\ & \circ h_1 \dashv_{\mathtt{i}}^{N'} h_2 \\ & \circ (\mathrm{Stride}_{\mathtt{i},N}(\ell_2) \times 0) \\ & \circ \mathrm{Down}_{\mathtt{i},N}(h_2, q) \end{aligned}$$

where

$$N' = N \cup B \cup (O_1 \setminus E_1) \cup O_2 \quad (14)$$

where O_1 and O_2 are the sets of outgoing edges of ℓ_1 and ℓ_2, respectively; E_1 is the exit edge of ℓ_1. Basically, N' ensures $h_1 \dashv_{\mathtt{i}}^{N'} h_2$ traverses no cycle around h_1 or h_2 so that the evolution can be computed by the basic algorithm.

When p is immediately enclosed in $\ell_{p,q}$, ℓ_1 does not exist. Under such case, the composition with $\mathrm{Up}_{\mathtt{i},N}(p, h_1) \circ (\mathrm{Stride}_{\mathtt{i},N}(\ell_1) \times 0)$ is removed from (14), h_1 is replaced by p. The same applies to ℓ_2.

To illustrate step one, we compute $5 \dashv_{\mathtt{i}} 3 \dashv_{\mathtt{i}} 5$ in Figure 8. Since 3 is enclosed in ℓ_1 and ℓ_3, this evolution is computed as $cycl_1$ as follows:

$$\begin{aligned} 5 \dashv_{\mathtt{i}}^{d} 3 \dashv_{\mathtt{i}}^{d} 5 &= cycl_0 \\ &= \mathrm{Up}_{\mathtt{i}}(5,3) \\ &\quad \circ (\mathrm{Stride}_{\mathtt{i}}(\ell_3) \times 0) \\ &\quad \circ \mathrm{Down}_{\mathtt{i}}(3,5) \\ &= \diamond \circ (\lhd_1 \times 0) \circ \lhd_1 \;=\; \lhd_1 \\ 5 \dashv_{\mathtt{i}} 3 \dashv_{\mathtt{i}} 5 &= cycl_1 \\ &= cycl_0 \sqcup (\mathrm{Up}_{\mathtt{i}}(5,1) \\ &\qquad \circ (\mathrm{Stride}_{\mathtt{i}}(\ell_1) \times 0) \\ &\qquad \circ \mathrm{Down}_{\mathtt{i}}(1,5)) \\ &= \lhd_1 \sqcup (\lhd_0 \circ (\lhd_{10} \times 0) \circ \lhd_3) = \lhd_1 \end{aligned}$$

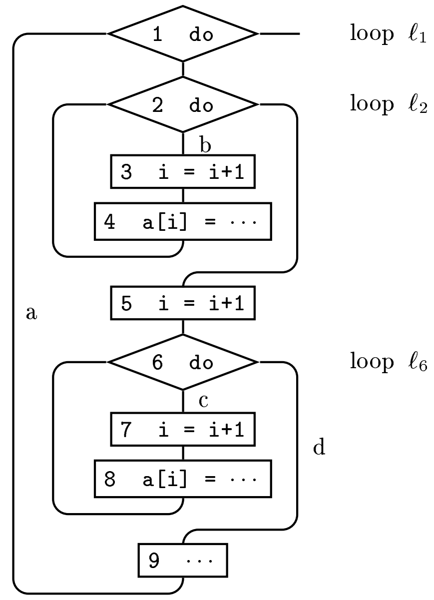


Figure 10: Evolution across loops

To illustrate (14), we compute $4 \dashv_{\mathtt{i}}^{a} 8$ in Figure 10, where ℓ_1 is the innermost loop that encloses 4 and 8.

$$\begin{aligned} 4 \dashv_{\mathtt{i}}^{a} 8 &= \mathrm{Up}_{\mathtt{i}}(4,2) \circ (\mathrm{Stride}_{\mathtt{i}}(\ell_2) \times 0) \\ &\quad \circ 2 \dashv_{\mathtt{i}}^{a,b,c,d} 6 \\ &\quad \circ (\mathrm{Stride}_{\mathtt{i}}(\ell_6) \times 0) \circ \mathrm{Down}_{\mathtt{i}}(6,8) \\ &= \diamond \circ (\lhd_1 \times 0) \circ \lhd_1 \circ (\lhd_1 \times 0) \circ \lhd_1 = \lhd_2 \end{aligned}$$

6.5 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 standard segments, i.e., up-strides, down-strides, and strides of loops. We can compute and tabulate these stride informations 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.

7. 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. The computation is performed in five passes:

Pass 1 computes local strides for every statement p involved in dependence test, and local strides for every loop ℓ in L. Using the basic algorithm, all local down-strides and local strides of loops can be computed in one traversal of the control-flow graph. To compute all local up-strides in a single traversal, however, present some difficulties. Consider statements, p and q, enclosed in a loop ℓ with a header h. If we compute $\mathrm{Up}_{\mathtt{i},N}(p)$ and then $\mathrm{Up}_{\mathtt{i},N}(q)$, some control-flow edges will be traversed twice. To overcome this problem, we can start the computation from h and follow the opposite direction of the control-flow edges in ℓ applying the basic algorithm. In this way, all local up-strides can be computed in a single traversal of the control-flow graph.

During each traversal, According to (1), for any node s, it takes one operation to compute out_s and $(e_s - 1)$ operations to compute in_s where e_s is the number of incoming edges of s. As a whole, this pass takes $2e$ operations.

Pass 2 computes $\mathrm{Down}_{\mathtt{i}}(h, p)$ and $\mathrm{Up}_{\mathtt{i}}(p, h)$ for every p and the header h of every enclosing loop of p, as well as $\mathrm{Down}_{\mathtt{i}}(h')$ and $\mathrm{Up}_{\mathtt{i}}(h')$ for every header h'. According to (7-10), each up-stride/down-stride takes at most 2 operations. In total, $4mk$ operations are needed.

Pass 3 computes $p \dashv_{\mathtt{i}}^{N} h \dashv_{\mathtt{i}}^{N} q$ for every triplet (p, q, h) and every enclosing loop of the triplet (by specifying different N), h being the header of an enclosing loop of p and q. According to (11) and (12), $p \dashv_{\mathtt{i}}^{N} h \dashv_{\mathtt{i}}^{N} q$ for different N (i.e., different enclosing loops of (p, q, h)) can be computed inductively in m steps. Each takes at most 4 operations. Then, counting all possible (p, q, h), this pass takes $4m^2k^2$ operations.

Pass 4 computes all $h_1 \dashv_{\mathtt{i}}^{N'} h_2$ where h_1 and h_2 can be any statement involved in the dependence test or the header of any surrounding loop of such statements. N' is designed such that $h_1 \dashv_{\mathtt{i}}^{N'} h_2$ will not traverse inside ℓ_1 or ℓ_2 or back-edges of any loop that surrounds h_1 and h_2. In fact, N' is fixed for any given h_1 and h_2.

Following the lines of the first pass, all $h_1 \dashv_{\mathtt{i}}^{N'} h_2$ *starting* from the same h_1 can be computed together in one traversal of the control-flow graph. Furthermore, all $h_1 \dashv_{\mathtt{i}}^{N'} h_2$ with h_1 from different loops traverse no common edges (assuming that loops are reduced to abstract nodes). Therefore, a conservative estimate is that evolutions starting from a statement p and from the header of the surrounding loop of p can be computed in one traversal of the control-flow graph (i.e., in e operations). Counting all the statements and summing up, this pass takes ek operations.

Pass 5 computes $p \dashv_{\mathtt{i}}^{N} q$ for every pair of p and q and every enclosing loop of the pair. According to (14) and (13), $p \dashv_{\mathtt{i}}^{N} q$ takes at most 7 operations. Counting all possible p, q, and N, this pass takes at most $7mk^2$ operations.

Putting them all together, the number of operations of the whole dependence test is bounded by,

$$2e + 4mk + 4m^2k^2 + ke + 7mk^2. \qquad (15)$$

The complexity of the dependence test is thus,

$$O(ek + m^2k^2). \qquad (16)$$

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 (15) and (16)). Therefore, compared to classical dependence tests *without induction variable recognition*, our scheme requires more steps.[4] However, (15) 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.

8. EXPERIMENTAL RESULTS

For our experimental studies, we used Polaris [5], 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) 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 [14], but these applications are left for future work.

In the experiment, we used Polaris to find candidate IIVs from the Perfect Club benchmark suite. Three programs have been omitted: `arc2d` and `track` because they contain no loop with IIVs, and `spice` because it

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

	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 4: Experiments with the Perfect Club benchmark suite

could not be handled by Polaris. 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 by hand. Table 4 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), that is, loops that 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. Such dependences can be handled either by finding closed form expressions and performing the substitution, or by the techniques described in the next paragraph.

Let us comment on the results. Our dependence test matches or outperforms Polaris on all but four loops with IIVs. These cases are studied in greater detail in Section 8.2. In addition, we detect 19 more loops whose only dependences came from operations on induction variables themselves that Polaris can not handle. Among them, one (in `mdg`) does have a closed form, but the dependence test in Polaris failed to handle the closed form expression. Twelve (11 in `bdna` and 1 in `dyfesm`) have no closed form expressions because the loop bounds involve array references, but using a parallel reduction scheme, they can be parallelized without much overhead. The other six (5 in `qcd` and 1 in `bdna`) 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.

We also found numerous cases where our technique reports less dependences than Polaris does. Such additional precision may be exploited by other parallelization algorithms—such as loop splitting and skewing or general expansion and scheduling algorithms—which are not implemented in Polaris.

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 would only miss three loops that can be parallelized by Polaris using closed form expressions, but finds 19 more parallel loops (or 13 without considering conditional induction variable updates) than can Polaris.

8.1 Successfully Analyzed Loop Patterns

In this section, we discuss several loops because they expose non-trivial induction variables and our dependence test is able to analyze them precisely. For some of these loops, Polaris reports false dependences.

Pattern 1: IVs spanning nested loops.
An example of such IVs has been described in Figure 1.a. Polaris succeeds in computing a closed form expression for IIV `ijk` but detects false dependences because `ijk` spans a triple-nested loop and its closed form expression is a non-affine function of loop counters and invariants. Conversely, our dependence test easily checks that `ijk` is strictly increasing between any two iterations of each loop and concludes that all of them are parallel.

Pattern 2: complex loop bounds.

```
— dyfesm sethm do15 — line 6120 —
do iss = 1,nblock-1
   irel = 1
   do i = 1,nnpss(iss)
```

```
      node = node+1
      iwherd(node,1) = iss
      iwherd(node,2) = irel
      irel = irel+nddf
   end do
end do
```

In this example, a closed form expression is still lacking for induction variable `node` because it would depend on the bound of the inner loop (`nnpss(iss)`) which is subscripted by the outer loop counter. Our analysis detects that variable `node` is strictly increasing between two references to `iwherd`. Thus, the outer loop contains no loop-carried dependence except for computing the value of `node`. In fact, values of `node` could be generated from the source code as a sum of values in array `nnpss`. The outer loop may be executed in two parts. A parallel reduction part sums elements in `nnpss`, and a simple parallel loop computes the rest of the code. Our analysis found 12 loops in `bdna` and `dyfsem` that could be parallelized this way .

Pattern 3: conditional increments.
An example of loops where IVs are conditionally incremented has been described in Figure 1.b. Because of conditionals, Polaris and other recognition techniques [2, 9, 11] fail to compute closed forms for IIVs. We can easily prove that there are no dependences on array `lisred` because `nred` is strictly increasing. However, parallelization is difficult because evaluation of `nred` still has to be sequential, yielding a pipelined execution scheme. Six loops in `qcd` and `bdna` share this pattern. One of these (in `qcd`) as shown below has increments interleaved with array accesses.

— qcd qqqlps do21 — line 1921 —

```
do ntrans = 1,2
   ...
   do i = 0,r3-1
      str(ctr) = temp
      ctr = ctr+1
   end do
   ...
   ctr = ctr+1
   ...
   if (r3.ne.0) then
      str(ctr) = 36
      ctr = ctr+1
   else
      do i = 0,t-1
         str(ctr) = 4
         ctr = ctr+1
      end do
      ...
   endif
   ...
end do
```

Because of the conditionals, Polaris finds no closed form expression for `ctr`, and neither would more powerful techniques such as [2, 9, 11]. Therefore, Polaris is able to parallelize the inner loops, but fails to discover that there is no dependence carried by the outer loop. The method proposed in [9] is able classify each static reference of `ctr` as "strictly monotonic" but not to compare `ctr` occurring in different statements (accessing to array `str`). It would thus fail to parallelize the outer loop.

Pattern 4: strides and offsets.
— adm rffti1 do108 — line 4964 —

```
do ii = 3,ido,2
   i = i+2
   fi = fi+1.
   arg = fi*argld
   wa(i-1) = cos(arg)
   wa(i) = sin(arg)
end do
```

It is quite simple to compute the closed form expression of `i` and to infer that the loop carries no dependence over array `wa`. However, our basic lattice is not expressive enough to capture the information that values of `i` for distinct iterations always differ by a distance greater than 1 (in this case, the minimal distance is 2). The distance-extended lattice solves the problem. Forty four loops with various offsets and strides have been found and successfully parallelized this way in `spec77`.

8.2 Patterns that Could not be Handled

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

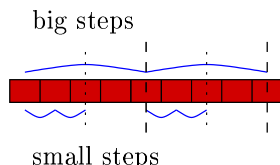
Pattern 1: Monotonic small- and big-step
— trfd olda do100 — line 331 —

```
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
```

In this case, variable `mrsij` is incremented by a "small" step (1) in every iteration of the inner loop, and is reassigned 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.

Polaris detects no dependences carried by any of the loops because the closed form expression of `mrsij` yielded disjoint intervals [4]. This is illustrated in the figure below:



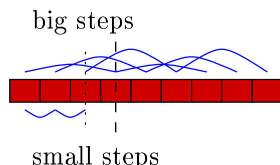
there are no dependences carried by the outer loop because the dotted lines—the last value of `mrsij`—always precede the dashed ones—the next value of `mrsij0`. Because of its inability to precisely handle arbitrary assignments (`mrsij = mrsij0`) and capture ranges of integer values, our technique may only parallelize the two inner loops of the three-nested loops based on the strict monotonicity of `mrsij`.

Pattern 2: 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 `nt`. This is illustrated in the figure below:



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`. Polaris cannot handle this pattern either.

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 [15], 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 [9] is based on a SSA representation [7] 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[11]. 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 [10], for non-nested loops only. Their technique is applied to optimizations such as efficient run-time array bounds checking. Lin and Padua [14] 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 [9], 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 [12] 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 [4].

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.

11. 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] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, USA, 1988.

[4] 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.

[5] 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.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1989.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.