Flow-Sensitive Loop-Variant Variable Classification in Linear Time

Yixin Shou[1], Robert van Engelen[1⋆], and Johnnie Birch[2]

[1] Florida State University, Tallahassee FL 32306
{shou,engelen}@cs.fsu.edu
[2] 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 more complicated 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 and Related Work

Induction variables (IVs) [1, 9, 11–13, 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 [8, 21]. Other example applications are array bounds check elimination [10], loop-level cache reuse analysis [3], software prefetching [2], loop blocking, variable privatization, IV elimination [1, 9, 11, 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.

⋆ Supported in part by NSF grant CCF-0702435.

Closer inspection of these benchmarks reveals that value progressions of *all* of these flow-sensitive variables can be bounded with tight bounding functions over the iteration space. Typically a pair of linear lower- and upper-bound functions on variables that have conditional increments suffices. Bounding the value progressions of these variables has the advantage of increased 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 (iteration-specific) bounds on variables, analysis and optimization can continue. For example, in [6, 20] it was shown that dependence analysis can be easily 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.

Automatic classification of flow-sensitive variables poses two challenges: 1) to find accurate bounds 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. And 2) to find a polynomial time algorithm with sufficient accuracy to classify and bound these variables.

A search method that uses full path enumeration to collect coupled variable update operations in a loop body may require 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.

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[3] with a classification of their fundamentally different characteristics.

The method by Gerlek, Stoltz and Wolfe [9] 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. The method suggests a *sequence strengthening*

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

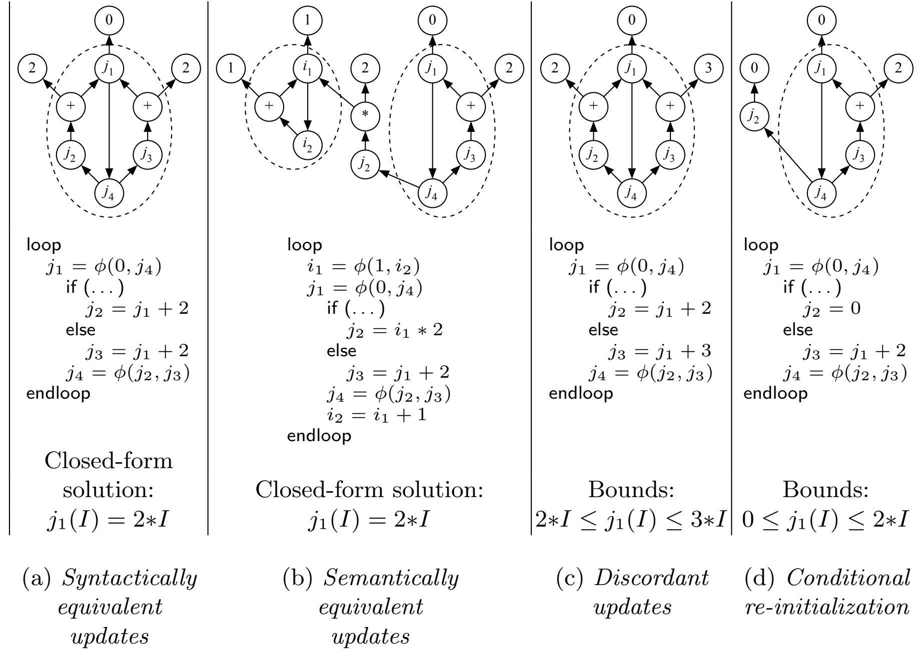


Fig. 1. Loops with Flow-Sensitive Loop-Variant Variable Updates

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 [11] 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 [5, 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.

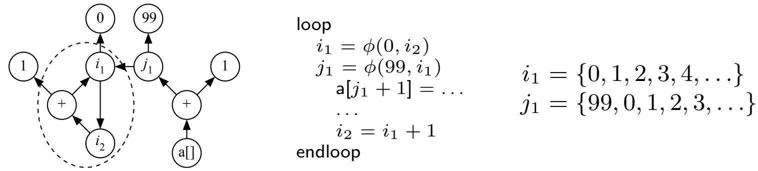


Fig. 2. SCC of the SSA Form of an Example Loop with a Wrap-around Variable

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 execution time as a consequence of full 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, which take a known sequence but have exceptional (re)start values. 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 [11]. A wrap-around variable is flow-sensitive: 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. These variables may cascade: any IV that depends on the value of a wrap-around variable is a wrap-around variable of one order higher [9] (two iterations with out-of-sequence values).

This paper presents a linear-time flow-sensitive loop-variant variable analysis algorithm based on the method by Gerlek et al. [9] and the CR# (CR-sharp) algebra [19]. This approach enables the analysis of coupled loop-variant variables in multiple SCCs Figure 1(a-b) (both formed by conditional and unconditional flow) and is essential to construct lower- and upper-bounding functions for flow-sensitive variables Figure 1(c-d).

The contributions of this paper can be summarized as follows:

- 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 that are updated in multiple branches of the loop body, with a running time that scales linearly with the size of the SSA region of a loop nest.
- An implementation in GCC 4.1 of the classifier.

The remainder of this paper is organized as follows. Section 2 gives CR# algebra preliminaries. Section 3 presents the linear time, flow-sensitive IV classification algorithm based on the CR# algebra. In Section 4 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 5 summarizes the conclusions.

2 Preliminaries

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 the next iteration by operator $\odot_1$ (either $+$ or $*$) and stride value f_1. When f_1 is a non-constant function in CR form this gives 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

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

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	...
$\{1, *, 2, +, 1\}_i$ value sequence $=$	1	2	6	24	120	720	...

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[4]:

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

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].

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 #-operator of the CR# algebra has the following semantics.

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

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

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

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.

To analyze conditionally updated variables in a loop, 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, $\{1, +, 1, *, 1\}$ 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 $\{1, +, 2, +, 1\}_i$.

A set of Lemmas that provide concept and proof of a simple algorithm for alignment of CR forms can be found in a technical report [14]. After alignment, the minimum and the maximum bounding CRs of two arbitrary CR forms is inductively defined, see also [14].

Consider two example CR forms, $\Phi_i = \{1, \#, 1, +, 2\}_i$ represents a wrap-around variable and $\Psi_i = \{1, *, 2\}_i$ is geometric. First, Φ_i and Ψ_i are aligned:

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

Then both sequences are bounded by the min and max sequences:

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

3 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.

3.1 Algorithms

1. Collect Recurrence Relations The first phase of the algorithm is performed by Collect-Recurrences shown in Figure 3. 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.

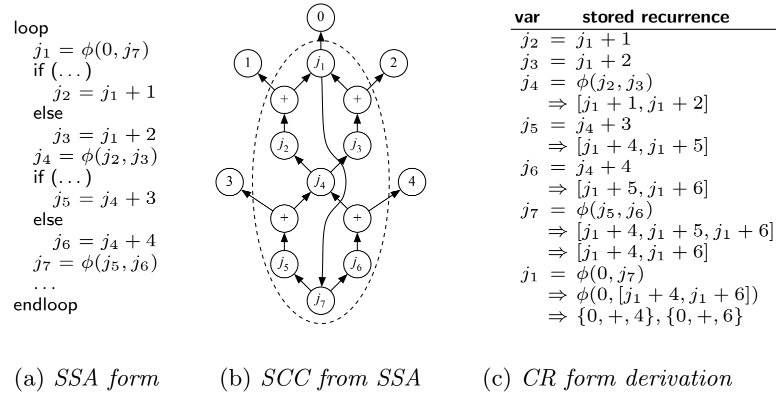
```
Algorithm COLLECT-RECURRENCES(v, S)
 - input:   program in SSA form, SSA variable v, and assignment S of the form var = expr
 - output: recurrence sequence pair or recurrence sequence list
if expr is of the form x then rec := CHECK(v,x), store (var, rec) and Return rec
else if expr is of the form of x ⊙ y then
      rec := CHECK(v,x) ⊙ CHECK(v,y), store (var, rec) and Return rec
else if expr is a loop header node φ(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 := ⟨var, (I, Seq)⟩
   Return p
else if expr is a conditional node φ(b_1, ···, b_n) then
   Check each branch of conditional φ node:
   B_1 := Check(v, b_1), ···,B_n := Check(v, b_n)
   Construct sequence list Seq := (B_1, ···, B_n), Compute bound on the Seq
   if the length of the Seq list > N_thresh then Return ⊥
   Store (var, Seq) and Return Seq
else Return ⊥
endif

Algorithm CHECK(v, x)
 - input:   loop header φ-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 Φ stored then
      if Φ's index variable loop level is deeper than current loop level then
         Apply the CR#^-1 rules to convert Φ to closed form f(I)
         Replace I's in f(I) with trip counts of index variables of the loop
         Return f
      else Return Φ
      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
```

Fig. 3. Collecting the Recurrence Relations from the SCCs of an SSA Loop Region

The process is illustrated with an example code in SSA form and corresponding SCC shown in Figures 4(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 4(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



(a) *SSA form* (b) *SCC from SSA* (c) *CR form derivation*

Fig. 4. Analysis of SSA ϕ-Node Join Points

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 can be found in a technical report [14].

2. Constructing CR Forms for Recurrences Relations Algorithm CR-Construction(p) shown in Figure 5 converts recurrence relations of a variable into CR form (the last step of the example shown in Figure 4(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. The CR forms are derived as follows, where j_1 is a first-order wrap-around variable:

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

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.

3. CR Alignment and Bounds To handle conditionally updated variables in a loop nest, we introduce an algorithm for CR alignment and bounds com-

```
Algorithm CR-Alignment-and-Bounds(pl)
 - input:   recurrences sequence list pair pl = ⟨v, (I, Seq)⟩
 - output: CR Bounds solution
if length of the Seq list n > N_thresh then Return ⊥
cr := CR-Construction(⟨v, (I, first recurrence in Seq list)⟩)
for each remaining recurrence e in Seq
   Construct pair p := ⟨v, (I, e)⟩
   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 ⊥
   endif
enddo
Store (v, cr) and Return cr

Algorithm CR-Construction(p)
 - input:   recurrences sequence pair p = ⟨v, (v_0, S)⟩, 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 + Ψ (Ψ can be CR or constant) then
        Φ := {v_0, +, Ψ}_loop, where loop is the innermost loop v located
(2) else if S is of the form v ∗ Ψ (Ψ can be CR or constant) then
        Φ := {v_0, ∗, Ψ}_loop
(3) else if S is of the form c ∗ v + Ψ, where c is constant or a singleton CR form and
          Ψ is a constant or a polynomial CR form then
        Φ := {φ_0, +, φ_1, +, ···, +, φ_{k+1}, ∗, φ_{k+2}}_loop, where
        φ_0 = v_0;   φ_j = (c − 1) ∗ φ_{j−1} + ψ_{j−1};   φ_{k+2} = c
(4) else if S is variable v then
        Φ := {v_0}_loop
(5) else
        Φ := {v_0, #, S}_loop
    endif
```

Fig. 5. Constructing CR Forms for Recurrence Relations

putation. 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 5 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 of [14]

and cr_3 is aligned using Lemma 1 of [14]. The minimal and maximum bound of these CR forms is obtained with Definition 3 in [14] as follows:

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

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

3.2 Complexity

In the worst case there are 2^n cycles in the SCC for n number of ϕ-node join points, see Figure 6. 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 4(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 get 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 by eliminating duplicate recurrence relations and eliminating relations that are already bounded by other relations, see e.g. Figure 4. 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. Since the average size of the recurrence list of the benchmark

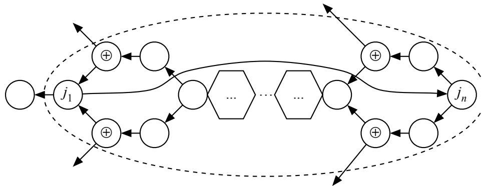


Fig. 6. An SCC with 2^n Cycles Constructed from a Loop with n ϕ-Nodes

```
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:
  $0 \le i \le I$
  $n - I \le j \le n$
  s = $\{n, +, -1\}$

(b) *Full path search results*

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

(c) *Linear-time results*

Fig. 7. Comparison of Full Path Search and Linear Time Algorithms

in CINT2000 ranges from 2.04 to 2.32, we found that $N_{\text{thresh}} = 10$ is sufficiently large to handle the SPEC2000 benchmarks accurately.

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.

3.3 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 [9, 11, 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 7 for a Quicksort partition loop. The full path search results are shown in Figure 7(b) and the linear-time results is in Figure 7(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 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.

4 Implementation and Experimental Results

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

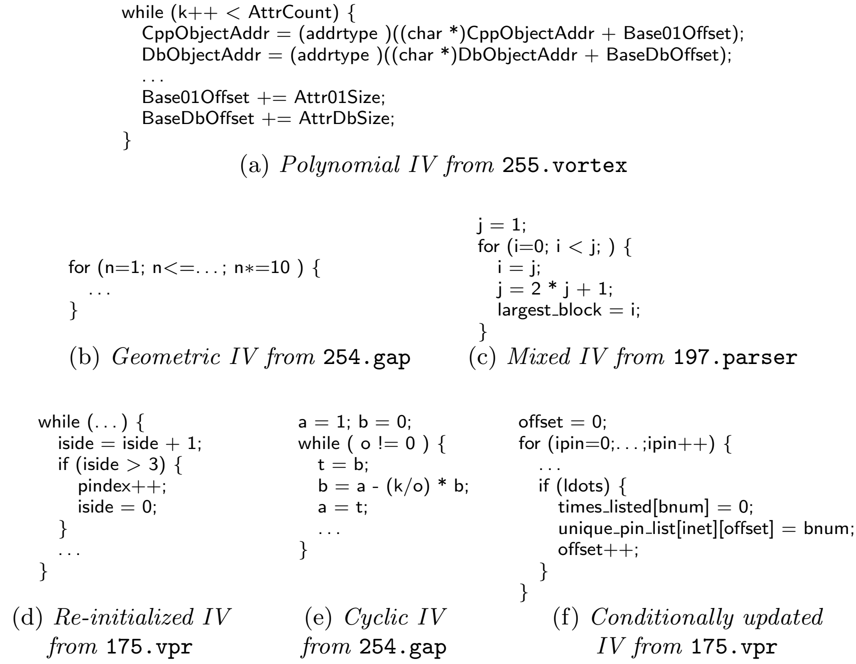


Fig. 8. Example Loops from the SPEC2000 Benchmarks

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 8(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 8(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 8(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 8(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).

Cyclic induction variables who have cyclic dependence between the recurrence relations of variables. For example, in Figure 8(e) variables a and b from

Table 1. 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%

cyclic IVs. In some cases cyclic IVs can be represented by geometric sequences [9, 11], 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 8(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 1 shows the experimental results of all induction variables categorized in SPEC2000[5] 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 1 the percentage of conditional induction variables ranges from 5.22% to 15.15% in CINT2000, with 9.32% on average. None of these

[5] 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.

are detected by GCC as well as other compilers, such as Open64 and Polaris [7] (Polaris uses advanced nonlinear IV recognition algorithms [13]). Our algorithm also identifies all polynomial, geometric, mix, cyclic and wrap-around induction variables. None of these are currently detected by GCC implementations.

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.

5 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. 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.
6. 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.
7. 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.

8. 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.
9. 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.
10. R. Gupta. A fresh look at optimizing array bound checking. *SIGPLAN Not.*, 25(6):272–282, 1990.
11. 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.
12. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Fransisco, CA, 1997.
13. 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.
14. Y. Shou, R. van Engelen, and J. Birch. Flow-sensitive loop-variant variable classification in linear time. Technical report, TR-071005, Computer Science Dept., Florida State University, 2007.
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.