Technical Report TR-071005, Department of Computer Science, Florida State University, October 2007.

Flow-Sensitive Loop-Variant Variable Classification in Linear Time

Yixin Shou *Robert van Engelen*[*]
Deptartment of Computer Science
Florida State University
Tallahassee, FL 32306
{shou,engelen}@cs.fsu.edu

Johnnie Birch
Deptartment of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
birch@cs.utsa.edu

Abstract

This paper presents an efficient algorithm for classifying generalized induction variables and flow-sensitive loop-variant variables that have arbitrary conditional update patterns along multiple paths in a loop nest. Variables are recognized and translated into closed-form functions, such as linear, polynomial, geometric, wrap-around, periodic, and mixer functions. The remaining flow-sensitive variables (those that have no closed forms) are bounded by tight bounding functions on their value sequences by bounds derived from our extensions of the Chains of Recurrences (CR#) algebra. The classification algorithm has a linear worst-case execution time in the size of the SSA region of a loop nest. Classification coverage and performance results for the SPEC2000 benchmarks are given and compared to other methods.

1 Introduction

Induction variables (IVs) [1, 10, 12, 13, 14, 23] are an important class of loop-variant variables whose value progressions form linear, polynomial, or geometric sequences. IV recognition plays a critical role in optimizing compilers as a prerequisite to loop analysis and transformation. For example, a loop-level optimizing compiler applies array dependence testing [23] in loop optimization, which requires an accurate analysis of memory access patterns of IV-indexed arrays and arrays accessed with pointer arithmetic [9, 21]. Other example applications are array bounds check elimination [11], loop-level cache reuse analysis [3], software prefetching [2], loop blocking, variable privatization, IV elimination [1, 10, 12, 22], and auto-parallelization and vectorization [23].

The relative occurrence frequency in modern codes of flow-sensitive loop-variant variables that exhibit more complicated update patterns compared to IVs is significant. The authors found that 9.32% of the total number of variables that occur in loops in CINT2000 are conditionally updated and 2.82% of the total number of variables in loops in CFP2000 are conditionally updated. By contrast to IVs, these variables have no known closed-form function equivalent. As a consequence, current IV recognition methods fail to classify them. The result is a pessimistic compiler analysis outcome and lower performance expectations.

Closer inspection of the SPEC2000 benchmarks reveals that value progressions of *all* of these flow-sensitive loop-variant variables can be bounded with tight bounding functions that are defined over the loop iteration space. Typically a pair of linear lower- and upper-bound functions on

[*]Supported in part by NSF grant CCF-0702435.

variables that have conditional increments suffices for simple cases, such as conditionally updated counters. However, more complicated cases do exist in these benchmarks, where variables have multiple "discordant" updates along different paths in the loop body. Bounding the value progressions of these variables has the advantage of increased loop analysis coverage. Bounding also significantly alleviates loop analysis accuracy problems in the presence of unknowns. Most compilers will simply give up on loop analysis and optimization when a single variable with a recurrence in a loop has an unknown value progression. With the availability of tight functional (i.e. iteration-specific) bounds on variables, analysis and optimization can continue. For example, in [7, 20] it was shown that current dependence analysis methods can be extended to handle such functional bounds. We believe this approach can also strengthen methods for array bounds check elimination, loop-level cache reuse analysis, software prefetching, and loop restructuring optimizations that require dependence analysis.

Efficient automatic classification of flow-sensitive variables has two challenges to overcome:

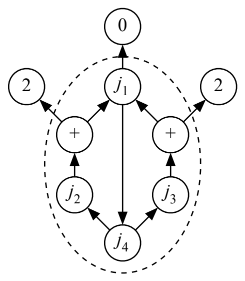
- How to symbolically construct accurate bounding functions on the value progressions of variables that are conditionally updated, conditionally reinitialized, and more generally, exhibit multiple coupled assignments in the branches of a loop body?
- How reduce the search cost for flow-sensitive variables across all branches in a loop nest when loops may have an exponentially growing number of control flow paths?

To collect coupled variable update operations in a loop, search methods that use loop body path enumeration can be used. However, full path enumeration requires an exponential number of steps to complete in the worst case. Furthermore, the use of bounds should be restricted to the necessary cases only. This means that the "traditional" form of IVs in loops should still be classified as linear, polynomial, and geometric. Thus, speed of a classification algorithm can only be traded in for accuracy of classifying flow-sensitive variables that have (multiple) conditional updates in loops.

This paper presents a linear-time flow-sensitive loop-variant variable analysis algorithm based on the method by Gerlek et al. [10] and the CR# (CR-sharp) algebra [19]. The contributions of this paper are:

- A systematic classification approach based on new CR# algebra extensions to analyze a large class of loop-variant variables "in one sweep" without the need for a-priori classification and recurrence solvers.
- A new algorithm for classification of flow-sensitive variables, i.e. variables that are arbitrarily updated in multiple branches of the loop body, with a running time that scales linearly with the size of the code region of a loop nest.
- An implementation in GCC 4.1 of the classifier.

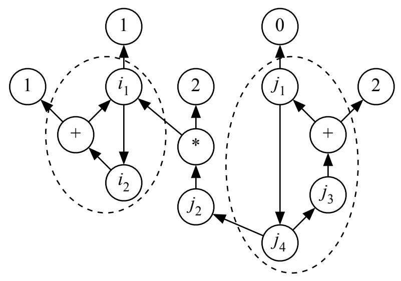
The remainder of this paper is organized as follows. In Section 2 we compare related work. Section 3 gives CR# algebra preliminaries. Section 4 presents the linear time, flow-sensitive IV classification algorithm based on the CR# algebra. In Section 5 results are presented using an implementation in GCC 4.1. Performance results on SPEC2000 show increased classification coverage with a very low running time overhead. Section 6 summarizes the conclusions.



```
loop
  $j_1 = \phi(0, j_4)$
    if (...)
      $j_2 = j_1 + 2$
    else
      $j_3 = j_1 + 2$
    $j_4 = \phi(j_2, j_3)$
endloop
```

Closed-form solution: $j_1(I) = 2{*}I$

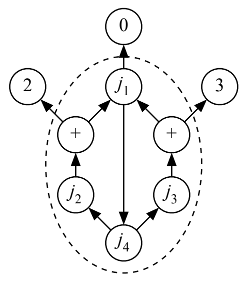
(a) *Syntactically equivalent updates*



```
loop
  $i_1 = \phi(1, i_2)$
  $j_1 = \phi(0, j_4)$
    if (...)
      $j_2 = i_1 * 2$
    else
      $j_3 = j_1 + 2$
    $j_4 = \phi(j_2, j_3)$
    $i_2 = i_1 + 1$
endloop
```

Closed-form solution: $j_1(I) = 2{*}I$

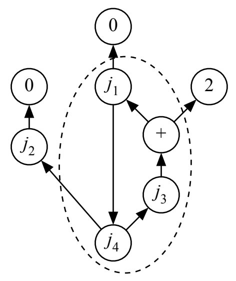
(b) *Semantically equivalent updates*



```
loop
  $j_1 = \phi(0, j_4)$
    if (...)
      $j_2 = j_1 + 2$
    else
      $j_3 = j_1 + 3$
    $j_4 = \phi(j_2, j_3)$
endloop
```

Bounds: $2{*}I \leq j_1(I) \leq 3{*}I$

(c) *Discordant updates*



```
loop
  $j_1 = \phi(0, j_4)$
    if (...)
      $j_2 = 0$
    else
      $j_3 = j_1 + 2$
    $j_4 = \phi(j_2, j_3)$
endloop
```

Bounds: $0 \leq j_1(I) \leq 2{*}I$

(d) *Conditional re-initialization*

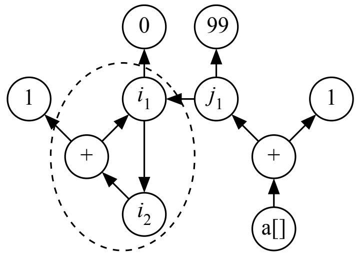
Figure 1: Loops with Flow-Sensitive Loop-Variant Variable Updates

2 Related Work

While the recognition of "traditional" forms of IVs is extensively described in the literature, there is a limited body of work on methods to analyze more complicated flow-sensitive loop-variant variables that have arbitrary conditional update patterns along multiple paths in a loop nest. We compared this related work to our approach. To compare the capabilities of all of these approaches, Figure 1 shows four example loop structures[1] with a classification of their fundamentally different characteristics.

The method by Gerlek, Stoltz and Wolfe [10] classifies IVs by detecting *Strongly Connected Components* (SCCs) in a FUD/SSA graph using a variant of Tarjan's algorithm [16]. Each SCC represents an IV or a loop-variant variable. A collection of interconnected SCCs represent a set of interdependent IVs. The IV classification proceeds by matching the update statement patterns for linear, geometric, periodic, and polynomial IVs and by constructing the closed-form characteristic function of each IV using a sequence-specific recurrence solver. Induction variable substitution (IVS) is then applied to replace induction expressions with equivalent closed-form functions.

[1]All examples in this report will be given in *Single Static Assignment* (SSA) form.



```
loop
  $i_1 = \phi(0, i_2)$
  $j_1 = \phi(99, i_1)$
    a$[j_1 + 1] = \ldots$
    $\ldots$
    $i_2 = i_1 + 1$
endloop
```

$$\begin{aligned} i_1 &= \{0, 1, 2, 3, 4, \ldots\} \\ j_1 &= \{99, 0, 1, 2, 3, \ldots\} \end{aligned}$$

Figure 2: SCC of the SSA Form of an Example Loop with a Wrap-around Variable

The method suggests a *sequence strengthening method* to handle restricted forms of conditionally-updated variables. However, the variables in Figure 1(a) and (c) would be loosely classified as a *monotonic variables*, without identifying its linear sequence or bounds.

Loops with *syntactic and semantically equivalent updates* Figure 1(a,b) require aggressive symbolic analysis and expression manipulation to prove equivalence of updates in branches. Haghighat and Polychronopoulos [12] present a *symbolic differencing* technique to capture induction variable sequences by applying abstract interpretation. Symbolic differencing with abstract interpretation is expensive. They do not handle the classes of loops shown in Figure 1(c,d).

Wu et al. [24] introduce a loop-variant variable analysis technique that constructs a lattice of *monotonic evolutions* of variables, which includes variables with *discordant updates* Figure 1(c). However, her approach only determines the *direction* in which a variable changes and other information such as strides are lost. Closed-form functions of IV progressions are not computed.

Recent work by several authors [6, 15, 17, 18] incorporates the *Chains of Recurrences* (CR) algebra [25] for IV recognition and manipulation. The use of CR forms eliminates the need for a-priori classification, pattern matching, and recurrence solvers. All of these approaches use a variation of an algorithm originally proposed by Van Engelen [18] to construct CR forms for IVs. The primary advantage of these methods is the manipulation of CR-based recurrence forms rather than closed-form functions, which gives greater coverage by including the recognition and manipulation of IVs that have no closed forms.

An extensive loop-variant variable recognition approach based on CR forms is presented in [20]. The approach captures value progressions of all types of conditionally-updated loop-variant variables Figure 1(a-d). The method uses *full path enumeration* on *Abstract Syntax Tree* (AST) forms. The algorithm has an exponential worst-case complexity due to the complexity of path enumeration.

The class of *re-initialized variables* Figure 1(d) and *wrap-around variables* shown in Figure 2 are special cases of "out-of-sequence variables" (OSV), which take a known sequence but have exceptional "out-of-sequence" restart values that occur at certain, but compile-time unpredictable, events in the loop iteration space. Even though the relative percentage of these types of variables in benchmarks is low (0.55% in CINT2000 and to 0.62% in CFP2000), their classification is important to enable loop restructuring [12]. The common *wrap-around variable* is a special case of an OSV: it is assigned a value outside the loop for the first iteration and then takes the value sequence of another IV for the remainder of the iterations. Wrap-around variables can cascade: any IV that depends on the value of a wrap-around variable is itself a wrap-around variable of one order higher [10] (two iterations with out-of-sequence values). Wrap-around variables are recognized by methods described in [6, 10, 12, 18].

By contrast to these methods, the approach presented in this paper enables analysis of coupled loop-variant variables in multiple SCCs Figure 1(a-b) (both formed by conditional and unconditional flow) in "one sweep" and constructs lower- and upper-bounding functions for flow-sensitive variables Figure 1(c-d).

3 Preliminaries

This section briefly introduces Chains of Recurrences (CR). For more details, see [4, 18, 25].

3.1 The CR Notation and Algebra

The CR notation and algebra was introduced by Zima [25] and later extended by Bachmann [4] and Van Engelen [18]. A *basic recurrence* Φ_i is of the form:

$$\Phi_i = \{\varphi_0, \odot_1, f_1\}_i$$

which represents a sequence of values starting with an initial value φ_0, updated in each iteration over a uni-distant grid by operator $\odot_1$ (either $+$ or $*$) and stride value f_1. When f_1 is constant, the sequence is either linear ($\odot = +$) or geometric ($\odot = *$). When f_1 is a non-constant function in CR form then we have a *chain of recurrences*:

$$\Phi_i = \{\varphi_0, \odot_1, \{\varphi_1, \odot_2, \{\varphi_2, \cdots, \odot_k, \{\varphi_k\}_i\}_i\}_i\}_i$$

which is usually written in flattened form

$$\Phi_i = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \cdots, \odot_k, \varphi_k\}_i$$

Any discrete real- or complex-valued function can be represented by a CR, assuming that the stride function $\varphi_k = f_k(i)$ is sufficiently expressive. For constant φ_k, the CR notation represents polynomials, exponentials, factorials, and trigonometric functions (exponentials in the complex domain). The value sequences of three example CR forms is illustrated below:

iteration $i =$	0	1	2	3	4	5	...
$\{2, +, 1\}_i$ value sequence $=$	2	3	4	5	6	7	...
$\{1, *, 2\}_i$ value sequence $=$	1	2	4	8	16	32	...
$\{1, *, \frac{1}{2}\}_i$ value sequence $=$	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	...
$\{0, +, 0, +, 1\}_i$ value sequence $=$	0	0	1	3	6	10	...
$\{1, *, 2, +, 1\}_i$ value sequence $=$	1	2	6	24	120	720	...
$\{0, +, 1, *, 2\}_i$ value sequence $=$	0	1	3	7	15	31	...

The semantics of the one-dimensional CR form is defined by the pseudo-code loop template shown in Figure 3. The loop computes the value sequence $val[i]$ of the CR form $\Phi_i = \{\varphi_0, \odot_1, \cdots, \odot_k, \varphi_k\}_i$ for $i = 0, \ldots, n$ using variables cr_j initialized with the (symbolic) CR coefficients.

For example, the value sequence of the CR form $\{1, *, 1, +, 1\}$, where $cr_0 = cr_1 = \varphi_2 = 1$, is computed for $n = 5$ as follows:

iteration $i =$	0	1	2	3	4	5
$val[i] =$	1	1	2	6	24	120
$cr_0 =$	1	2	6	24	120	2880
$cr_1 =$	2	3	4	5	6	7
$\varphi_2 =$	1	1	1	1	1	1

```
$cr_0$     $= \varphi_0$
$cr_1$     $= \varphi_1$
 $\vdots$       $= \vdots$
$cr_{k-1}$ $= \varphi_{k-1}$
for $i = 0$ to $n-1$
   $val[i]$ $= cr_0$
   $cr_0$     $= cr_0$     $\odot_1$ $cr_1$
   $cr_1$     $= cr_1$     $\odot_2$ $cr_2$
    $\vdots$       $= \vdots$         $\vdots$     $\vdots$
   $cr_{k-1}$ $= cr_{k-1}$ $\odot_k$ $\varphi_k$
endfor
```

Figure 3: Semantics of the CR Form $\Phi_i = \{\varphi, \odot_1, \cdots, \odot_k, \varphi_k\}_i$ Expressed as a Loop Template

During each iteration i, the value at $val[i]$ is set to cr_0, cr_0 is set to $cr_0 * cr_1$, and cr_1 is set to $cr_1 + \varphi_2$, thereby producing the well-known factorial sequence $val[i] = i!$.

Multi-variate CRs (MCR) are CRs with coefficients that are CRs in a higher dimension [4]. Multi-dimensional loops are used to evaluate MCRs over grids.

The power of CR forms is exploited with the CR algebra: its simplification rules produce CRs for multivariate functions and functions in CR form can be easily combined. Below is a selection of CR algebra rules[2]:

$$\begin{array}{rcl}
c * \{\varphi_0, +, f_1\}_i & \Rightarrow & \{c*\varphi_0, +, c*f_1\}_i \\
\{\varphi_0, +, f_1\}_i \pm c & \Rightarrow & \{\varphi_0 \pm c, +, f_1\}_i \\
\{\varphi_0, +, f_1\}_i \pm \{\psi_0, +, g_1\}_i & \Rightarrow & \{\varphi_0 \pm \psi_0, +, f_1 \pm g_1\}_i \\
\{\varphi_0, +, f_1\}_i * \{\psi_0, +, g_1\}_i & \Rightarrow & \{\varphi_0*\psi_0, +, \{\varphi_0, +, f_1\}_i*g_1+\{\psi_0, +, g_1\}_i*f_1+f_1*g_1\}_i
\end{array}$$

CR rules are applicable to IV manipulation. For example, suppose i is a loop counter with CR $\{0, +, 1\}_i$ and j is a linear IV with CR $\{j_0, +, 2\}_i$ which has a symbolic unknown initial value j_0. Then expression $i^2 + j$ is simplified to

$$\{0, +, 1\}_i * \{0, +, 1\}_i + \{j_0, +, 2\}_i \Rightarrow \{0, +, 1, +, 2\} + \{j_0, +, 2\}_i \Rightarrow \{j_0, +, 3, +, 2\}_i$$

The closed form function f of this CR is $f(I) = j_0 + I * (I + 2)$, which is derived by the application of the CR inverse rules defined in [17]. A lattice of CR forms for simplification and methods for IV analysis is introduced in [19]. In [17] Van Engelen proved that the CR algebra as a term rewriting system (TRS) is complete (confluent and terminating). Therefore, CR forms are *normal forms* of the CR algebra TRS.

3.2 The CR# Algebra

The CR# (CR-sharp) algebra is an extension of the CR algebra with new operators, algebra rules, and CR form alignment operations to derive CR bounding functions. The key idea is to replace the $\odot$ operator in a CR form with a selection operator while retaining the formal semantics of a CR.

[2]See [17] for the complete list of CR algebra simplification rules.

3.2.1 The Delay Operator of the CR# Algebra

Delay operator # was introduced in the CR algebra as an extension (CR#) by Van Engelen in [19].

Definition 1 *The* delay operator *# is a right-selection operation defined by*

$$(x\#y) = y \qquad\qquad \textit{for any } x \textit{ and } y.$$

CRs with #-operators will be referred to as *delayed CRs*. The #-operator allows several initial values to take effect before the rest of the sequence kicks in:

iteration $i =$	0	1	2	3	4	5	...
$\{9, \#, 1, +, 2\}_i$ value sequence =	9	1	3	5	7	9	...
$\{1, *, 1, \#, 2\}_i$ value sequence =	1	1	2	4	8	16	...

Delayed CRs are an essential instrument to analyze "out-of-sequence variables".

3.2.2 CR# Alignment and CR# Bounds

To effectively analyze the value sequences of conditionally updated variables that are present in a loop nest, new rules for CR# alignment and CR# bounds construction are introduced. Two or more CR forms of different lengths or with different operations can be aligned for comparison.

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, the CR $\{1, +, 1, *, 1\}$ is aligned with the CR $\{0, +, 2, *, 2\}_i$. The CR $\{1, +, 2\}_i$ is not aligned with the CR $\{1, *, 2\}_i$, because the operators differ. The CR $\{1, +, 2\}_i$ is not aligned with the CR $\{1, +, 2, +, 1\}_i$, because the lengths differ.

To align a (delayed) CR form of a mixed polynomial and geometric function to a longer (delayed) CR form, + operators can be inserted. This allows for pairwise alignment of CRs by moving the $*$ operators to higher order CR coefficients.

Lemma 1 *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 in Section 3.1 the sequence $f[i]$ for $\{1, *, r\}_i$ and $g[i]$ for $\{1, +, r-1, *, r\}_i$ are computed by the following two loop templates:

```
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 = cr_0 * $r$         $g[i]$ = cr_0
endfor                       cr_0 = cr_0 + cr_1
                             cr_1 = 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 in [19] 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 [5, 26], 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 $\Phi_i = \{\varphi_0, \odot_1, \cdots, \odot_{k-1}, \varphi_{k-1}, *, \varphi_k\}_i$ 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*

$$\Phi_i = \{\varphi_0, \odot_1, \cdots, \odot_{k-1}, \varphi_{k-1}, \underbrace{+, \varphi_{k-1}(\varphi_k-1), +, \varphi_{k-1}(\varphi_k-1)^2, +, \cdots, +, \varphi_{k-1}(\varphi_k-1)^m}_{\textit{inserted}}, *, \varphi_k\}_i$$

without changing the value sequence of Φ_i.

A delay operator can be inserted in a CR form according to the following lemma.

Lemma 2 *Let $\Phi_i = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \ldots, \odot_k, \varphi_k\}_i$ be a (multivariate) CR form. Then,*

$$\Phi_i = \{\varphi_0, \#, \mathcal{F}\Phi_i\}_i$$

where F is called the forward shift operator*, defined by*

$$\mathcal{F}\Phi_i = \{\psi_0, \odot_1, \psi_1, \odot_2, \ldots, \odot_k, \psi_k\}_i$$

with $\psi_j = \varphi_j \odot_{j+1} \varphi_{j+1}$ for $j = 0, \ldots, k-1$ and $\psi_k = \varphi_k$.

Proof. The value sequence $\mathsf{val}[i]$ of CR form $\Phi_i = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \ldots, \odot_k, \varphi_k\}_i$ for $i = 0, ..., n-1$ is computed by the following CR loop template:

```
$\mathsf{cr}_0$     $= \varphi_0$
$\mathsf{cr}_1$     $= \varphi_1$
 :      =  :
$\mathsf{cr}_{k-1}$ $= \varphi_{k-1}$
for $i = 0$ to $n-1$
   $\mathsf{val}[i]$ $= \mathsf{cr}_0$
   $\mathsf{cr}_0$     $= \mathsf{cr}_0 \odot_1 \mathsf{cr}_1$
   $\mathsf{cr}_1$     $= \mathsf{cr}_1 \odot_2 \mathsf{cr}_2$
    :      =  :      :   :
   $\mathsf{cr}_{k-1}$ $= \mathsf{cr}_{k-1} \odot_k \varphi_k$
endfor
```

After loop peeling, we obtain

```
$\mathsf{cr}_0$     $= \varphi_0$
$\mathsf{cr}_1$     $= \varphi_1$
 $\vdots$      $=$ $\vdots$
$\mathsf{cr}_{k-1}$ $= \varphi_{k-1}$
$\mathsf{val}[0]$ $= \mathsf{cr}_0$
$\mathsf{cr}_0$     $= \mathsf{cr}_0 \odot_1 \mathsf{cr}_1$
$\mathsf{cr}_1$     $= \mathsf{cr}_1 \odot_2 \mathsf{cr}_2$
 $\vdots$      $=$ $\vdots$     $\vdots$   $\vdots$
$\mathsf{cr}_{k-1}$ $= \mathsf{cr}_{k-1} \odot_k \varphi_k$
for $i = 1$ to $n-1$
   $\mathsf{val}[i]$ $= \mathsf{cr}_0$
   $\mathsf{cr}_0$     $= \mathsf{cr}_0 \odot_1 \mathsf{cr}_1$
   $\mathsf{cr}_1$     $= \mathsf{cr}_1 \odot_2 \mathsf{cr}_2$
    $\vdots$      $=$ $\vdots$     $\vdots$   $\vdots$
   $\mathsf{cr}_{k-1}$ $= \mathsf{cr}_{k-1} \odot_k \varphi_k$
endfor
```

After rewriting the initialization assignments, we have

```
$\mathsf{s}$       $= \varphi_0$
$\mathsf{cr}_0$     $= \varphi_0 \odot_1 \varphi_1$
$\mathsf{cr}_1$     $= \varphi_1 \odot_2 \varphi_2$
 $\vdots$      $=$ $\vdots$
$\mathsf{cr}_{k-1}$ $= \varphi_{k-1} \odot_k \varphi_k$
$\mathsf{val}[0]$ $= \mathsf{s}$
for $i = 1$ to $n-1$
   $\mathsf{val}[i]$ $= \mathsf{cr}_0$
   $\mathsf{cr}_0$     $= \mathsf{cr}_0 \odot_1 \mathsf{cr}_1$
   $\mathsf{cr}_1$     $= \mathsf{cr}_1 \odot_2 \mathsf{cr}_2$
    $\vdots$      $=$ $\vdots$     $\vdots$   $\vdots$
   $\mathsf{cr}_{k-1}$ $= \mathsf{cr}_{k-1} \odot_k \varphi_k$
endfor
```

By using s as a wrap-around variable in the loop to sink val[0]=s back into the loop, we obtain

```
$\mathsf{s}$       $= \varphi_0$
$\mathsf{cr}_0$     $= \varphi_0 \odot_1 \varphi_1$
$\mathsf{cr}_1$     $= \varphi_1 \odot_2 \varphi_2$
 $\vdots$      $=$ $\vdots$
$\mathsf{cr}_{k-1}$ $= \varphi_{k-1} \odot_k \varphi_k$
for $i = 0$ to $n-1$
   $\mathsf{val}[i]$ $= \mathsf{s}$
   $\mathsf{s}$       $= \mathsf{cr}_0$
   $\mathsf{cr}_0$     $= \mathsf{cr}_0 \odot_1 \mathsf{cr}_1$
   $\mathsf{cr}_1$     $= \mathsf{cr}_1 \odot_2 \mathsf{cr}_2$
    $\vdots$      $=$ $\vdots$     $\vdots$   $\vdots$
   $\mathsf{cr}_{k-1}$ $= \mathsf{cr}_{k-1} \odot_k \varphi_k$
endfor
```

To see that this sequence computation is equivalent to the sequence of $\{\varphi_0, \#, \mathcal{F}\Phi_i\}_i$, we use the definition of the forward shift operator $\mathcal{F}$. The value sequence $\mathsf{val}[i]$ of CR form $\{\varphi_0, \#, \mathcal{F}\Phi_i\}_i$ is computed by the CR template:

```
s       = $\varphi_0$
cr$_0$     = $\psi_0$     = $\varphi_0$     $\odot_1$ $\varphi_1$
cr$_1$     = $\psi_1$     = $\varphi_1$     $\odot_2$ $\varphi_2$
 :      =  :
cr$_{k-1}$ = $\psi_{k-1}$ = $\varphi_{k-1}$ $\odot_k$ $\varphi_k$
for $i$ = 0 to $n-1$
   val[$i$]  = s
   s       = s       #   cr$_0$
   cr$_0$     = cr$_0$     $\odot_1$ cr$_1$
   cr$_1$     = cr$_1$     $\odot_2$ cr$_2$
    :      =  :       :   :
   cr$_{k-1}$ = cr$_{k-1}$ $\odot_k$ $\varphi_k$
endfor
```

Because the assignment $\mathsf{s} = \mathsf{s} \# \mathsf{cr}_0$ is identical to $\mathsf{s} = \mathsf{cr}_0$, the value computations are semantically equivelent. Thus, this completes the proof that

$$\Phi_i = \{\varphi_0, \#, \mathcal{F}\Phi_i\}_i$$

□

To align two CR forms of unequal length, the shorter CR can be lengthened by adding dummy operations as follows.

Lemma 3 *Let $\Phi_i = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \cdots, \odot_k, \varphi_k\}_i$ be a (multivariate) CR form, where φ_k is invariant of i. Then, the following identities hold*

$$\begin{aligned}
\Phi_i &= \{\varphi_0, \odot_1, \varphi_1, \odot_2, \cdots, \odot_k, \varphi_k, +, 0\}_i \\
\Phi_i &= \{\varphi_0, \odot_1, \varphi_1, \odot_2, \cdots, \odot_k, \varphi_k, *, 1\}_i \\
\Phi_i &= \{\varphi_0, \odot_1, \varphi_1, \odot_2, \cdots, \odot_k, \varphi_k, \#, \varphi_k\}_i
\end{aligned}$$

Proof. The proof immediately follows from the CR semantics defined in Section 3.1, 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, or by the operation $\mathsf{cr}_k \# \mathsf{cr}_k$).
□

From Lemmas 1, 2, and 3 it follows that any two CR forms can be aligned. Consider for example

$$\Phi_i = \{1, \#, 1, +, 2\}_i = \{1, \#, 1, +, 2, *, 1\}_i$$
$$\Psi_i = \{1, *, 2\}_i = \{1, \#, 2, *, 2\}_i = \{1, \#, 2, +, 2, *, 2\}_i$$

Alignment allows comparisons to be made between pairwise CR coefficients to determine bounds. By comparing the coefficients of two CR forms we can determine the min/max bounds of two CR forms as defined as follows.

Definition 3 *The* minimum *of two CR form is inductively defined by*

$$\min(\{\varphi_0, \#, f_1\}_i, \{\psi_0, \#, g_1\}_i) = \{\min(\varphi_0, \psi_0), \#, \min(f_1, g_1)\}_i$$
$$\min(\{\varphi_0, +, f_1\}_i, \{\psi_0, +, g_1\}_i) = \{\min(\varphi_0, \psi_0), +, \min(f_1, g_1)\}_i$$
$$\min(\{\varphi_0, *, f_1\}_i, \{\psi_0, *, g_1\}_i)$$
$$= \begin{cases} \{\min(\varphi_0, \psi_0), *, \min(f_1, g_1)\}_i & \\ & \text{if } \varphi_0{>}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{min(\varphi_0, \psi_0), *, \max(f_1, g_1)\}_i & \\ & \text{if } \varphi_0{<}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\varphi_0, *, f_1\}_i & \text{if } \varphi_0{<}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\psi_0, *, g_1\}_i & \text{if } \varphi_0{>}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{-\max(|\varphi_0|, |\psi_0|), *, \max(|f_1|, |g_1|)\}_i & \text{if } f_1{<}0 \vee g_1{<}0 \\ \bot & \text{otherwise} \end{cases}$$

where the sign of the coefficients is determined using the monotonicity properties of the coefficients. The maximum *of two CR forms is inductively defined by*

$$\max(\{\varphi_0, \#, f_1\}_i, \{\psi_0, \#, g_1\}_i) = \{\max(\varphi_0, \psi_0), \#, \max(f_1, g_1)\}_i$$
$$\max(\{\varphi_0, +, f_1\}_i, \{\psi_0, +, g_1\}_i) = \{\max(\varphi_0, \psi_0), +, \max(f_1, g_1)\}_i$$
$$\max(\{\varphi_0, *, f_1\}_i, \{\psi_0, *, g_1\}_i)$$
$$= \begin{cases} \{\max(\varphi_0, \psi_0), *, \max(f_1, g_1)\}_i & \\ & \text{if } \varphi_0{>}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\max(\varphi_0, \psi_0), *, \min(f_1, g_1)\}_i & \\ & \text{if } \varphi_0{<}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\varphi_0, *, f_1\}_i & \text{if } \varphi_0{>}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\psi_0, *, g_1\}_i & \text{if } \varphi_0{<}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\max(|\varphi_0|, |\psi_0|), *, \max(|f_1|, |g_1|)\}_i & \text{if } f_1{<}0 \vee g_1{<}0 \\ \bot & \text{otherwise} \end{cases}$$

Using this definition it is possible to construct bounding functions in CR form over sets of CRs. Consider for example

$$\min(\{1, \#, 1, +, 2, *, 1\}_i, \{1, \#, 2, +, 2, *, 2\}_i) = \{1, \#, 1, +, 2, *, 1\}_i$$
$$\max(\{1, \#, 1, +, 2, *, 1\}_i, \{1, \#, 2, +, 2, *, 2\}_i) = \{1, \#, 2, +, 2, *, 2\}_i$$

Thus, the sequence of values of the CR form $\{1, \#, 1, +, 2, *, 1\}_i$ provides a lower bound and the sequence of values of the CR form $\{1, \#, 2, +, 2, *, 2\}_i$ provides an upper bound on the two CRs. This accurately captures the following behavior of the variable k (SSA variable k_1) in the loop:

```
loop
  $j_1 = \phi(1, j_2)$        // $j_1 = \{1, +, 2\}$
  $k_1 = \phi(1, k_4)$        // $\{1, \#, 1, +, 2, *, 1\} \leq k_1 \leq \{1, \#, 2, +, 2, *, 2\}$
    if (...)
      $k_2 = j_1$             // update represented by $\{1, \#, 1, +, 2\}$
    else
      $k_3 = 2 * k_1$         // update represented by $\{1, *, 2\}$
    $j_2 = j_1 + 2$           // update represented by $\{1, +, 2\}$
    $k_4 = \phi(k_2, k_3)$    // merge and align $\{1, \#, 1, +, 2\}$ and $\{1, *, 2\}$
endloop
```

4 Flow-Sensitive Loop-Variant Variable Classification

This section presents an algorithm to classify flow-sensitive loop-variant variables in linear time based on CR forms. The algorithm has three parts: COLLECT-RECURRENCES, CR-CONSTRUCTION and CR-ALIGNMENT-AND-BOUNDS. These routines are described first, followed by an analysis of complexity and accuracy.

4.1 Algorithms

4.1.1 Collect Recurrence Relations

The first phase of the algorithm is performed by COLLECT-RECURRENCES shown in Figure 4. The routine computes the set of recurrence relations for a variable v defined in an assignment S and this is repeated for each variable of a loop header ϕ-node. The algorithm visits each node in each SCCs to compute sets of recurrence relations of loop-variant variables. The sets are cached at the nodes for retrieval when revisited via a cycle, which ensures that nodes and edges are visited only once.

The process is illustrated with an example code in SSA form and corresponding SCC shown in Figures 5(a) and (b). The loop exhibits conditional updates of variable j. Starting from the loop header ϕ-node j_1, the algorithm follows the SSA edges recursively to collect the recurrence relations for each SSA variable in the SCC. The ϕ function for j_1 merges the initial value `0` outside the loop and the update j_7 inside the loop. Since conditional ϕ-node j_7 merges two arguments j_5 and j_6, to collect the recurrence sequence for j_7, the recurrence sequences for j_5 and j_6 must be collected first, which means j_7 depends on j_5 and j_6. Thus, j_5 was checked first for j_7 and j_4 was reached by following the SSA edges from j_5. The search continues until the starting loop header ϕ-node j_1 is reached. The symbol j_1 was returned and the recursive calling stops. Therefore, the recurrence sequence for j_2 can be obtained based on j_1, which is $j_1 + 1$. Similarly, based on this dependence chain, the recurrences propagated for each SSA variable are shown in Figure 5(c).

Note that due to control flow variable j_4 has two recurrences. Consequently, all variables that depend on j_4 have at least two recurrences. However, as the recurrences are propagated they degenerate into lower and upper sequences to limit the algorithmic complexity. Finally, the recurrence pair for loop header ϕ-node j_1 is constructed with initial value 0 and bounding recurrence sequences $j_1 + 4$ and $j_1 + 6$.

To compute the recurrences for variables in a multi-dimensional loop, the algorithm starts with the analysis of the inner loop. More details with examples of multiple-dimensional loops are discussed in later section.

4.1.2 Constructing CR Forms for Recurrences Relations

Algorithm CR-CONSTRUCTION(p) shown in Figure 6 converts recurrence relations of a variable into CR form (the last step of the example shown in Figure 5(c)), where p denotes a recurrences sequence pair with initial value v_0 of variable v and recurrence sequence S. If variable v does not appear in recurrence sequence S, then v is a conditionally reinitialized variable or wrap around variable of any order.

To illustrate this process, consider a classic form of a wrap-around variable shown in Figure 2.

```
**Algorithm** COLLECT-RECURRENCES($v, S$)
 - **input:**   program in SSA form, SSA variable $v$, and assignment $S$ of the form $var = expr$
 - **output:** recurrence sequence pair or recurrence sequence list
if $expr$ is a variable $x$ then
  $rec$ := CHECK($v$,$x$) and store the pair ($var$, $rec$)
  Return $rec$
else if $expr$ is of the form $x \odot y$ then
  $rec$ := CHECK($v$,$x$) $\odot$ CHECK($v$,$y$) and store the pair ($var$, $rec$)
  Return $rec$
else if $expr$ is a loop header node $\phi(x,y)$ ($x$ is defined outside the current loop and $y$ is defined inside the current loop) then
  $I$ := CHECK($var$,$x$) and $Seq$ := CHECK($var$,$y$)
  Construct pair $p := \langle var, (I, Seq)\rangle$
  Return $p$
else if $expr$ is a conditional node $\phi(b_1, \cdots, b_n)$ then
  Check each branch of conditional $\phi$ node: $B_1 := Check(v, b_1), \cdots, B_n := Check(v, b_n)$
  Construct sequence list $Seq := (B_1, \cdots, B_n)$
  Compute bound on $Seq$
  if the length of the $Seq$ list $> N_{\mathrm{thresh}}$ then Return $\perp$
  Store the pair ($var$, $Seq$)
  Return $Seq$
else
  Return $\perp$
endif
**Algorithm** CHECK($v, x$)
 - **input:**   loop header $\phi$-node variable $v$ and operand $x$
 - **output:** recurrence sequence expression list
if $x$ is loop invariant or constant then
  Return $x$
else if $x$ is an SSA variable then
  if $x$ is $v$ then Return $x$
  else if $x$ has a CR form or recurrence $\Phi$ stored then
    if $\Phi$'s index variable loop level is deeper than current loop level then
      Apply the $CR\#^{-1}$ rules to convert $\Phi$ to closed form $f(I)$
      Replace $I$'s in $f(I)$ with trip counts of index variables of the loop
      Return $f$
    else
      Return $\Phi$
    endif
  else if the loop depth where $x$ located is lower than the loop depth where $v$ located then
    Return $x$
  else
    Return COLLECT-RECURRENCES($v$, the statement $S$ that defines $x$)
  endif
endif
```

Figure 4: Collecting the Recurrence Relations from the SCCs of an SSA Loop Region

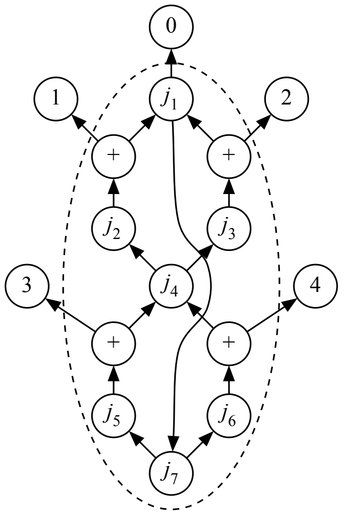
The CR forms are derived as follows, where j_1 is a first-order wrap-around variable:

$$\begin{aligned} i_1 : &\quad \langle i_1, (0, i_1 + 1)\rangle \Rightarrow \{0, +, 1\} \\ j_1 : &\quad \langle j_1, (99, i_1)\rangle \Rightarrow \{99, \#, 0, +, 1\} \\ j_1 + 1 &= \{99, \#, 0, +, 1\} + 1 = \{100, \#, 1, +, 1\} \end{aligned}$$

Now CR-CONSTRUCTION takes the pair $\langle i_1, (0, i_1 + 1)\rangle$ for variable i_1 as the input. The CR form for i_1 is computed with rule **(1)** of the algorithm. Similarly, the CR form for j_1 is computed based on rule **(5)** of the algorithm. The application of the CR# algebra enables efficient manipulation and simplification of expressions with wrap-around variables, such as the analysis of array subscript $j_1 + 1$ in Figure 2.

```
loop
  $j_1 = \phi(0, j_7)$
  if (...)
    $j_2 = j_1 + 1$
  else
    $j_3 = j_1 + 2$
  $j_4 = \phi(j_2, j_3)$
  if (...)
    $j_5 = j_4 + 3$
  else
    $j_6 = j_4 + 4$
  $j_7 = \phi(j_5, j_6)$
  ...
endloop
```

(a) *SSA form*



(b) *SCC from SSA*

var		stored recurrence
j_2	$=$	$j_1 + 1$
j_3	$=$	$j_1 + 2$
j_4	$=$	$\phi(j_2, j_3)$
	$\Rightarrow$	$[j_1 + 1, j_1 + 2]$
j_5	$=$	$j_4 + 3$
	$\Rightarrow$	$[j_1 + 4, j_1 + 5]$
j_6	$=$	$j_4 + 4$
	$\Rightarrow$	$[j_1 + 5, j_1 + 6]$
j_7	$=$	$\phi(j_5, j_6)$
	$\Rightarrow$	$[j_1 + 4, j_1 + 5, j_1 + 6]$
	$\Rightarrow$	$[j_1 + 4, j_1 + 6]$
j_1	$=$	$\phi(0, j_7)$
	$\Rightarrow$	$\phi(0, [j_1 + 4, j_1 + 6])$
	$\Rightarrow$	$\{0, +, 4\}, \{0, +, 6\}$

(c) *CR form derivation*

Figure 5: Analysis of SSA ϕ-Node Join Points

4.1.3 CR Alignment and Bounds

To handle conditionally updated variables in a loop nest, we introduce an algorithm for CR alignment and bounds computation. The key idea is that two or more CR forms of different lengths or with different operations can be aligned to enable pair-wise coefficient comparisons to efficiently construct bounding functions on the combined sequences. The CR-based bounds are important to determine the iteration-specific bounds on sequences as illustrated in Figures 1(c) and (d).

Algorithm CR-Alignment-and-Bounds shown in Figure 6 aligns multiple CRs and computes bounding functions, which are two CR forms that represent lower- and upper-bound sequences.

Consider an example variable j_1 which has three different recurrences due to control flow. The input recurrence list pair for the algorithm CR-Alignment-and-Bounds is:

$$pl = \langle j_1,\ (1,\ j_1 + 3\ \rightarrow\ 2 * j_1 + 1\ \rightarrow\ 2 * j_1\ )\rangle$$

Algorithm CR-Construction computes CR forms for each recurrence in this list. We have three different CR forms:

$$\begin{aligned} cr_1 &= \{1, +, 3\} = \{1, +, 3, *, 1\} \\ cr_2 &= \{1, +, 2, *, 2\} = \{1, +, 2, *, 2\} \\ cr_3 &= \{1, *, 2\} = \{1, +, 1, *, 2\} \end{aligned}$$

where cr_1, cr_2, and cr_3 are computed with rules **(1)**, **(3)** and **(2)** in CR-Construction, respectively. CR form cr_1 is aligned using Lemma 3 and cr_3 is aligned using Lemma 1. The minimal and maximum bound of these CR forms is obtained with Definition 3 as follows:

$$\min(\{1, +, 3, *, 1\}, \{1, +, 2, *, 2\}, \{1, +, 1, *, 2\}) = \{1, +, 1, *, 1\} \overset{CR\#^{-1}}{\Rightarrow} I + 1$$

$$\max(\{1, +, 3, *, 1\}, \{1, +, 2, *, 2\}, \{1, +, 1, *, 2\}) = \{1, +, 3, *, 2\} \overset{CR\#^{-1}}{\Rightarrow} 3 * 2^I - 2$$

Therefore, we have the bounds $I + 1 \leq j_1 \leq 3 * 2^I - 2$ for iteration $I = 0, \ldots, n$.

```
Algorithm CR-Alignment-and-Bounds($pl$)
 - input:  recurrences sequence list pair $pl = \langle v, (I, Seq)\rangle$
 - output: CR Bounds solution
if length of the $Seq$ list $n > N_{\mathrm{thresh}}$ then
   Return $\perp$
$cr$ := CR-Construction($\langle v$, ($I$, first recurrence in $Seq$ list)$\rangle$)
for each remaining recurrence $e$ in $Seq$
   Construct pair $p := \langle v, (I, e)\rangle$
   $cr_1$ := CR-Construction($p$)
   Align $cr$ with $cr_1$
   if CR alignment succeeds then
      Compute the bounds of $cr$ and $cr_1$ to $cr$
   else
      Return $\perp$
   endif
enddo
Store ($v$, $cr$) and Return $cr$

Algorithm CR-Construction($p$)
 - input:  recurrences sequence pair $p = \langle v, (v_0, S)\rangle$, where $v_0$ is initial value
           of variable $v$ and $S$ is the recurrence sequence for $v$
 - output: CR Solution
(1) if $S$ is of the form $v + \Psi$ ($\Psi$ can be CR or constant) then
        $\Phi := \{v_0, +, \Psi\}_{loop}$, where $loop$ is the innermost loop $v$ located
(2) else if $S$ is of the form $v * \Psi$ ($\Psi$ can be CR or constant) then
        $\Phi := \{v_0, *, \Psi\}_{loop}$
(3) else if $S$ is of the form $c * v + \Psi$, where $c$ is constant or a singleton CR form and
          $\Psi$ is a constant or a polynomial CR form then
        $\Phi := \{\varphi_0, +, \varphi_1, +, \cdots, +, \varphi_{k+1}, *, \varphi_{k+2}\}_{loop}$, where
        $\varphi_0 = v_0$;   $\varphi_j = (c-1) * \varphi_{j-1} + \psi_{j-1}$;   $\varphi_{k+2} = c$
(4) else if $S$ is variable $v$ then
        $\Phi := \{v_0\}_{loop}$
(5) else
        $\Phi := \{v_0, \#, S\}_{loop}$
    endif
```

Figure 6: Constructing CR Forms for Recurrence Relations

4.2 Examples

Consider two examples shown in Figure 7 and Figure 8. The loop nest shown in Figure 7(a) exhibits conditional updates of variable i and j. The recurrence system and its solution are shown in Figure 7(c) and (d). From the SSA form shown in Figure 7(b), an SCC for loop header ϕ-node i_2 can be found which contains two conditional ϕ-terms i_3 and i_4. Three different recurrences are obtained by traversing this SCC starting from loop header ϕ-term i_2. Based on three recurrences in the sequence list pair, the CR solutions for loop header ϕ-term i_2 are obtained. The CR alignment and min, max bounding functions are applied to the set of CR forms, resulting in the lower and upper dynamic value range bounds for variable i_2 which is shown in Figure 7(d). The same analysis procedure is applied for loop header ϕ-term j_1 and the CR bounds for j_1 is shown in Figure 7(d).

Consider the triangular loop nest shown in Figure 8. The algorithm starts with the analysis of the inner loop of the loop shown in Figure 8(a). From the SSA form shown in Figure 8(b), an SCC for loop header ϕ-term j_1 can be found and recurrence pair for j_1 is shown in Figure 8(c). The algorithm return variable k_1 as the initial value for the recurrence of loop header ϕ-term k_4 since k_1 is defined in the outer loop of $loop2$ where variable k_4 is defined. Thus, the CR forms for j_1 and k_4 shown in Figure 8(d) are obtained using the CR construction algorithm CR-Construction. The same analysis is applied for loop header ϕ-term i_1 when we analyze the outer loop.

```
i = 2
j = 3
for (...)
  if ... then
    i = 0
  else
    j = i
    if ... then
      continue
    endif
  endif
  i = i + 1
endfor
```

(a) *Loop Nest*

```
loop
  $j_1 = \phi(j_2, 3)$
  $i_2 = \phi(i_4, 2)$
  $i_1 = i_2 + 1$
  $i_3 = \phi(1, i_1)$
  ...
  $j_4 = \phi(j_1, i_2)$
  $i_5 = i_3$
  $j_2 = \phi(i_2, j_4)$
  $i_4 = \phi(i_2, i_5)$
  ...
endloop
```

(b) *SSA Form*

i_2:
$\langle i_2, (2, i_2) \rightarrow (2, 1) \rightarrow (2, i_2 + 1)\rangle$

j_1:
$\langle j_1, (3, j_1) \rightarrow (3, i_2)\rangle$

(c) *Recurrences list pairs*

i_2 :
min: $\{2, \#, 1\}$
max: $\{2, \#, 3, +, 1\}$

j_1 :
min: $\{3, \#, 2, \#, 1\}$
max: $\{3, \#, 3, \#, 3, +, 1\}$

(d) *CR Solutions*

Figure 7: Conditional and Wrap Around Recurrences

To analyze the outer loop header ϕ-term k_1, the algorithm Collect-Recurrences starts from k_1 and follows the SSA edge to the variable k_4. Since k_4 belongs to the inner loop, the CR$\#^{-1}$ rules are applied to convert the CR form of k_4 to closed form and then update the closed form with trip count of the inner loop. The total effect is to compute the aggregate value of the recurrence updates to k_4 in the inner loop. The recurrence form of the variable k_1 is shown in Figure 8(c) with the aggregate recurrence updates. After substituting i_1 in the recurrence pair with its CR form and simplify it, the recurrence form for k_1, which is $\langle k_1, (0, k_1 + \{3, +, 3\})\rangle$, is obtained. The CR form for k_1 shown in Figure 8(d) is obtained by CR construction algorithm.

4.3 Complexity

In the worst case there are 2^n cycles in the SCC for n number of ϕ-node join points, see Figure 9. Methods based on full path enumeration require 2^n traversals from j_1 to j_n. However, the presented algorithm is linear in the size of the SSA region of a loop nest as explained as follows.

The algorithms CollectRecurrences and Check perform a recursive depth-first traversal of the SSA graph to visit each node to collect recurrences. When the Collect-Recurrences algorithm visits a node in the SSA graph, the recurrence collected for this SSA variable is stored in a cache for later retrieval. Whenever this node is visited again via another data flow path, the cached recurrence forms are used. Thus, it is guaranteed that the algorithm visits each node and each edge in the SSA graph only once, which has the same complexity as Tarjan's algorithm [16].

For example, in Figure 5(c) each SSA node in the SCC cycle has recurrences stored and updated during the traversal of the SCC. Assume that the algorithm visits the leftmost successor of ϕ-nodes first. To obtain the recurrence for variable j_7, the edges from j_5 was followed first to collect the recurrence for node j_4 in depth-first manner. The recurrence stored for j_4 guarantee all the successor node of j_4 in the graph and the node j_4 itself will not be revisited via edge from j_6.

Note that each time a new set of recurrence pairs at a conditional ϕ-node is merged this potentially increases the recurrence set by a factor of two. However, the set is reduced immediately

```
k = 0
for (i=1 to n)
  for (j=1 to i)
    k = k + 3
  endfor
endfor
```

(a) *Loop Nest*

```
loop1
  $i_1 = \phi(i_2, 1)$
  $k_1 = \phi(k_3, 0)$
  $i_2 = i_1 + 1$
  loop2
    $j_1 = \phi(j_2, 1)$
    $k_4 = \phi(k_3, k_1)$
    $k_3 = k_4 + 3$
    $j_2 = j_1 + 1$
  endloop2
endloop1
```

(b) *SSA Form*

Recurrences pairs after inner loop analysis

$\langle j_1, (1, j_1 + 1)\rangle$

$\langle k_4, (k_1, k_4 + 3)\rangle$

Recurrences pairs after outer loop analysis

$\langle i_1, (1, i_1 + 1)\rangle$

$\langle k_1, (0, k_1 + 3 * i_1)\rangle$

(c) *Recurrences*

$i_1 : \{1, +, 1\}_{loop1}$

$k_1 : \{0, +, 3, +, 3\}_{loop1}$

$j_1 : \{1, +, 1\}_{loop2}$

$k_4 : \{k_1, +, 3\}_{loop2}$

(d) *CR Solutions*

Figure 8: Recurrences in Multiple Dimensional Loop

by eliminating duplicate recurrence relations and eliminating relations that are already bounded by other relations, see e.g. Figure 5. The size of the set of recurrence relations cannot exceed N_{thresh}, which is a predetermined constant threshold. A low threshold speeds up the algorithm but limits the accuracy.

Table 1 shows the statistical data on the size of recurrence relation list for conditional loop-variant variables collected by our algorithm from the SPEC2000 benchmarks. The first column in the table lists the benchmarks. Only those benchmarks which have at least one conditional loop-variant variable are listed in the table. The columns labeled “Maximum”, “Mean”, and “Standard Deviation” show the value of the maximum size, average size of the recurrence list and the standard deviation value for each benchmark.

From the result of Table 1 it can be concluded that the average size of the recurrence list for conditional updated variables ranges from 2.00 to 2.41 and the standard deviation value for each benchmark ranges from 0.0 to 0.61, which is very small. Clearly, $N_{\text{thresh}} = 10$ is sufficiently large to handle the SPEC2000 benchmarks accurately and given that the standard deviation is small we expect that this threshold value will be effective to handle a wide range of real-world application codes in general.

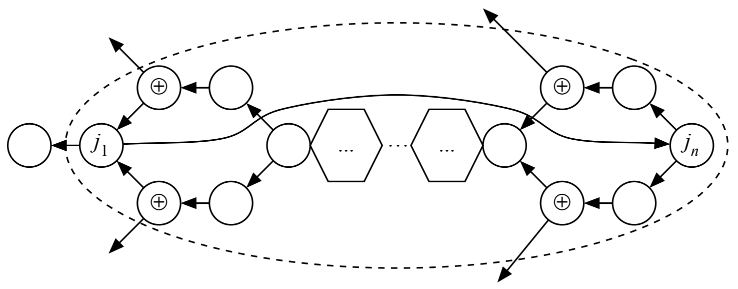


Figure 9: An SCC with 2^n Cycles Constructed from a Loop with n ϕ-Nodes

Table 1: The Size of Recurrence List for Conditional Variable in SPEC2000

Benchmark	Max	Mean	St.dev.
CINT2000			
`164.gzip`	3	2.41	0.49
`175.vpr`	3	2.08	0.27
`181.mcf`	3	2.22	0.33
`186.crafty`	3	2.05	0.21
`197.parser`	3	2.04	0.20
`254.gap`	4	2.26	0.49
`255.vortex`	3	2.08	0.28
`256.bzip2`	4	2.32	0.61
`300.twolf`	3	2.06	0.24
Average	3.22	2.17	0.35
CFP2000			
`173.applu`	2	2	0.0
`177.mesa`	2	2	0.0
`179.art`	2	2	0.0
`183.equake`	2	2	0.0
`187.facerec`	2	2	0.0
`188.ammp`	3	2.2	0.4
`189.lucas`	2	2	0.0
`200.sixtrack`	2	2	0.0
Average	2.13	2.025	0.04

Because the cost for analyzing an SSA node operation is constant and the cost of recurrence updates at nodes is bounded by N_{thresh}, the worst-case complexity is $\mathcal{O}(|SSA|)$, where $|SSA|$ denotes the size of the SSA region.

4.4 Accuracy

The algorithm recognizes IVs with closed forms accurately when IVs are not conditionally updated, thereby producing classifications that cover linear, polynomial, geometric, periodic, and mixer functions, similar to other nonlinear IV recognition algorithms [10, 12, 22]. For conditionally updated loop-variant variables that have no closed forms the algorithm produces bounds.

By comparison, in certain exceptional cases, the full path analysis algorithm [20] is more accurate in producing bounds than the linear time algorithm presented in this paper. This phenomenon occurs when variables are coupled or combined in induction expressions. In that case their original relationship may be lost, which results in looser bounds than full path analysis. However, the greatest disadvantage of the full path analysis method is its exponential execution time.

To illustrate the effect of coupling on the accuracy of the algorithms, an example comparison is shown in Figure 10 for a Quicksort partition loop. The full path search results are shown in Figure 10(b) and the linear-time results is in Figure 10(c). Full path analysis computes CR solution for variable i, j, and s in the example loop separately for two paths of the program. The

```
i = 0
j = n
do
  if (...)
     i = i + 1
  else
     j = j − 1
  s = j − i
...
while (s > 0)
```

(a) *Loop*

```
Path 1:
   i = {0, +, 1}
   j = n
   s = j - i = {n, +, −1}
Path 2:
   j = {n, +, −1}
   i = 0
   s = j - i = {n, +, −1}
Solution for iteration I:
   0 ≤ i ≤ I
   n−I ≤ j ≤ n
   s = {n, +, −1}
```

(b) *Full path search results*

Variable	*Min CR*	*Max CR*
i	$\{0\}$	$\{0,+,1\}$
j	$\{\mathsf{n},+,-1\}$	$\{\mathsf{n}\}$
s = j-i	$\{\mathsf{n},+,-2\}$	$\{\mathsf{n}\}$

(c) *Linear-time results*

Figure 10: Comparison of Full Path Search and Linear Time Algorithms

CR result $\{n,+,-1\}$ for variable $s = j - i$ is equal in two paths because on of the updates $i = i + 1$ and $j = j - 1$ is always taken. Instead of the single CR form for s, the CR solutions of the faster algorithm for variable s are bounded by $\{n,+,-2\}$ and $\{n\}$, which is less accurate than full path search.

5 Implementation and Experimental Results

The following classes of loop-variant variables are recognized and classified by the algorithm.

Linear induction variables are represented by nested CR forms $\{a,+,s\}_i$, where a is the integer-valued initial value and s is the integer-valued stride in the direction of i. The coefficient a can be a nested CR form in another loop dimension. Linear IVs are the most common IV category.

Polynomial induction variables are represented by nested CR forms of length k, where k is the order of the polynomial. All $\odot$ operations in the CR form are additions, i.e. $\odot = +$. For example, the variable CppObjectAddr and DbObjectAddr in Figure 11(a) are pointer IV with polynomial CR form $\{\mathsf{DbObjectAddr},+,0,+,\mathsf{AttrDbSize}\}$ and $\{\mathsf{CppObjectAddr},+,0,+,\mathsf{Attr01Size}\}$.

Geometric induction variables are represented by the CR form $\{a,*,r\}_i$, where a and r are loop invariant. For example, the variable n in Figure 11(b) are Geometric induction variable with CR form $\{1,*,10\}$.

Mix induction variables with CR forms that contain both $\odot = +$ and $*$. For example, the variable i and j in Figure 11(c) have CR form $\{0,+,1,*,2\}$ and $\{1,+,2,*,2\}$ respectively.

Out-of-sequence (OSV) variables are *re-initialized variables* and *wrap-around variables.* They are represented by (a set of) CR forms $\{a,\#,s\}_i$, where a is the initial out-of-sequence value and s is a nested CR form. In Figure 11(d), variable iside in the loop of `175.vpr` benchmark is bounded by the CR-form range $[\{-1,\#,+,0\},\{-1,\#,+,1\}]$ (iside is a re-initialized variable).

```
while (k++ < AttrCount) {
   CppObjectAddr = (addrtype )((char *)CppObjectAddr + Base01Offset);
   DbObjectAddr = (addrtype )((char *)DbObjectAddr + BaseDbOffset);
   ...
   Base01Offset += Attr01Size;
   BaseDbOffset += AttrDbSize;
}
```

(a) *Polynomial IV from* `255.vortex`

```
for (n=1; n<=...; n*=10 ) {
   ...
}
```

(b) *Geometric IV from* `254.gap`

```
j = 1;
for (i=0; i < j; ) {
   i = j;
   j = 2 * j + 1;
   largest_block = i;
}
```

(c) *Mixed IV from* `197.parser`

```
while (...) {
   iside = iside + 1;
   if (iside > 3) {
      pindex++;
      iside = 0;
   }
   ...
}
```

(d) *Re-initialized IV from* `175.vpr`

```
a = 1; b = 0;
while ( o != 0 ) {
   t = b;
   b = a - (k/o) * b;
   a = t;
   ...
}
```

(e) *Cyclic IV from* `254.gap`

```
offset = 0;
for (ipin=0;...;ipin++) {
   ...
   if (ldots) {
      times_listed[bnum] = 0;
      unique_pin_list[inet][offset] = bnum;
      offset++;
   }
}
```

(f) *Conditionally updated IV from* `175.vpr`

Figure 11: Example Loops from the SPEC2000 Benchmarks

Cyclic induction variables who have cyclic dependence between the recurrence relations of variables. For example, in Figure 11(e) variables a and b from cyclic IVs. In some cases cyclic IVs can be represented by geometric sequences [10, 12], but most cyclic forms represent special functions (e.g. the Fibonacci sequence is such an example). Some cyclic forms can be degenerated into monotonic sequences, by replacing a variable's update with an unknown [19].

Conditional induction variables are represented by the CR $\{[a, b], \odot, s\}$, where s is a nested bounded CR form and $\odot$ can be $+$, $*$, or $\#$. Variable offset in Figure 11(f) is bounded by the CR sequence range $[0, \{0, +, 1\}]$.

Unknown variables have unknown initial values or unknown update values. These unknown are typically function returns, updates with (unbounded) symbolic variables, or bit-operator recurrences. Some of these are identified as monotonic. For example, an IV with initial value 0 and a "random" positive stride function has a CR $\{0, +, \top\}$, where the stride is represented by the lattice value $\top$.

Table 2: Loop-variant Variable Classification in SPEC2000

Benchmark	Linear	Polyn'l	Geom.	OSV	Cyclic	Cond'l	Mix	Unknown
CINT2000								
`164.gzip`	59.45%	0.00%	0.00%	0.79%	0.00%	7.48%	0.00%	32.29%
`175.vpr`	59.47%	0.00%	0.21%	0.21%	0.00%	9.05%	0.00%	31.07%
`181.mcf`	38.18%	0.00%	0.00%	0.00%	0.00%	10.91%	0.00%	50.91%
`186.crafty`	47.91%	0.00%	0.00%	0.00%	0.00%	12.71%	0.00%	39.37%
`197.parser`	35.19%	0.00%	0.00%	0.51%	0.00%	5.22%	0.51%	58.58%
`254.gap`	62.73%	0.00%	2.52%	1.00%	0.33%	5.85%	0.38%	27.51%
`255.vortex`	66.06%	3.03%	0.61%	2.42%	0.00%	15.15%	0.00%	12.73%
`256.bzip2`	54.67%	0.00%	0.93%	0.00%	0.00%	12.15%	1.40%	30.84%
`300.twolf`	40.21%	0.00%	0.00%	0.00%	0.00%	5.35%	0.00%	54.45%
Average	51.54%	0.34%	0.47%	0.55%	0.04%	9.32%	0.25%	37.53%
CFP2000								
`168.wupwise`	80.20%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	19.80%
`171.swim`	96.30%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	3.70%
`172.mgrid`	84.06%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	15.94%
`173.applu`	94.77%	0.00%	0.00%	0.00%	0.00%	1.31%	0.00%	3.92%
`177.mesa`	79.57%	0.00%	0.30%	0.00%	0.00%	12.73%	0.00%	7.40%
`179.art`	73.12%	0.00%	0.00%	0.00%	0.00%	4.30%	0.00%	22.58%
`183.equake`	81.25%	0.00%	0.00%	2.08%	1.04%	3.12%	0.00%	13.54%
`187.facerec`	86.92%	0.00%	0.42%	0.00%	0.00%	2.53%	0.00%	10.13%
`188.ammp`	59.89%	0.00%	0.00%	2.54%	0.00%	3.95%	0.00%	33.62%
`189.lucas`	87.68%	0.00%	1.48%	0.00%	0.00%	1.97%	0.99%	7.88%
`200.sixtrack`	83.87%	0.00%	2.15%	2.15%	0.00%	1.08%	1.08%	9.68%
Average	82.51%	0.00%	0.40%	0.62%	0.09%	2.82%	0.19%	13.47%

Table 2 shows the experimental results of all induction variables categorized in SPEC2000[3] with our algorithm. The first column in the table names the benchmark. The columns labeled "Linear", "Polynomial", "Geometric", "OSV", "Cyclic", "Conditional", "Mix" and "Unknown" show the percentage of each loop-variant variable category as a percentage of the total number of loop-variant variables in each benchmark.

From the results of Table 2 the percentage of conditional induction variables ranges from 5.22% to 15.15% in CINT2000, with 9.32% on average. None of these are detected by GCC as well as other compilers, such as Open64 and Polaris [8]. The algorithm also identifies polynomial, geometric, mix, cyclic and wrap-around induction variables, which have been given significant attention in the past [10, 12, 14]. None of these are currently detected by the standard implementation of GCC.

Table 3 shows the sub-classification results of the conditional loop-variant variables category in SPEC2000. This sub-classification identifies the percentage of all conditionally updated variables

[3]Three CINT2000 and three CFP2000 benchmarks results are not listed because of GCC 4.1-specific compilation errors that are not related to our implementation.

Table 3: Classification for Conditional Loop-Variant Variable in SPEC2000

Benchmark	Linear	Polyn'l	Geom.	OSV	Mix	Unknown
CINT2000						
164.gzip	42.11%	0.00%	0.00%	5.26%	15.79%	36.84%
175.vpr	72.73%	0.00%	0.00%	9.09%	0.00%	18.18%
181.mcf	66.67%	0.00%	0.00%	0.00%	16.67%	16.67%
186.crafty	32.81%	0.00%	0.00%	10.94%	7.81%	48.44%
197.parser	70.97%	0.00%	0.00%	19.35%	0.00%	9.68%
254.gap	45.53%	0.00%	0.00%	20.33%	6.50%	27.64%
255.vortex	60.00%	4.00%	0.00%	24.00%	0.00%	12.00%
256.bzip2	61.54%	0.00%	0.00%	23.08%	7.69%	7.69%
300.twolf	51.61%	0.00%	0.00%	40.32%	0.00%	8.06%
Average	55.99%	0.44%	0.00%	16.93%	6.05%	20.58%
CFP2000						
173.applu	100%	0.00%	0.00%	0.00%	0.00%	0.00%
177.mesa	63.26%	0.00%	0.00%	1.40%	0.47%	34.88%
179.art	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%
183.equake	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%
187.facerec	83.33%	0.00%	0.00%	0.00%	0.00%	16.67%
188.ammp	28.57%	0.00%	7.14%	14.29%	0.00%	50.00%
189.lucas	75.00%	0.00%	0.00%	0.00%	25.00%	0.00%
200.sixtrack	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Average	68.77%	0.00%	0.89%	14.46%	3.18%	12.69%

(taken as 100%), which are "Linear", "Polynomial", "Geometric", "OSV", "Mix" and "Unknown". Only those benchmarks which have at least one conditional loop-variant variable are listed in the table. For example, from the 7.48% of conditionally updated variables in `164.gzip`, 42.11% is subclassified as linear. Thus, 3.15% of the variables are conditionally updated variables with linear closed forms.

From the results of Table 3, linear conditional variables are the most common category. The average percentage of linear conditional variable takes 55.99% and 68.77% in CINT2000 and CFP2000 separately. Note that these closed forms are not detected by other compilers. We also identify all re-initialized variables, the "OSV" category. The "Mix" conditional variable has an average percentage of 6.05% and 3.18% in CINT2000 and CFP2000, and was separately identified with our implementation from OSV using the bounding technique. This result shows that CR alignment is applied frequently for bounding the CR forms.

To evaluate the execution time performance of our CR implementation in GCC, we measured the compilation time of CR construction for the SPEC2000 benchmarks. CR construction accounts for 1.75% percent of the compilation time of GCC in average. The additional time is less than one second for most benchmarks. This shows that the performance of our algorithm is quite good.

6 Conclusion

This paper presented a linear-time loop-variant variable analysis algorithm that effectively analyzes flow-sensitive variables that are conditionally updated. We believe that the strength of our algorithm lies in its ability to analyze nonlinear and non-closed index expressions in the loop nests with higher accuracy than pure monotonic analysis. This benefits many compiler optimizations, such as loop restructuring and loop parallelizing transformations that require accurate data dependence analysis.

The experimental results of our algorithm applied to the SPEC2000 benchmarks shows that a high percentage of flow-sensitive variables are detected and accurately analyzed requiring only a small fraction of the total compilation time (1.75%). The result is a more comprehensive classifications of variables, including additional linear, polynomial, geometric, and wrap-around variables when these are conditionally updated.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, Reading MA, 1985.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

[3] D. Andrade, M. Arenaz, B. Fraguela, J. T. no, and R. Doallo. Automated and accurate cache behavior analysis for codes with irregular access patterns. In *Concurrency and Computation: Practice and Experience (to appear)*, 2007.

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

[5] O. Bachmann, P. Wang, and E. Zima. 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)*, pages 242–249, Oxford, 1994. ACM.

[6] D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for GCC. In *Proceedings of the 2004 GCC Developers' Summit*, pages 37–54, 2004.

[7] J. Birch, R. van Engelen, K. Gallivan, and Y. Shou. An empirical evaluation of chains of recurrences for array dependence testing. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 295–304, New York, NY, USA, 2006. ACM Press.

[8] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced program restructuring for high-performance computers with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[9] B. Franke and M. O'Boyle. Array recovery and high-level transformations for dsp applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(2):132–162, 2003.

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

[11] R. Gupta. A fresh look at optimizing array bound checking. *SIGPLAN Not.*, 25(6):272–282, 1990.

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

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

[14] W. Pottenger and R. Eigenmann. Parallelization in the presence of generalized induction and reduction variables. Technical report, 1396, Univ. of Illinois at Urbana Champaign, Center for Supercomputing Research & Development, 1995.

[15] Y. Shou, R. van Engelen, J. Birch, and K. Gallivan. Toward efficient flow-sensitive induction variable analysis and dependence testing for loop optimization. In *proceedings of the ACM SouthEast Conference*, pages 1–6, 2006.

[16] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[17] R. van Engelen. Symbolic evaluation of chains of recurrences for loop optimization. Technical report, TR-000102, Computer Science Dept., Florida State University, 2000.

[18] R. van Engelen. Efficient symbolic analysis for optimizing compilers. In *proceedings of the ETAPS Conference on Compiler Construction 2001, LNCS 2027*, pages 118–132, 2001.

[19] R. van Engelen. The CR# algebra and its application in loop analysis and optimization. Technical report, TR-041223, Computer Science Dept., Florida State University, 2004.

[20] R. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 106–115, 2004.

[21] R. van Engelen and K. Gallivan. 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*, pages 80–89, Maui, Hawaii, 2001.

[22] M. Wolfe. Beyond induction variables. In *ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation*, pages 162–174, San Fransisco, CA, 1992.

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

[24] P. Wu, A. Cohen, J. Hoeflinger, and D. Padua. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 78–91, 2001.

[25] E. Zima. Recurrent relations and speed-up of computations using computer algebra systems. In *proceedings of DISCO'92*, pages 152–161. LNCS 721, 1992.

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