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

Array Data Dependence Testing with the Chains of Recurrences Algebra*

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

Abstract

This paper presents a new approach to array-based dependence testing in the presence of nonlinear and non-closed array index expressions and pointer references. Conventional data dependence testing requires induction variable substitution to replace recurrences with closed forms. We take a radically different approach to dependence testing by turning the analysis problem up-side-down. We convert closed forms to recurrences for dependence analysis. Because the set of functions defined by recurrences is a superset of functions with closed forms, we show that more dependence problems can be successfully analyzed by rephrasing the array-based dependence analysis as a search problem for a solution to a system of recurrences.

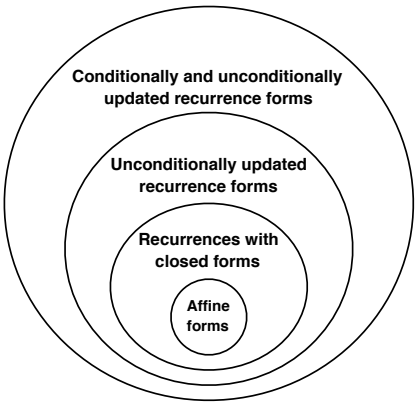

Figure 1. Recurrence Hierarchy

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 [5, 15, 17, 20, 23, 24, 27, 28, 29, 31, 32, 33, 35, 38, 41, 44, 45, 53, 58]. Current dependence analyzers are quite powerful and are able to solve complicated dependence problems, e.g. using the polyhedral model [6, 30]. However, recent work by Psarris et al. [34, 36], Franke and O'Boyle [22], Wu et al. [54], van Engelen et al. [48, 50] and earlier work by Shen, Li, and Yew [43], Haghighat [26], and Collard et al. [18] 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 array-based dependence testing on nonlinear array index expressions and pointer references in loops with conditionally updated induction variables and common forms of pointer arithmetic. Our approach is radically different compared to conventional induction variable substitution (IVS). Induction variable detection and substitution [2, 23, 26, 40, 52] are common methods to replace linear and nonlinear induction variables with closed form expressions to enable array-based dependence analysis. In our approach, we turn this problem up-side-down and convert closed forms to recurrences rather than deriving closed-form index expressions for the recurrences of induction variables. The recurrence forms are determined from a loop nest. But in contrast to strength reduction [1] the actual code is not changed. Recurrences provide greater coverage for analyzing dependence problems compared to conventional methods that require closed forms, because *the set of index functions defined by recurrences is a superset of functions with closed forms*, as depicted in the recurrence hierarchy shown in Figure 1. In this paper we show that more dependence problems can be successfully analyzed by rephrasing the array-based dependence test problem into a problem finding a solution to a system of recurrences [50]. In this paper we also show that the solution to a recurrence system can be determined using standard dependence test algorithms such as the Banerjee test [5] and range test [14].

* Supported in part by NSF grants CCR-0105422, CCR-0208892, EIA-0072043 and DOE grant DEFG02-02ER25543.

```
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 2. TRFD Benchmark: `olda` Routine

Note that in Figure 1 the set of *affine forms* contains all integer-valued multivariate linear polynomial index functions, which includes for example all linear combinations of the basic induction variables of a loop nest. Most compilers implement dependence tests on affine forms of index expressions [5, 15, 17, 20, 23, 24, 27, 28, 29, 31, 32, 33, 35, 38, 44, 45, 53, 58] possibly in combination with value range analysis techniques [11, 14, 21] to enable symbolic and nonlinear dependence testing (e.g. in the Polaris compiler [12]).

The set of *recurrences with closed forms* contains affine and nonlinear functions that can be converted to a multivariate recurrence form, for example using the chains of recurrences (CR) algebra [3, 47, 48]. This set includes the characteristic functions of generalized induction variables (GIVs) [26, 47] that describe polynomial and geometric progressions. To determine the closed-form characteristic function of a GIV, the variable updates in a loop nest must be unconditional[1]. An example code with unconditionally updated nonlinear induction variables is shown in Figure 2. The TRFD code is part of he Perfect Benchmark suite of programs, which has been extensively studied for performance optimizations and parallelization, see e.g. [13, 26]. Restructuring and parallelizing compilers traditionally rely on the determination of a closed-form characteristic function for dependence testing and symbolic value range analysis. For example, Polaris [12] is able to determine the absence of an output dependence on the array xijkl by aggressively applying induction variable substitution to determine the closed-form characteristic function of the ijkl induction variable, which is a multivariate polynomial over the $\langle i, j, k, \ell \rangle$ index space.

The set of functions defined by *unconditionally updated recurrence forms* includes recurrences that have no closed forms. This set includes recurrences that cannot be converted to closed form because a system of (coupled) recurrences may not have a closed-form solution in general. For example, variable j in the recurrences $j = j + k; k = k * i$, for $i = 1, \ldots, n$ with initial values $j = 0$ and $k = 1$, has no closed-form function over index i. Its evaluation requires a sequential tabulation of the values of the recurrence system. This class of recurrences is part of the recurrence hierarchy for completeness, but this class is only of academic interest, because these recurrence problems are unlikely to occur in real-world programs. Despite of this, our approach handles these cases by applying data dependence testing on the recurrence system. Thus, j is an admissible variable in an array index expression in our dependence framework.

1 In [26] it is shown how semantically equivalent conditional updates in multiple paths can be traced to form a single characteristic function.

```
DQ.ntrans=1,2
  DO i=0,r3–1
    str(ctr)=temp
    ctr=ctr+1
  ENDDO
  ...
  ctr=ctr+1
  ...
  IF (r3.NE.0) THEN
    str(ctr)=36
    ctr=ctr+1
  ELSE
    DO i=0,t–1
      str(ctr)=4
      ctr=ctr+1
    ENDDO
    ...
  ENDIF
  ...
ENDDO
```

Figure 3. QCD Benchmark: `qqqlps` Routine

The set of index functions with *conditionally and unconditionally updated recurrence forms* is the target of our dependence testing approach. Consider for example the QCD program from the Perfect Benchmark suite shown in Figure 3. Variable ctr has no closed form, due to a conditional update. Current restructuring compilers cannot apply dependence testing to this code using existing techniques. Our approach can handle this code by applying a data dependence test based on the recurrence system defined by variable ctr.

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 [22, 48]. An important class of programs are digital signal processor (DSP) codes for filtering operations. The implementations of these algorithms exhibit conditionally or unconditionally updated nonlinear induction variables and pointer updates. Our pointer reference dependence analysis can handle these specialized algorithms, including radix-2 FFTs [48].

```
int *f = ..., *lsp = ...;
...
f += 2; lsp += 2;
for (i = 2; i <= 5; i++) {
   *f = f[-2];
   for (j = 1; j < i; j++, f– –)
      *f += f[-2]–2*(*lsp)*f[-1];
   *f –= 2*(*lsp);
   f += i; lsp += 2;
}
```

Figure 4. ETSI Codec: `Get_lsp_pol` Routine

Consider for example the code segment of the `Get_lsp_pol` routine of the `LSP_AZ` module of the GSM Enhanced Full Rate speech codec [19] shown in Figure 4. The loop nest is triangular and involves a nonlinear data-dependent pointer update. Our data dependence test is applicable to the original pointer-based code by treating the f and lsp pointers as induction variables to establish a recurrence system to determine if the loop nest has forward, anti, or output dependences on the arrays accessed by the f and lsp pointers.

Related to our pointer-based dependence analysis is the array recovery method by Franke and O'Boyle [22]. Their method converts pointer references to array accesses to enable conventional array-based compiler analysis on the closed-form affine index expressions. However, 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 or code transformations.

Most closely related to our work is the work by Wu et al. [54]. They propose an approach for dependence testing without closed form computations. Similar to our method, the application of induction variable substitution 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 programs of the Perfect Benchmark suite. In contrast, our method uses the inherent monotonicity information of the recurrence forms to determine that the loops in these benchmarks are dependence free. Their method also does not apply dependence testing to pointer arithmetic. In addition, our recurrence forms are easily converted to closed forms for IVS using the inverse CR algebra [47].

In our earlier work on GIV recognition [46, 47] it was observed that symbolic differencing [26] is unsafe and that the method by Gerlek et al. [23] requires the application of several different recurrence solvers. In contrast, the complexity of our GIV recognition is safe and the complexity is comparable to constant folding [1], which is a relatively inexpensive method. In [46] we also proved that the CR algebra is complete and closed under the formation of characteristic functions of GIVs, which is an important property for the applicability of our recurrence framework to induction variable recognition and dependence testing.

This paper is an extended version of [49]. In this paper (and [49]) we use recurrence forms to determine if array and pointer references are free of forward, anti, or output dependences in a loop nest. Our recurrence formulation increases the accuracy of standard dependence algorithms such as the extreme value test and value range test by including the analysis of nonlinear induction variables, conditionally updated variables, and pointer arithmetic. We will show how these tests can be directly applied to our recurrence system. Because trusted dependence algorithms can be used in our enhanced analysis environment, a high level of flexibility of the implementation and greater assurance on the soundness of the approach are achieved compared to ad-hoc approaches.

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 formalism. For more details, we refer to [3, 47, 48]. The formalism was originally developed by Zima [55, 56, 57] and later improved by Bachmann, Zima, and Wang [3, 4] to expedite the evaluation of multivariate functions on regular grids. Our work includes the addition of new CR algebra rules [46] and applications of the CR formalism for the detection and substitution of GIVs [47], for array recovery through pointer-to-array conversion [48], and for value range analysis [10]. 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 [3]):

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

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 $i = 0, \ldots, n-1$ defined by the loop template

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

The loop produces the sequence $\mathsf{val}[i]$ 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 Construction

The CR algebra [4, 47, 57] defines a set of term rewriting rules $\mathcal{CR}$ shown in Figure 5 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].

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.

We proved that the CR algebra is closed under the formation of the (multivariate) characteristic function of a GIV. The set of rewrite rules of the algebra is also complete [46], which means that CR forms for multivariate GIVs are normal forms. Another advantage is that the manipulation of CR forms is type safe, which ensures that the coefficients of CR forms of integer-valued polynomial functions and GIVs are also integer valued.

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 recurrence $k = k + i$ with initial value $k = 0$. The recurrence of k in CR form is $\Phi(k) = \{0, +, 0, +, 1\}_i$. The CR construction of the example expression yields:

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

The multivariate CR form is a normal form [46] for the multivariate polynomial expression. The example CR form can be converted to closed form polynomial using the inverse CR algebra [47] described in the next section.

2.4. Closed Forms

The inverse mapping $\mathcal{CR}^{-1}$ shown in Figure 5 converts CR forms to closed-form functions. Consider for example the CR form $\{\{1, +, n\}_j, +, 1, +, 2\}_i$ from the example given in the previous section. The closed form multivariate polynomial characteristic function is $1 + i^2 + n * j$. To compute the closed form, we use our extension of the CR algebra [47, 48] by applying the inverse rules to convert a CR to a closed-form symbolic expression. In general, multivariate GIVs, i.e. sums of multivariate polynomials and geometric functions, can always be converted to closed form formulae using efficient matrix-vector products [51] discussed in Section 2.5.

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, see also Figure 1.

2.5. Newton Matrices

Because linear and polynomial induction variables are more common compared to geometric sequences, it is important to consider the efficiency of the symbolic manipulation of recurrences for polynomial forms. Addition and subtraction of the CR forms of polynomials require just $\mathcal{O}(k)$ operations using the $\mathcal{CR}$ rules shown in Figure 5, where k is the order of the polynomials. Bachmann describes an algorithm [3] for polynomial multiplication in $\mathcal{O}(k^2)$ operations, while the application of $\mathcal{CR}$ rule 14 shown in Figure 5

$\mathcal{CR}$				
#	LHS		RHS	Condition
1	$\{\phi_0, +, 0\}_i$	$\Rightarrow$	ϕ_0	
2	$\{\phi_0, *, 1\}_i$	$\Rightarrow$	ϕ_0	
3	$\{0, *, f_1\}_i$	$\Rightarrow$	0	
4	$-\{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\{-\phi_0, +, -f_1\}_i$	
5	$-\{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\{-\phi_0, *, f_1\}_i$	
6	$\{\phi_0, +, f_1\}_i \pm E$	$\Rightarrow$	$\{\phi_0 \pm E, +, f_1\}_i$	when E is i-loop invariant
7	$\{\phi_0, *, f_1\}_i \pm E$	$\Rightarrow$	$\{\phi_0 \pm E, +, \phi_0 * (f_1 - 1), *, f_1\}_i$	when E and f_1 are i-loop invariant
8	$E * \{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\{E * \phi_0, +, E * f_1\}_i$	when E is i-loop invariant
9	$E * \{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\{E * \phi_0, *, f_1\}_i$	when E is i-loop invariant
10	$E/\{\phi_0, +, f_1\}_1$	$\Rightarrow$	$1/\{\phi_0/E, +, f_1/E\}_i$	when $E \neq 1$ is i-loop invariant
11	$E/\{\phi_0, *, f_1\}_1$	$\Rightarrow$	$\{E/\phi_0, *, 1/f_1\}_i$	when E is i-loop invariant
12	$\{\phi_0, +, f_1\}_i \pm \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 \pm \psi_0, +, f_1 \pm g_1\}_i$	
13	$\{\phi_0, *, f_1\}_i \pm \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 \pm \psi_0, +, \{\phi_0 * (f_1 - 1), *, f_1\}_i \pm g_1\}_i$	when f_1 is i-loop invariant
14	$\{\phi_0, +, f_1\}_i * \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 * \psi_0, +, \{\phi_0, +, f_1\}_i * g_1 + \{\psi_0, +, g_1\}_i * f_1 + f_1 * g_1\}_i$	
15	$\{\phi_0, *, f_1\}_i * \{\psi_0, *, g_1\}_i$	$\Rightarrow$	$\{\phi_0 * \psi_0, *, f_1 * g_1\}_i$	
16	$\{\phi_0, *, f_1\}_i^E$	$\Rightarrow$	$\{\phi_0^E, *, f_1^E\}_i$	when E is i-loop invariant
17	$\{\phi_0, *, f_1\}_i^{\{\psi_0, +, g_1\}_i}$	$\Rightarrow$	$\{{\phi_0}^{\psi_0}, *, \{\phi_0, *, f_1\}_i^{g_1} * {f_1}^{\{\psi_0, +, g_1\}_i} * f_1^{g_1}\}_i$	
18	$E^{\{\phi_0, +, f_1\}_i}$	$\Rightarrow$	$\{E^{\phi_0}, *, E^{f_1}\}_i$	when E is i-loop invariant
19	$\{\phi_0, +, f_1\}_i^n$	$\Rightarrow$	$\begin{cases} \{\phi_0, +, f_1\}_i * \{\phi_0, +, f_1\}_i^{n-1} & \textbf{if } n \in \mathbb{Z}, n > 1 \\ 1/\{\phi_0, +, f_1\}_i^{-n} & \textbf{if } n \in \mathbb{Z}, n < 0 \end{cases}$	
20	$\{\phi_0, +, f_1\}_i!$	$\Rightarrow$	$\begin{cases} \{\phi_0!, *, \left(\prod_{j=1}^{f_1} \{\phi_0 + j, +, f_1\}_i\right)\}_i & \textbf{if } f_1 \geq 0 \\ \{\phi_0!, *, \left(\prod_{j=1}^{\lvert f_1 \rvert} \{\phi_0 + j, +, f_1\}_i\right)^{-1}\}_i & \textbf{if } f_1 < 0 \end{cases}$	
21	$\{\phi_0, +, \phi_1, *, f_2\}$	$\Rightarrow$	$\{\phi_0, *, f_2\}_i$	when $\frac{\phi_1}{\phi_0} = f_2 - 1$

$\mathcal{CR}^{-1}$				
#	*LHS*		*RHS*	*Condition*
1	$\{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\phi_0 + \{0, +, f_1\}_i$	when $\phi_0 \neq 0$
2	$\{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\phi_0 * \{1, *, f_1\}_i$	when $\phi_0 \neq 1$
3	$\{0, +, -f_1\}_i$	$\Rightarrow$	$-\{0, +, f_1\}_i$	
4	$\{0, +, f_1 + g_1\}_i$	$\Rightarrow$	$\{0, +, f_1\}_i + \{0, +, g_1\}_i$	
5	$\{0, +, f_1 * g_1\}_i$	$\Rightarrow$	$f_1 * \{0, +, g_1\}_i$	when i does not occur in f_1
6	$\{0, +, f_1^i\}_i$	$\Rightarrow$	$\frac{f_1^i - 1}{f_1 - 1}$	when i does not occur in f_1 and $f_1 \neq 1$
7	$\{0, +, f_1^{g_1 + h_1}\}_i$	$\Rightarrow$	$\{0, +, f_1^{g_1} * f_1^{h_1}\}_i$	
8	$\{0, +, f_1^{g_1 * h_1}\}_i$	$\Rightarrow$	$\{0, +, (f_1^{g_1})^{h_i}\}_i$	when i does not occur in f_1 and g_1
9	$\{0, +, f_1\}_i$	$\Rightarrow$	$i * f_1$	when i does not occur in f_1
10	$\{0, +, i\}_i$	$\Rightarrow$	$\frac{i^2 - i}{2}$	
11	$\{0, +, i^n\}_i$	$\Rightarrow$	$\sum_{k=0}^{n} \frac{\binom{n+1}{k}}{n+1} B_k \, i^{n-k+1}$	for $n \in \mathbb{N}$, B_k is k^{th} Bernoulli number
12	$\{1, *, -f_1\}_i$	$\Rightarrow$	$(-1)^i \{1, *, f_1\}_i$	
13	$\{1, *, \frac{1}{f_1}\}_i$	$\Rightarrow$	$\{1, *, f_1\}_i^{-1}$	
14	$\{1, *, f_1 * g_1\}_i$	$\Rightarrow$	$\{1, *, f_1\}_i * \{1, *, g_1\}_i$	
15	$\{1, *, f_1^{g_1}\}_i$	$\Rightarrow$	$f_1^{\{1, *, g_1\}_i}$	when i does not occur in f_1
16	$\{1, *, g_1^{f_1}\}_i$	$\Rightarrow$	$\{1, *, g_1\}_i{}^{f_1}$	when i does not occur in f_1
17	$\{1, *, f_1\}_i$	$\Rightarrow$	f_1^i	when i does not occur in f_1
18	$\{1, *, i\}_i$	$\Rightarrow$	0^i	
19	$\{1, *, i + f_1\}_i$	$\Rightarrow$	$\frac{(i + f_1 - 1)!}{(f_1 - 1)!}$	when i does not occur in f_1 and $f_1 \geq 1$
20	$\{1, *, f_1 - i\}_i$	$\Rightarrow$	$(-1)^i * \frac{(i - f_1 - 1)!}{(-f_1 - 1)!}$	when i does not occur in f_1 and $f_1 \leq -1$

Figure 5. The $\mathcal{CR}$ and $\mathcal{CR}^{-1}$ Term Rewriting System Representations of the CR Algebra

- **input:** $p[0:k]$
- **output:** $\phi[0:k]$
Local integer array $m[0:k]$
for j = 0 to k
  ϕ_j := 0
  m_j := 0
ϕ_0 := p_0
if $k \leq 0$ then
  stop
ϕ_1 := p_1
m_1 := 1
for i = 2 to k
  for j = i to 1 step -1
  m_j := j*(m_{j-1}+m_j)
  ϕ_j += m_j*p_i
 	
(a) *Compute* $\mathbf{\Phi} = \mathbf{N}_k\mathbf{p}$
	- **input:** $\phi[0:k]$
	- **output:** $p[0:k]$
	Local rational array $m[0:k]$
	for j = 0 to k
	  p_j := 0
	  m_j := 0
	p_0 := ϕ_0
	if $k \leq 0$ then
	  stop
	p_1 := ϕ_1
	m_1 := 1
	for i = 2 to k
	  for j = i to 1 step -1
	  m_j := (m_{j-1}−(i−1)*m_j)/i
	  p_j += m_j*ϕ_i
	
	(b) *Compute* $\mathbf{p} = \mathbf{N}_k^{-1}\mathbf{\Phi}$

Figure 6. Conversion Algorithms

requires $\mathcal{O}(k^3)$ operations. Zima describes an efficient algorithm for CR division [56].

We use the Newton matrix [3] to compute the CR form of a polynomial in $\mathcal{O}(k^2)$ steps. For example, the Newton matrix for $k = 3$ is

$$\mathbf{N}_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 6 \\ 0 & 0 & 0 & 6 \end{bmatrix} \quad .$$

The coefficients of the CR form $\Phi(i)$ of a polynomial $p(i) = p_0 + p_1 i + \cdots + p_k i^k$ is obtained by the matrix-vector product $\mathbf{N}_k\vec{p}$ with Newton matrix $\mathbf{N}_k$ and the vector of polynomial coefficients $\vec{p} = [p_0, \ldots, p_k]$.

The algorithm shown in Figure 6(a) symbolically computes the coefficients ϕ_j for the CR form $\Phi(i) = \{\phi_0, +, \ldots, +, \phi_k\}_i$ of a polynomial $p(i) = p_0 + p_1 i + \cdots + p_k i^k$ in $\mathcal{O}(k^2)$ operations with $\mathcal{O}(k)$ temporary storage space. The algorithm uses a two-term recurrence [3].

The algorithm shown in Figure 6(b) computes the closed-form polynomial of a CR form using the inverse Newton matrix $\mathbf{N}_k^{-1}$. For example, the inverse Newton triangle matrix for $k = 3$ is

$$\mathbf{N}_3^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -\frac{1}{2} & \frac{1}{3} \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{6} \end{bmatrix} \quad .$$

The vector of polynomial coefficients is computed by the product $\vec{p} = \mathbf{N}_k^{-1}\vec{\phi}$ of a CR form $\Phi(i)$ with coefficients $\vec{\phi} = [\phi_0, \ldots, \phi_k]$.

Bachmann [3] proved the correctness of the algorithm shown in Figure 6(a) using well-known recurrences of the Sterling numbers. Here we prove the fundamental relationship between the Newton matrix and the CR algebra by deriving the Newton matrix using properties of the CR algebra directly.

Lemma 1 *Let $\mathbf{N}_k$ denote the Newton matrix and let $\mathbf{p} = [p_0, \ldots, p_k]$ be the coefficients of a polynomial $p(i) = p_0 + p_1 i + \cdots + p_k i^k$. Then,*

$$\mathbf{\Phi} = \mathbf{N}_k\mathbf{p}$$

are the coefficients of the CR $\{\phi_0, +, \cdots, +, \phi_k\}_i$ for $p(i)$.

Proof. First, we rewrite polynomial p in Horner form

$$p(i) = p_0 + i(p_1 + i(p_2 + \ldots + i\,p_k)) \quad .$$

Since polynomials are evaluated on a domain $i = 0, \ldots, n-1$ (normalized loop bounds), we can replace i with the CR form $i = \{0, +, 1\}_i$

$$p_0 + \{0, +, 1\}_i(p_1 + \{0, +, 1\}_i(p_2 + \ldots + \{0, +, 1\}_i\, p_k)) \quad .$$

To obtain this form, we define the symbolic translation of $p(i)$ to a CR by

$$\begin{array}{rcl} \mathcal{CR}(p_0) & = & p_0 \\ \mathcal{CR}(\mathbf{p}) & = & p_0 + \mathcal{I}(\mathcal{CR}([p_1, \ldots, p_k]^{\mathrm{T}})) \end{array}$$

where

$$\begin{array}{rcl} \mathcal{I}(\phi_0) & = & \{0, +, \phi_0\}_i \\ \mathcal{I}(\{\phi_0, +, f_1\}_i) & = & \{0, +, \mathcal{I}(f_1) + \{\phi_0, +, f_1\}_i + f_1\}_i \end{array} \quad .$$

The recursion in the definition of $\mathcal{I}$ is based on

$$\{0, +, 1\}_i * \{\phi_0, +, f_1\}_i = \{0, +, \{0, +, 1\}_i * f_1 + \{\phi_0, +, f_1\}_i + f_1\}_i$$

using $\mathcal{CR}$ rule 14 for multiplication with $\{0, +, 1\}_i$. The $\{\}_i$ CR notation is eliminated by representing CRs as vectors. To operate on vectors we define new functions $\mathcal{N}$ and $\mathcal{A}$ for $\mathcal{CR}$ and $\mathcal{I}$, respectively, by

$$\mathcal{N}(\mathbf{p}) = [p_0, p_1', \ldots, p_k']^{\mathrm{T}}$$

where

$$\mathbf{p}' = \mathcal{A}(\mathcal{N}([p_1, \ldots, p_k]^{\mathrm{T}}))$$

with $\mathcal{N}(p_0) = p_0$ and

$$\mathcal{A}(\mathbf{p}) = [0, p_1'', \ldots, p_k'']^{\mathrm{T}} + \mathbf{p} + [p_1, \ldots, p_k, 0]^{\mathrm{T}}$$

where

$$\mathbf{p}'' = \mathcal{A}([p_1, \ldots, p_k]^{\mathrm{T}})$$

with $\mathcal{A}(p_0) = p_0$.

The operations $\mathcal{N}$ and $\mathcal{A}$ can be implemented by matrices $\mathbf{N}_k$ and $\mathbf{A}_k$, such that $\mathcal{N}(\mathbf{p}) = \mathbf{N}_k\mathbf{p}$ and $\mathcal{A}(\mathbf{p}) = \mathbf{A}_k\mathbf{p}$, as follows

$$\begin{array}{rcl} \mathbf{N}_0 & = & 1 \\ \mathbf{N}_k & = & \left[\begin{array}{c|c} 1 & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{A}_{k-1}\mathbf{N}_{k-1} \end{array}\right] \end{array} \qquad (1)$$

and

$$\begin{array}{rcl} \mathbf{A}_0 & = & 1 \\ \mathbf{A}_k & = & \left[\begin{array}{c|c} 0 & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{A}_{k-1} \end{array}\right] + \mathbf{I}_k + \mathbf{Z}_k \end{array} \qquad (2)$$

where $\mathbf{I}_k$ is the identity matrix of order $k+1$ and $\mathbf{Z}_k$ is the right-shifted identity matrix. Solving the recursion in (2) gives the coefficients of $\mathbf{A}_k$ by

$$a_{i,j} = \begin{cases} i & \text{if } i = j \text{ or } i = j-1 \\ 0 & \text{otherwise} \end{cases} .$$

By the recursive formulation (1) the coefficients of $\mathbf{N}_k$ exhibit the two-term recurrence

$$m_{i+1,j+1} = \sum_{\ell=1}^{k} a_{i,\ell} m_{\ell,j} = i(m_{i,j} + m_{i+1,j})$$

which gives the coefficients of $\mathbf{N}_k$ by

$$m_{i,j} = \begin{cases} 1 & \text{if } i = 1 \text{ and } j = 1 \\ (i-1)(m_{i-1,j-1} + m_{i,j-1}) & \text{if } i \geq 2 \text{ and } j \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

for all $i = 1, \ldots, k+1$ and $j = 1, \ldots, k+1$. □

2.6. Relation to Compiler Analysis

CR forms are more amenable to symbolic analysis compared to closed forms, because the monotonic properties of the function and its extreme values can be more accurately determined using CR forms [46], see also Section 3.4. Determining the monotonic properties of (compositions) of array index expressions is important in dependence testing for loop restructuring and parallelization, which will be further discussed in Section 4.

The application of CR construction for symbolic manipulation in compiler analysis is clear when we first consider the types of linear and nonlinear index functions and expressions commonly encountered in practice in compiler analysis dealing with array index expressions and generalized induction variables. The next section presents our framework for the detection of induction variables to compute a recurrence system for array-based dependence testing.

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 to a multivariate CR by a conversion algorithm based on matrix-vector multiplication with Newton matrices [3, 51].

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 (see our proof in [46]). By definition [25], 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 $a\,r^i$.

For loop parallelization it is desirable to eliminate the cross-iteration dependences induced by the recurrences defined by 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 $\mathcal{CR}^{-1}$ described in the previous section.

The CR algebra rules for CR construction and conversion to closed forms are implemented in our CR library for SUIF using a representation of CRs based on arrays of symbolic coefficients for efficient manipulation.

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 to compute CR forms for conditionally updated induction variables and pointers.

3.1. General Recurrence Form of a GIV

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

```
V = V0
for i = 0 to n–1
   ...
   V = α * V + p(i)
   ...
endfor
```

where α 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 $\alpha = 0$ (V is equal to polynomial p), $\alpha = 1$ (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 2 *Let* $\Psi_i = \{\psi_0, +, \psi_1, +, \cdots, +, \psi_k\}_i$ *be the CR form of polynomial* $p(i)$. *Then, the CR form of the recurrence* $V = \alpha * V + p(i)$ *is* $\Phi(V) = \{\phi_0, +, \phi_1, +, \cdots, +, \phi_{k+1}, *, \phi_{k+2}\}_i$ *where*

$$\phi_0 = V_0; \qquad \phi_j = (\alpha - 1)\phi_{j-1} + \psi_{j-1}; \qquad \phi_{k+2} = \alpha$$

Proof. The sequence of the recurrence $V = \alpha * V + p(i)$, with initial value $V = V_0$, for iterations $i = 0, \ldots, n-1$ is

$$\begin{array}{rcl} i = 0 & \Rightarrow & V_0 \\ i = 1 & \Rightarrow & \alpha V_0 + p(0) \\ i = 2 & \Rightarrow & \alpha(\alpha V_0 + p(0)) + p(1) \\ i = 3 & \Rightarrow & \alpha(\alpha(\alpha V_0 + p(0)) + p(1)) + p(2) \\ \vdots & \Rightarrow & \vdots \end{array}$$

Polynomial $p(i)$ has CR $\Phi_i = \{\psi_0, +, \psi_1, +, \cdots, +, \psi_k\}_i$. According to the CR semantics, Section 2.2, the sequence of $p(i)$ calculated by the loop template $p(i) = \mathsf{val}[i]$ is

$$\begin{array}{rcl} p(0) & = & \psi_0 \\ p(1) & = & \psi_0 + \psi_1 \\ p(2) & = & \psi_0 + 2\psi_1 + \psi_2 \\ \vdots & = & \vdots \end{array}$$

Replacing the left-hand sides with the right-hand sides in the recurrence above yields

$$\begin{array}{rcl} i = 0 & \Rightarrow & V_0 \\ i = 1 & \Rightarrow & \alpha V_0 + \psi_0 \\ i = 2 & \Rightarrow & \alpha(\alpha V_0 + \psi_0) + \psi_0 + \psi_1 \\ i = 3 & \Rightarrow & \alpha(\alpha(\alpha V_0 + \psi_0) + \psi_0 + \psi_1) + \psi_0 + 2\psi_1 + \psi_2 \\ \vdots & \Rightarrow & \vdots \end{array}$$

The CR form $\Phi(V)$ of V can be determined using the Newton series of this progression. The Newton series (the lower left diagonal of the difference table) of the sequence of the recurrence is

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

The terms continue to expand up to nonzero coefficient ψ_k. After that, the sequence continues as multiples of $\alpha - 1$ times the previous row. Therefore, the remainder of the sequence is a geometric progression with ratio α. 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 a generalized induction variable.

- For $\alpha = 0$, 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 as we showed in [47], where the initial value of V may be unrelated to p.

- For $\alpha = 1$ we have a recurrence of the form

  $$V = V + p(i)$$

  Therefore, according to Lemma 2 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).

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

  $$V = \alpha * V$$

  Therefore, according to Lemma 2 we obtain

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

  This equation also holds for any symbolic expression α (not only constant). Hence, when α has a CR form we obtain

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

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

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 code of a loop body is often structured by a programmer in such a way that the recurrence of a generalized induction variable in the loop nest may not exactly match the recurrence pattern $V = \alpha * V + p(i)$. Multiple updates to a single induction variable may occur in the loop nest (e.g. variable ijkl in Figure 2), multiple induction variables may be coupled (e.g. variables ij and ijkl in Figure 2), and control flow may require intra-procedural analysis and control path analysis in a loop nest (see e.g. Figure 3), all of which obscures the recurrence pattern. To recognize recurrence patterns in the presence of coupled induction variables, we use a forward substitution approach introduced in our earlier work [47] and schematically illustrated in Figure 7 (a) and (b). Repeated forward substitution yields a set of normalized assignments in which each variable is assigned at most once, i.e. similar to single static assignment (SSA) forms, which facilitates recurrence pattern recognition. Nested loops are analyzed from the innermost to the outermost loop level to compute multivariate recurrences [47]. In this paper we apply this technique to conditionally updated induction variables by selectively traversing paths through the loop body to determine sets of recurrence patterns for induction variables as illustrated in the example shown in Figure 7 (c) and (d) where V and U both may have two different recurrence patterns. Because the switching behavior of the flow in the loop is unknown, the conditional recurrence of variables V and U have no closed-form equivalents. Further details on the path-based substitution algorithm are presented in Section 3.5.

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

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

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

```
for i = a to b
  ...
  if ... then
    V = expr1
  else
    V = expr2
  endif
  ...
  { U = ... expr1 ..., U = ... expr2 ... }
  ...
endfor
```
(d) *After Forward Substitution*

Figure 7. Forward Substitution

```
...
for i = 0 to n–1
  /* L(i) ≤ V ≤ U(i) */
  ...
  if ... then
    V = α * V + p(i)
  else
    V = β * V + q(i)
  endif
  /* L(i+1) ≤ V ≤ U(i+1) */
  ...
endfor
```
(a) *Bounding Functions*

```
k = 1
for i = 0 to n–1
  /* i + 1 ≤ k ≤ 2^i */
  a[k] = b[i]
  if ... then
    k = k + 1
  else
    k = 2 * k
  endif
endfor
```
(b) *Example*

Figure 8. Dynamic L and U Bounds on V

3.4. Bounding Functions

To analyze the range of values of conditionally updated recurrences we developed an algorithm to compute *dynamic value range bounds*, consisting of indexed lower and upper bound functions on the values of a set of conditional recurrences. Because the bounding functions are indexed by points in the iteration space, the dynamic bounds are more accurate compared to static bounds that are independent of the iteration space. Static bounds are commonly used in value range analysis [15, 16] for nonlinear dependence testing [11, 14]. Because our value range information is dynamic, our nonlinear data dependence testing can be more accurate [10]. An example application will be discussed in Section 4.

3.4.1. Dynamic Value Range Bounds. Dynamic value range bounds are functions over the iteration space that bound the possible sequences of a set of (conditional) recurrences. Figure 8(a) shows the $L(i)$ and $U(i)$ bounds on the sequence of induction variable V, where V is conditionally updated using two different recurrence forms. The $L(i)$ and $U(i)$ bounding functions are indexed by the index space of the (multidimensional) loop nest (only a one dimensional loop is shown in the figure). Figure 8(b) shows an actual example. Note that the array access is dependence free because k is strictly monotonically increasing. The determination of bounding information and monotonicity is crucial for accurate dependence testing.

To determine dynamic value range bounds, we developed a new method to compute *min* and *max* bounding functions of a set of (multivariate) CR forms. The bounding functions are computed in CR form.

Definition 1 *Let $\Phi_i = \{\phi_0, \odot_1, f_1\}_i$ and let $\Psi_i = \{\psi_0, \odot_1, g_1\}_i$ be (multivariate) CR forms over the same index variable i, where f_1 and g_1 are the nested CR "tails" of Φ_i and Ψ_i with the remainder of the coefficients (e.g. using the common nested representation of CR forms with basic recurrences (BRs) [3]).*

The minimum *CR form is inductively defined by*

$$
\begin{aligned}
&min(\{\phi_0, +, f_1\}_i, \{\psi_0, +, g_1\}_i) \\
&\quad = \{min(\phi_0, \psi_0), +, min(f_1, g_1)\}_i \\
&min(\{\phi_0, *, f_1\}_i, \{\psi_0, *, g_1\}_i) \\
&\quad = \begin{cases}
\{min(\phi_0, \psi_0), *, min(f_1, g_1)\}_i & \text{if } \phi_0{>}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\
\{min(\phi_0, \psi_0), *, max(f_1, g_1)\}_i & \text{if } \phi_0{<}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\
\{\phi_0, *, f_1\}_i & \text{if } \phi_0{<}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\
\{\psi_0, *, g_1\}_i & \text{if } \phi_0{>}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\
\{-max(|\phi_0|, |\psi_0|), *, max(|f_1|, |g_1|)\}_i & \text{if } f_1{<}0 \vee g_1{<}0
\end{cases}
\end{aligned}
$$

and the maximum *CR form is inductively defined by*

$$
\begin{aligned}
&max(\{\phi_0, +, f_1\}_i, \{\psi_0, +, g_1\}_i) \\
&\quad = \{max(\phi_0, \psi_0), +, max(f_1, g_1)\}_i \\
&max(\{\phi_0, *, f_1\}_i, \{\psi_0, *, g_1\}_i) \\
&\quad = \begin{cases}
\{max(\phi_0, \psi_0), *, max(f_1, g_1)\}_i & \text{if } \phi_0{>}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\
\{max(\phi_0, \psi_0), *, min(f_1, g_1)\}_i & \text{if } \phi_0{<}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\
\{\phi_0, *, f_1\}_i & \text{if } \phi_0{>}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\
\{\psi_0, *, g_1\}_i & \text{if } \phi_0{<}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\
\{max(|\phi_0|, |\psi_0|), *, max(|f_1|, |g_1|)\}_i & \text{if } f_1{<}0 \vee g_1{<}0
\end{cases}
\end{aligned}
$$

The *min* and *max* operators are associative for polynomial CR forms. Under certain conditions the operators are also associative for geometric forms, but not in general.

The *min* and *max* dynamic bounding functions applied to two CR forms require the CRs to be aligned where the operators between the CR forms match up.

Definition 2 *Two CR forms Φ_i and Ψ_i over the same index variable i are* aligned *if they have the same length k and the operators $\odot_j$, $j = 1, \ldots, k$, form a pairwise match.*

For example, $\{1,+,1,*,1\}$ is aligned with $\{0,+,2,*,2\}_i$, but $\{1,+,2\}_i$ is not aligned with $\{1,*,2\}_i$ and $\{1,+,2\}_i$ is not aligned with $\{1,+,2,+,1\}_i$.

To align two CR forms of unequal length, the shorter CR can be lengthened by adding dummy operations without changing the sequence it represents.

Lemma 3 *Let $\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \cdots, \odot_k, \phi_k\}_i$ be a CR form, where ϕ_k is invariant of i. Then, the following two identities hold*

$$\begin{aligned}\Phi_i &= \{\phi_0, \odot_1, \phi_1, \odot_2, \cdots, \odot_k, \phi_k, +, 0\}_i \\ \Phi_i &= \{\phi_0, \odot_1, \phi_1, \odot_2, \cdots, \odot_k, \phi_k, *, 1\}_i\end{aligned}$$

Proof. The proof immediately follows as a consequence of the CR semantics defined in Section 2.2, because the initial value of the induction variable cr_k for coefficient ϕ_k is set to ϕ_k and the value of cr_k is unchanged in the loop (either by adding zero or multiplying by one). □

When the operators of two CR forms do no match, the results of the following lemma and corollary are used.

Lemma 4 *Let $\Phi_i = \{a, *, r\}_i$ be a geometric CR form with initial value a and ratio r (r is invariant of i). Then,*

$$\Phi_i = \{a, +, a(r-1), +, a(r-1)^2, +, \cdots, +, a(r-1)^m, *, r\}_i$$

for any positive integer $m > 0$.

Proof. The proof is by induction on m.

- For the base case $m = 1$ we show that $\{a, *, r\}_i = \{a, +, a(r-1), *, r\}_i$ in two steps.
  1. Consider $a = 1$. By the definition of the CR semantics Section 2.2 the sequence $f[i]$ for $\{1, *, r\}_i$ and $g[i]$ for $\{1, +, r-1, *, r\}_i$ are computed by

cr_0 = 1	cr_0 = 1
for i = 0 to n–1	cr_1 = r–1
$f[i]$ = cr_0	for i = 0 to n–1
cr_0 = $\mathsf{cr}_0 * r$	$g[i]$ = cr_0
endfor	cr_0 = cr_0 + cr_1
	cr_1 = $\mathsf{cr}_1 * r$
	endfor

     (a) For iteration $i = 0$, we find that $f[0] = g[0]$
     (b) For iterations $i = 1, \ldots, n-1$, we find that

$$\begin{aligned}f[i] &= \prod_{j=0}^{i-1} r \\ &= r^i \\ g[i] &= 1 + \sum_{j=0}^{i-1}(r-1)r^j \\ &= 1 + \sum_{j=0}^{i-1} r\, r^j - \sum_{j=0}^{i-1} r^j \\ &= 1 + \sum_{j=1}^{i} r^j - \sum_{j=0}^{i-1} r^j \\ &= r^i\end{aligned}$$

  2. Consider $a \neq 1$. It follows from the CR algebra Figure 5 that $\{a, *, r\}_i = a\{1, *, r\}_i$ and $a\{1, +, r-1, *, r\}_i = \{a, +, a(r-1), *, r\}_i$, and therefore that

$$\begin{aligned}\{a, *, r\}_i &= a\{1, *, r\}_i \\ &= a\{1, +, r-1, *, r\}_i \\ &= \{a, +, a(r-1), *, r\}_i\end{aligned}$$

- Suppose the equation holds for $k = m - 1$. We have

  $$\Phi_i = \{a, +, a(r-1), +, a(r-1)^2, +, \cdots, +, a(r-1)^k, *, r\}_i$$

  Because the "flat" CR form Φ_i is identical to a nested CR form [4, 57], we use the base case to rewrite the tail part of the nested CR form as follows

$$\begin{aligned}&\{a, +, a(r-1), +, \cdots, +, a(r-1)^k, *, r\}_i \\ &= \{a, +, a(r-1), +, \cdots, +, \{a(r-1)^k, *, r\}_i\}_i \\ &= \{a, +, a(r-1), +, \cdots, +, \{a(r-1)^k, +, a(r-1)^k(r-1), *, r\}_i\}_i \\ &= \{a, +, a(r-1), +, \cdots, +, \{a(r-1)^k, +, a(r-1)^{k+1}, *, r\}_i\}_i \\ &= \{a, +, a(r-1), +, \cdots, +, a(r-1)^{m-1}, +, a(r-1)^m, *, r\}_i\end{aligned}$$

  Thus, it follows from the induction hypothesis that $\Phi_i = \{a, +, a(r-1), +, a(r-1)^2, +, \cdots, +, a(r-1)^m, *, r\}_i$. □

Corollary 1 *Let $\Phi_i = \{\phi_0, \odot_1, \cdots, \odot_{k-1}, \phi_{k-1}, *, \phi_k\}_i$ such that ϕ_k is invariant of i. Then, any number $m > 0$ of + operators can be inserted at the $(k-1)^{\text{th}}$ coefficient as follows*

$$\Phi_i = \{\phi_0, \odot_1, \cdots, \odot_{k-1}, \phi_{k-1}, \underbrace{+, \phi_{k-1}(\phi_k - 1), +, \phi_{k-1}(\phi_k-1)^2, +, \cdots, +, \phi_{k-1}(\phi_k-1)^m}_{\textit{inserted}}, *, \phi_k\}_i$$

without changing the sequence of Φ_i.

Consider for example $\Phi_i = \{1, +, 1\}_i$ and $\Psi_i = \{1, *, 2\}_i$. The CR forms are aligned using Lemmas 3 and 4

$$\begin{aligned}\Phi_i &= \{1, +, 1\}_i \\ &= \{1, +, 1, *, 1\}_i \\ \Psi_i &= \{1, *, 2\}_i \\ &= \{1, +, 1, *, 2\}_i\end{aligned}$$

After alignment the *min* and *max* can be applied

$$\begin{aligned}&min(\{1, +, 1, *, 1\}_i, \{1, +, 1, *, 2\}_i) \\ &= \{1, min(\{1, *, 1\}_i, \{1, *, 2\}_i)\}_i \\ &= \{1, +, 1, *, 1\}_i \\ &max(\{1, +, 1, *, 1\}_i, \{1, +, 1, *, 2\}_i) \\ &= \{1, max(\{1, *, 1\}_i, \{1, *, 2\}_i)\}_i \\ &= \{1, +, 1, *, 2\}_i\end{aligned}$$

The closed forms of the *min* and *max* CRs are $L(i) = i + 1$ and $U(i) = 2^i$ respectively. These bounds are used in Figure 8(b). The dynamic bounds of a conditionally updated induction variable V were calculated by the *min* and *max* of the CR forms of the conditional recurrences.

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, where (symbolic) constant bounds are required.

To determine the direction of a recurrence, we define the step function of a CR.

Definition 3 *The* step *function $\Delta\Phi_i$ of a CR form $\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \ldots, \odot_k, \phi_k\}_i$ is defined by*

$$\Delta\Phi_i = \{\phi_1, \odot_2, \ldots, \odot_k, \phi_k\}_i$$

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*

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

where he 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:

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

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

Note that $\Phi_i = \{\mathcal{V}\Phi_i, \odot_1, \Delta\Phi_i\}_i$.

Definition 4 *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

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

It is important to point out that the $\mathcal{L}$ and $\mathcal{U}$ bounds applied to the recurrence of a monotonic function gives the *exact* (symbolic) value range of the function on a discrete domain, when the function is monotonic on the discrete grid rather than in the continuous domain. A function that is monotonic on discrete grid points is not necessarily monotonic in the continuous domain.

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

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 [47]. These CR forms describe sequences of possible values for the conditionally updated variables in 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 9, 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. For non-enumeration controlled loops such as while-loops, a virtual iteration variable i is introduced with initial value $a = 0$ and stride $s = 1$.

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 recurrence 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.

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 9. Algorithm for Constructing a Recurrence System from a Loop

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:

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 $\langle V, X\rangle \in A$ in topological order ($\prec$) DO
  IF $\langle V, X\rangle$ is marked *for deletion* THEN
    Let $A := A \backslash \{\langle V, X\rangle\}$
  ELSE
    Let $X := \mathcal{CR}(X)$ /* CR construction: replace all i in X by $\{a, +, s\}_i$ and apply CR algebra rules */
    IF X is of the form $V + \Psi_i$, where Ψ_i is a constant or closed-form expression over i or a CR form THEN
      Let $\Phi := \{V_0, +, \Psi_i\}_i$
      SUBSTITUTE(V, Φ, A)
    ELSE IF X is of the form $V * \Psi_i$, where Ψ_i is a constant or closed-form expression over i or a CR form THEN
      Let $\Phi := \{V_0, *, \Psi_i\}_i$
      SUBSTITUTE(V, Φ, A)
    ELSE IF X is of the form $c * V + \Psi_i$, 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 $\Phi := \{\phi_0, +, \phi_1, +, \cdots, +, \phi_{k+1}, *, \phi_{k+2}\}_i$, where
        $\phi_0 = V_0; \quad \phi_j = (c-1)\phi_{j-1} + \psi_{j-1}; \quad \phi_{k+2} = c$
      SUBSTITUTE(V, Φ, A)
    ELSE IF V does not occur in X THEN /* potential wrap-around variable */
      Mark V *wrap-around*
      IF $\langle V, X\rangle$ is marked as *use-before-def* THEN
        Let $\Phi := \{V_0 - \mathcal{V}(\mathcal{B}(X)), *, 0\}_i + \mathcal{B}(X)$
      ELSE
        Let $A := A \backslash \{\langle V, X\rangle\}$
        Let $\Phi := X$
      ENDIF
      SUBSTITUTE(V, Φ, A)
    ELSE /* cannot solve the recurrence for V */
      SUBSTITUTE$(V, \perp, A)$
    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 $\langle V, X\rangle$ in A with $\langle V, \Phi\rangle$
FOR each $\langle U, Y\rangle \in A$, $\langle V, X\rangle \prec \langle U, Y\rangle$ DO
  Mark $\langle U, Y\rangle \in A$ *for deletion*
  Let $Y' := Y[V \leftarrow \Phi]$ /* substitute each use of V with Φ */
  Let $A := A \cup \{\langle U, Y'\rangle\}$
ENDDO

Figure 10. Algorithm for Solving Recurrence Systems

```
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 de-

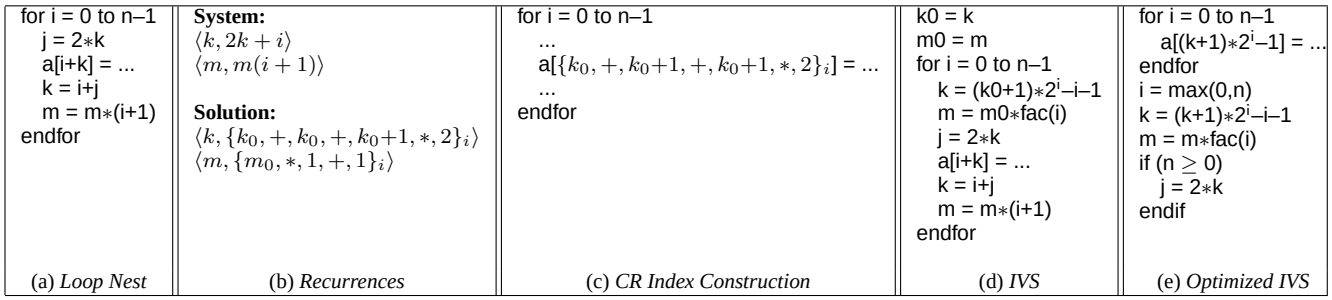

Figure 11. Nonlinear Recurrences

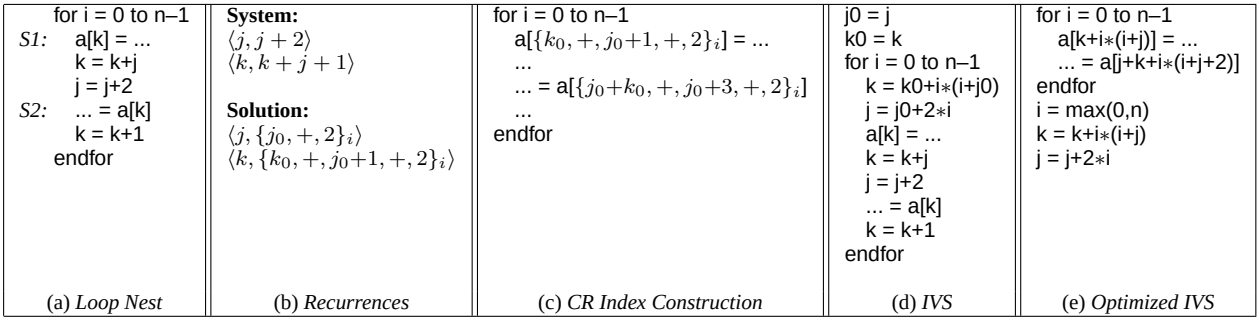

Figure 12. Coupled Nonlinear Recurrences with Multiple Updates

pendencies, symbolic forward substitution, symbolic loop-invariant expressions, and wrap-around variables.

3.6.1. Nonlinear Recurrences. Consider the loop nest shown in Figure 11(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 11(b). The solutions of the recurrences of k and m are computed in CR form. Figure 11(c) depicts the result of CR index construction (see Section 2.3), 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 11(d). The result of conventional restructuring compiler optimizations applied to the IVS code is shown in Figure 11(e). The final adjustments to j, k, and m shown in Figure 11(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 12(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 12(b). Figure 11(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

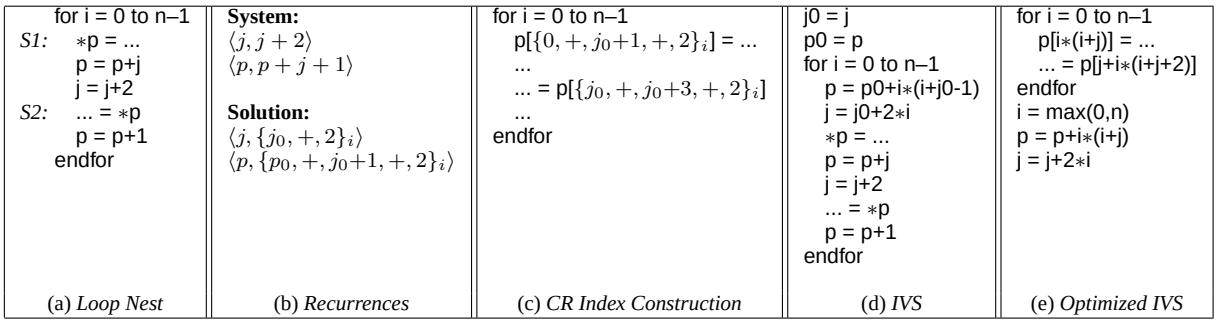

Figure 13. Coupled Nonlinear Pointer Recurrences with Multiple Updates

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 12(d) and (e).

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 13(a) has recurrences and the solutions in CR form shown in Figure 13(b). Figure 13(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 13(d). Conventional restructuring compiler optimization leads to the loop nest shown in Figure 13(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 14(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 14(a). The recurrence system of the inner loop and its solution are shown in Figure 14(b). The CR index of the pointer access shown in Figure 14(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 14(e).

Next, the algorithm proceeds with the outer loop (using the virtually added pointer update information) shown in Figure 14(f). The recurrence system of the outer loop and its solution are shown in Figure 14(g). The CR index of the pointer access shown in Figure 14(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.

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.

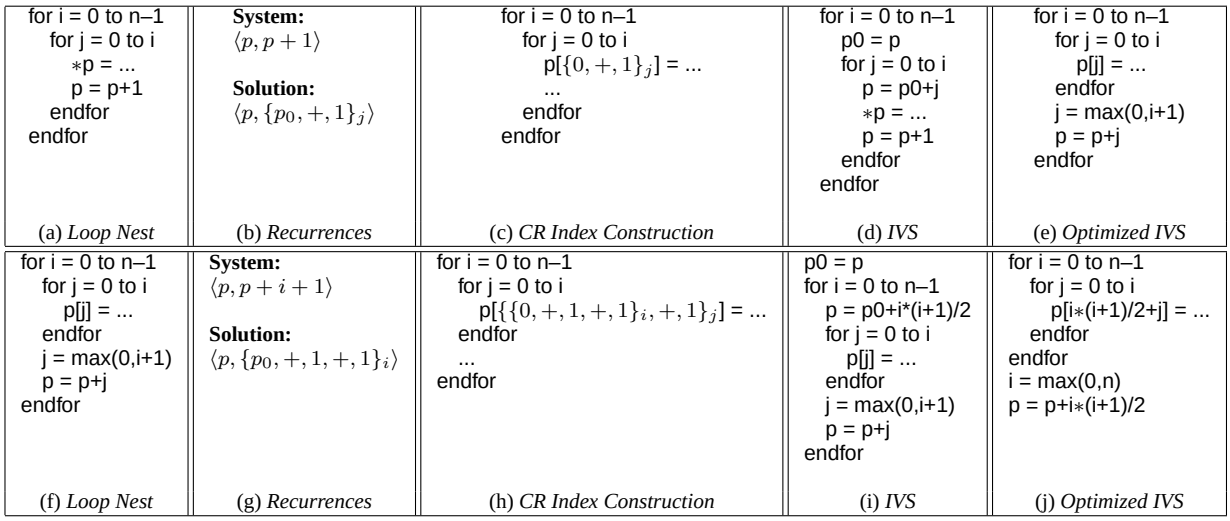

Figure 14. Recurrences in Multidimensional Non-rectangular Loop Nest

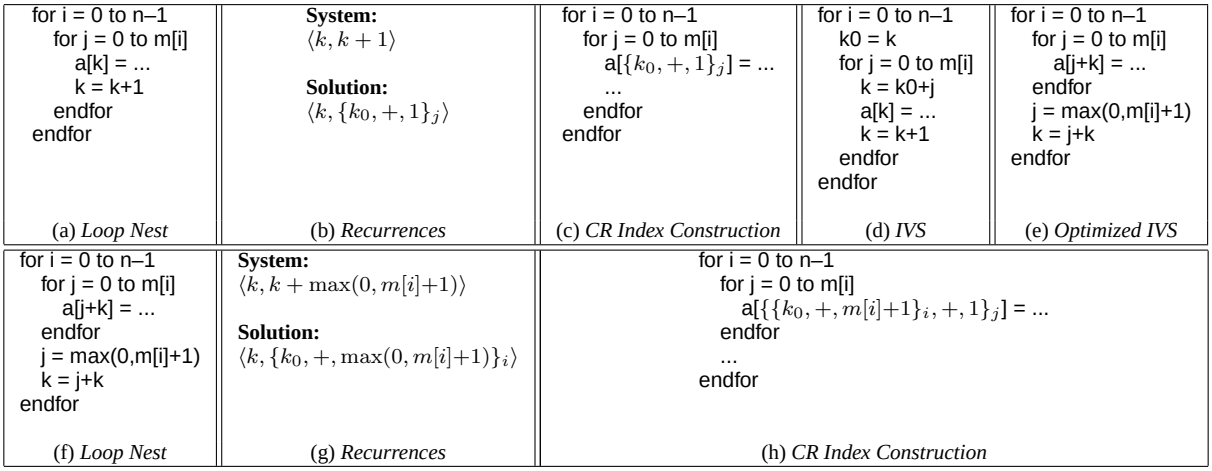

Figure 15. Recurrences with Irregular Symbolic Strides

Because the CR forms of the recurrences have closed forms, IVS can be applied resulting in the loop nest shown in Figure 14(i) and the optimized code shown in Figure 14(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 15(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 15(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 15(e), where k is adjusted for recurrence analysis in the outer loop. The algorithm produces the recurrence system and solution shown in Figure 15(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 15(h). There

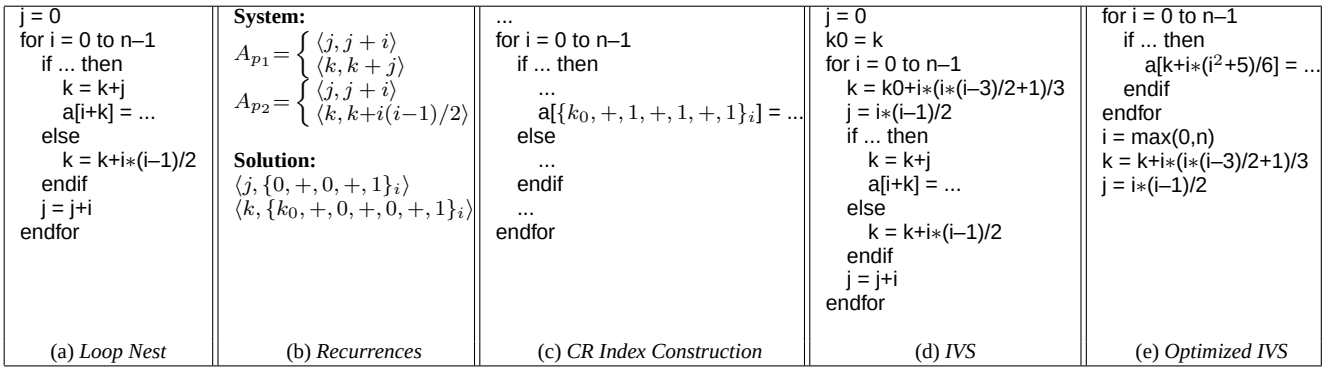

Figure 16. Conditionally Updated Variables with Single Recurrence Solution

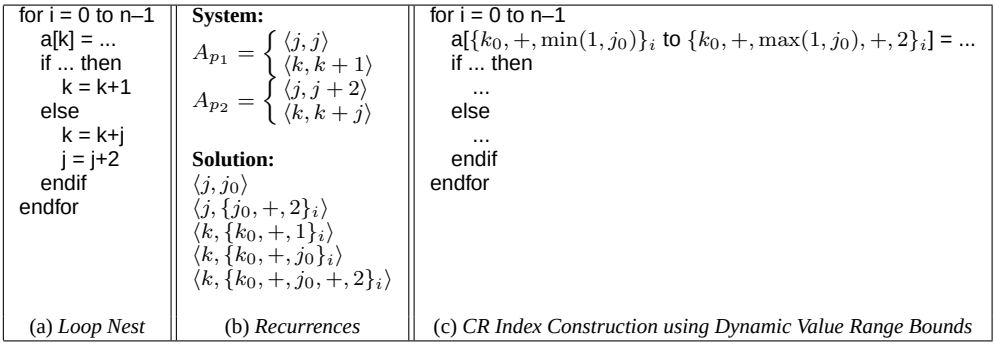

Figure 17. Conditionally Updated Variables with Multiple Recurrence Solutions

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 16(a). The loop exhibits conditional updates of variable k. The recurrence system and its solution are shown in Figure 16(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 16(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 16(d) and (e).

3.6.7. Conditionally Updated Variables with Multiple Recurrence Solutions. Consider the loop nest shown in Figure 17(a). The loop exhibits conditional updates of variables j and k. The recurrence system and its solution are shown in Figure 17(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. Non-Enumeration-Controlled Loops

The recurrence analysis algorithm handles non-enumeration controlled loops such as while-loops. The calculation of recurrence forms does not require a loop trip count. Thus, the loop exit condition can be arbi-

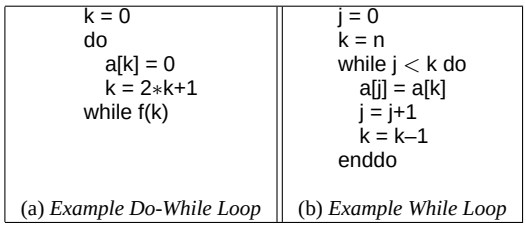

Figure 18. Recurrences in Logically Controlled Loops

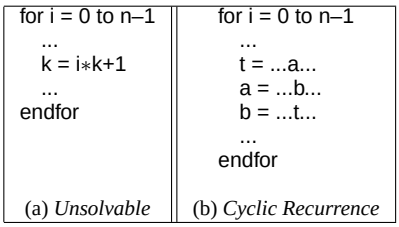

Figure 19. Recurrence Patterns not Recognized

trary. The loop iteration counter, when provided, is used by the algorithm to convert closed-form index expressions to recurrences. The absence of a counter does not inhibit the application of the method, since there are no uses of the counter. Note that the algorithm uses a new (virtual) loop iteration counter to bind the recurrences of induction variables to the loop (see step 1 of the algorithm in Section 3.5. Any unique name for the loop counter suffices to ensure that nested loops in a loop nest are distinguishable to enable the computation of multivariate recurrences.

Consider the loop shown in Figure 18(a). The algorithm selects a unique loop counter to bind the recurrences to the loop, say I. The recurrence form of k is $\{0,+,1,*,2\}_I$ (the algorithm uses Lemma 2). Note that the closed form of k is $2^I - 1$, where I is the selected loop counter variable. Because all writes operations to a are strictly monotonic the loop has no output dependence.

It is important to be able to compute the trip count of a non-enumeration controlled loop, when possible, in order to use the trip count as a constraint in the dependence system. When the exit condition is based on a constraint on a linear or quadratic recurrence, the trip count can be easily determined from the recurrence forms in the exit condition.

Consider for example the loop shown in Figure 18(b). The recurrence form of j is $\{0,+,1\}_I$ and for k is $\{\mathsf{n},+,-1\}_I$. The exit condition in recurrence form is $\{0,+,1\}_I < \{\mathsf{n},+,-1\}_I$. After rearranging terms $\{0,+,1\}_I - \{\mathsf{n},+,-1\}_I < 0$, this is simplified to $\{-\mathsf{n},+,2\}_I < 0$. From the closed form $-\mathsf{n} + 2 * I < 0$ of this recurrence constraint it is easy to see that the trip count of the loop is $I < \lfloor \mathsf{n}/2 \rfloor$. With this constraint the dependence equation $\{0,+,1\}_{I^d} = \{\mathsf{n},+,-1\}_{I^u}$ (i.e. $I^d = \mathsf{n} - I^u$ in closed form), which has no solution when testing for cross-iteration flow dependence (i.e. $<$ dependence direction).

3.8. Recurrence Patterns Not Recognized

This section presents two recurrence patterns that cannot be solved by our recurrence analysis algorithm.

3.8.1. Unsolvable Recurrence Patterns. Some recurrence patterns exist that cannot be solved, such as the recurrence shown in Figure 19(a). The recurrence cannot be solved by our algorithm because it has neither a CR form nor a closed-form equivalent.

3.8.2. Cyclic Recurrence Relations. These relations cannot be analyzed by our algorithm, as shown in Figure 19(b), because the recurrence system constructed from the loop nest must have a partial order $\prec$ on the assignments. Note that our algorithm can handle coupled recurrences with cyclic dependences, but not cyclic recurrencse. Cyclic recurrence systems require a separate solver for periodic sequences, as in [23]. We propose an extension of our algorithm using partial loop unrolling to solve periodic recurrences. The unroll factor is the LCM of the sizes of the strongly connected components in the graph spanned be $\prec$ (edges) on the variables (vertices).

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 compute 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 2 depicting a segment of the original TRFD code. The CR forms of ijkl and l obtained by CR construction are

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

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

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

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 $\mathsf{left} > \mathsf{m}(\mathsf{m}{+}1)/2$, 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 [5], 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,

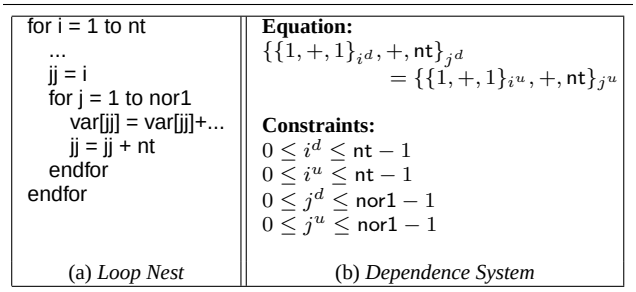

Figure 20. A Linear Dependence System

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 20(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 [54], 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.

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 20(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:

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

The simplified dependence equation from Figure 20(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 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).

```
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 21. A Nonlinear Dependence System

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

$$\left.\begin{matrix}1\\ \{1,+,1\}_{i^d}\end{matrix}\right\} \le i^u \le \mathsf{nt}-1 \quad\Bigg|\quad 0 \le i^d \le \begin{cases}\mathsf{nt}-2\\ \{-1,+,1\}_{i^u}\end{cases}$$

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

The dependence equation is

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

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 21(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 21(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:

$$\left.\begin{matrix}1\\ \{1,+,1\}_{i^d}\end{matrix}\right\} \le i^u \le n-1 \quad\Bigg|\quad 0 \le i^d \le \begin{cases}n-2\\ \{-1,+,1\}_{i^u}\end{cases}$$

$$\left.\begin{matrix}1\\ \{1,+,1\}_{j^d}\end{matrix}\right\} \le j^u \le \{0,+,1\}_{i^u} \quad\Bigg|\quad 0 \le j^d \le \begin{cases}\{-1,+,1\}_{i^d}\\ \{-1,+,1\}_{j^u}\end{cases}$$

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

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

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 22(a). This example is taken from [37], 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 22(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.

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

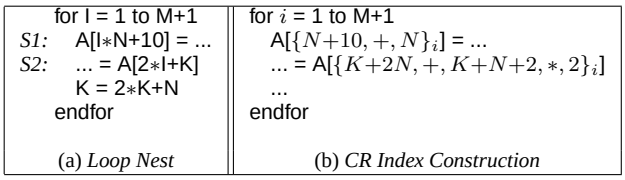

Figure 22. Nonlinear Range Test Example

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.

4.4. Dependence Testing on RTL Code

Register transfer list (RTL) notation is a popular intermediate code representation for low-level instructions used by a variety of compilers, such as *gcc* and *vpo*. The RTL notation is uniform and provides an orthogonal instruction set based on predicated assignments. The *vpo* [7]. compiler optimizes code using a control-flow graph (CFG) representation of the program [1]. Each basic block in the CFG contains a sequence of consecutive register transfer list (RTL) instructions. The RTL notation used by *gcc* is distinctly more Lisp-like compared to the *vpo* RTL. However, the two RTL variants are conceptually the same. Other intermediate representations used by compilers, such as three-address code and indirect triples [1] are similar and can be mapped to RTL instructions.

Our recurrence analysis algorithm recovers induction variables at the three-address code level or RTL level by extracting the assignments to memory and registers (using ADDRECURRENCES in Figure 9) to set up the recurrence system and by applying forward substitution (SUBSTITUTE in Figure 10) to solve the recurrence system.

Figure 23(a) and (b) depict an example loop and the intermediate optimized RTL code generated by *vpo*, respectively. The standard notational conventions and semantics of RTL assignments to registers enables the application of our recurrence analysis algorithm to the integer-valued registers in the code. Register `r[10]` forms a recurrence, because `r[10]` is updated in block `L2` and live at the back edge (determined with data flow analysis [1]). The recurrence analysis algorithm applied to the RTL code results in the recurrence system shown in Figure 23(c). The output dependence equation is shown in Figure 23(d). Because basic induction variables may no longer be associated with loop structures at the (optimized) RTL code level, the basic block number `L2` is used as a reference to the iteration space. In this example, the exit condition can be deduced from the RTL code to determine the size of the iteration

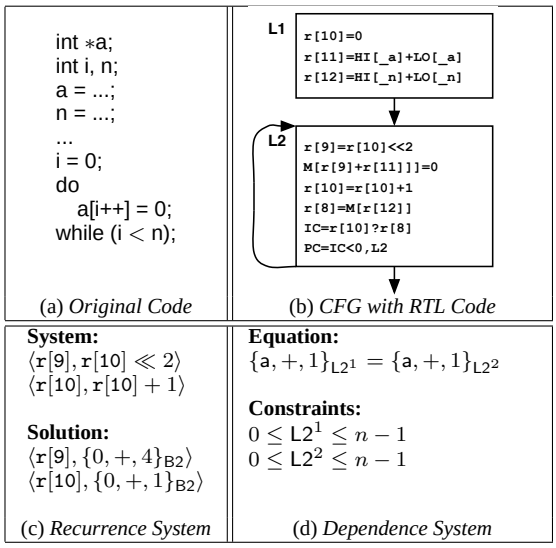

Figure 23. Example RTL Code

space for the dependence system. If the size of the iteration space cannot be deduced from the code, an unknown (symbolic) bound can be substituted instead.

Note that the size of the elements of array `a` are 4 bytes. The memory access `M[r[9]+r[11]]` with `r[9]=r[10]<<2` and `r[11]` is the base address of `a` is represented by the pointer access pattern $\{\mathtt{a},+,1\}_{\mathtt{L2}}$, where the second coefficient is adjusted to represent array element accesses instead of bytes. Because the access pattern $\{\mathtt{a},+,1\}_{\mathtt{L2}}$ is strictly monotonically increasing, there is no output dependence.

We will also demonstrate the application of EVT to disprove output dependence as follows. To test the $<$ output dependence, the constraints are

$$\left.\begin{matrix}1\\ \{1,+,1\}_{\mathtt{L2}^1}\end{matrix}\right\} \leq \mathtt{L2}^2 \leq \mathtt{n}-1 \quad \Big| \quad 0 \leq \mathtt{L2}^1 \leq \begin{cases}\mathtt{n}-2\\ \{-1,+,1\}_{\mathtt{L2}^2}\end{cases}$$

Rearranging the terms in the dependence equation we get

$$\{\{0,+,1\}_{\mathtt{L2}^2},+,-1\}_{\mathtt{L2}^1} = 0$$

The equation has no solution, because

$$\begin{aligned}&\mathcal{L}(\{\{0,+,1\}_{\mathtt{L2}^2},+,-1\}_{\mathtt{L2}^1})\\ &= \mathcal{L}(\{k,+,1\}_{\mathtt{L2}^2} - \{-1,+,1\}_{\mathtt{L2}^1})\\ &= 1 > 0\end{aligned}$$

To test the $>$ output dependence, the constraints are

$$\left.\begin{matrix}1\\ \{1,+,1\}_{\mathtt{L2}^2}\end{matrix}\right\} \leq \mathtt{L2}^1 \leq \mathtt{n}-1 \quad \Big| \quad 0 \leq \mathtt{L2}^2 \leq \begin{cases}\mathtt{n}-2\\ \{-1,+,1\}_{\mathtt{L2}^1}\end{cases}$$

Rearranging the terms in the dependence equation we get

$$\{\{0,+,1\}_{\mathtt{L2}^1},+,-1\}_{\mathtt{L2}^2} = 0$$

No output dependence exists, because the equation has no solution:

$$\begin{aligned}&\mathcal{L}(\{\{0,+,1\}_{\mathtt{L2}^1},+,-1\}_{\mathtt{L2}^2})\\ &= \mathcal{L}(\{k,+,1\}_{\mathtt{L2}^1} - \{-1,+,1\}_{\mathtt{L2}^2})\\ &= 1 > 0\end{aligned}$$

```
void preemphasis(short *signal, short L, short g)
{  short *p1, *p2, i;
   p1 = signal + L – 1;
   p2 = p1 – 1;
   ...
   for (i = 0; i ≤ L – 2; i++)
      *p1––  –= g * *p2––;
   ...
}
```

(a) *Original Source code*

System:
$\langle p1, p1 - 2\rangle$
$\langle p2, p2 - 2\rangle$
$\langle tmp1, p1\rangle$
$\langle tmp2, p2\rangle$
$\langle i, i + 1\rangle$

Solution:
$\langle p1, \{signal + L - 1, +, -1\}_{L2}\rangle$
$\langle p2, \{signal + L - 2, +, -1\}_{L2}\rangle$
$\langle tmp1, \{signal + L - 1, +, -1\}_{L2}\rangle$
$\langle tmp2, \{signal + L - 2, +, -1\}_{L2}\rangle$
$\langle i, \{0, +, 1\}_{L2}\rangle$

(b) *Recurrence System based on CFG*

Equation:
$\{signal + L - 1, +, -1\}_{L2^d}$
$= \{signal + L - 2, +, -1\}_{L2^u}$

Constraints:
$0 \leq L2^d \leq L - 2$
$0 \leq L2^u \leq L - 2$

(c) *Dependence Equation for the use* *tmp2 *and def* *tmp1 *in block* L2

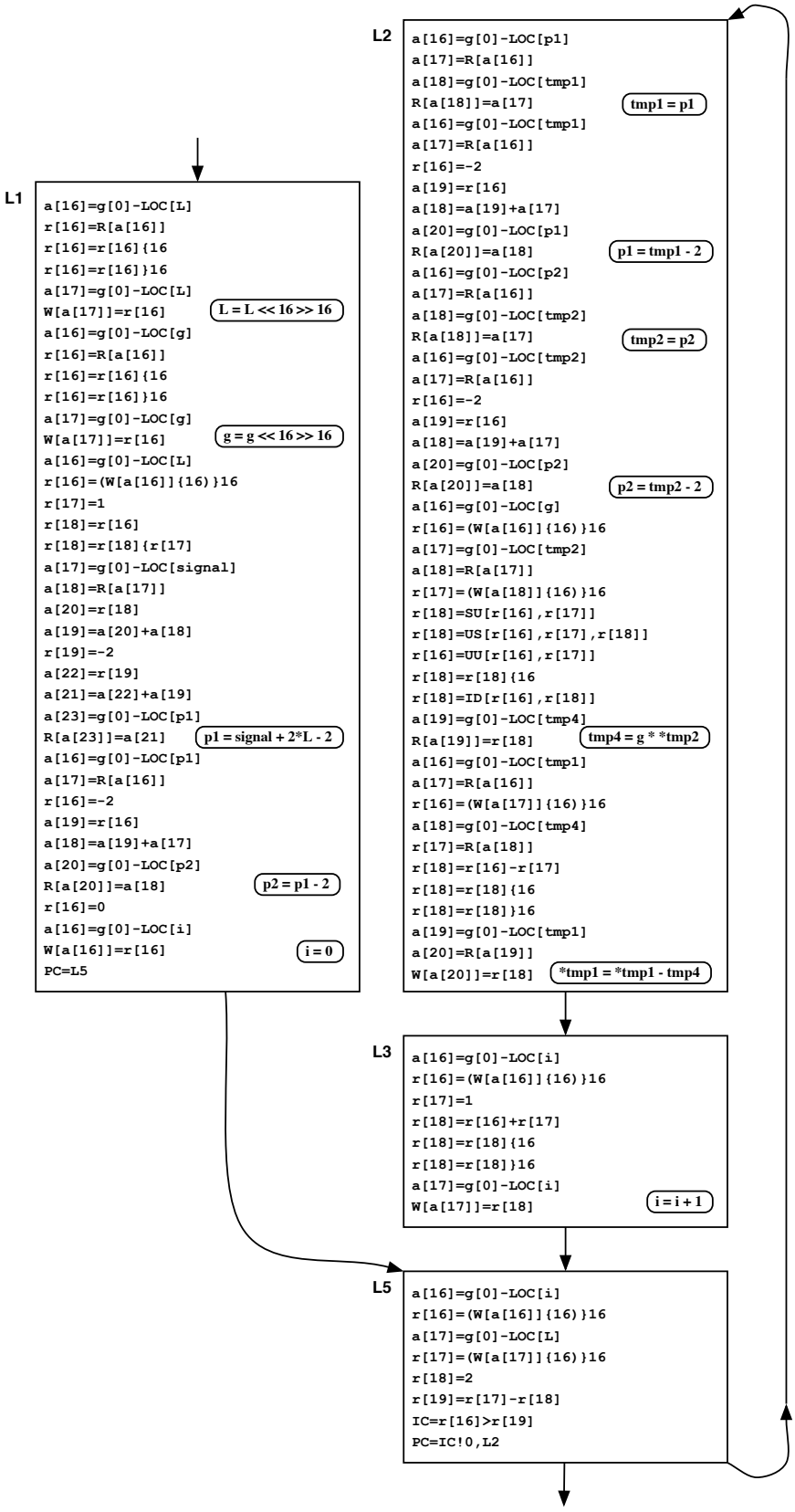

Figure 24. ETSI Codec: Source Code and CFG of the `preemphasis` Routine

A more extensive example taken from the ETSI Speech codec is illustrated in Figure 24. Figure 24(a) shows the source code and the CFG generated by *vpo* is shown on the right. For sake of convenience the CFG is annotated with the memory stores derived from the RTL assignments. The stores are calculated by forward substitution of the RTL defs to the uses determined by the data flow analysis. The application of the recurrence analysis algorithm produces the recurrence system shown in Figure 24(b). The coefficients of the recurrences are adjusted to refer to array elements rather than bytes by scaling the increments down by a factor of two, because the pointers p1 and p2 point to 16 bit integers.

The dependence equation for the use of $*$tmp2 and def of $*$tmp1 in block L2 is shown in Figure 24(c) with the constraints on the loop iteration space. The recurrence system and equation are determined from the CFG. Note that none of the high-level source code details are required. The names of the local variables and function arguments are provided in the RTL code[2]. In this case the number of loop iterations $\mathsf{L}-1$ can be determined as a symbolic expression from the RTL code. If this is not possible, because of the low-level representation, rather than attempting to derive the loop exit condition an unknown symbolic value n can be used to represent the unknown number of loop iterations. This simplification can be made for many types of loop dependence problems without losing accuracy, except in cases where the number of loop iterations is correlated with the size of the accessed array regions (e.g. to detect non-overlapping regions).

To test the $<$ loop carried flow dependence direction, the constraints are

$$\left.\begin{matrix}1\\ \{1,+,1\}_{\mathsf{L2}^d}\end{matrix}\right\} \leq \mathsf{L2}^u \leq \mathsf{L}-2 \quad \Big| \quad 0 \leq \mathsf{L2}^d \leq \begin{Bmatrix}\mathsf{L}-3\\ \{-1,+,1\}_{\mathsf{L2}^u}\end{Bmatrix}$$

After rearranging the terms in the dependence equation we get

$$\{\{-1,+,-1\}_{\mathsf{L2}^u},+,1\}_{\mathsf{L2}^d} = 0$$

The equation has no solution, because

$$\begin{aligned}&\mathcal{U}(\{\{-1,+,-1\}_{\mathsf{L2}^u},+,1\}_{\mathsf{L2}^d})\\ &= \mathcal{U}(\{-1,+,-1\}_{\mathsf{L2}^u} + \{-1,+,1\}_{\mathsf{L2}^u})\\ &= -2 < 0\end{aligned}$$

Therefore, the loop has no forward loop-carried dependence from the $*$tmp1 defs to the $*$tmp2 uses. Likewise, the loop has no forward loop-carried dependence from the $*$tmp1 uses to the $*$tmp1 defs (the dependence equation is similar and the details are not shown).

2 Even the argument and local names are irrelevant, because arguments and locals can be renamed.

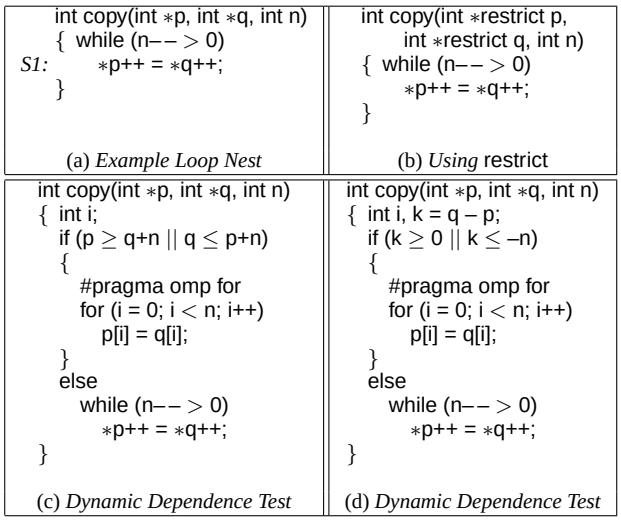

Figure 25. Dynamic Dependence Testing

4.5. Pointer Aliases and Dynamic Dependence Testing

Pointer-based dependence test requires aliasing analysis [32] to prevent inaccurate dependence testing in the presence of pointers that share the same memory region. Points-to analysis [42] constructs a model of pointer based structures which could be used to determine aliases. However, static pointer alias analysis can be very conservative often leading to a conclusion that a large set of pointers can be aliased. In contrast, we propose to use runtime alias analysis [53] to aggressively apply compiler transformations.

We compare two different approaches to dynamic data dependence testing.

- A simple overlap test based on memory interval analysis [53] can be used. When the intervals of memory accesses performed by pointers do not overlap, dependence cannot exist, see also [8, 9]. The overlap check is applicable when pointer (and array) references are affine. To extend the approach to nonlinear references, we use the CR range analysis applied to a pointer to determine the interval information.

  Consider for example Figure 25(a). The CR form of p and q are $\{\mathsf{p},+,1\}_I$ and $\{\mathsf{q},+,1\}_I$ after loop normalization $I = 0,\ldots,\mathsf{n}-1$, where I is a new index variable created for the while-loop (see Section 3.7) and $n-1$ is the bound on the trip count calculated from the recurrence of n (see also Section 3.7). The range of p is $[\mathcal{L}(\{\mathsf{p},+,1\}_I), \mathcal{U}(\{\mathsf{p},+,1\}_I)] = [\mathsf{p}, \mathsf{p}+\mathsf{n}-1]$ and the range of q is $[\mathsf{q}, \mathsf{q}+\mathsf{n}-1]$. The runtime overlap check is shown in Figure 25(c).

- Apply the (nonlinear) inexact EVT test or use an exact

method such as Fourier-Motzkin or Omega test [39, 53] to determine a reduced system of constraints on unknowns to be met for proving that a loop nest is dependence free at compile time or run time. The pointer alias analysis is simply integrated into the test as follows. Note that the distance between the pointers p and q can be used as a parameter in the dependence system by asserting that $\mathsf{q} = \mathsf{p} + k$, where k is an unknown. Because the value of k can be determined at runtime, a dynamic dependence test verifies whether the constraints on k are met to conclude that the loop is dependence free.

Consider for example the application of EVT on the example shown in Figure 25(a) after loop normalization $I = 0, \ldots, \mathsf{n} - 1$. There exists a fixed offset value k such that $\mathsf{q} = \mathsf{p} + k$ before the loop nest, so the initial value of q is replaced by $\mathsf{p} + k$ in the recurrence system for dependence analysis. The CR forms of p and q are $\{\mathsf{p}, +, 1\}_I$ and $\{\mathsf{p} + k, +, 1\}_{+I}$, respectively. The dependence equation is

$$\{\mathsf{p}, +, 1\}_{I^d} = \{\mathsf{p} + k, +, 1\}_{I^u}$$

After rearranging the terms we have

$$\{\{k, +, 1\}_{I^u}, +, -1\}_{I^d} = 0$$

To test the $<$ dependence direction, the constraints are

$$\left.\begin{matrix} 1 \\ \{1, +, 1\}_{I^d} \end{matrix}\right\} \leq I^u \leq \mathsf{n} - 1 \quad \Big| \quad 0 \leq I^d \leq \begin{cases} \mathsf{n} - 2 \\ \{-1, +, 1\}_{I^u} \end{cases}$$

The equation has no solution if

$$\begin{array}{l} \mathcal{L}(\{\{k, +, 1\}_{I^u}, +, -1\}_{I^d}) > 0 \\ \mathcal{L}(\{k, +, 1\}_{I^u} - \{-1, +, 1\}_{I^u}) > 0 \\ \mathcal{L}(k + 1) > 0 \\ k > -1 \end{array}$$

and the dependence equation has no solution if

$$\begin{array}{l} \mathcal{U}(\{\{k, +, 1\}_{I^u}, +, -1\}_{I^d}) < 0 \\ \mathcal{U}(\{k, +, 1\}_{I^u}) < 0 \\ \mathcal{U}(k + \mathsf{n} - 1) < 0 \\ k < 1 - \mathsf{n} \end{array}$$

These constraints are sufficient to verify at runtime to enable dynamic dependence analysis. The transformed code is shown in Figure 25(d). Note that the unknown k is calculated by taking the difference between pointer locations p and q, which according to the semantics of C presents the number of array elements separating p and q.

Note that this approach may create a set of constraints on unknowns, such as k, that is infeasible. To eliminate redundant constraints and verify feasibility of the solution, Fourier-Motzkin elimination [53] can be used.

Note that the restrict keyword can be used in C/C++ application codes to assert that a pointer variable has exclusive access to a memory region, as shown in Figure 25(b). However, compiler hints like restrict are fragile because programmers are responsible to obey the semantics. The automatic detection of aliases at compile time or runtime improves the robustness of the dependence test without relying on program annotations.

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] AMMERGUALLAT, Z., AND HARRISON III, W. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (White Plains, NY, 1990), pp. 283–295.

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

[4] 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.

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

[6] BASTOUL, C. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques* (2004). to appear.

[7] BENITEZ, M. E., AND DAVIDSON, J. W. A Portable Global Optimizer and Linker. In *Proceedings of the SIGPLAN '88*

Symposium on Programming Language Design and Implementation (June 1988), pp. 329–338.

[8] BIK, A. *The Software Vectorization Handbook*. Intel Press, 2004.

[9] BIK, A., GIRKAR, M., AND HAGHIGHAT, M. Incorporating Intel MMX technology into a Java JIT compiler. *Scientific Computing 7*, 2 (1999), 167–184.

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

[11] BLUME, AND EIGENMANN. Nonlinear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems 9*, 12 (December 1998), 1180–1194.

[12] BLUME, W., DOALLO, R., EIGENMANN, R., GROUT, J., HOEFLINGER, J., LAWRENCE, T., LEE, J., PADUA, D., PAEK, Y., POTTENGER, B., RAUCHWERGER, L., AND TU, P. Advanced program restructuring for high-performance computers with Polaris, 1996.

[13] BLUME, W., AND EIGENMANN, R. Performance analysis of parallelizing compilers on the perfect benchmark programs. *IEEE Transactions on Parallel and Distributed Systems 3*, 6 (Nov. 1992), 643–656.

[14] BLUME, W., AND EIGENMANN, R. The range test: a dependence test for symbolic non-linear expressions. In *proceedings of Supercomputing* (1994), pp. 528–537.

[15] 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.

[16] BLUME, W., AND EIGENMANN, R. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium* (April 1995), pp. 357–363.

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

[18] 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.

[19] EUROPEAN TELECOMMUNICATION STANDARD (ETSI). Digital cellular telecommunications system: ANSI-C code for the GSM enhanced full rate (EFR) speech codec. Available from `http://www.etsi.org`.

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

[21] FAHRINGER, T., AND STOLZ, B. A unified symbolic evaluation framework for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems 11*, 11 (Nov. 2000).

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

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

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

[30] LI, W., AND PINGALI, K. A singular loop transformation framework based on non-singular matrices. *Parallel Programming 22*, 2 (1994), 183–205.

[31] 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.

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

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

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

[35] 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).

[36] 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).

[37] 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.

[38] 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.

[39] PUGH, W., AND WONNACOTT, D. Eliminating false data dependences using the Omega test. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Fransisco, CA, June 1992), pp. 140–151.

[40] REDON, X., AND FEAUTRIER, P. Detection of recurrences in sequential programs with loops. In 5^{th} *International Parallel Architectures and Languages Europe* (1993), pp. 132–145.

[41] 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.

[42] RUGINA, R., AND RINARD, M. C. Pointer analysis for multithreaded programs. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (1999), pp. 77–90.

[43] 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.

[44] 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.

[45] 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.

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

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

[48] VAN ENGELEN, R., 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.

[49] VAN ENGELEN, R. A., BIRCH, J., AND GALLIVAN, K. A. Array data dependence testing with the chains of recurrences algebra. In *proceedings of the IEEE International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems (IWIA)* (January 2004), pp. 70–81.

[50] 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.

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

[52] WOLFE, M. Beyond induction variables. In *ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation* (San Fransisco, CA, 1992), pp. 162–174.

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

[54] 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.

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

[56] 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.

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

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