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.
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 \textit{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 computational 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 \textit{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 lized.

- 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 \textit{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
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:

- unknown evolution;
- monotonically increasing;
- strictly monotonically increasing;
- monotonically decreasing;
- strictly monotonically decreasing;
- constant evolution;
- no evolution.

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

![Figure 2: The lattice of evolution states](image)

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.

![Table 1: Transfer functions of evolution values](image)
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 $p \cdot q$ is $\perp$. Intuitively, $p \cdot q$ tells us how the value of $q$ 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 $p \cdot q \cdot i$ 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 "..." 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.

- $3 \cdot 3 = \top$, since $i = 0$ is traversed along some paths from 3 to 3;
- $5 \cdot 4 = \perp$, since $b$ is excluded, 4 can not be reached from 5;
- $4 \cdot 5 = \perp$, after $b$ is excluded, there is only one path from 4 to 5, and it traverses $i = i + 1$;
- $3 \cdot 6 = \perp$, 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, $4 \cdot 5$ is evaluated to $\perp$ and $5 \cdot 4$ is $\top$. Second, the transfer function for arbitrary assignments is very conservative: $3 \cdot i$ 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, $3 \cdot i$ should rather be $\epsilon$. 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, $p \cdot q$ 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 $\text{pred}(s)$ 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, $\text{state}_q$ that holds the final result of the computation. The following data-flow equations have to be solved:

$$in_s = \bigcup_{i \in \text{pred}(s)} \text{out}_i, \quad \text{out}_s = f_s(in_s) \quad (1)$$

To compute $p \cdot q$, the iterative algorithm starts from node $p$. Initially, we have $in_p = \top$, $\text{state}_q = \perp$, and $\text{out}_s = \perp$ for all statement $s$. The algorithm iteratively solves the data-flow equations (1). During this process, every time $in_s$ is updated, the new state is joined to $\text{state}_q$, i.e., $\text{state}_q \leftarrow \text{state}_q \cup in_s$. The algorithm terminates when it reaches a fixed point, and $\text{state}_q$ yields the result of $p \cdot q$. An early termination is also possible as soon as $\text{state}_q$ 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 $\text{state}_q$ 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 $\cdot i$, 3 in Figure 3 yields $in_3 = \top$ at the first iteration after traversing 5, and $in_3 = \perp$ after traversing 5, 3, 4, 5, yet $5 \cdot 3 = \text{state}_3 = \perp$.

2.3 Evolution Composition

The composition operator ($\diamond$) "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 $p \cdot q \cdot r \cdot q$ is $\text{state}_q$ of the second evolution which is computed by the iterative algorithm starting with $in_s \leftarrow p \cdot i \cdot r$ (instead of $in_s \leftarrow \top$). Composition can be used to compute evolutions that must traverse an intermediate node. In fact,

$$p \cdot q \cdot r \cdot q = p \cdot q \cdot r \cdot q \quad (2)$$

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

\footnotesize

\begin{itemize}
  \item $\cdot i \cdot 3 = \perp$, since $i = 0$ is traversed along some paths from 3 to 3;
  \item $5 \cdot 4 = \perp$, since $b$ is excluded, 4 can not be reached from 5;
  \item $4 \cdot 5 = \perp$, after $b$ is excluded, there is only one path from 4 to 5, and it traverses $i = i + 1$;
  \item $3 \cdot 6 = \perp$, 3 may directly follow the exit edge of loop 3 to reach 6 without traversing $i = i + 1$.
\end{itemize}

\footnotesize

\[\text{state}_q = \top, \quad \text{out}_s = \perp \quad (1)\]

\footnotesize

\[\text{state}_q = \top, \quad \text{out}_s = \perp \quad (2)\]

When $q = p$, the initial value of $in_p$ ($\top$) is not joined to $\text{state}_p$ since it is not computed (updated) from the data-flow equations.
that composition is idempotent, i.e., \( a \circ a = a \); and \( \phi \) is the neutral element, i.e., \( \phi \circ a = a \). 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 \( \bullet \), which can be computed as the composition of the evolutions along 4 and 5 (i.e., \( \circ \bullet \)). Thus, we may re-define transfer functions given in Table 1 in terms of composition as follows:

- Identity\((\text{state})\) = \( \text{state} \)
- Forward\((\text{state})\) = \( \text{state} \circ \triangleleft \)
- Backward\((\text{state})\) = \( \text{state} \circ \triangleright \)
- Arbitrary\((\text{state})\) = \( \text{state} \circ \top \)

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 \( p \circ_i^N q \) where expression \( e \) is the sum of several terms. When \( e \) is a constant expression, \( p \circ_i^N q \) is constant (\( \phi \)) if \( q \) can be reached from \( p \) (otherwise, \( \perp \)). Considering expression \( -i \), the state of \( p \circ_i^{N_i} q \) is always the "opposite" of the state of \( p \circ_i^{N_i} q \). Operator \( - \) computes the "opposite" of an evolution state:

\[
\text{state} = \begin{array}{cccccccc}
\perp & \triangleleft & \triangleleft & \triangleleft & \triangleright & \triangleright & \top & \\
\perp & \circ & \circ & \circ & \circ & \circ & \circ & \\
\end{array}
\]

In general, when \( e \) is of the form \( ax \), \( p \circ_i^{N_i} q \) equals \( p \circ_i^{N_i} q \) when \( a > 0 \), or \( -(p \circ_i^{N_i} q) \) 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 \( 2i-3j+6 \).

\[
p \circ_i^{N_i} q = p \circ_i^{N_i} q \circ -(p \circ_i^{N_i} q) \circ p \circ_i^{N_i} q.
\]

### 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 \( (3 \circ_i 5 = \triangleleft) \), 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 \( 5 \circ_i 3 \), then values of \( i \) at 3 and 5 are always distinct, meaning that 3 and 5 are not dependent. In this example, however, \( 5 \circ_i 3 \) is evaluated to \( \perp \), 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 \( p \circ_i^{N_i} q \) and \( q \circ_i^{N_i} p \) cannot both be strict (\( \triangleleft, \triangleright \)) or unreachable (\( \perp \)). To prove the absence of any dependence, we may check that the evolutions between \( p \) and \( q \) are either \( \triangleleft, \triangleright \), or \( \perp \). 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 \( \ell \) 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 \( \ell \). 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 \( \ell \), if:

\[
p \circ_i^{B} \in \{\perp, \triangleleft, \triangleright\} \land q \circ_i^{B} \in \{\perp, \triangleleft, \triangleright\}
\]

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

Consider the previous example again, since \( 3 \circ_i^{B} \)}
5 = ⊥ and 5 ⊮ 3 = ⊥, 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 \( \ell \), but the same iteration of any loop that surrounds \( \ell \). Thus, evolutions used in the test must traverse a back-edge of \( \ell \), and must not traverse any exit edge of \( \ell \). 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 \( \ell \), if:

\[
p \stackrel{\ell E}{\rightarrow} h \quad q \in \{ \bot, \llcorner, \lrcorner \} \wedge q \stackrel{\ell E}{\rightarrow} p \in \{ \bot, \llcorner, \lrcorner \} \tag{4}
\]

where \( E \) is the set of exit edges of \( \ell \) and \( h \) is the header of \( \ell \).

In the previous example, since \( 3 \stackrel{1}{\rightarrow} 1 \stackrel{1}{\rightarrow} 5 = \bot \) and \( 5 \stackrel{-1}{\rightarrow} 1 \stackrel{-1}{\rightarrow} 3 = \bot \), 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 requires 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 \( n \geq 0 \), \( \llcorner_n \) and \( \lrcorner_n \) 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 \( \llcorner_1 \) and \( \lrcorner_1 \) correspond to \( \bot \) and \( \top \) of the original lattice, respectively, where the only information is that “the value has changed” in some direction; and \( \llcorner_0 / \lrcorner_0 \) correspond to \( \llcorner / \lrcorner \), 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.

<table>
<thead>
<tr>
<th>( \sqcup )</th>
<th>( \llcorner_p )</th>
<th>( \lrcorner_p )</th>
<th>( \top )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>( \llcorner_q )</td>
<td>( \llcorner_{p+q} )</td>
<td>( \llcorner_q )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( \lrcorner_q )</td>
<td>( \lrcorner_p )</td>
<td>( \lrcorner_p )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
</tbody>
</table>

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 \( \llcorner_p \) with \( \llcorner_q \) yields \( \llcorner_{p+q} \). According to this new definition, \( \circ \) is still commutative and associative, but not idempotent anymore. E.g., \( \llcorner_p \circ \llcorner_p = \llcorner_{2p} \).

<table>
<thead>
<tr>
<th>( \circ )</th>
<th>( \llcorner_p )</th>
<th>( \lrcorner_p )</th>
<th>( \top )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>( \llcorner_q )</td>
<td>( \llcorner_{p+q} )</td>
<td>( \llcorner_q )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( \lrcorner_q )</td>
<td>( \lrcorner_p )</td>
<td>( \lrcorner_p )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
</tbody>
</table>

Table 3: Distance-extended composition

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

<table>
<thead>
<tr>
<th>( \text{state} \times \text{n} )</th>
<th>( \llcorner_n )</th>
<th>( \lrcorner_n )</th>
<th>( \top )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>( \llcorner_q )</td>
<td>( \llcorner_{p+q} )</td>
<td>( \llcorner_q )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( \lrcorner_q )</td>
<td>( \lrcorner_p )</td>
<td>( \lrcorner_p )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
</tbody>
</table>

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 \( i - i+p \) 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 \( \llcorner_2 \).
This extension makes the lattice infinite, even of infinite height. However, since \( state \circ state \sqsubseteq state \) 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, \( i+d \geq i \) if \( d \geq 0 \) at \( q \). Therefore, we may compute the evolution of \( i \) from \( p \) to \( q \), then compose it with \( <q \). 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 \( \ell \), 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 \( \ell \), respectively.

- If \( d > 0 \), there is no intra-loop dependence between \( p \) and \( q \) for loop \( \ell \) when,

\[
p \vdash^d \text{h} \quad q \circ <d \not\in \{T, \varnothing, <b, \triangleright_0\} \\
\quad \land \quad q \vdash^E \text{p} \circ \triangleright_0 \not\in \{T, \varnothing, <b, \triangleright_0\}.
\]

If \( d < 0 \), \( <d \) (resp. \( \triangleright_0 \)) is replaced by \( \triangleright_{-d} \) (resp. \( \triangleright_{-d} \)) in (5).

- If \( d > 0 \), there is no loop-carried dependence between \( p \) and \( q \) for loop \( \ell \) when,

\[
p \vdash^E \text{h} \quad q \circ <d \not\in \{T, \varnothing, <b, \triangleright_0\} \\
\quad \land \quad q \vdash^E \text{h} \circ \triangleright_0 \not\in \{T, \varnothing, <b, \triangleright_0\}.
\]

If \( d < 0 \), \( <d \) (resp. \( \triangleright_0 \)) is replaced by \( \triangleright_{-d} \) (resp. \( \triangleright_{-d} \)) 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,

\[
3 \vdash_{k} 1 \vdash_{k} 5 \circ <1 \quad = \quad <2 \circ <1 = <3 \\
5 \vdash_{k} 1 \vdash_{k} 3 \circ \triangleright_1 \quad = \quad <2 \circ \triangleright_1 = <3
\]

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:

- \( h_\ell \) denotes the header of any loop \( \ell \).
- \( \ell_p \) 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 exists 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 \( \ell \) with header \( h \), two nodes \( p \) and \( q \) in \( \ell \), and a “pruned” control-flow graph that excludes edges in a set \( N \). There are two cases:

- The first case is when \( q \not= h \). The algorithm returns the value of \( p \vdash^{h_{\ell}} q \) where \( O \) is the set of outgoing edges of \( \ell \). Testing reachability is straightforward: the algorithm returns \( \bot \) 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 \( \text{pred}(s) \) be the predecessors of \( s \) inside loop \( \ell \). Initially, the algorithm sets \( in_\ell = \bot \). Then, \( in \), and \( out \), are computed for each node in using equation (1). When \( in_\ell \) is computed, the algorithm returns \( in_\ell \) and terminates.

- The second case is when \( q = h \). The algorithm returns \( \bot \), if all back-edges of \( \ell \) 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 \( out \), is computed, the final result, \( in_\ell \), is computed as the join of all \( out \) from the back-edges.

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

\(^3\)When \( p \) is the header of a loop \( \ell \), the innermost enclosing loop of \( p \) is the loop that immediately encloses \( \ell \).
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 $2 \cdot \Delta_1^3 6$ following the depth-first order: $i_2 = \infty$, $out_2 = \infty$, $i_3 = \infty$, $out_3 = \langle 1 \rangle$, $i_4 = \infty$, $out_4 = \infty$, $i_5 = \infty$, $out_5 = \langle 2 \rangle$, and $in_6 = \langle 1 \rangle \cup \langle 2 \rangle = \langle 1 \rangle$. The result is $2 \cdot \Delta_1^3 6 = in_6 = \langle 1 \rangle$. Now, consider an evolution that ends at a loop header, $2 \cdot \Delta_1^5 1$. The basic algorithm computes the evolution in two steps: it first computes $out_6 = \langle 1 \rangle$; then, 6 being the only predecessor of 1 in the loop, the result is $2 \cdot \Delta_1^5 1 = in_1 = \langle 1 \rangle$ 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 \( \ell \) with a header \( h \) and a statement \( p \) enclosed in \( \ell \). We define three strides between \( p \) and \( h \):

- \( \text{Up}_{\ell,N}(p, h) \) denotes an evolution from \( p \) up to the first \( h \) reached, excluding edges in \( \ell \). It captures the value difference of \( i \) from an instance of \( p \) to the first instance of \( h \) that follows it. \( \text{Up}_{\ell,N}(p, h) \) is called an up-stride of \( p \). When \( p \) is immediately enclosed by \( \ell \), \( \text{Up}_{\ell,N}(p, h) \) is the local up-stride of \( p \).

- \( \text{Down}_{\ell,N}(h, p) \) denotes an evolution from \( h \) down to \( p \) without traversing \( h \) twice, excluding edges in \( \ell \). It is also called a down-stride of \( p \). When \( p \) is immediately enclosed by \( \ell \), \( \text{Down}_{\ell,N}(h, p) \) is the local down-stride of \( p \).

- \( \text{Stride}_{\ell,N}(\ell) \) denotes an evolution from \( h \) to the next \( h \) excluding edges in \( \ell \) and the exit edge of \( \ell \). It is a special case of \( \text{Down}_{\ell,N}(h, p) \) and \( \text{Up}_{\ell,N}(p, h) \) when \( p = h \). Since \( \text{Stride}_{\ell,N}(\ell) \) captures the effect of iterating exactly one iteration of \( \ell \), it is called the local stride of \( \ell \).

Let us illustrate strides on the example in Figure 8.

Figure 7: Using the basic algorithm

- \( \text{Down}_1(3, 5) = \langle 1 \rangle \) is the evolution along the path 3, 4, 5;
- \( \text{Up}_1(5, 3) = \infty \) is the evolution along the path 5, 3;
- \( \text{Stride}_1(\ell_1) = \langle 1 \rangle \) is the evolution along the path 3, 4, 5, 3;
- \( \text{Down}_1(1, 3) = \langle 2 \rangle \) is the evolution along all paths 1, 2, 3, 4, 5, 3, \( \cdots \). This path traverses once, but may not traverse any node in \( \ell_3 \);
- \( \text{Up}_1(3, 1) = \langle 6 \rangle \) is the evolution along all paths 3, 4, 5, \( \cdots \) 6, 1. It traverses 6, but may not traverse any node in \( \ell_3 \);
- \( \text{Down}_1(1, 5) = \langle 10 \rangle \) is the evolution along paths 5, 3, 4, 5, 3, \( \cdots \) 6, 1.
- \( \text{Stride}_1(\ell_1) = \langle 110 \rangle \) is the evolution that traverses one iteration of loop \( \ell_1 \). It traverses 2 once and the entire inner loop once (knowing \( \ell_3 \) 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 \( \text{Stride}_{\ell,N}(\ell) \). The algorithm starts from the innermost loops and proceeds outwards. Consider an innermost loop \( \ell_2 \). \( \text{Stride}_{\ell,N}(\ell_2) \) can be computed by the basic algorithm. Then, \( \ell_2 \) is reduced to a single abstract node \( x \), where exit edges of \( \ell_2 \) 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,

\[ f_2(\text{in}) = \text{in} \circ (\text{Stride}_{\ell,N}(\ell_2) \times \Delta x). \]
where \( i_{e} \) is the minimal number of iterations of \( \ell_{e} \). Loops with early exits or continue statements have \( i_{e} = 1 \). When all the innermost loops 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, \( \text{Stride}_e(\ell_3) \) is computed in three steps.

1. \( \text{Stride}_e(\ell_3) = <1_1 \) is computed by the basic algorithm.

2. Loop \( \ell_3 \) is abstracted to a single node \( x \) as shown in Figure 9; knowing \( \ell_3 \) executes exactly 8 iterations, the transfer function is
   \[
   f_x(in) = in \circ (\langle 1_1 \times 8 \rangle) = in \circ <8>.
   \]

3. \( \text{Stride}_e(\ell_1) = <1_{10} \) is computed by the basic algorithm on the transformed graph.

When \( p \) is not a loop header, \( \text{Down}_{1,N}(h, p) \) and \( \text{Up}_{1,N}(p, h_p) \) can be computed by the same algorithm using the appropriate starting and ending nodes.

When \( p \) is the header of a loop \( \ell_0 \) that is immediately enclosed by \( \ell_1 \), local strides of \( p(h_0) \) are computed:

\[
\text{Up}_{1,N}(h_0, h_1) = \text{Up}_{1,N}[\text{Up}(h_0, h_1) \circ (\text{Stride}_{1,N}(f) \times 0)]
\]

\[
\text{Down}_{1,N}(h_1, h_0) = \text{Down}_{1,N}[\text{Down}(h_1, h_0) \circ (\text{Stride}_{1,N}(f) \times 0)].
\]

where \( O \) is the set of outgoing edges of \( h_0 \), and \( E \) is the exit edge of \( \ell_0 \). By excluding edges entering loop \( \ell_0 \), \( \text{Down}_{1,N}[O](h_1, h_0) \) and \( \text{Up}_{1,N}[O](h_0, h_1) \) can be computed by the basic algorithm.

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

\[
\text{Up}_{1,N}(3, 1) = \text{Up}_{1,N}(3, 1) \circ \text{Stride}_{1}(\ell_3) \times 0
\]

\[
= \text{Do wn}_{1,N}(\ell_3) \circ (\text{Stride}_{1}(\ell_3) \times 0)
\]

\[
\text{Down}_{1,N}(1, 3) = \text{Do wn}_{1,N}(1, 3) \circ \text{Stride}_{1}(\ell_3) \times 0
\]

\[
= \text{Do wn}_{1}(\ell_3) \circ (\text{Stride}_{1}(\ell_3) \times 0)
\]

### 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 \( p \circ \text{cycl}_q \)). They are also used in dependence tests to determine loop-carried dependences.

Consider such an evolution, \( p \circ \text{cycl}_q \). Where loop \( \ell_0 \) encloses both \( p \) and \( q \) and is enclosed in a loop nest \( \ell_0, \ldots, \ell_1 \). The algorithm proceeds as follows. For each loop \( \ell_k \) where \( 0 \leq k \leq n \), it computes an intermediate result \( \text{cycl}_k \) starting from \( \ell_0 \). \( \text{cycl}_k \) is the evolution of \( i \) from \( p \) to \( q \) without traveling outside \( \ell_k \). Finally, \( p \circ \text{cycl}_q \) is computed by \( \text{cycl}_n \), which is confined to the outermost loop \( \ell_n \), is computed as \( \text{cycl}_n \). We first describe how to compute \( \text{cycl}_k \). Any path from \( p \) to \( q \) via \( h_k \) within \( \ell_0 \) can be split into three segments:

\[
\text{cycl}_0 = \text{Up}_{1,N}(p, h_0) \circ (\text{Stride}_{1,N}(\ell_0) \times 0)
\]

Now, consider any \( \text{cycl}_{k+1} \). Then, the corresponding evolution is either confined to \( \ell_k \) (i.e., \( \text{cycl}_{k} \)) or travels outside \( \ell_k \) but within \( \ell_{k+1} \) (i.e., from \( p \) to \( q \)).
To illustrate step one, we compute $\omega_{h_{k+1}}$. Hence,

$$cyc h_{k+1} = cyc h_{k} \cup (Up_{1, N}(p, h_{k+1})$$

$\circ (Stride_{1, N}(t_{k+1} \times 0))$

$\circ Down_{1, N}(h_{k+1}, p))$ (12)

This leads to an inductive computation of $p \cdot N_{i}^{h_{0}} q$ as $cyc h_{i}$ using (11) and (12).

2. The second step computes general evolutions in form of $p \cdot N_{i}^{h_{0}} q$. Let $t_{p, q}$ be the loop that immediately encloses $p$ and $q$. Paths traversed by $p \cdot N_{i}^{h_{0}} q$ can be separated in two sets, depending on whether they traverse $h_{p, q}$ or not:

$$p \cdot N_{i}^{h_{0}} q = p \cdot N_{i}^{h_{0} \cup B} q \cup p \cdot N_{i}^{h_{p, q}} q$$ (13)

where $B$ is the set of back-edges of $t_{p, q}$. Since the second term in (13) is computed in the first step, we focus on describing the computation of $p \cdot N_{i}^{h_{0} \cup B} q$.

Let $t_{1}$ (resp. $t_{2}$) be the outermost enclosing loop of $p$ (resp. $q$) within $t_{p, q}$. We would like to split $p \cdot N_{i}^{h_{0} \cup B} q$ around $h_{1}$ and $h_{2}$ into the following segments:

$$p \cdot N_{i}^{h_{0} \cup B} q = Up_{1, N}(p, h_{1})$$

$\circ (Stride_{1, N}(t_{1} \times 0))$

$\circ h_{1} N_{i}^{h_{0} \cup B} h_{2}$

$\circ (Stride_{1, N}(t_{2} \times 0))$

$\circ Down_{1, N}(h_{2}, q)$

where

$$N' = N \cup B \cup (O_{1} \setminus E_{1}) \cup O_{2}$$ (14)

where $O_{1}$ and $O_{2}$ are the sets of outgoing edges of $t_{1}$ and $t_{2}$, respectively; $E_{1}$ is the exit edge of $t_{1}$. Basically, $N'$ ensures $h_{1} N_{i}^{h_{0} \cup B} h_{2}$ 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 $t_{p, q}$, $t_{1}$ does not exist. Under such case, the composition with $Up_{1, N}(p, h_{1}) \circ (Stride_{1, N}(t_{1} \times 0))$ is removed from (14), $h_{1}$ is replaced by $p$. The same applies to $t_{2}$.

To illustrate step one, we compute $5 \cdot N_{i}^{3} 3 \cdot N_{i}^{5}$ in Figure 8. Since $3$ is enclosed in $t_{1}$ and $t_{5}$, this evolution is computed as $cyc h_{i}$ as follows:

$$5 \cdot N_{i}^{3} 3 \cdot N_{i}^{5} = cyc h_{3}$$

$$= Up_{1}(3, 5)$$

$\circ (Stride_{1}(t_{3} \times 0))$

$\circ Down_{1}(3, 5)$

$$= \circ \circ (<1 \times 0) \circ <1 = <1$$

$$5 \cdot N_{i}^{3} 3 \cdot N_{i}^{5} = cyc h_{3}$$

$$= cyc h_{3} \cup (Up_{1}(3, 1))$$

$\circ (Stride_{1}(t_{1} \times 0))$

$\circ Down_{1}(1, 5)$

$$= <1 \cup (<10 \circ (<10 \times 0) \circ <1) = <1$$

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 \( p^{-i_{1}} h^{-i_{2}} q \) and \( p^{-i_{3}} q \), 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 \( \ell \) in \( L \). Using the basic algorithm, all local down- and local strides of loops can be computed in one traversal of the control-flow graph. To compute all local up- and local strides in a single traversal, however, present some difficulties. Consider statements \( p \) and \( q \) enclosed in a loop \( \ell \) with a header \( h \). If we compute \( U_{p, h}(q) \) and then \( U_{p, h}(q) \), 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 \( \ell \) applying the basic algorithm. In this way, all local up- and local 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 \( \text{out}_{s} \) and \( (e_{s} - 1) \) operations to compute \( \text{in}_{s} \) where \( e_{s} \) is the number of incoming edges of \( s \). As a whole, this pass takes \( 2e \) operations.

**Pass 2** computes \( D_{p}(h, p) \) and \( U_{p}(p, h) \) for every \( p \) and the header \( h \) of every enclosing loop of \( p \), as well as \( D_{p}(h') \) and \( U_{p}(h') \) 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 \( p^{-i_{1}} h^{-i_{2}} q \) 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), \( p^{-i_{1}} h^{-i_{2}} q \) 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 \( 4m^{2}k^{2} \) operations.

**Pass 4** computes all \( h'^{-i_{1}'} h_{2} \) 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 \( h_{1}^{-i_{1}'} h_{2} \) will not traverse inside \( I_{1} \) or \( I_{2} \) 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 \( h_{1}^{-i_{1}'} h_{2} \) starting from the same \( h_{1} \) can be computed together in one traversal of the control-flow graph. Furthermore, all \( h_{1}^{-i_{1}'} h_{2} \) 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., \( e \) operations). Counting all the statements and summing up, this pass takes \( ek \) operations.

**Pass 5** computes \( p^{-i_{1}'} q \) for every pair of \( p \) and \( q \) and every enclosing loop of the pair. According to (14) and (13), \( p^{-i_{1}'} q \) 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:

\[
2e + 4mk + 4m^{2}k^{2} + ke + 7mk^{2}. \tag{15}
\]

The complexity of the dependence test is thus:

\[
O(ek + m^{2}k^{2}). \tag{16}
\]

Any flow-sensitive, statement-wise dependence test for \( k \) statements in a loop nest of depth \( m \) must take at least \( mk^{2} \) steps. However, equations (15) estimates the number of operations involved in the dependence test: the formula gives a fairly accurate account of the cost of the test. On the other hand, 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 idiomatic 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. The context of dependence testing for array accesses, we focus on *integer induction variables* (IVs) which are used in array subscripts, and we do not deal with IVs unrelated to any dependences, e.g., IVs used in subscripts for arrays that only appear in right-hand side. Other IVs 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 IVs from the Perfect Club benchmark suite. Three programs have been omitted: *arc2d* and *track* because they contain no loop with IVs, and *spice* because it

\[ n \text{ times, when testing a large number of array accesses, i.e., when } k \text{ is close to } e. \]
Parallel Loops with IVs

<table>
<thead>
<tr>
<th>Total</th>
<th>Subscript</th>
<th>Targeted</th>
</tr>
</thead>
<tbody>
<tr>
<td>adm</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>bdna</td>
<td>63</td>
<td>62</td>
</tr>
<tr>
<td>dyfesm</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>flo52</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>mdg</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>mg3d</td>
<td>97</td>
<td>97</td>
</tr>
<tr>
<td>ocean</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>qcd</td>
<td>69</td>
<td>69</td>
</tr>
<tr>
<td>spec77</td>
<td>99</td>
<td>59</td>
</tr>
<tr>
<td>trfd</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Polaris</th>
<th>Monotonic</th>
<th>w/ Distance</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>22</td>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>58</td>
<td>63</td>
<td>63</td>
<td>63</td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 4: Experiments with the Perfect Club benchmark suite

Let us comment on the results. Our dependence test matches or outperforms Polaris on all but four loops with IVs. 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.**

```plaintext
$\text{dyfesm \ sethm do15 \ line \ 6120}$
$\text{do \ iss = 1, nblock-1}$
$\text{irel \ - \ 1}$
$\text{do \ i = 1, npss(iss)}$
```
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 (`nnps(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 `nnps`. The outer loop may be executed in two parts. A parallel reduction part sums elements in `nnps`, 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 IVs. 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.

```plaintext
node = node+1
iwherd(node,1) = iSS
iwherd(node,2) = irel
irel = irel+ndddf
end do
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.**

```plaintext
adm rfft1 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**

```plaintext
trfd olda do100 — line 331 —
do mrs = 1,nrs
  ...
  ... mrsij = mrsij0
  ...
 ...
do mj = 1,mi
    ...
    ...
    ...
end do
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 \( m_{rsij} \) yielded disjoint intervals [4]. This is illustrated in the figure below:

```
big steps

small steps
```

there are no dependences carried by the outer loop because the dotted lines—the last value of \( m_{rsij} \)—always precede the dashed ones—the next value of \( m_{rsij0} \). Because of its inability to precisely handle arbitrary assignments (\( m_{rsij} = m_{rsij0} \)) 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 \( m_{rsij} \).

**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 correco do1000 — line 980 —
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:

```
big steps

small steps
```

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 Am marguellat 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 Haghhighat and Polychronopoulos for the Parafase 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 Specialelli 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 \( \phi \)-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
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].

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 88-70657 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