Value Range Analysis of Conditionally Updated Variables and Pointers*

Johnnie Birch Robert van Engelen Kyle Gallivan
Department of Computer Science and
School of Computational Science and Information Technology
Florida State University

Abstract

Pointer-based array traversals are often used by C programmers to optimize sequential memory access. However, the use of pointer arithmetic prevents compilers from effectively applying dependence testing and loop analysis for optimization and parallelization. Because pointer variables can be viewed as reference-type induction variables, conventional induction variable recognition and substitution methods can be used to translate pointer-based code to array-based code, thereby enabling the application of traditional loop restructuring compiler optimizations for parallelization. However, conditionally updated induction variables and pointers are prohibiting the application of these optimizations, due to the inability to determine closed-form function equivalents. This paper presents a method for value range analysis on conditionally updated induction variables and pointers which have no closed-form functions. The value range method determines the memory access patterns of pointers and arrays indexed with conditionally updated induction variables. Important applications of this method are in array bounds analysis and data dependence testing.

1 Introduction

Accurate value range analysis is crucial for restructuring and parallelizing compilers. Several compiler analysis methods rely on value range analysis, such as array-based data dependence testing, array and loop bounds checking, pointer analysis, feasible path analysis, and loop timing analysis. Current value range analysis methods are powerful and are capable of handling symbolic and nonlinear expressions. These methods extensively use symbolic analysis techniques, such as abstract interpretation and symbolic differencing, to determine the closed-form functions of induction variables for range analysis. However, recent work by Psarris [20, 21, 22], Shen, Li, and Yew [23], and Haghighat [17] mention the difficulty compilers have in analyzing expressions containing conditionally updated variables, whose recurrence relations have no closed form. Consider for example Figure 1(a) depicting an example loop with conditionally updated induction variables. Because conditionally updated variables in loop nests do not have closed forms, current value range methods are ineffective dealing with loops containing conditional variable updates.

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

```
int i, j = 0, k = 0;
for (i = 0; i < n; i++) {
  ...
  A[f(i,j,k)] = A[g(i,j,k)];
  ...
  if (C[i])
    j = j + k;
  else
    k = k + 1;
  ...
}
```

(a) *Conditionally Updated Variables*

```
int i, *p = A, *q = B;
for (i = 0; i < n; i++) {
  ...
  *p += *q++;
  ...
  if (C[i])
    p += 2;
  ...
}
```

(b) *Conditionally Updated Pointer*

Figure 1: Example Loops with Conditionally Updated Variables and Pointers

In addition, current value range analyzers are mainly developed to analyze the range of values of numerical expressions. However, because pointers are frequently used in C code to step through arrays, there is a need to effectively analyze the range of memory locations accessed by pointers, e.g. for data dependence analysis and array bounds checking. Pointer-to-array conversion, also known as array recovery, can be used to determine the closed forms of induction pointers [16, 27], which in turn can be used by a value range analyzer to determine the array bounds. However, none of these algorithms handle conditionally updated pointers, see Figure 1(b) for an example.

In this paper we present an algorithm for value range analysis that incorporates a novel method for the analysis of affine and nonlinear (index) expressions, generalized induction variables, and recurrence relations of conditionally updated variables and pointers in loops. To this end, the method derives bounding functions for conditionally updated variables for accurate value range analysis. The results of this analysis can be used to enhance the effectiveness of other analysis methods. For example, data dependence analysis applied to array A shown in Figure 1(a) may succeed with this value range information, where f and g are affine or nonlinear index functions. Furthermore, the presented method handles conditional pointer updates to determine the range of memory locations accessed by arrays as well as by pointers.

2 Related Work

Range analysis techniques play an integral role in symbolic analyzers. In [8, 10] Blume and Eigenmann describe techniques for computing variable ranges and propagating those computed values based on control flow information. In [6, 7] they propose a technique for dependence testing based on value range analysis. In [17] Haghighat and Polychronopoulos describe an engine for the evaluation of polynomial expressions, and propose to use these properties in dependence testing and performance prediction. Approaches for analyzing conditionally executed statements are also given. In [13] Fahringer, and later in [14, 15] Fahringer and Scholz, describe the use of value range analysis in their symbolic engine. These analyzers heavily rely on abstract interpretation and symbolic evaluation.

In contrast to these methods, our analysis method is based on a the manipulation of symbolic

expressions with the Chains of Recurrence (CR) algebra, which is significantly less expensive compared to abstract interpretation and symbolic differencing [24]. The CR algebra was first introduced by Zima [29] and later extended by Bachmann, Wang, and Zima [4, 28, 30, 3, 31]. In [24] we augmented the algebra and presented an algorithm for generalized induction variable (GIV) recognition and induction variable substitution (IVS) based on CR forms.

Allen and Johnson [2] used their vectorization and parallelization framework as an intermediate language for induction variable substitution to generate pointer expressions that are more amenable to vectorization than the original representation. However, their approach does not fully convert pointers to index expressions. Muchnick [19] mentions the regeneration of array indexes from pointer-based array traversal, but no explicit details are given. Franke and O'Boyle [16] developed a compiler transformation to convert pointer-based accesses to explicit array accesses for array recovery. However, their work has several assumptions and restrictions. In particular, their method is restricted to structured loops with a constant upper bound and all pointer arithmetic has to be data independent and conditional pointer updates are not permitted. Furthermore, pointer assignments, apart from initializations to some start element of the array to be traversed, are not allowed.

In contrast, our pointer analysis algorithm presented in [27] can handle non-rectangular loops, more general pointer initializations, and the most common types of data dependent and independent pointer updates. Our value range analysis algorithm builds on that work as well as on our work on timing estimators [11, 12].

3 Applications

Value range analysis is applied to support many compiler analysis and code transformation methods. In this section we discuss some of those applications.

3.1 Array Bounds Checking

Array bounds checking determines whether the value of an array expression is within specified bounds in all of its uses in a program. The task can be very expensive as there is overhead code to execute for each iteration we want to check. An example runtime array bounds checking in Fortran is illustrated in the code fragment below:

```
REAL A[0:100]
k = 0
m = 0
DO j = 0, 10
  DO i = 0, j – 1
C  /* Check k >= 0 and k =< 100 */
    A[k] = ...
    k = k + 1
C  /* Check m >= 0 and m =< 100 */
    A[m] = ...
  ENDDO
  IF (...) m = m + j
ENDDO
```

The closed form of k is $(j^2 - j)/2 + i$, which is a function over the loop iteration variables i and j. Because, $0 \leq (j^2 - j)/2 + i \leq 100$, the bounds check for A[k] can be eliminated. Note that the conditionally updated induction variable m does no have a closed form. Therefore, the bounds check cannot be removed using current methods. However, our method determines the bounding functions for m:

$$0 \leq m \leq \frac{j^2-j}{2} < 100$$

Therefore, the array bounds check for A[m] can be removed.

3.2 Dependence Testing

Dependence testing with value range information determines if there is any potential overlap between the reads and writes to an array in a loop nest. This is illustrated in the example code fragment below:

```
DO j = 1, N
  DO i = j, 10
    A[−j] = A[i∗j−1] + val
  ENDDO
ENDDO
```

The inner loop can be parallelized if the compiler can show that $i * j - 1 \geq -j$ which means that the values of array A are read before being written. Blume and Eigenmann's range analysis [9] calculates the value range of the difference between i∗j−1 and −j as follows:

$$\begin{aligned}
[i*j+j-1, i*j+j-1] &= [[j,10]*j+j-1, [j,10]*j+j-1] \\
&= [j^2+j-1, 11*j-1] \\
&= [[1,\infty]^2+[1,\infty]-1, 11*[1,\infty]-1] \\
&= [1,\infty] > 0
\end{aligned}$$

where $[lb, ub]$ denotes a range of values between lb and ub. Likewise, our CR-based value range analysis determines the lower bound of the difference using CR forms:

$$\begin{aligned}
L(i*j+j-1) &= L(\{\{1,+,1\}_j,+,1\}_i * \{1,+,1\}_j + \{1,+,1\}_j - 1) \\
&= L(\{\{1,+,4,+,2\}_j,+,\{1,+,1\}_j\}_i) \\
&= 1 > 0
\end{aligned}$$

where L is the CR-based lower bound function presented in Section 5. Since the resulting range information indicates that the difference is positive, there is no cross iteration dependence and the loop nest can be parallelized. Note that the bound is derived using the CR algebra rules applied to the CR form $\{1,+,1\}_j$ of j and nested CR form $\{\{1,+,1\}_j,+,1\}_i$ of i as defined by the loop iteration space, where $j \geq 1$ and $i \geq j$ with unit strides.

In [26] we present a more extensive data dependence algorithm based on CR forms to implement an efficient nonlinear version of the Banerjee Bounds test [5] that can handle a larger set of nonlinear and symbolic dependence problems compared to the work by Blume and Eigenmann on nonlinear data dependence testing.

3.3 Dependence Testing in Loops with Conditionally Updated Variables

When conditional variable updates occur in a loop nest, current methods for value range analysis for dependence testing fail. Consider for example the following loop nest:

```
k = 0
DO j = 1, N
  DO i = j, 10
    A[k] = A[i*j] + val
  ENDDO
  IF (...) k = k + 1
ENDDO
```

Variable k has no closed form. However, the CR-based range analysis detects that

$$0 \leq \mathsf{k} \leq \mathsf{j} - 1$$

Therefore, we use the upper bound CR form $\{0, +, 1\}_j$ of k in the lower bound derivation of the difference between i*j and k to obtain:

$$\begin{array}{rcl} L(i * j - k - 1) & = & L(\{\{1, +, 1\}_j, +, 1\}_i * \{1, +, 1\}_j) - k \\ & = & L(\{\{1, +, 1\}_j, +, 1\}_i * \{1, +, 1\}_j - \{0, +, 1\}_j) \\ & = & L(\{\{1, +, 2, +, 2\}_j, +, \{1, +, 1\}_j\}_i) \\ & = & 1 > 0 \end{array}$$

Since the resulting range information indicates that the difference is positive, there is no cross iteration dependence and the inner loop nest can be parallelized.

3.4 Dependence Testing in Loops with Pointer-Based Array Accesses

When (conditional) pointer updates occur in a loop nest and pointer arithmetic is used to access arrays, current value range analysis methods are inadequate and dependence testing fails. Array recovery can be used to recover arrays and eliminate pointer arithmetic, but these methods are severely restricted. Consider for example the following loop nest:

```
p = A; q = A;
for (i = 0; i < n; i++)
{ for (j = 0; j <= i; j++)
    *p = *++q + val; // equivalent to A[i] = A[(i*i+i)/2+j+1]
  p++;
}
```

Our array recovery algorithm determines the closed forms [27], while the method of Franke and O'Boyle [16] cannot be applied due to the nonlinear forms. Because the difference between (i*i+i)/2+j+1 and i is positive the loop can be parallelized.

The method presented in this paper can also handle conditional pointer updates, as shown in the following example loop:

```
p = A; q = A + 2*n;
for (i = 0; i < n; i++)
{ *p += *q--;
  if (...) p++;
}
```

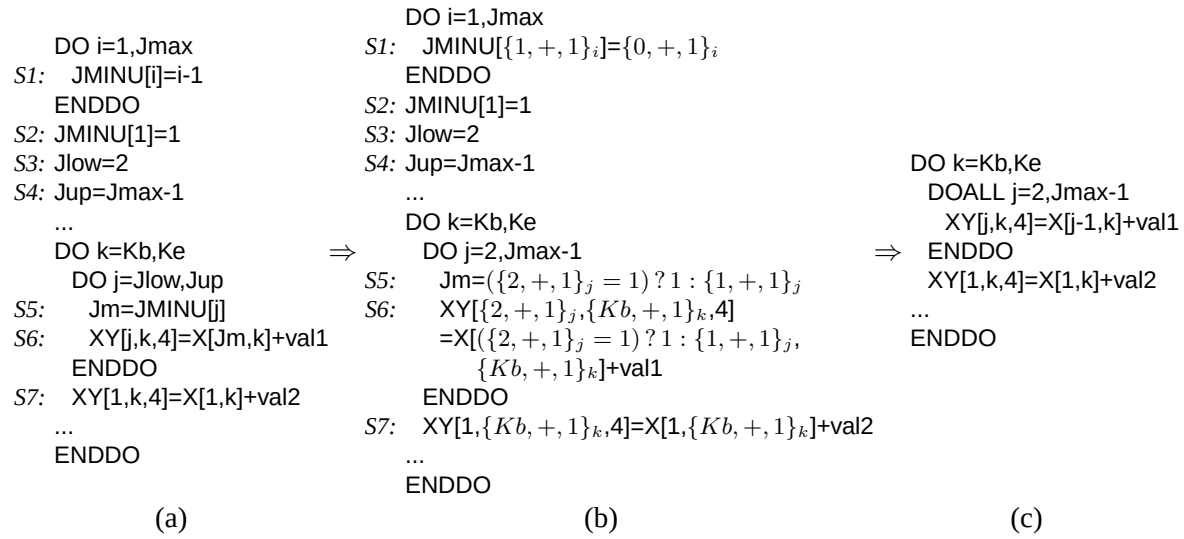


Figure 2: Code Segment of ARC2D

The bounding functions of the pointer-access description (PAD) [27] of pointer p are

$$\mathsf{A} \leq \mathsf{p} \leq \mathsf{A} + \mathsf{n} - 1$$

Therefore, the difference between the memory locations accessed by p and q are determined

$$\begin{aligned} L(q-p) &= L(\{\mathsf{A}+2\mathsf{n},+,-1\}_i - \{\mathsf{A},+,1\}_i) \\ &= L(\{2\mathsf{n},+,-2\}_i) \\ &= 2 > 0 \end{aligned}$$

where $\{\mathsf{A}+2\mathsf{n},+,-1\}_i$ is the PAD of p and $\{\mathsf{A},+,1\}_i$ is the PAD of q. Because the difference is positive, there is no cross iteration dependence.

3.5 Global Value Range Propagation

Maslov [18] describes an algorithm for global value propagation of affine constraints using F-relations. We improve on this method to handle nonlinear functions.

The value range propagation with CR forms is illustrated with the ARC2D code segment of the Perfect Benchmarks$^{(\mathrm{R})}$ shown in Figure 2. This example illustrates the technique for affine expressions. However, the CR forms are not limited to be affine and may encompass piecewise polynomials and geometric sequences.

The contents of array JMINU set by statements *S1* and *S3* is represented by the conditional expression $(i = 1)\,?\,1 : i-1$. JMINU in *S6* is indexed by j, which has the CR form $\{1,+,1\}_j$. Expanding JMINU with its definition and replacing i with $\{1,+,1\}_j$ gives $(\{2,+,1\}_j = 1)\,?\,1 : \{0,+,1\}_j$. The conditional CR-expression $(\{2,+,1\}_j = 1)\,?\,1 : \{0,+,1\}_j$ simplifies to $\{0,+,1\}_j$ because the value 1 is not in the sequence described by $\{2,+,1\}_j$ by testing $L\{2,+,1\}_j = 2 > 1$, thereby indicating that Jm is a true linear induction variable.

The j-index of XY in *S6* is monotonic (monotonicity is discussed in Section 4), which allows parallelization of the j-loop, as shown in Figure 2 (c). In addition, there is no output dependence between *S6* and *S7*, because the difference between the index expressions is positive, i.e. $L(\{2,+,1\}_j - 1) = L\{1,+,1\}_j = 1 > 0$.

4 Monotonicity

To determine an accurate value range of expressions involving (conditionally) updated (generalized) induction variables the monotonic properties of the expression as a function of the variables is determined. The CR form of a function enables the determination of monotonicity in a straightforward manner.

Consider for example the expression $3 * j + i + 2 * k + 1$, where $i \geq 0$ and $j \geq 0$ are the index variables that span a two-dimensional loop iteration space with unit stride and k is a generalized induction variable with characteristic function $\chi(i) = (i^2 - i)/2$. The CR formation of this expression proceeds as follows (see [25] for the CR algebra rules):

$$
\begin{array}{ll}
\mathcal{CR}_k(\mathcal{CR}_j(\mathcal{CR}_i(3*j+i+2*k+1))) & \\
\quad = \mathcal{CR}_k(\mathcal{CR}_j(3*j+\{0,+,1\}_i+2*k+1)) & \text{(replacing } i\text{)} \\
\quad = \mathcal{CR}_k(3*\{0,+,1\}_j+\{0,+,1\}_i+2*k+1) & \text{(replacing } j\text{)} \\
\quad = 3*\{0,+,1\}_j+\{0,+,1\}_i+2*\{0,+,0,+,1\}_i+1 & \text{(replacing } k\text{)} \\
\quad = \{\{1,+,3\}_j,+,1,+,2\}_i & \text{(normalize with CR algebra)}
\end{array}
$$

where the characteristic function $\chi(i)$ of k in CR normal form is $\Phi(k) = \{0,+,0,+,1\}_i$. The resulting multivariate CR form $\{\{1,+,3\}_j,+,1,+,2\}_i$ is monotonic in both the i and j directions, because the coefficients of the CR forms are all positive, indicating that the initial value of the function is positive and that the value of the function increases by increasing i and j.

To analyze the properties of a CR form, we define two basic operations V and Δ on CR forms to obtain the *initial value* of a CR form and the *stepping* function of a CR form, respectively.

Definition 1 *The* initial value $V\Phi_i$ *of a CR form* Φ_i *is*

$$V\Phi_i = V\{\phi_0, \odot_1, \ldots, \odot_k, \phi_k\}_i = \phi_0$$

Monotonic properties of a CR forms are determined by the stepping functions. The sign and size of the step function of a CR is used to determine the direction and growth of the GIVs and induction expressions in a loop iteration space. The initial value of a CR form is the first coefficient, which is the starting value of the CR form evaluated on a unit grid in the i-direction.

Definition 2 *The* step *function* $\Delta\Phi_i$ *of a CR form* Φ_i *is*

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

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

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

For example, $\Delta_i\{\{1,+,3\}_j,+,1,+,2\}_i = \{1,+2\}_i$ and $\Delta_j\{\{1,+,3\}_j,+,1,+,2\}_i = 3$. Because the lower bound $L\{1,+,2\}_i = 1 > 0$ and $3 > 0$, the CR form is monotonic with respect to i and j. Therefore, the extreme values of $3 * j + i + 2 * k + 1$ on $i \geq 0$ and $j \geq 0$ lie on the endpoints of the iteration space spanned by i and j.

5 Lower and Upper Bounds on CR Forms

In the previous sections we used the lower bound L on a CR form. In this section we formally define the lower and upper bound functions L and U, respectively. The lower and upper bounds are exact if the function is monotonic. Because the CR form of a polynomial is unique [25] and monotonicity can be determined from the CR coefficients directly, the CR representation of a polynomial does not suffer from the problem that the tightness of a range analyzer is dependent on how the analyzed expression is formed [1], such as with Horner's rule for polynomials.

Definition 3 *The* lower bound $L\Phi_i$ *of a multi-variate CR form* Φ_i *is*

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

and the upper bound $U\Phi_i$ *of a multi-variate CR form* Φ_i *is*

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

where $\mathcal{CR}_i^{-1}(\Phi_i)$ *is the closed form of* Φ_i *with respect to* i *(i.e. nested CR forms are not converted), and where*

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

Based on CR construction and simplification rules, and CR^{-1} inverse rules described in [25], we have developed an algorithm that efficiently evaluates the monotonic properties and bounds of expressions containing GIVs.

6 Value Range Analysis Algorithm

We modified and extended our GIV analysis algorithm [25, 27] to handle conditionally updated variables that may or may not have closed forms. In the modified GIV recognition algorithm, we formulate the multi-variate CR forms for each non-aliased scalar integer/pointer variable for each path in a loop nest. In this way, a *set of* CR forms or PADs for a variable is determined in our current implementation, rather than a single CR form as in [25]. These CR forms and PADs describe sequences of possible values for the conditionally updated variables in a loop. We note

that the CR forms describe the possible values of the variable with respect to the paths in the loop. We combine the set of CR forms to compute the CR forms for functions that bound the possible range of values of the conditionally updated variables. To this end, we define *min* and *max* bounding functions for a set of CR forms over an index space.

Definition 4 *Let* $\{\Phi_i^1, \ldots, \Phi_i^n\}$ *be a set of* n *multi-variate polynomial CR forms over* i*. Then the* minimum *CR form is defined by*

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

and the maximum *CR form is defined by*

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

These bounding functions are polynomial CR forms calculated from the set of CR forms of the conditionally updated variables in the index space spanned by a loop nest.

The polynomial min and max bounds derived in CR form are always more accurate compared to bounds derived from polynomials coefficients. Consider for example $p(i) = \frac{1}{2}(i^2 - i)$ which has CR form $\{0, +, 0, +, 1\}_i$ for $i \geq 0$ and $q(i) = 0$. Then, $min(\{0, +, 0, +, 1\}_i, 0) = 0 = q(i)$ and $max(\{0, +, 0, +, 1\}_i, 0) = \{0, +, 0, +, 1\}_i = p(i)$, while taking the max over the polynomial coefficients gives $max(p, q) = max(\frac{1}{2}, 0)i^2 max(-1, 0)i = \frac{1}{2}i^2 > p(i)$, which is inexact.

The extended and modified GIV recognition algorithm for modeling conditionally updated variables using the min and max bounding functions proceeds as follows:

1. For each path p in the body of a (nested) loop, find the set A_p of variable-update pairs $\langle v, e\rangle$ with our single-path CR-based GIV recognition algorithm [25].

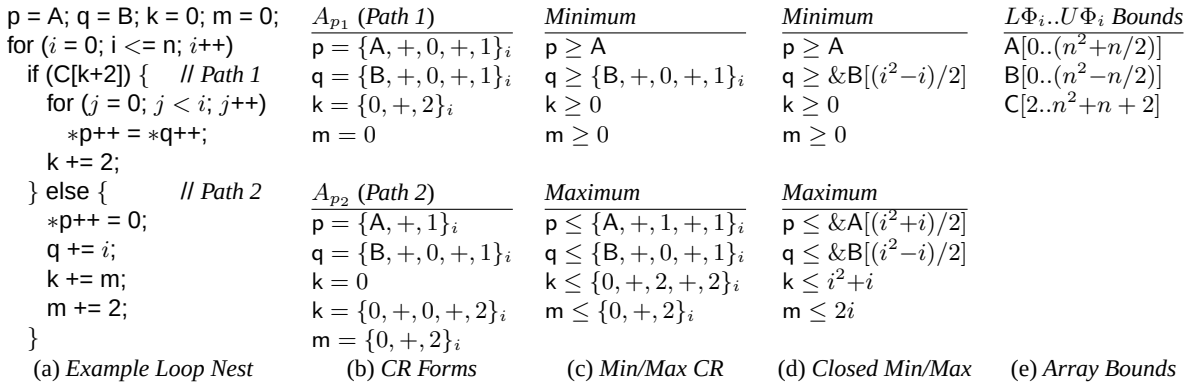
2. For each variable-update pair $\langle v, e\rangle$ defined in topological order $\prec$ in the combined set $\bigcup A_p$, compute alternative CR forms $\Phi^j(v) = \mathcal{CR}(e)$ by replacing each induction variable in expression e with its previously computed CR form. The $\prec$ relation defines a topological order on the pairs in the set by

   $$\langle v_1, e_1\rangle \prec \langle v_2, e_2\rangle \qquad \text{if } v_1 \neq v_2 \text{ and } v_1 \text{ occurs in } e_2$$

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

3. For each variable v collect the CR forms $\Phi^j(v)$ from the pairs $\langle v, \Phi^j(v)\rangle \in \bigcup A_p$, where the $\Phi^j(v)$ were computed in step 2. Compute the *min* and *max* bounding functions over the set $\{\Phi^j(v)\}$ for variable v.

4. When the *min* and *max* bounding functions are identical, the bounding function forms a single (multi-variate) characteristic function of a GIV. The closed form GIV is used for induction variable substitution and array recovery.



(a) *Example Loop Nest* (b) *CR Forms* (c) *Min/Max CR* (d) *Closed Min/Max* (e) *Array Bounds*

Figure 3: Analysis of an Example Code Segment Containing Conditional Updates

The cost of the algorithm to derive CR forms and bounding functions is linear in the number of paths in a loop nest, if the order of the polynomial induction variables is bounded by a constant. When the polynomial order is bounded, the CR algebra requires a bounded number of rewrite steps to derive CR forms. The topological ordering $\prec$ can be implemented using bit-vectors to reduce the complexity of the algorithm with respect to the pass over the combined set $\bigcup A_p$ in step 2 of the algorithm.

In Figure 3 the algorithm is applied to an example code segment with conditional variable updates. In step (b) the CR forms are formed for the two paths. In step (c) the CR forms are combined to form min and max bounding functions. In step (d) the closed-forms of the CR-forms are shown for comparison. Next in step (e), array bounds are determined from the min and max forms. As a result, the value range of the the pointer p and q accesses to arrays A, B, and C determine the array bounds of these arrays. Note that p, q are pointers that behave as nonlinear induction variables. Also note that conditionally updated variables k and m are coupled and k forms a nonlinear induction variable with respect to *Path 2*.

7 Conclusions.

The use of pointer arithmetic and conditionally updated induction variables and pointers prevent compilers from effectively applying dependence testing and loop analysis for optimization and parallelization. As a consequence, many C codes such as DSP applications cannot be effectively optimized. In this paper we have presented a new value range analysis technique that utilizes CR-forms for evaluation of expressions containing conditionally updated induction variables and pointers. We have shown several applications of the method for loop analysis, such as dependence testing and array bounds checking.

The implementation of the CR algebra and the analysis algorithm in the SUIF and Polaris compilers is close to completion. For more information and updates on the status of this project, please refer to `http://www.cs.fsu.edu/~birch/research/crDemo3.php` .

References

[1] ALEFELD, G. Interval arithmetic tools for range approximation and inclusion of zeros. In *Error Control and Adaptivity in Scientific Computing*, H. Bulgak and C. Zenger, Eds. Kluwer Academic Publishers, 1999, pp. 1–21.

[2] ALLEN, R., AND JOHNSON, S. Compiling C for vectorization, parallelization, and inline expansion. In *Proc. of the SIGPLAN 1988 Conference of Programming Languages Design and Implementation* (Atlanta, GA, June 1988), pp. 241–249.

[3] BACHMANN, O. *Chains of recurrences*. PhD thesis, 1997.

[4] BACHMANN, O., WANG, P. S., AND ZIMA, E. V. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *International Symposium on Symbolic and Algebraic Computation* (1994), pp. 242–249.

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

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

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

[8] BLUME, W., AND EIGENMANN, R. Demand-driven symbolic range propagation. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing* (Columbus, OH, 1995), pp. 141–160.

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

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

[11] ENGELEN, R. A. V., AND GALLIVAN, K. A. Tight non-linear loop timing estimation. In *Proceedings of the 2002 International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems* (2002), pp. 21–26.

[12] ENGELEN, R. V., GALLIVAN, K., AND WALSH, B. Tight timing estimation with newton-gregory formulae. In *Proceedings of the 2003 Compilers for Parallel Computers* (2003), pp. 321–329.

[13] FAHRINGER, T. Efficient symbolic analysis for parallelizing compilers and performance estimators. *The Journal of Supercomputing 12*, 3 (1998).

[14] FAHRINGER, T., AND SCHOLZ, B. Symbolic evaluation for parallelizing compilers. In *Proceedings of the 11th international conference on Supercomputing* (1997), ACM Press, pp. 261–268.

[15] FAHRINGER, T., AND SCHOLZ, B. A unified symbolic evaluation framework for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems 11*, 11 (November 2000), 1105–1125.

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

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

[18] MASLOV, V. Enhancing array dataflow dependence analysis with on-demand global value propagation. In 9^{th} *ACM International Conference on Supercomputing* (Barcelona, Spain, July 1995), ACM Press, pp. 265–269.

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

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

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

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

[23] SHEN, Z., LI, Z., AND YEW, P.-C. An empirical study on array subscripts and data dependencies.

[24] VAN ENGELEN, R. A. Efficient symbolic analysis for optimizing compilers. *Lecture Notes in Computer Science 2027* (2001).

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

[26] VAN ENGELEN, R. A., BIRCH, J., SHOU, Y., WALSH, B., AND GALLIVAN, K. A. A unified framework for nonlinear dependence testing and symbolic analysis. In *proceedings of the 18th Annual ACM International Conference on Supercomputing* (2004).

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

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

[29] ZIMA, E. V. Recurrent relations and speed-up of computations using computer algebra systems. In *Proc. of DISCO, Bath, U.K., April 1992* (1992), pp. 152–161.

[30] ZIMA, E. V. Simplification and optimization transformations of chains of recurrences. In *International Symposium on Symbolic and Algebraic Computation* (1995), pp. 42–50.

[31] ZIMA, E. V. On computational properties of chains of recurrences. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation* (2001), ACM Press, p. 345.