This is an extended version of a paper published in the proceedings of the IEEE IWIA workshop, 2004.

Array Data Dependence Testing with the Chains of Recurrences Algebra*

Robert A. van Engelen Johnnie Birch Kyle A. Gallivan
Department of Computer Science and School of Computational Science and Information Technology
Florida State University
FL32306, USA

Abstract

This paper presents a new approach to dependence testing in the presence of nonlinear and non-closed array index expressions and pointer references. The chains of recurrences formalism and algebra is used to analyze the recurrence relations of induction variables, and for constructing recurrence forms of array index expressions and pointer references. We use these recurrence forms to determine if the array and pointer references are free of dependences in a loop nest. Our recurrence formulation enhances the accuracy of standard dependence algorithms such as the extreme value test and range test. Because the recurrence forms are easily converted to closed forms (when they exist), induction variable substitution and array recovery can be delayed until after the loop is analyzed.

1. Introduction

Accurate dependence testing is critical for the effectiveness of restructuring and parallelizing compilers. Several types of loop optimizations for improving program performance rely on exact or inexact array data dependence testing [4, 6, 7, 9, 11, 12, 15, 16, 17, 18, 19, 20, 22, 25, 26, 28, 29, 35, 40]. Current dependence analyzers are quite powerful and are able to solve complicated dependence problems on affine array index expressions. However, recent work by Psarris et al. [21, 23], Franke and O'Boyle [10], Wu et al. [36], van Engelen et al. [33, 34] and earlier work by Shen, Li, and Yew [27], Haghighat [14], and Collard et al. [8] mention the difficulty dependence analyzers have with nonlinear symbolic expressions, pointer arithmetic, and conditional control flow in loop nests.

This paper presents a new approach to dependence testing on nonlinear array index expressions and pointer references in loops with conditionally updated induction variables and common forms of pointer arithmetic.

Most closely related to our work is the work by Wu et al. [36]. They propose an approach for dependence testing without closed form computations. As with our method, the application of *induction variable substitution* (IVS) can be delayed until after dependence testing. However, their method cannot handle dependence problems in which induction variable step sizes are relevant, such as in the TRFD and MDG benchmarks. In contrast, our method uses the inherent monotonicity information of the recurrence forms to determine that the loops in these benchmarks are dependence free. In addition, our recurrence forms are easily converted to closed forms (when they exist) and our method does not require the application of an extra IVS algorithm.

Our dependence test is also applicable to pointer references. Because pointers are frequently used in C code to step through arrays, there is a need to effectively analyze the dependences of pointer references to assess parallelism and enable performance-critical optimizations. The *array recovery* method by Franke and O'Boyle [10] converts pointer references to array accesses to enable conventional array-based compiler analysis. Their work has several assumptions and restrictions. In particular, their method is restricted to structured loops with constant bounds and all pointer arithmetic must be data independent. Furthermore, pointer assignments within a loop nest are not permitted. In contrast, our method directly applies dependence testing on pointer references without restrictions and avoids or delays array recovery.

Our approach to dependence testing exploits the fact that any affine, polynomial, or geometric index expression composed over a set of *generalized induction variables* (GIVs) forms a recurrence relation. Because the chains of recurrences algebra is closed under the addition and multiplication of polynomials and geometric functions, the computation of the recurrence relations of index expressions and pointer references is straightforward. Our nonlinear dependence test uses these recurrence forms to solve a dependence problem. When closed forms of recurrence relations

* Supported in part by NSF grants CCR-0105422, CCR-0208892, and DOE grant DEFG02-02ER25543.

```
        p = A
        ...
        for i = a to b
          ...
S1:     A[...i...j...k...m...] = ...
S2:     ... = A[...i...j...k...m...]
S3:     ... = *p++
        j = j+k
        k = k+1
        m = m<<1
          ...
        endfor
```

Figure 1. Array Data Dependence Testing

do not exist, our test can, for many cases, still determine whether array and pointer accesses are independent.

More specifically, consider the example loop nest shown in Figure 1. The loop exhibits linear induction variables i and k, a quadratic GIV j, and a geometric GIV m. The array index expressions of A in statements *S1* and *S2* are assumed to be compositions of the induction variables i, j, k, and/or m using addition, subtraction, division, and multiplication. Our data dependence analysis computes recurrence forms for the induction variables j, k, and m, and pointer access p in *S3*, and determines the recurrence forms for the index expressions of A in *S1* and *S2* using the chains of recurrence algebra. Our data dependence test compares the recurrence forms of these index expressions to determine whether a loop-carried dependence may exist between statements *S1*, *S2*, and *S3*.

The remainder of this paper is organized as follows. In Section 2 we briefly introduce the chains of recurrences formalism and algebra. The chains of recurrences notation is used throughout this paper. Section 3 presents and algorithm for solving recurrence systems. The objective of the algorithm is to find the recurrence forms of induction variables and pointer updates in a loop nest. The algorithm does not attempt to construct closed forms, but rather computes the solutions in the chains of recurrences form for data dependence testing. Our data dependence tests are discussed in Section 4. Finally, Section 5 summarizes our results.

2. The Chains of Recurrences Formalism

This section briefly introduces the chains of recurrences (CR) formalism. For more details, we refer to [2, 32, 34]. The formalism was originally developed by Zima [37, 38, 39] and later improved by Bachmann, Zima, and Wang [2, 3] to expedite the evaluation of multivariate functions on regular grids. Our work includes extensions and applications of the CR formalism for the detection and substitution of generalized induction variables [32], for array recovery through pointer-to-array conversion [34], and for value range analysis [5]. The application to data dependence testing is the main focus of this paper.

2.1. Basic Formulation

A function or closed-form expression evaluated over a unit-distant grid with index i can be rewritten into a mathematically equivalent CR of the form (see [2]):

$$\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \cdots, \odot_k, \phi_k\}_i$$

where ϕ are coefficients consisting of constants or functions (symbolic expressions) independent of i, or nested CR forms, and $\odot$ are the operators $\odot = +$ or $\odot = *$. The coefficient ϕ_k may be a function of i, i.e. $\phi_k = f_k(i)$.

2.2. CR Semantics

A CR form $\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \cdots, \odot_k, \phi_k\}_i$ represents a set of recurrence relations over a grid $i = 0, \ldots, n-1$ as defined by the following template:

$$\begin{array}{lll}
\mathsf{cr}_0 & = \phi_0 & \\
\mathsf{cr}_1 & = \phi_1 & \\
\vdots & = \vdots & \\
\mathsf{cr}_{k-1} & = \phi_{k-1} & \\
\multicolumn{3}{l}{\text{for } i = 0 \text{ to } n-1} \\
\quad \mathsf{val}[i] & = \mathsf{cr}_0 & \\
\quad \mathsf{cr}_0 & = \mathsf{cr}_0 & \odot_1\ \mathsf{cr}_1 \\
\quad \mathsf{cr}_1 & = \mathsf{cr}_1 & \odot_2\ \mathsf{cr}_2 \\
\quad \vdots & = \vdots & \vdots\ \ \vdots \\
\quad \mathsf{cr}_{k-1} & = \mathsf{cr}_{k-1} & \odot_k\ \phi_k \\
\multicolumn{3}{l}{\text{endfor}}
\end{array}$$

The loop produces the sequence val[i] of the CR form. This sequence is one-dimensional. A multidimensional loop nest is constructed for multivariate CR forms (CR forms with nested CR form coefficients), where the indices of the outermost loops are the indices of the innermost CR forms.

2.3. CR Algebra

The CR algebra defines a set of rewrite rules for the construction of CR forms for closed-form formulae. The application of the rewrite rules is straightforward and not computationally intensive. The required symbolic processing is comparable to classical constant-folding [1]. CR algebra rules are applied to add, multiply, and raise CR forms to a power. Consider for example the following subset of rules:

$$\begin{array}{c}
c * \{\phi_0, +, f_1\}_i \Rightarrow \{c\,\phi_0, +, c\,f_1\}_i \\
\{\phi_0, +, f_1\}_i + \{\psi_0, +, g_1\}_i \Rightarrow \{\phi_0 + \psi_0, +, f_1 + g_1\}_i \\
\{\phi_0, +, f_1\}_i * \{\psi_0, +, g_1\}_i \Rightarrow \{\phi_0\,\psi_0, +, \Phi_i\,g_1 + \Psi_i\,f_1 + f_1\,g_1\}_i
\end{array}$$

where f_1 and g_1 are the "tails" of $\Phi_i = \{\phi_0, +, f_1\}_i$ and $\Psi_i = \{\psi_0, +, g_1\}_i$, i.e. CR forms defining the second coefficient to the last. These rules are implemented in our library for SUIF as operations on arrays of CR coefficients.

The CR algebra is closed under the formation of the characteristic function of a GIV. Therefore, multivariate polynomials and geometric sequences have CR form equivalents.

In [30] we proved that CR forms for multivariate GIVs are normal forms. Another advantage is that the manipulation of CR forms is type safe, which means that the coefficients of CR forms of integer-valued polynomial functions and GIVs are also integer valued.

2.4. CR Construction

The CR algebra provides an efficient mechanism to construct CR forms for symbolic expressions evaluated in multidimensional iteration spaces. The translation of a closed-form symbolic expression $e_{i_1,\ldots,i_n}$ defined over a set of index variables $i_1,\ldots,i_n$ to a multivariate nested CR form is defined by:

$$\begin{array}{rcl} \mathcal{CR}(e_{i_1,\ldots,i_n}) & = & \mathcal{CR}(\mathcal{CR}(\cdots\mathcal{CR}(e_{i_1})_{i_2}\cdots)_{i_n}) \\ \mathcal{CR}(e_{i_j}) & = & e[i_j \leftarrow \Phi(i_j)] \end{array}$$

where $\Phi(i_j)$ is the CR representation of the index variable i_j. When the index variables $i_1,\ldots,i_n$ span a unit-distance grid with origin $(x_1,\ldots,x_n)$, then $\Phi(i_j) = \{x_j,+,1\}_{i_j}$ for all $j = 1,\ldots,n$. The mapping replaces variables i_j with their corresponding CR forms using substitution, denoted by $e[i_j \leftarrow \Phi(i_j)]$. The CR algebra is then applied to normalize the expression to (nested) CR forms.

CR construction with the CR algebra computes CR forms for arbitrary symbolic expressions. Consider for example the nonlinear index expression $n*j+i+2*k+1$, where $i \geq 0$ and $j \geq 0$ are index variables that span a two-dimensional iteration space with unit distance and k is an induction variable with polynomial CR form $\Phi(k) = \{0,+,0,+,1\}_i$. The CR construction yields:

$$\begin{array}{ll} & \mathcal{CR}(\mathcal{CR}(\mathcal{CR}(n*j{+}i{+}2*k{+}1))) \\ = & \mathcal{CR}(\mathcal{CR}(n*j{+}\{0,+,1\}_i{+}2*k{+}1)) & \text{(replacing } i\text{)} \\ = & \mathcal{CR}(n*\{0,+,1\}_j{+}\{0,+,1\}_i{+}2*k{+}1) & \text{(replacing } j\text{)} \\ = & n*\{0,+,1\}_j{+}\{0,+,1\}_i{+}2*\{0,+,0,+,1\}_i{+}1 & \text{(replacing } k\text{)} \\ = & \{\{1,+,n\}_j,+,1,+,2\}_i & \text{(normalize)} \end{array}$$

The CR form is also amenable to symbolic analysis to determine the properties of the function it represents, such as monotonicity and extreme values. Determining the properties of (compositions) of array index expressions is important in dependence testing for loop optimization and parallelization.

2.5. Relation to Compiler Analysis

The importance of CR construction as an analytical tool for compiler analysis is clear when considering the following classes of functions and expressions commonly encountered in practice when dealing with array index expressions and generalized induction variables.

Affine index expressions are uniquely represented by nested CR forms $\{a,+,s\}_i$ of order 1, where a is the integer-valued initial value or a nested CR form and s is the integer-valued stride in the direction of i. The formation of nested CR forms for affine expressions of dimension d requires just $\mathcal{O}(d)$ steps.

Multivariate Polynomial expressions are uniquely represented by nested CR forms of length k, where k is the maximum order of the polynomial. All $\odot$ operations in the CR form are additions, i.e. $\odot = +$. A d-dimensional k-order polynomial can be translated in $\mathcal{O}(d\,k^2)$ steps by a conversion algorithm based on matrix-vector multiplication with Newton matrices [2, 31].

Geometric expressions $a\,r^i$ are uniquely represented by the CR form $\{a,*,r\}_i$.

Characteristic functions of GIVs are uniquely represented by CR forms [30]. By definition [13], the characteristic function $\chi(i) = p(i) + a\,r^i$ of a GIV is the sum of a polynomial $p(i)$ and a geometric series $a\,r^i$.

2.6. Closed Forms

For loop parallelization it is desirable to eliminate the cross-iteration dependences induced by the recurrences defined by the induction variable updates. Methods such as IVS introduce closed forms in a loop nest to eliminate such recurrences. For the application of IVS we use the inverse mapping $\mathcal{CR}^{-1}$ to convert CR forms to closed-form functions. The inverse mapping uses our extension of the CR algebra with inverse rules [32, 34]. Multivariate generalized induction variables, i.e. sums of multivariate polynomials and geometric functions, can always be converted to closed form formulae. In particular, the coefficients of a polynomial CR form are identical to the Newton series of the polynomial. Therefore, Newton's formula for the interpolating polynomial can be used to symbolically compute closed forms $\chi(i) = \sum_{j=0}^{k} \phi_j \binom{i}{j}$ for polynomial CR forms $\Phi_i = \{\phi_0,+,\cdots,+,\phi_k\}_i$.

The inverse CR rules are applied component-wise on a multivariate CR using $\mathcal{CR}_i^{-1}$ or in all directions at once, denoted by $\mathcal{CR}^{-1}$. For certain recurrence forms a closed form may not exist. For example, when the last coefficient of a CR form is not a (symbolic) constant but a function of the CR index i, no closed form can be constructed. However, our data dependence test does not require closed forms, because the method analyzes and compares the CR forms of array index expressions.

3. Solving Systems of Recurrences

Solving the systems of recurrences defined by induction variables in a loop nest facilitates CR construction for data dependence testing, general loop analysis, and loop parallelization. CR construction applied to index expressions

and loop bounds containing induction variables requires the CR forms of these variables. The CR forms of induction variables are obtained from a loop nest using a recurrence solver. This section presents a recurrence solver for generalized induction variables that computes CR forms for conditionally updated induction variables and pointers.

3.1. General Recurrence Form

Consider the general recurrence form of a generalized induction variable in a loop:

V = V_0
for i = a to b
  ...
  V = $c * V$ + $p(i)$
  ...
endfor

where c is a numeric constant or an i-loop invariant symbolic expression, and p is polynomial in i (expressed in closed form or recurrence). Common recurrence forms found in benchmark codes have either $c = 0$ (V is equal to polynomial p), $c = 1$ (V is the partial sum of polynomial p, where p is often a numeric or symbolic constant), or $p(i) = 0$ (V is geometric).

Lemma 1 *Let $\Psi_i = \{\psi_0, +, \psi_1, +, \cdots, +, \psi_k\}_i$ be the CR form of polynomial $p(i)$. Then, the CR form $\Phi(V) = \{\phi_0, +, \phi_1, +, \cdots, +, \phi_k, *, \phi_{k+1}\}_i$ where the CR coefficients of $\Phi(V)$ are defined by*

$$\phi_0 = V_0; \qquad \phi_j = (c-1)\phi_{j-1} + \psi_{j-1}; \qquad \phi_{k+1} = c$$

Proof. The sequence of values of the recurrence is $V_0, cV_0 + p(0), c(cV_0 + p(0)) + p(1), c(c(cV_0 + p(0)) + p(1)) + p(2), \ldots$. The values $p(0), p(1), p(2), \ldots$ have CR coefficient forms $p(0) = \psi_0, p(1) = \psi_0 + \psi_1, p(2) = \psi_0 + 2\psi_1 + \psi_2, \ldots$ as defined by the loop template for Ψ_i. By substituting $p(0), p(1), p(2), \ldots$ in the sequence of values of the recurrence and by computing the symbolic difference table of the sequence, we obtain the Newton series $\phi_0, \phi_1, \phi_2, \ldots$ (lower left diagonal of the difference table):

$$\begin{array}{rcl}
\phi_0 & = & V_0 \\
\phi_1 & = & (c-1)V_0 + \psi_0 \\
\phi_2 & = & (c-1)^2 V_0 + (c-1)\psi_0 + \psi_1 \\
\phi_3 & = & (c-1)^3 V_0 + (c-1)^2\psi_0 + (c-1)\psi_1 + \psi_2 \\
\vdots & = & \vdots
\end{array}$$

The terms continue to expand up to nonzero coefficient ψ_k. After that, the series continues as multiples of $c-1$ times the previous row. Therefore, the remainder of the sequence is a geometric progression with ratio c. Combining these results, we obtain the inductive definition of $\Phi(V)$. □

3.2. Special Cases

We consider several special cases of the general recurrence form of an generalized induction variable.

- When $c = 0$, we have a non-recursive assignment

  V = $p(i)$

  Therefore, we compute the CR form $\Psi_i = \mathcal{CR}(p(i))$

  $$\Phi(V) = \Psi_i$$

  In fact, this holds for any symbolic expression $p(i)$ (not only polynomials). However, special care has to be taken to model *wrap around* induction variables in loop nests [32], where the initial value of V may be unrelated to p.

- When $c = 1$ we have a recurrence of the form

  V = V + $p(i)$

  Therefore, according to Lemma 1 we obtain

  $$\Phi(V) = \{V, +, \Psi_i, *, 1\}_i = \{V, +, \Psi_i\}_i$$

  with $\Psi_i = \mathcal{CR}(p(i))$ for any symbolic expression $p(i)$ (not only polynomials).

- When $p(i) = 0$ we have

  V = $c * V$

  Therefore, according to Lemma 1 we obtain

  $$\Phi(V) = \{V, *, c\}_i$$

  However, this also holds for any symbolic expression c (not only constant). Hence,

  $$\Phi(V) = \{V, *, \Psi_i\}_i$$

  with $\Psi_i = \mathcal{CR}(c)$.

In the above, the nested CR forms $\{V, +, \Psi_i\}_i$ and $\{V, *, \Psi_i\}_i$ are flattened to a single CR form by replacing Ψ_i with its constituent coefficients.

3.3. Coupled Recurrences

The recurrences of a set of related generalized induction variables in a loop nest may not form a simple pattern that is easy to recognize and solve. For example, multiple updates to an induction variable may occur, which may obscure the recurrence pattern. To deal with nontrivial recurrence patterns, we use a substitution approach [32] to normalize the set of assignments to scalar variables. The approach detects recurrence patterns by traversing the analyzed path through a loop body from the last to the first statement to construct the set of assignments to the scalar variables from a loop body. The set of assignments is constructed using a substitution algorithm that follows the def-use chains to replace variable uses with definitions as is illustrated in Figure 2. The result is a set of normalized assignments in which each variable is assigned at most once, similar to single static assignment (SSA) forms.

```
for i = a to b
   ...
   V = rec-expr1
   ...
   U = rec-expr2
   ...
   U = ... V ... U ...
   ...
endfor
```
(a) *Multiple Assignments*

```
for i = a to b
   ...
   V = rec-expr1
   ...
   U = ... rec-expr1 ... rec-expr2 ...
   ...
endfor
```
(b) *After Substitution*

Figure 2. Normalization of Assignments

```
for i = a to b
   ... = ... V ...
   if ... then
      V = rec-expr1
   else
      V = rec-expr2
   endif
   ... = ... V ...
endfor
```
(a) *Conditional Updates*

```
for i = a to b
   /* f(i) ≤ V ≤ g(i) */
   ... = ... V ...
   if ... then
      V = rec-expr1
   else
      V = rec-expr2
   endif
   /* f(i+1) ≤ V ≤ g(i+1) */
   ... = ... V ...
endfor
```
(b) *Dynamic Bounds*

Figure 3. Conditional Recurrences

3.4. Conditional Recurrences

Because conditionally updated induction variables may not have closed forms, we developed bounding functions to determine the dynamic range of values of these variables, where the dynamic range is a function of the loop counter variable. Because the bounding functions are indexed, the evaluation of the dynamic range leads to more accurate data dependence testing [5].

3.4.1. Dynamic Value Range Bounds. Dynamic value range bounds are functions of the indices of the iteration space that bound the possible sequence of values of a set of recurrences. We introduce *min* and *max* dynamic bounding functions of a set of CR forms. The bounding functions are CR forms that bound the sequence of values defined by the CR forms in the set. Therefore, the *min* and *max* functions are CR forms of the functions f and g that bound the values of the conditionally updated variable V as shown in in Figures 3(a) and (b).

Let $\{\Phi_i^1, \ldots, \Phi_i^n\}$ be a set of n multivariate polynomial CR forms over the same index variable i. The *minimum* CR form of the set is defined by

$$min(\Phi_i^1, \ldots, \Phi_i^n) = \{min(\mathcal{V}\Phi_i^1, \ldots, \mathcal{V}\Phi_i^n), +, min(\Delta\Phi_i^1, \ldots, \Delta\Phi_i^n)\}_i$$

and the *maximum* CR form of the set is recursively defined by

$$max(\Phi_i^1, \ldots, \Phi_i^n) = \{max(\mathcal{V}\Phi_i^1, \ldots, \mathcal{V}\Phi_i^n), +, max(\Delta\Phi_i^1, \ldots, \Delta\Phi_i^n)\}_i$$

where the *step* function $\Delta\Phi_i$ of a CR form Φ_i is defined by

$$\begin{aligned}\Delta\Phi_i &= \Delta\{\phi_0, \odot_1, \phi_1, \odot_2, \ldots, \odot_k, \phi_k\}_i \\ &= \{\phi_1, \odot_2, \ldots, \odot_k, \phi_k\}_i\end{aligned}$$

The *direction-wise step* function $\Delta_j\Phi_i$ of a multivariate CR form Φ_i is the step function with respect to an index variable j

$$\Delta_j\Phi_i = \begin{cases} \Delta\Phi_i & \text{if } i = j \\ \Delta_j\mathcal{V}\Phi_i & \text{otherwise} \end{cases}$$

The direction-wise step information indicates the growth rate of a function on an axis in the iteration space.

The initial value of $\mathcal{V}\Phi_i$ of a CR form is the first coefficient, which is the starting value of the CR form evaluated on a unit grid in the i-direction:

$$\mathcal{V}\Phi_i = \phi_0$$

3.4.2. Static Bounds. The determination of the constant static bounds on the range of possible values of a function is necessary for data dependence testing, value range analysis, and loop bounds analysis.

The *lower bound* $\mathcal{L}\Phi_i$ of a multivariate CR form Φ_i evaluated on $i = 0, \ldots, n$, $n \geq 0$, is

$$\mathcal{L}\Phi_i = \begin{cases} \mathcal{L}\,\mathcal{V}\Phi_i & \text{if } \mathcal{L}\,\mathcal{M}\Phi_i \geq 0 \\ \mathcal{L}\,\mathcal{CR}_i^{-1}(\Phi_i)[i \leftarrow n] & \text{if } \mathcal{U}\,\mathcal{M}\Phi_i \leq 0 \\ \mathcal{L}\,\mathcal{CR}_i^{-1}(\Phi_i) & \text{otherwise} \end{cases}$$

and the *upper bound* $\mathcal{U}\Phi_i$ of a multivariate CR form Φ_i is

$$\mathcal{U}\Phi_i = \begin{cases} \mathcal{U}\,\mathcal{V}\Phi_i & \text{if } \mathcal{U}\,\mathcal{M}\Phi_i \leq 0 \\ \mathcal{U}\,\mathcal{CR}_i^{-1}(\Phi_i)[i \leftarrow n] & \text{if } \mathcal{L}\,\mathcal{M}\Phi_i \geq 0 \\ \mathcal{U}\,\mathcal{CR}_i^{-1}(\Phi_i) & \text{otherwise} \end{cases}$$

where $\mathcal{CR}_i^{-1}(\Phi_i)$ is the closed form of Φ_i with respect to i (i.e. nested CR forms are not converted), and where $\mathcal{M}$ is used in tests for monotonicity of a CR form defined by

$$\mathcal{M}\Phi_i = \begin{cases} \Delta\Phi_i & \text{if } \odot_1 = + \\ \Delta\Phi_i - 1 & \text{if } \odot_1 = * \wedge \mathcal{L}\,\mathcal{V}\Phi_1 \geq 0 \wedge \mathcal{L}\,\Delta\Phi_i > 0 \\ 1 - \Delta\Phi_i & \text{if } \odot_1 = * \wedge \mathcal{U}\,\mathcal{V}\Phi_1 < 0 \wedge \mathcal{L}\,\Delta\Phi_i > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

The $\mathcal{L}$ and $\mathcal{U}$ bounds have important applications in our dependence tests discussed in Sections 3.6 and 4.

3.5. Algorithm

The algorithm presented in this section extends our previous induction variable analysis algorithm by handling conditionally updated variables in recurrences, where the recurrences may or may not have closed forms. In the new algorithm we compute multivariate CR forms for each non-aliased scalar integer and pointer variable by considering each path in a loop nest. In this way, a set of CR forms for a variable is determined, rather than a single CR form as in our previous work [32]. These CR forms describe sequences of possible values for the conditionally updated variables in a loop.

Algorithm FINDRECURRENCES(i, a, s, B, A)
Constructs the recurrence system A from the AST of loop body B
- **input:** iteration counter variable i with initial value a and stride s, and loop body B
- **output:** recurrence system A consisting of a set of $\langle V, X\rangle \in A$ pairs denoting assignments $V := X$

```
Let A := ∅
FOR each control-flow path p (up to a back edge) in B DO
  Let A_p := ∅
  FOR each statement S_k ∈ B from the last (k = |B|) to the first statement (k = 1) on path p DO
    IF S_k is an assignment statement V := X
      AND V is an integer or pointer variable
      AND X has no function calls and array accesses THEN
        UPDATE(V, X, A_p)
      Mark ⟨V, X⟩ use-before-def if V has a use on path p before this assignment
    ENDIF
  ENDDO
  ADDRECURRENCES(A, A_p)
ENDDO
```

Algorithm UPDATE(V, X, A_p)
Update the recurrence of variable V with expression X in the recurrence system A_p
- **input:** variable V, expression X, and recurrence system A_p
- **output:** updated recurrence system A_p

```
IF V ∉ Dom(A_p) THEN /* if V is not defined in A_p */
  Let A_p:=A_p ∪ {⟨V, X⟩}
ENDIF
FOR each ⟨U, Y⟩ ∈ A_p DO
  Replace each use of variable V in Y with X
ENDDO
```

Algorithm ADDRECURRENCES(A, A_p)
Add the path-specific recurrences A_p to the general recurrence system A
- **input:** recurrence systems A and A_p
- **output:** updated recurrence system A

```
IF A = ∅ THEN
  Let A := A_p
ELSE
  FOR each ⟨V, X⟩ ∈ A_p DO
    IF V ∉ Dom(A) THEN
      Let A := A ∪ {⟨V, V⟩}
    ENDIF
    Let A := A ∪ {⟨V, X⟩}
  ENDDO
  FOR each ⟨V, X⟩ ∈ A DO
    IF V ∉ Dom(A_p) THEN
      Let A := A ∪ {⟨V, V⟩}
    ENDIF
  ENDDO
ENDIF
```

Figure 4. Algorithm for Constructing a Recurrence System from a Loop

The algorithm is applied recursively from the innermost loops to the outermost loops in a (not necessarily perfectly nested) loop nest:

1. Compute the set A of variable assignments using the induction variable recognition algorithm FINDRECURRENCES(i, a, s, B, A) shown in Figure 4, where i is the name of the loop counter variable, a is the (symbolic) initial value of i, s is the (symbolic) stride, and B is the AST of the loop body.

2. Solve the recurrence system A by computing the CR forms using algorithm SOLVERECURRENCES(i, a, s, A). The $\prec$ relation used by this algorithm defines a topological order on the pairs in the set A by

$$\langle V, X\rangle \prec \langle U, Y\rangle \qquad \text{if } V \neq U \text{ and } V \text{ occurs in } Y$$

The relation ensures that the computation of the CR forms for all variables can proceed in one sweep, by first computing the CR forms for variables that do not depend on any other variables. These CR forms are then used to compute the CR forms for variables that depend on the CR forms of other variables.

3. For each variable V collect the CR forms $\Phi^j(V)$ from the pairs $\langle V, \Phi^j(V)\rangle \in A$. When only one CR form $\Phi(V)$ exists for V, obtain the closed form of the re-

Algorithm SOLVERECURRENCES(i, a, s, A)
Computes the CR-form solutions of a set of coupled recurrences over a one-dimensional iteration space
- **input:** iteration counter variable i with initial value a and stride s, and the recurrence system A consisting of a set of $\langle V, X\rangle \in A$ pairs denoting assignments $V := X$
- **output:** coupled recurrences in A are converted to uncoupled CR expressions

```
FOR each ⟨V, X⟩ ∈ A in topological order (≺) DO
  IF ⟨V, X⟩ is marked for deletion THEN
    Let A := A\{⟨V, X⟩}
  ELSE
    Let X := CR(X) /* CR construction: replace all i in X by {a, +, s}_i and apply CR algebra rules */
    IF X is of the form V + Ψ_i, where Ψ_i is a constant or closed-form expression over i or a CR form THEN
      Let Φ := {V_0, +, Ψ_i}_i
      SUBSTITUTE(V, Φ, A)
    ELSE IF X is of the form V * Ψ_i, where Ψ_i is a constant or closed-form expression over i or a CR form THEN
      Let Φ := {V_0, *, Ψ_i}_i
      SUBSTITUTE(V, Φ, A)
    ELSE IF X is of the form c * V + Ψ_i, where c is a constant or an i-loop invariant expression
         and Ψ_i is a constant or an i-loop invariant expression or a polynomial CR form THEN
      Let Φ := {φ_0, +, φ_1, +, ···, +, φ_k, *, φ_{k+1}}_i, where
          φ_0 = V_0;   φ_j = (c − 1)φ_{j−1} + ψ_{j−1};   φ_{k+1} = c
      SUBSTITUTE(V, Φ, A)
    ELSE IF V does not occur in X THEN /* potential wrap-around variable */
      Mark V wrap-around
      IF ⟨V, X⟩ is marked as use-before-def THEN
        Let Φ := {V_0 − V(B(X)), *, 0}_i + B(X)
      ELSE
        Let A := A\{⟨V, X⟩}
        Let Φ := X
      ENDIF
      SUBSTITUTE(V, Φ, A)
    ELSE /* cannot solve the recurrence for V */
      SUBSTITUTE(V, ⊥, A)
    ENDIF
  ENDIF
ENDDO
```

Algorithm SUBSTITUTE(V, Φ, A)
Substitute all occurrences of V by Φ in the recurrence system A
- **input:** variable V, CR form Φ, and recurrence system A
- **output:** updated recurrence system A

```
Replace ⟨V, X⟩ in A with ⟨V, Φ⟩
FOR each ⟨U, Y⟩ ∈ A, ⟨V, X⟩ ≺ ⟨U, Y⟩ DO
  Mark ⟨U, Y⟩ ∈ A for deletion
  Let Y' := Y[V ← Φ] /* substitute each use of V with Φ */
  Let A := A ∪ {⟨U, Y'⟩}
ENDDO
```

Figure 5. Algorithm for Solving Recurrence Systems

currence for V given by $\mathcal{CR}^{-1}(\Phi(V))$. When multiple CR forms exist, compute the *min* and *max* bounding functions over the set $\{\Phi^j(V)\}$ to determine the dynamic range of values of the variable through the loop iteration. The CR form and/or the dynamic range are used by the data dependence test.

4. To facilitate the recognition of induction variables in outer loops, the set A is used to add (conditional) variable updates at the end of the analyzed loop nest. These updates are virtual and only used to reveal the induction variables to the outer loops for further analysis. More specifically, for each variable V a set of conditional assignments are added corresponding to the tuples $\langle V, \Phi^j(V)\rangle \in A$, which is similar to the following template:

```
for i = a to b step s
  ...
endfor
i = max(0, ⌊(b − a)/s + 1⌋)
case (random(1 to j))
  of 1: V = CR^{-1}(Φ^1(V))
  of 2: V = CR^{-1}(Φ^2(V))
  ...
  of j: V = CR^{-1}(Φ^j(V))
endcase
```

A virtual case block is added for each variable. The conditional flow ensures that only one of the updates is visible on a path through the outer loop body. It is important to note that the addition of the block is virtual and only used to provide a feed back mechanism

to ensure that the recurrences are analyzed by the application of the algorithm to the outer loops.

5. As an optional step in the algorithm, IVS is applied when all variables V in the set A have single closed forms. IVS normalizes the loop and adds initializing assignments to variables V to the start of the loop and its body to remove cross-iteration dependences induced by the induction variable updates:

```
V0 = V
 :  =  :
for i = 0 to ⌊(b − a)/s + 1⌋
   V = CR^{-1}(Φ(V))
    :  =  :
   B /* normalized loop body */
endfor
i = max(0,⌊(b − a)/s + 1⌋)
V = CR^{-1}(Φ(V))
 :  =  :
```

The loop can be optimized by forward substitution to eliminate the assignments in the loop body. The elimination of the assignments requires the addition of assignments in the loop epilogue to adjust the values of the induction variables after the execution of the loop, as shown in the code template above. Special care is taken for potential wrap-around variables, whose final assignments must be guarded by a test on the nonzero trip property of the loop.

3.6. Recurrence Patterns Recognized

In this section we discuss several loops with non-trivial recurrences patterns defined by induction variable updates. Our algorithm handles the most complicated classes of GIVs, such as those found in the TRFD and MDG benchmarks. The algorithm can handle multiple assignments to induction variables, generalized induction variables in loops with symbolic bounds and strides, symbolic integer division, conditional induction expressions, cyclic induction dependencies, symbolic forward substitution, symbolic loop-invariant expressions, and wrap-around variables.

3.6.1. Nonlinear Recurrences. Consider the loop nest shown in Figure 6(a). The loop has a potential wrap-around induction variable j and nonlinear induction variables k and m. Because there is no use of j before the definition of j in the path through the loop body, the recurrence system discards j and solves for k and m, as shown in Figure 6(b). The solutions of the recurrences of k and m are computed in CR form. Figure 6(c) depicts the result of CR index construction (see Section 2.4), where the array access is determined by the CR form obtained from the solution to the recurrence system and by applying CR construction to the index expression.

The loop can be parallelized if the induction variables can be eliminated using IVS and if no output dependence on the assignment to a[i+k] exists. No output dependence can exist if the array index i+k is strictly monotonically increasing or decreasing. Therefore, we test $\mathcal{LM}\{k_0,+,k_0{+}1,+,k_0{+}1,*,2\}_i > 0$ or $\mathcal{UM}\{k_0,+,k_0{+}1,+,k_0{+}1,*,2\}_i < 0$. The first constraint is met when $k_0 + 1 > 0$ and the latter constraint is met when $k_0 + 1 < 0$. Hence, if $k_0 \neq -1$ no dependence can exist and the loop is parallelizable

The closed forms of the CR forms for variables k and m are used in the non-optimized IVS converted code shown in Figure 6(d). The result of conventional restructuring compiler optimizations applied to the IVS code is shown in Figure 6(e). The final adjustments to j, k, and m shown in Figure 6(e) are necessary to enable any uses of these variables after the loop. Because j is a potential wrap-around variable (detected by SOLVERECURRENCES), its final adjustment is conditional on the nonzero trip property of the loop.

3.6.2. Coupled Recurrences with Multiple Updates. Consider the loop nest shown in Figure 7(a) with coupled induction variables j and k. The loop contains two updates of k. The algorithm computes the recurrences and their solutions in CR form as shown in Figure 7(b). Figure 6(c) depicts the result of CR index construction, where the array accesses are determined by the CR form obtained from the solution to the recurrence system and by applying CR construction to the index expression (in which all variables are replaced by their definitions using forward substitution). We test for dependence between statements *S1* and *S2* to verify whether the loop can be parallelized.

To disprove loop-carried flow dependence between statements *S1* and *S2*, we have to show that there is no use *S2* after the definition *S1* of a[k] in subsequent iterations. The symbolic non-constant distance between the use *S2* and definition *S2* is a function defined by the CR form $\{j_0{+}k_0,+,j_0{+}3,+,2\}_i - \{k_0,+,j_0{+}1,+,2\}_i = \{j_0,+,2\}_i$, which is linear in i, i.e the function $j_0 + 2i$. This means that the distance starts with the initial value j_0 of j and grows by stride two through the iterations. Thus, no loop-carried flow dependence between *S1* and *S2* exists if $j_0 \geq 0$.

We also apply our nonlinear version of the GCD test for disproving dependence by considering whether the reads *S2* and writes *S1* to array a are interleaved. This occurs when the GCD of the CR coefficients $j_0 + 1, j_0 + 3, 2$ does not divide j_0 based on the dependence equation $\{j_0{+}k_0,+,j_0{+}3,+,2\}_i = \{k_0,+,j_0{+}1,+,2\}_i$. Note that when j_0 is odd, no dependence can exist.

Combining these results, the loop can be parallelized when $j_0 \geq 0$ or when j_0 is odd. To further parallelize the loop, IVS is applied as shown in Figures 7(d) and (e).

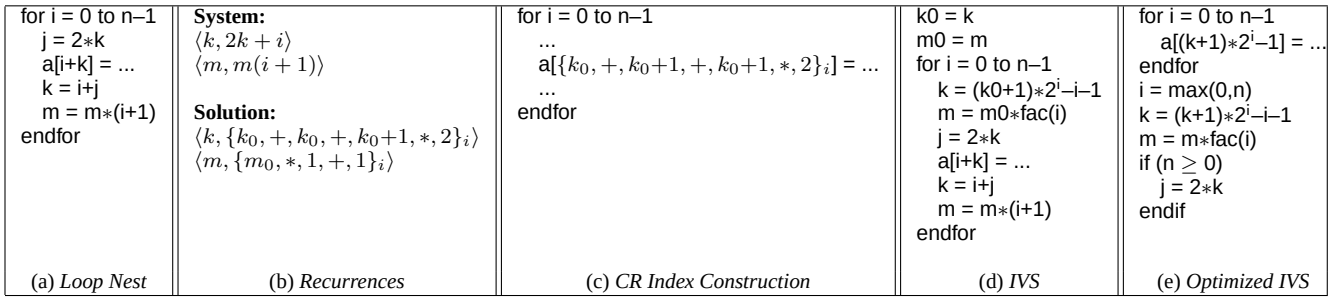

Figure 6. Nonlinear Recurrences

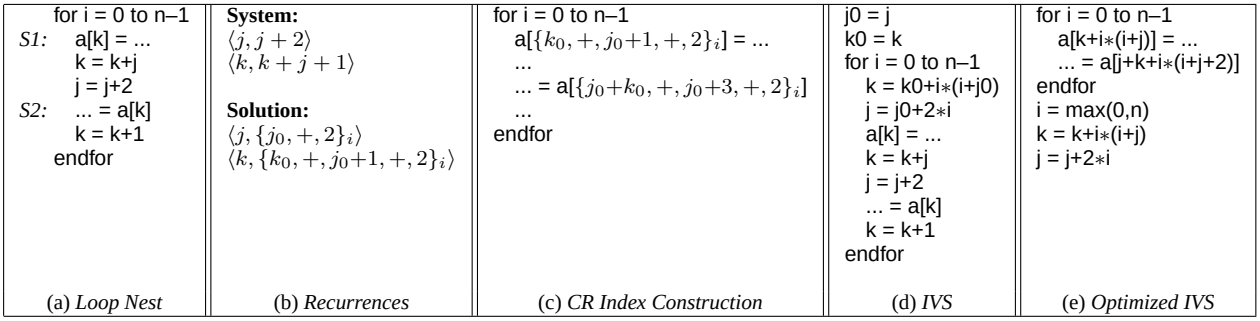

Figure 7. Coupled Nonlinear Recurrences with Multiple Updates

3.6.3. Coupled Pointer Recurrences with Multiple Updates. This example is similar to that of Section 3.6.2, but differs with respect to the use of pointer references to access memory. The loop nest shown in Figure 8(a) has recurrences and the solutions in CR form shown in Figure 8(b). Figure 8(c) depicts the result of CR index construction applied to the induction variables and pointer arithmetic. As in Section 3.6.2 we test for dependence between statements *S1* and *S2* to verify whether the loop can be parallelized.

To disprove loop-carried flow dependence between statements *S1* and *S2*, we compute the symbolic non-constant distance $\{j_0, +, j_0{+}3, +, 2\}_i - \{0, +, j_0{+}1, +, 2\}_i = \{j_0, +, 2\}_i$ between the use *S2* and definition *S2* in CR form. No flow dependence between *S1* and *S2* can exist if $j_0 \geq 0$. In addition, the GCD of the CR coefficients $j_0 + 1, j_0 + 3, 2$ does not divide j_0 if j_0 is odd. Therefore, when $j_0 \geq 0$ or when j_0 is odd, the loop can be parallelized. The application of IVS results in the non-optimized loop nest shown in Figure 8(d). Conventional restructuring compiler optimization leads to the loop nest shown in Figure 8(e), where the pointer accesses are replaced by array accesses. The result is a loop nest that reflects the application of array recovery methods.

3.6.4. Multidimensional Loops. Consider the triangular loop nest shown in Figure 9(a). The sequence of memory writes by p is strictly monotonic in the inner and outer loop nest. Therefore, no loop-carried output dependence can exist. The algorithm disproves dependence as follows.

The algorithms starts with the analysis of the inner loop shown in Figure 9(a). The recurrence system of the inner loop and its solution are shown in Figure 9(b). The CR index of the pointer access shown in Figure 9(c) is obtained by CR construction. To analyze the outer loop, the algorithm virtually adds an update to the pointer p at the the loop exit. The addition of a variable update to p is similar to the IVS code shown in Figure 9(e).

Next, the algorithm proceeds with the outer loop (using the virtually added pointer update information) shown in Figure 9(f). The recurrence system of the outer loop and its solution are shown in Figure 9(g). The CR index of the pointer access shown in Figure 9(h) is obtained by CR construction using the recurrence solution.

The simplification of $\{p_0, +, \max(0, \{1, +, 1\}_i\}_i$ to $\{p_0, +, 1, +, 1\}_i$ in the recurrence solution is accomplished by the addition of four new CR algebra rules:

$$\begin{array}{rcll} \max(\Phi_i, \Psi_i) & \Rightarrow & \Phi_i & \text{if } \mathcal{L}(\Phi_i - \Psi_i) \geq 0 \\ \max(\Phi_i, \Psi_i) & \Rightarrow & \Psi_i & \text{if } \mathcal{U}(\Phi_i - \Psi_i) \leq 0 \\ \min(\Phi_i, \Psi_i) & \Rightarrow & \Phi_i & \text{if } \mathcal{U}(\Phi_i - \Psi_i) \leq 0 \\ \min(\Phi_i, \Psi_i) & \Rightarrow & \Psi_i & \text{if } \mathcal{L}(\Phi_i - \Psi_i) \geq 0 \end{array}$$

These rules may enable the construction of a closed form for a CR form with min and max terms.

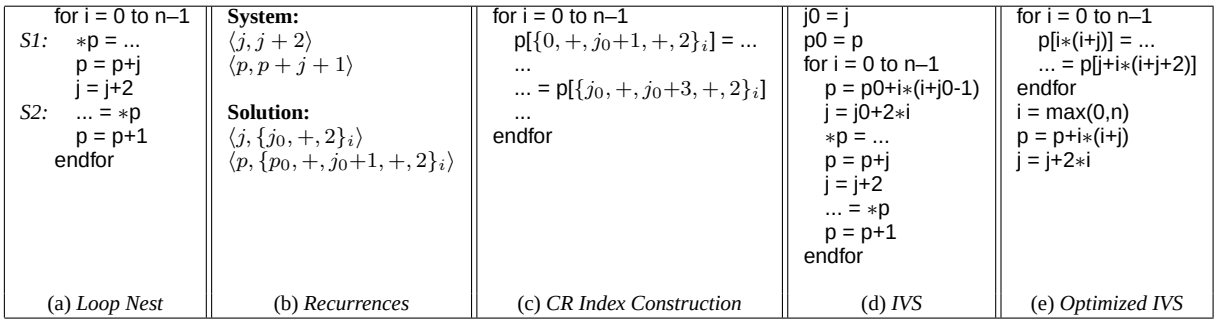

Figure 8. Coupled Nonlinear Pointer Recurrences with Multiple Updates

To determine if a loop-carried output dependence exists, we test whether the sequence of memory location accessed by the writes to p in the loop is strictly monotonic. Because $\mathcal{L}\Delta_j\{\{0,+,1,+,1\}_i,+,1\}_j = 1$ and $\mathcal{L}\Delta_i\{\{0,+,1,+,1\}_i,+,1\}_j = 1$, the sequence is strictly monotonic in the j and i directions, respectively. Therefore, the loop nest can be parallelized.

Because the CR forms of the recurrences have closed forms, IVS can be applied resulting in the loop nest shown in Figure 9(i) and the optimized code shown in Figure 9(j).

3.6.5. Recurrences with Irregular Symbolic Strides. Induction variables with irregular symbolic strides do not have closed forms. Current restructuring compilers cannot test for dependence when the recurrences in a loop nest have no closed forms. Our algorithm can determine a dependence system for these cases.

Consider the loop nest shown in Figure 10(a), where the inner loop nest is bounded by an outer-loop dependent unknown value m[i]. The algorithm proceeds by analyzing the inner loop first. The results are shown in Figures 10(b) and (c). The analysis of the outer loop requires the aggregation of the updates to the induction variables in the inner loop. The virtually added update statements to the exit of the inner loop are similar to the updates shown in Figure 10(e), where k is adjusted for recurrence analysis in the outer loop. The algorithm produces the recurrence system and solution shown in Figure 10(g) by analyzing the outer loop. Note that the solution does not have a closed form, because of the presence of the non-constant CR coefficient $\max(0, m[i] + 1)$. However, the CR construction of the index expression of the array access a[k] can still proceed. Because $m[i] \geq 0$ in the inner loop nest, the CR form of the index expression is $\{\{k_0, +, \max(0, m[i] + 1)\}_i, +, 1\}_j = \{\{k_0, +, m[i]+1\}_i, +, 1\}_j$ as shown in Figure 10(h). There is no loop-carried output dependence in the i and j directions, because $\mathcal{L}\Delta_i\{\{k_0, +, m[i] + 1, +, 1\}_i, +, 1\}_j = 1$ and $\mathcal{L}\Delta_j\{\{k_0, +, m[i] + 1, +, 1\}_i, +, 1\}_j = 1$.

3.6.6. Conditionally Updated Variables with Single Recurrence Solution. Consider the loop nest shown in Figure 11(a). The loop exhibits conditional updates of variable k. The recurrence system and its solution are shown in Figure 11(b). The variables j and k both have a single solution, despite the differences of the recurrences forms A_{p_1} and A_{p_2} on the two paths p_1 and p_2 through the loop body. This illustrates the importance of the fact that CR forms are normal forms for GIVs thereby enabling the detection of semantically equivalent recurrences.

CR construction applied to the array index expression of a results in the description of the array access in CR form shown in Figure 11(c). There are no loop-carried output dependences, because the function of the CR form $\{k_0, +, 1, +, 1, +, 1\}_i$ is strictly monotonically increasing.

Closed forms of the recurrences can be computed, because the variables of the recurrences have single solutions. The closed forms are used for IVS as shown in Figures 11(d) and (e).

3.6.7. Conditionally Updated Variables with Multiple Recurrence Solutions. Consider the loop nest shown in Figure 12(a). The loop exhibits conditional updates of variables j and k. The recurrence system and its solution are shown in Figure 12(b). In this case, the variables j and k do not have a single recurrence solution. The set of recurrence solutions is used for the CR construction of the array index expression. The *min* and *max* bounding functions are applied to the set of CR forms obtained for the array index k, resulting in the lower and upper dynamic value range bounds $\{k_0, +, \min(1, j_0)\}_i$ and $\{k_0, +, \max(1, j_0), +, 2\}_i$, respectively. Because the lower and upper bound functions on the array index expression are both strictly monotonically increasing, the array access are strictly monotonically increasing and no loop-carried output dependence exists.

3.7. Recurrence Patterns Not Recognized

In this section we give some examples of recurrence patterns that cannot be recognized and/or solved by our algorithm.

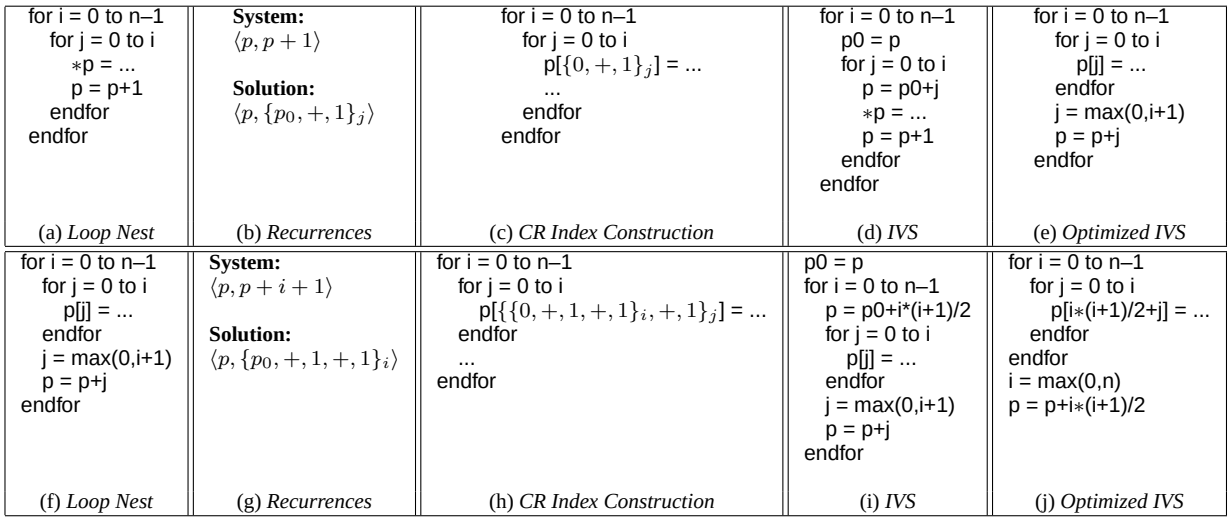

Figure 9. Recurrences in Multidimensional Non-rectangular Loop Nest

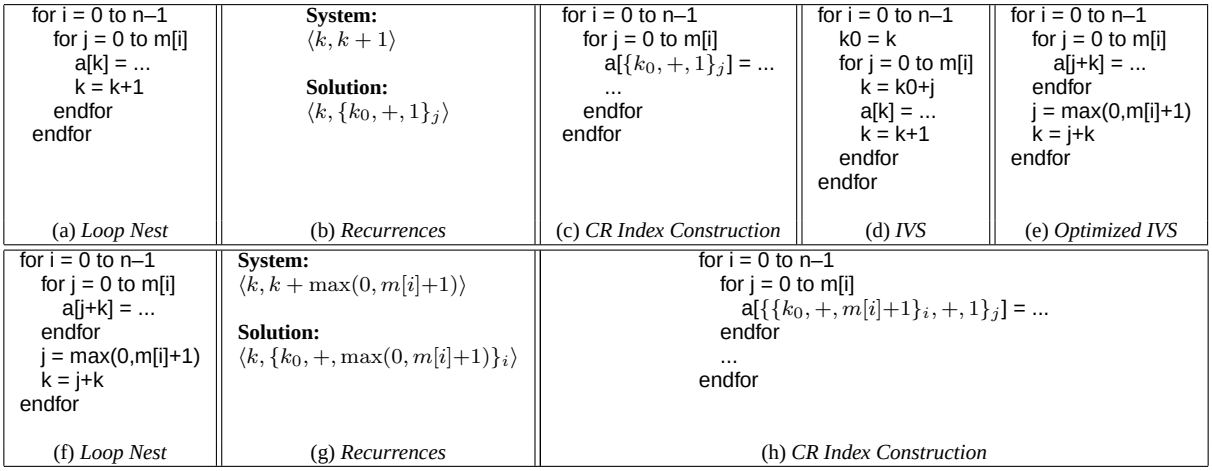

Figure 10. Recurrences with Irregular Symbolic Strides

3.7.1. Unsolvable Recurrence Patterns. Some recurrence patterns exist that cannot be solved, such as the recurrence shown in Figure 13(a). The recurrence cannot be solved by our algorithm because it has neither a CR form nor a closed-form equivalent.

3.7.2. Cyclic Recurrence Relations. These relations cannot be analyzed by our algorithm, as shown in Figure 13(b). The reason is that the recurrence system constructed from the loop nest must have a partial order $\prec$ on the assignments to solve the system using substitution. Note that this does not prohibit the coupling of recurrences.

3.7.3. No Min/Max Dynamic Bounds. These bounds cannot be formulated for conditionally updated variables with possible geometric progressions, as is shown in Figure 13(c). Therefore, dependence testing with our algorithm cannot be applied to array index expressions containing these variables.

4. Nonlinear Dependence System Solvers

This section introduces three dependence solvers. The solvers are based on our recurrence solver and do not require closed-form index expressions. The dependence solvers construct dependence systems based on the CR forms of index expressions. The dependence tests can be applied to loop nests with conditionally updated induction variables and pointers. The objective of the tests is to com-

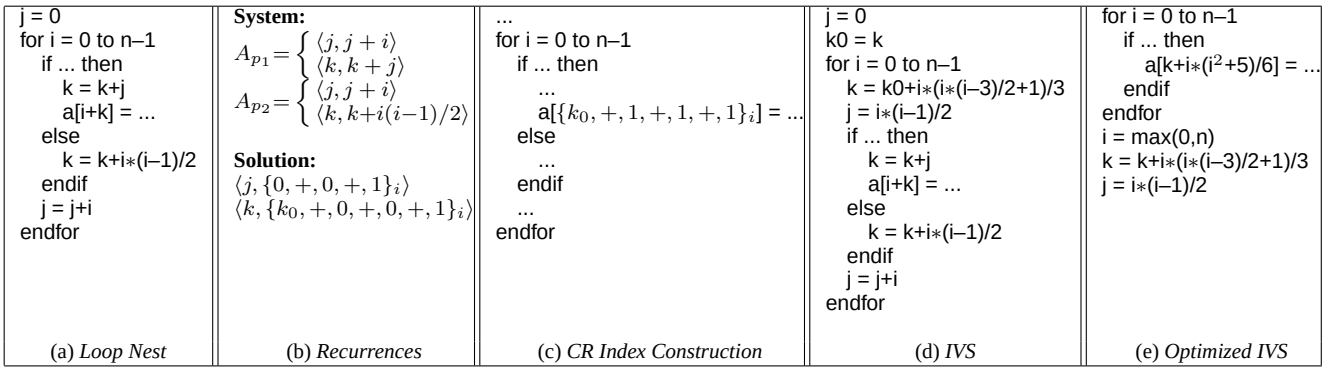

Figure 11. Conditionally Updated Variables with Single Recurrence Solution

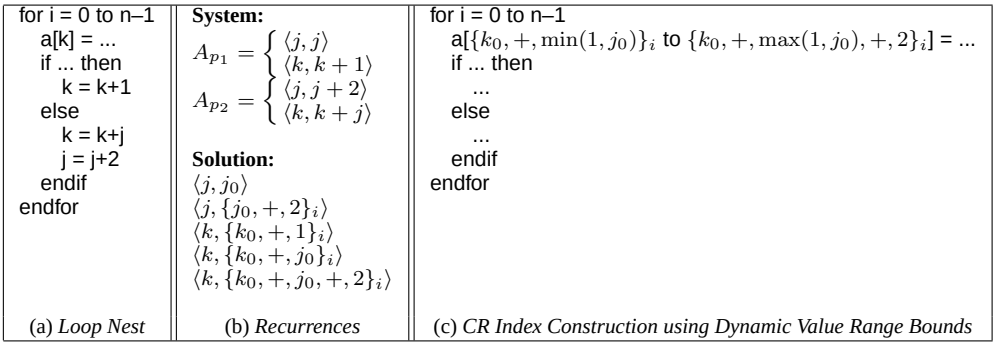

Figure 12. Conditionally Updated Variables with Multiple Recurrence Solutions

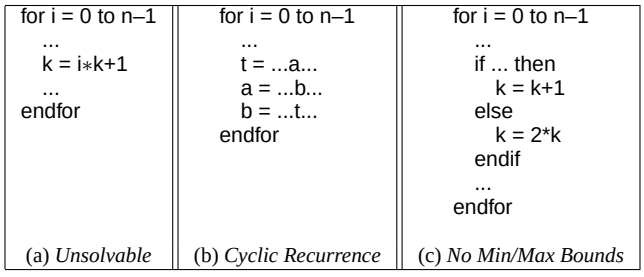

Figure 13. Recurrence Patterns not Recognized

pute the conditions under which a solution to a dependence system exists, rather than just testing for potential dependence. This allows us to generate multi-version code with parallelized versions of the code fragments when admissible by the symbolic constraints.

4.1. Monotonicity Test

This is a relatively inexpensive test to verify whether loop-carried output dependences exist for a single array or pointer reference. The test verifies the monotonic property of an array index expression and pointer reference. More elaborate dependence testing involving multiple array and pointer accesses is performed with our nonlinear version of the extreme value test described in the next section.

Consider Figure 14 depicting a segment of the original TRFD code. The CR forms of ijkl and l obtained by CR construction are

$$\begin{aligned}\Phi(\text{ijkl}) = \{\{\{\{2,+,\text{left+m(m+1)/2+2},+,\text{left+m(m+1)/2+1}\}_i \\ ,+,\text{left+m(m+1)/2}\}_j \\ ,+,\{2,+,1\}_i,+,1\}_k \\ ,+,1\}_l\end{aligned}$$

and $\Phi(l) = \{1,+,1\}_l$ respectively. The CR form Φ(ijkl) has the following four step functions in the i, j, k, and l direction, respectively:

$$\begin{aligned}\Delta_i\Phi(\text{ijkl}) &= \{\text{left+m(m+1)/2+2},+,\text{left+m(m+1)/2+1}\}_i \\ \Delta_j\Phi(\text{ijkl}) &= \text{left+m(m+1)/2} \\ \Delta_k\Phi(\text{ijkl}) &= \{\{2,+,1\}_i,+,1\}_k \\ \Delta_l\Phi(\text{ijkl}) &= 1\end{aligned}$$

```
ijkl=0
ij=0
DO i=1,m
  DO j=1,i
    ij=ij+1
    ijkl=ijkl+i-j+1
    DO k=i+1,m
      DO l=1,k
        ijkl=ijkl+1
        xijkl[ijkl]=xkl[l]
      ENDDO
    ENDDO
    ijkl=ijkl+ij+left
  ENDDO
ENDDO
```

Figure 14. TRFD Code Segment

Note that the step functions in the k and l directions are nonnegative, because the CR coefficients are nonnegative. Therefore, the growth of the ijkl induction variable in the k, l direction of the index space is nonnegative and the addressing of the xijkl[ijkl] is strictly monotonically increasing in the inner k, l loop nest, allowing the inner two loop nests to be parallelized.

Also note that the growth of ijkl in the entire i, j, k, l index space is nonnegative if left $>$ m(m+1)/2, which is in fact the case when considering the larger part of the benchmark code (not shown).

4.2. Nonlinear Extreme Value Test

This nonlinear dependence test is based on the Banerjee bounds test [4], also known as the extreme value test (EVT). The test computes direction vector hierarchy information by performing symbolic subscript-by-subscript testing for multidimensional loops. The test is inexact. However, the test is efficient to determine direction vector hierarchy information. The test builds the direction vector hierarchy by solving a set of dependence equations one at a time.

Our extended EVT subsumes these characteristics by enhancing the test to cover common nonlinear array index expressions and uses of pointer arithmetic without requiring closed forms. Thus, our nonlinear EVT can determine absence of dependence for a larger set of dependence problems compared to the standard EVT. The implementation of our algorithm is identical to the original EVT method, except that CR forms and $\mathcal{L}$ and $\mathcal{U}$ bounds are used in the computations.

Consider for example the dependences of the loop nest shown in Figure 15(a), which is part of the MDG benchmark code. The loop nest cannot be analyzed by Polaris, despite the fact that the dependence system is affine (obtained after IVS). The recurrence pattern also cannot be handled by the *monotonic evolution* test [36], because a comparison is required between the stride of the inner loop and the outer loop bound. In contrast, our CR-based extreme value test succeeds in disproving loop-carried flow dependence.

```
for i = 1 to nt
   ...
   jj = i
   for j = 1 to nor1
      var[jj] = var[jj]+...
      jj = jj + nt
   endfor
endfor
```

(a) *Loop Nest*

Equation:

$$\{\{1,+,1\}_{i^d},+,\mathsf{nt}\}_{j^d} = \{\{1,+,1\}_{i^u},+,\mathsf{nt}\}_{j^u}$$

Constraints:

$$0 \le i^d \le \mathsf{nt}-1$$
$$0 \le i^u \le \mathsf{nt}-1$$
$$0 \le j^d \le \mathsf{nor1}-1$$
$$0 \le j^u \le \mathsf{nor1}-1$$

(b) *Dependence System*

Figure 15. A Linear Dependence System

The recurrence solver and CR construction algorithms compute the multivariate CR form of the var array index expression, which is $\{\{1,+,1\}_i,+,\mathsf{nt}\}_j$, to set up the dependence equation system shown in Figure 15(b).

Testing for $(=,<)$ dependence, with $i^d = i^u$ and $j^d < j^u$, gives the normalized set of bounds for j^u and j^d:

$$\left.\begin{array}{r}1\\ \{1,+,1\}_{j^d}\end{array}\right\} \le j^u \le \mathsf{nor1}-1 \quad \Big| \quad 0 \le j^d \le \begin{cases}\mathsf{nor1}-2\\ \{-1,+,1\}_{j^u}\end{cases}$$

The simplified dependence equation from Figure 15(b) with $i^d = i^u$ is

$$\{\{0,+,\mathsf{nt}\}_{j^u},+,-\mathsf{nt}\}_{j^d} = 0$$

When applying direction vector constraints to determine the dependence hierarchy, terms must cancel when possible to ensure accuracy. Therefore, the j^u variable is selected to dominate the j^d variable in the equation, such that replacement of j^d by its upper bound constraint $\{-1,+,1\}_{j^u}$ will lead to possible cancellations in the application of the CR algebra simplification rules. The choice of dominating variable depends on the direction of the dependence test.

We proceed by computing the lower bound of the equation's left hand side

$$\begin{array}{ll}
\mathcal{L}\{\{0,+,\mathsf{nt}\}_{j^u},+,-\mathsf{nt}\}_{j^d} & \\
= \mathcal{L}(\{\{0,+,\mathsf{nt}\}_{j^u} - \mathsf{nt}\, j^d)[j^d \leftarrow \{-1,+,1\}_{j^u}]) & (\mathsf{nt} \ge 1)\\
= \mathcal{L}(\{\{0,+,\mathsf{nt}\}_{j^u} - \mathsf{nt}\,\{-1,+,1\}_{j^u}) & (\text{subst.})\\
= \mathcal{L}(\{\{0,+,\mathsf{nt}\}_{j^u} + \{\mathsf{nt},+,-\mathsf{nt}\}_{j^u}) & (\text{simplify})\\
= \mathcal{L}(\mathsf{nt}) & (\text{simplify})\\
= 1 & (\mathsf{nt} \ge 1)
\end{array}$$

Because the lower bound of the left-hand side of the equation is positive, the $(=,<)$ dependence is disproved (note that for the above $\mathsf{nt} \ge 1$ holds in the loop nest).

Testing for $(<,<)$ dependence, with $i^d < i^u$ and $j^d < j^u$, gives the normalized set of bounds:

$$\left.\begin{array}{r}1\\ \{1,+,1\}_{i^d}\end{array}\right\} \le i^u \le \mathsf{nt}-1 \quad \Big| \quad 0 \le i^d \le \begin{cases}\mathsf{nt}-2\\ \{-1,+,1\}_{i^u}\end{cases}$$
$$\left.\begin{array}{r}1\\ \{1,+,1\}_{j^d}\end{array}\right\} \le j^u \le \mathsf{nor1}-1 \quad \Big| \quad 0 \le j^d \le \begin{cases}\mathsf{nor1}-2\\ \{-1,+,1\}_{j^u}\end{cases}$$

The dependence equation is

$$\{\{\{\{0,+,-1\}_{i^u},+,1\}_{i^d},+,-\mathsf{nt}\}_{j^u},+,\mathsf{nt}\}_{j^d} = 0$$

```
p = A
q = A
for i = 0 to n–1
   for j = 0 to i
      p = p + 1
      *q = *p
   endfor
   q = q + 1
endfor
```

(a) *Loop Nest*

Equation:

$\{\mathsf{A},+,1\}_{i^d} = \{\{\mathsf{A}{+}1,+,1,+,1\}_{i^u},+,1\}_{j^u}$

Constraints:

$0 \le i^d \le n-1$
$0 \le i^u \le n-1$
$0 \le j^d \le i^d$
$0 \le j^u \le i^u$

(b) *Dependence System*

Figure 16. A Nonlinear Dependence System

The lower bound of the equation's left hand side is

$$
\begin{aligned}
&\mathcal{L}\{\{\{\{0,+,-1\}_{i^u},+,1\}_{i^d},+,-\mathsf{nt}\}_{j^u},+,\mathsf{nt}\}_{j^d} \\
&= \mathcal{L}\{\{\{0,+,-1\}_{i^u},+,1\}_{i^d},+,-\mathsf{nt}\}_{j^u} && (\mathsf{nt}\ge 1)\\
&= \mathcal{L}(\{\{\{0,+,-1\}_{i^u},+,1\}_{i^d} - \mathsf{nt}\, j^u)[j^u \leftarrow \mathsf{nor1}{-}1]) && (\mathsf{nt}\ge 1)\\
&= \mathcal{L}(\{\{0,+,-1\}_{i^u},+,1\}_{i^d} - \mathsf{nt}(\mathsf{nor1}{-}1)) && \text{(subst.)}\\
&= \mathcal{L}\{\{-\mathsf{nt}\,(\mathsf{nor1}-1),+,-1\}_{i^u},+,1\}_{i^d} && \text{(simplify)}\\
&= \mathcal{L}\{-\mathsf{nt}\,(\mathsf{nor1}{-}1),+,-1\}_{i^u}\\
&= \mathcal{L}(-\mathsf{nt}\,(\mathsf{nor1}{-}1) - (\mathsf{nt}{-}1)) && \text{(subst. + simplify)}\\
&= -\mathcal{U}(\mathsf{nt}\,(\mathsf{nor1}{-}1)) - \mathcal{U}(\mathsf{nt}{-}1)\\
&= -\infty
\end{aligned}
$$

This result is inconclusive. However, the upper bound of the equation's left hand side is negative:

$$
\begin{aligned}
&\mathcal{U}\{\{\{\{0,+,-1\}_{i^u},+,1\}_{i^d},+,-\mathsf{nt}\}_{j^u},+,\mathsf{nt}\}_{j^d} \\
&= \mathcal{U}(\{\{\{0,+,-1\}_{i^u},+,1\}_{i^d},+,-\mathsf{nt}\}_{j^u} + \mathsf{nt}\,\{-1,+,1\}_{j^u})\\
&= \mathcal{U}\{\{-\mathsf{nt},+,-1\}_{i^u},+,1\}_{i^d} && \text{(simplify)}\\
&= \mathcal{U}(\{-\mathsf{nt},+,-1\}_{i^u} + \{-1,+,1\}_{i^u}) && \text{(subst. + simplify)}\\
&= \mathcal{U}(-\mathsf{nt}{-}1) && \text{(simplify)}\\
&= -\mathcal{L}(\mathsf{nt}) - 1\\
&= -2 && (\mathsf{nt}\ge 1)
\end{aligned}
$$

Therefore, the $(<,<)$ dependence is disproved.

Our nonlinear extreme value test also handles nonlinear recurrences. Consider the example triangular loop nest depicted in Figure 16(a). Note that pointers p and q read and write to the same array A. The recurrence solver and CR construction algorithms compute the multivariate CR forms of the p and q pointer accesses, which are $\{\{\mathsf{A}{+}1,+,1,+,1\}_i,+,1\}_j$ and $\{\mathsf{A},+,1\}_i$, respectively. The dependence system is shown in Figure 16(b). The normalized dependence equation is

$$\{\{\{-1,+,-1,+,-1\}_{i^u},+,1\}_{i^d},+,-1\}_{j^u} = 0$$

Testing flow dependence, $i^d < i^u$ and $j^d < j^u$ gives the normalized set of bounds:

$$
\begin{array}{l|l}
\left.\begin{matrix}1\\ \{1,+,1\}_{i^d}\end{matrix}\right\} \le i^u \le n-1 & 0 \le i^d \le \left\{\begin{matrix}n-2\\ \{-1,+,1\}_{i^u}\end{matrix}\right.\\
\left.\begin{matrix}1\\ \{1,+,1\}_{j^d}\end{matrix}\right\} \le j^u \le \{0,+,1\}_{i^u} & 0 \le j^d \le \left\{\begin{matrix}\{-1,+,1\}_{i^d}\\ \{-1,+,1\}_{j^u}\end{matrix}\right.
\end{array}
$$

Using these constraints, we compute the lower and upper bounds:

```
    for I = 1 to M+1
S1:    A[I*N+10] = ...
S2:    ... = A[2*I+K]
       K = 2*K+N
    endfor
```

(a) *Loop Nest*

```
for i = 1 to M+1
   A[{N+10,+,N}_i] = ...
   ... = A[{K+2N,+,K+N+2,*,2}_i]
   ...
endfor
```

(b) *CR Index Construction*

Figure 17. Nonlinear Range Test Example

$$
\begin{aligned}
&\mathcal{L}\{\{\{-1,+,-1,+,-1\}_{i^u},+,1\}_{i^d},+,-1\}_{j^u} \\
&= \mathcal{L}(\{\{\{-1,+,-1,+,-1\}_{i^u},+,1\}_{i^d} - j^u)[j^u \leftarrow \{0,+,1\}_{i^u}])\\
&= \mathcal{L}(\{\{-1,+,-1,+,-1\}_{i^u},+,1\}_{i^d} - \{0,+,1\}_{i^u}) && \text{(subst.)}\\
&= \mathcal{L}\{\{-1,+,-2,+,-1\}_{i^u},+,1\}_{i^d} && \text{(simplify)}\\
&= \mathcal{L}\{-1,+,-2,+,-1\}_{i^u}\\
&= \mathcal{L}((-1-(3i^u-(i^u)^2)/2)[i^u \leftarrow n-1])\\
&= \mathcal{L}((-n^2-n)/2) && \text{(subst.)}\\
&= (-\mathcal{U}(n^2) - \mathcal{U}(n))/2\\
&= -\infty\\
&\mathcal{U}\{\{\{-1,+,-1,+,-1\}_{i^u},+,1\}_{i^d},+,-1\}_{j^u}\\
&= \mathcal{U}\{\{-1,+,-1,+,-1\}_{i^u},+,1\}_{i^d}\\
&= \mathcal{U}(\{\{-1,+,-1,+,-1\}_{i^u} + i^d)[i^d \leftarrow \{-1,+,1\}_{i^u}])\\
&= \mathcal{U}(\{-1,+,-1,+,-1\}_{i^u} + \{-1,+,1\}_{i^u}) && \text{(subst.)}\\
&= \mathcal{U}\{-2,+,0,+,-1\}_{i^u} && \text{(simplify)}\\
&= -2
\end{aligned}
$$

Because the equation has no solution since zero does not lie between $-\infty$ and -2, our nonlinear extreme value test disproves $(<,<)$ flow dependence.

4.3. Nonlinear Range Test

This dependence test performs pairwise comparisons between array index expressions to determine the direction of the dependence. The comparisons are performed on the CR forms of array index expressions obtained by the recurrence solver and CR construction algorithm. The difference between the CR forms of two index expressions is a CR form that describes the index distance as a function of the iteration space. Therefore, the extreme values of the function indicates the direction of the dependence for the entire loop iteration space of the loop nest. This test is suitable to find the conditions under which loop-carried dependence does not exist, rather than just testing for the absence of dependence.

Consider for example the loop nest shown in Figure 17(a). This example is taken from [24], because the example was used by the authors to demonstrate the impossibility by current compilers to analyze the dependences for loop parallelization. In contrast, our dependence test handles this case by deriving the conditions under which no loop-carried flow dependence exists. The recurrence solver and CR construction algorithms compute the CR forms of the index expressions as shown in Figure 17(b). The dependence direction $<$ is disproved if

$$\mathcal{L}(\{K{+}2N,+,K{+}N{+}2,*,2\}_i - \{N{+}10,+,N\}_i) \ge 0$$

That is, to verify that all uses of A in subsequent iterations do not depend on the definitions of A we determine that the lower bound of the distance as a function of i over the normalized iteration space $i = 0, \ldots, \mathsf{M}$ is nonnegative.

$$\begin{aligned}
&\mathcal{L}(\{K{+}2N, +, K{+}N{+}2, *, 2\}_i - \{N{+}10, +, N\}_i) \\
&= \mathcal{L}\{K{+}N{-}10, +, K{+}2, *, 2\}_i \\
&= \begin{cases} K{+}N{-}10 & \text{if } K{+}2 \geq 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\
&\geq 0 \qquad \text{if } K{+}N \geq 10 \text{ and } K \geq -2
\end{aligned}$$

Therefore, no loop-carried flow dependence exist when $K{+}N \geq 10$ and $K \geq -2$. Since these conditions are easily checked at runtime, a parallelized loop nest can be generated that is conditionally executed depending on the runtime evaluation of these guards.

5. Conclusions

This paper presented a new approach to dependence testing in the presence of nonlinear and non-closed array index expressions and pointer references in loop nests. Dependences are analyzed using the chains of recurrences formalism and algebra for analyzing the recurrence relations of induction variables and for constructing recurrence forms of array index expressions and pointer references without computing closed forms. Our approach to dependence testing exploits the fact that any affine, polynomial, or geometric index expression composed over a set of generalized induction variables forms a recurrence relation. Because the chains of recurrences algebra is closed under the addition and multiplication of polynomials and geometric functions, the computation of the recurrence relations of index expressions and pointer references is straightforward. Our nonlinear dependence test uses these recurrence forms to solve a dependence problem. When closed forms of recurrence relations do not exist, our test can, any many cases, still determine whether array and pointer accesses are independent.

References

[1] AHO, A., SETHI, R., AND ULLMAN, J. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.

[2] BACHMANN, O. *Chains of Recurrences*. PhD thesis, Kent State University, College of Arts and Sciences, 1996.

[3] BACHMANN, O., WANG, P., AND ZIMA, E. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *proceedings of the International Symposium on Symbolic and Algebraic Computing (ISSAC)* (Oxford, 1994), ACM, pp. 242–249.

[4] BANERJEE, U. *Dependence Analysis for Supercomputing*. Kluwer, Boston, 1988.

[5] BIRCH, J., VAN ENGELEN, R., AND GALLIVAN, K. Value range analysis of conditionally updated variables and pointers. In *proceedings of Compilers for Parallel Computing (CPC)* (2004), pp. 265–276.

[6] BLUME, W., AND EIGENMANN, R. Demand-driven, symbolic range propagation. In *proceedings of the* 8^{th} *International workshop on Languages and Compilers for Parallel Computing* (Columbus, Ohio, USA, Aug. 1995), pp. 141–160.

[7] BURKE, M., AND CYTRON, R. Interprocedural dependence analysis and parallelization. In *proceedings of the Symposium on Compiler Construction* (1986), pp. 162–175.

[8] COLLARD, J.-F., BARTHOU, D., AND FEAUTRIER, P. Fuzzy array dataflow analysis. In *proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1995), pp. 92–101.

[9] FAHRINGER, T. Efficient symbolic analysis for parallelizing compilers and performance estimators. *Supercomputing 12*, 3 (May 1998), 227–252.

[10] FRANKE, B., AND O'BOYLE, M. Compiler transformation of pointers to explicit array accesses in DSP applications. In *proceedings of the ETAPS Conference on Compiler Construction 2001, LNCS 2027* (2001), pp. 69–85.

[11] GERLEK, M., STOLZ, E., AND WOLFE, M. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems (TOPLAS) 17*, 1 (Jan. 1995), 85–122.

[12] GOFF, G., KENNEDY, K., AND TSENG, C.-W. Practical dependence testing. In *proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)* (Toronto, Ontario, Canada, June 1991), vol. 26, pp. 15–29.

[13] HAGHIGHAT, M. R. *Symbolic Analysis for Parallelizing Compilers*. Kluwer Academic Publishers, 1995.

[14] HAGHIGHAT, M. R., AND POLYCHRONOPOULOS, C. D. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems 18*, 4 (July 1996), 477–518.

[15] HAVLAK, P. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, 1994.

[16] HAVLAK, P., AND KENNEDY, K. Experience with interprocedural analysis of array side effects. pp. 952–961.

[17] KUCK, D. *The Structure of Computers and Computations*, vol. 1. John Wiley and Sons, New York, 1987.

[18] MAYDAN, D. E., HENNESSY, J. L., AND LAM, M. S. Efficient and exact data dependence analysis. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (1991), ACM Press, pp. 1–14.

[19] MUCHNICK, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Fransisco, CA, 1997.

[20] POLYCHRONOPOULOS, C. *Parallel Programming and Compilers*. Kluwer, Boston, 1988.

[21] PSARRIS, K. Program analysis techniques for transforming programs for parallel systems. *Parallel Computing 28*, 3 (2003), 455–469.

[22] PSARRIS, K., AND KYRIAKOPOULOS, K. Measuring the accuracy and efficiency of the data dependence tests. In *proceedings of the International Conference on Parallel and Distributed Computing Systems* (2001).

[23] PSARRIS, K., AND KYRIAKOPOULOS, K. The impact of data dependence analysis on compilation and program parallelization. In *proceedings of the ACM International Conference on Supercomputing (ICS)* (2003).

[24] PSARRIS, K., AND KYRIAKOPOULOS, K. An experimental evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems 15*, 3 (March 2004), 196–213.

[25] PUGH, W. Counting solutions to Presburger formulas: How and why. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Orlando, FL, June 1994), pp. 121–134.

[26] RUGINA, R., AND RINARD, M. Symbolic bounds analysis of array indices, and accessed memory regions. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Vancouver, British Columbia, Canada, June 2000), pp. 182–195.

[27] SHEN, Z., LI, Z., AND YEW, P.-C. An empirical study on array subscripts and data dependencies. In *proceedings of the International Conference on Parallel Processing* (1989), vol. 2, pp. 145–152.

[28] SU, E., LAIN, A., RAMASWAMY, S., PALERMO, D., HODGES, E., AND BANERJEE, P. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *proceedings of the* 9^{th} *ACM International Conference on Supercomputing (ICS)* (Barcelona, Spain, July 1995), ACM Press, pp. 424–433.

[29] TU, P., AND PADUA, D. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *proceedings of the* 9^{th} *ACM International Conference on Supercomputing (ICS)* (New York, July 1995), ACM Press, pp. 414–423.

[30] VAN ENGELEN, R. Symbolic evaluation of chains of recurrences for loop optimization. Tech. rep., TR-000102, Computer Science Dept., Florida State University, 2000.

[31] VAN ENGELEN, R., GALLIVAN, K., AND WALSH, B. Tight timing estimation with the Newton-Gregory formulae. In *proceedings of CPC 2003* (Amsterdam, Netherlands, January 2003), pp. 321–330.

[32] VAN ENGELEN, R. A. Efficient symbolic analysis for optimizing compilers. In *proceedings of the ETAPS Conference on Compiler Construction 2001, LNCS 2027* (2001), pp. 118–132.

[33] VAN ENGELEN, R. A., BIRCH, J., SHOU, Y., WALSH, B., AND GALLIVAN, K. A. A unified framework for nonlinear dependence testing and symbolic analysis. In *proceedings of the ACM International Conference on Supercomputing (ICS)* (2004), pp. 106–115.

[34] VAN ENGELEN, R. A., AND GALLIVAN, K. An efficient algorithm for pointer-to-array access conversion for compiling and optimizing DSP applications. In *proceedings of the International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems (IWIA) 2001* (Maui, Hawaii, 2001), pp. 80–89.

[35] WOLFE, M. *High Performance Compilers for Parallel Computers*. Addison-Wesley, Redwood City, CA, 1996.

[36] WU, P., COHEN, A., HOEFLINGER, J., AND PADUA, D. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *proceedings of the ACM International Conference on Supercomputing (ICS)* (2001), pp. 78–91.

[37] ZIMA, E. Recurrent relations and speed-up of computations using computer algebra systems. In *proceedings of DISCO'92* (1992), LNCS 721, pp. 152–161.

[38] ZIMA, E. Simplification and optimization transformations of chains of recurrences. In *proceedings of the International Symposium on Symbolic and Algebraic Computing* (Montreal, Canada, 1995), ACM.

[39] ZIMA, E. V. Automatic construction of systems of recurrence relations. *USSR Computational Mathematics and Mathematical Physics 24*, 11-12 (1986), 193–197.

[40] ZIMA, H., AND CHAPMAN, B. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.