A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis

Robert A. van Engelen[*]
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4530
engelen@cs.fsu.edu

J. Birch, Y. Shou, B. Walsh
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4530

Kyle A. Gallivan[†]
School of Computational Science and Information Technology
Florida State University
Tallahassee, FL 32306-4120
gallivan@csit.fsu.edu

ABSTRACT

This paper presents a unified approach for generalized induction variable recognition and substitution, pointer analysis, analysis of conditionally updated variables, value range analysis, array region analysis, and nonlinear dependence testing. The analysis techniques share a well-defined uniform approach based on the chains of recurrences algebra. The uniform algebraic approach provides a powerful unified framework for developing analysis algorithms for restructuring compilers. The paper introduces a new set of analysis algorithms that accurately handle conditional control flow, pointer arithmetic, and nonlinear symbolic expressions in loops, which are known to be problematic for conventional restructuring compilers.

Categories and Subject Descriptors

D.3.4 [**Software**]: Processors—*Compilers, Optimization*

General Terms

Compiler Algorithms

1. INTRODUCTION

For timely and efficient program execution, programmers must rely on compilers to restructure and/or parallelize code. The effectiveness of a restructuring compiler depends heavily on the accuracy of the analysis system it employs. Consequently, the importance of accurate symbolic analysis and data dependence testing has been recognized by many [8, 12, 14, 23, 27, 28, 32, 33, 36, 38, 39, 40, 43, 46, 48, 50, 51, 57, 64].

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

[†]Supported in part by NSF grants CCR-0105422, and EIA-0072043.

Current dependence analyzers and frameworks for symbolic analysis in restructuring compilers are quite powerful. However, recent work by Psarris [41, 44], Franke and O'Boyle [26], Wu et al. [59], and earlier work by Shen, Li, and Yew [49], Haghighat [31], and Collard et al. [16] mention the difficulty dependence analyzers have with nonlinear symbolic expressions, pointer arithmetic, and conditional control flow in loop nests.

This paper presents a unified framework for developing compiler analysis methods, such as generalized induction variable recognition, induction variable substitution, pointer analysis, array recovery, value range analysis, array region and bounds analysis, and nonlinear dependence testing. The analysis methods in this framework share a well-defined uniform approach based on the chains of recurrences algebra to accurately handle nonlinear symbolic expressions, pointer arithmetic, and conditional control flow in loops. This shared uniform algebraic approach provides a powerful unified analysis framework for restructuring compilers that naturally handles conditional control flow with variable and pointer updates in loops. The main contributions of this this paper can be briefly summarized by:

- providing an efficient technique for the detection and substitution of generalized induction variables (GIVs);
- improving array recovery methods by analyzing both linear and nonlinear pointer updates;
- enhancing value range analysis and array region analysis by determining the range of values of conditionally updated coupled induction variables and pointers;
- increasing the accuracy of data dependence analysis by computing dependence equations for affine and nonlinear array indices, such as those formed by multivariate polynomial and geometric index expressions and index expressions involving conditionally updated variables and pointers.
- providing an effective formalism to develop nonlinear dependence tests that do not depend on closed forms or code adjustments, such as GIV substitution or array recovery.

In contrast, most of the symbolic analysis methods used by current state-of-the-art restructuring compilers require the formation of closed-form expressions at the source-code level, which involves the use of a GIV recognition and substitution algorithm to handle induction variables in loops. GIVs are a general class of induction variables that form polynomial and geometric progressions through loop iterations [5, 20, 21, 27, 29, 56]. Methods for GIV

```
int i, j = 0, k = 0, m = 1;
for (i = a; i <= b; i++) {
   . . .
   A[f(i,j,k,m)] = A[g(i,j,k,m)];
   . . .
   j = j + k;
   k = k + 1;
   m = m << 1;
   . . .
}
```

(a) *Original*

```
int i;
for (i = 0; i <= b-a; i++) {
   . . .
   A[f(i+a, (i^2-i)/2, i, 1<<i)]
   = A[g(i+a, (i^2-i)/2, i, 1<<i)]
   . . .
}
```

(b) *After IVS*

Figure 1: IVS

recognition, such as *sequence classification* by Gerlek et al. [27] and Haghighat's *symbolic differencing* method [29, 30], construct closed-form *characteristic functions* by analyzing the GIV updates in a loop. The removal of the updates and the replacement of the GIV's uses with semantically equivalent closed-form expressions is known as *Induction Variable Substitution* (IVS).

IVS transformation is illustrated in Figure 1. The example loop in Figure 1(a) exhibits linear induction variables i and k, a quadratic GIV j, and a geometric GIV m. After IVS shown in Figure 1(b), array index expressions f and g over i, j, k, m may or may not be affine, depending on the actual uses of i, j, k and/or m, e.g. array access A[2$*i$+a] is affine while A[$(i^2-i)/2$] and A[1<<i] are not.

We state some well-known facts about IVS and related compiler analysis techniques that depend on GIV recognition:

- IVS ensures that the cross-iteration dependencies of GIV updates are eliminated to further the parallelization of a loop nest [27, 31, 56].
- Array-based data dependence testing is applied after IVS to enable loop restructuring and optimization. If the array index expressions obtained by IVS are affine, conventional data dependence testing methods can be used [8, 28, 35, 42, 45, 57, 64]. If the resulting array index expressions are non-affine, nonlinear data dependence testing methods must be used [11, 37, 38, 58].
- The computation of closed forms for IVS is expensive and not always required for dependence testing [59].
- The presence of pointer arithmetic in a loop nest introduces aliasing problems hampering or disabling data dependence analysis [48]. Array recovery [26] can be an effective method to convert pointer accesses to array accesses in a loop nest, see Figure 2. The resulting array accesses can then be analyzed using conventional methods that operate on the closed-form index expresssions of array accesses.
- When dependence testing fails after IVS and array recovery and the loop nest cannot be restructured for parallelization, it is still possible to apply various data access optimizations, such as removal of array bound checks, software pipelining, blocking, data prefetching, and other locality enhancing optimizations.
- IVS fails when variable updates occur in multiple execution paths in the loop nest, because it is not generally possible to construct a single closed-form characteristic function when a variable has a single conditional update, has multiple conditional updates, and/or is dependent on variables that are conditionally updated [59], as is illustrated in Figure 3(a).
- Array recovery fails when pointers are conditionally updated, see Figure 3(b).

```
int i, *p = A, *q = B+n;
for (i = 0; i < n; i++) {
   . . .
   *p++ = *--q;
   . . .
}
```

(a) *Original*

```
int i;
for (i = 0; i < n; i++) {
   . . .
   A[i] = B[n-i-1];
   . . .
}
```

(b) *After Array Recovery*

Figure 2: Array Recovery

```
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 3: Loops with Conditional Updates

Our unified approach for compiler analysis handles variable and pointer updates in conditional paths. In addition, the framework simplifies the composition of compiler analysis stages, because the nonlinear dependence test and related symbolic analysis in our approach do not require closed forms and IVS and array recovery can be delayed (see also [59]). Therefore, value range and array region analysis can proceed without IVS. Likewise, dependence testing can proceed without IVS or array recovery.

This paper briefly introduces the mathematical foundations of the chains of recurrences formalism. For more information on the use of the algebra in the context of compilers, the reader is referred to [54]. The rest of the paper presents the unified analysis algorithms and their applications. Throughout the paper, we will demonstrate the effectiveness of these analysis methods on a number of illustrative examples.

The remainder of this paper is organized as follows. We briefly describe the uniform formal representation of symbolic expressions with chains of recurrences in Section 2. We also present our method for GIV recognition and manipulation, which forms the basis of the symbolic analysis. The uniform framework for symbolic analysis and dependence testing is presented in Section 3. The analysis uses the uniform formal representation for analyzing conditionally updated variables and pointers to derive value ranges for array region analysis and dependence testing. Section 4 gives related work. Finally, the paper is summarized in Section 5.

2. UNIFORM FORMAL NOTATION

This section introduces the uniform formal notation used in our compiler framework for the manipulation of loop counter variables, pointers, and GIVs in affine and non-linear expressions.

2.1 Chains of Recurrences

The uniform notation in our framework is based on the *Chains of Recurrences* (CR) form of functions. The CR formalism was originally developed by Zima [61, 62, 63] and later improved by Bachmann, Zima, and Wang [6, 7] to expedite the evaluation of multivariate functions on regular grids. Nested CR forms represent multivariate closed-form functions over sets of index variables.

DEFINITION 1. *A function or closed-form expression over an index variable i can be rewritten into a mathematically equivalent CR of the form (see [6]):*

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

where ϕ are coefficients consisting of constants, symbolic expressions, or nested CR forms, and $\odot = +$ or $\odot = *$.

To analyze and manipulate symbolic expressions in a loop iteration space, we extended the CR algebra with new rules [54] and implemented a symbolic manipulation system developed in C as a library for SUIF and Polaris. The implementation of the CR algebra is not computationally intensive and comparable to classical constant-folding [1]. Consider for example the CR algebra rules

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

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

DEFINITION 2. *The mapping from a closed-form symbolic expression $e_{i_1,\ldots,i_n}$ defined over a set of index variables $i_1, \ldots, i_n$ to a nested CR form is defined by*

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

where $\Phi(i_j)$ is the CR representation of the index variable i_j. When the index variables $i_1, \ldots, i_n$ span a unit-distance grid with origin $(x_1, \ldots, x_n)$, then $\Phi(i_j) = \{x_j, +, 1\}_{i_j}$ for all $j = 1, \ldots, n$.

The $\mathcal{CR}$ mapping recursively replaces induction variables i_j in an expression with their corresponding CR forms, using the substitution $e[i_j \leftarrow \Phi(i_j)]$. The induction variables can be linear, such as the loop counter variables, or non-linear, such as GIVs. The CR algebra is then applied to form (nested) CR forms.

The importance of CR formation for the analysis and manipulation of symbolic expressions is clear when considering the following classes of functions fundamental to compiler analysis:

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

Multivariate Polynomial expressions are uniquely represented by nested CR forms of length k, where k is the maximum order of the polynomial. All $\odot$ operations in the CR form are additions, i.e. $\odot = +$. A d-dimensional k-order polynomial can be translated in $\mathcal{O}(d\,k^2)$ steps by a conversion algorithm based on matrix-vector multiplication with Newton matrices [6, 53].

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

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

The CR algebra is closed under the formation of the characteristic function of a GIV. In addition, in [52] we proved that the (nested) CR forms are normal forms for GIVs. Therefore, semantically equivalent induction expressions can be easily identified.

DEFINITION 3. *The inverse mapping $\mathcal{CR}^{-1}$ converts CR forms to closed-form expressions using our extension of the CR algebra rules [54]. To obtain the closed-form $\chi(i) = \mathcal{CR}^{-1}(\Phi(v))$ of the polynomial term of a GIV v represented by the CR form $\Phi(v) = \{\phi_0, +, \ldots, +, \phi_k\}_i$ Newton's formula for the interpolating polynomial can be used:*

$$\chi(i) = \textstyle\sum_{j=0}^{k} \phi_j \binom{i}{j}$$

The inverse mapping is applied component-wise on a multivariate GIV, i.e. $\mathcal{CR}_i^{-1}$, or in all directions $\mathcal{CR}^{-1}$ at once. Both the $\mathcal{CR}$ mapping and its inverse $\mathcal{CR}^{-1}$ are used in our framework for symbolic analysis.

EXAMPLE 1. *Consider the expression $n * j + i + 2 * k + 1$, where $i \geq 0$ and $j \geq 0$ are the index variables that span the two-dimensional loop iteration space with unit stride and k is a GIV with characteristic function $\chi(i) = (i^2{-}i)/2$. The CR formation of this expression proceeds as follows:*

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

where the characteristic function $\chi(i)$ of k in CR normal form is $\Phi(k) = \{0, +, 0, +, 1\}_i$.

By application of Newton's formula we obtain the closed form $\mathcal{CR}^{-1}(\{\{1, +, n\}_j, +, 1, +, 2\}_i) = n * j + i^2 + 1$. $\diamond$

2.2 GIV Recognition and IVS

The characteristic function of a GIV from a loop nest is obtained using our GIV recognition algorithm [54]. The GIV recognition algorithm analyzes a loop nest (not necessarily perfectly nested) from the innermost loops, which are the primary candidates for compiler optimization, to the outermost loops. The worst-case computational complexity of the method is $\mathcal{O}(k\,n\,\log(s)\,m^2)$, see [54], where k is the maximum loop nesting level, n is the length of the source code fragment analyzed, s is the number of GIVs, and m is the maximum polynomial order of the GIVs in the fragment. Typically, k and m are quite small (e.g. ≤ 4 for the Perfect Benchmark suite). The number of GIVs s per loop varies, but usually does not exceed more than a handful of variables. The practical cost for GIV recognition is determined by the algorithm's AST traversal cost of a loop body to collect the GIV update operations and to store them in symbolic form.

Our GIV recognition method is straightforward to implement, yet equally powerful as other more complicated methods for GIV recognition such as differencing [30] and methods based on solving recurrence systems [27]. The GIV recognition algorithm can handle multiple assignments to induction variables, generalized induction variables in loops with symbolic bounds and strides, symbolic integer division, conditional induction expressions, cyclic induction dependencies, symbolic forward substitution, symbolic loop-invariant expressions, and wrap-around variables. Our algorithm handles the most complicated classes of GIVs, such as those found in the TRFD and MDG benchmarks [54].

Typically, closed forms are computed to apply IVS in order to perform subscript-to-subscript dependence testing. Our approach uses $\mathcal{CR}^{-1}$ to compute closed forms. However, the CR forms of GIVs can be used directly by our nonlinear data dependence tests, without requiring IVS. In Section 3.4 we develop an algorithm for nonlinear dependence testing with conditionally updated induction variables that have neither closed-forms nor single CR forms.

```
ijkl=0
ij=0
DO i=1,m                          DO i=0,m-1
  DO j=1,i                          DO j=0,i-1
    ij=ij+1
    ijkl=ijkl+i-j+1
    DO k=i+1,m                        DO k=0,m-i-2
      DO l=1,k                          DO l=0,k
        ijkl=ijkl+1
        xijkl[ijkl]=xkl[l]                xijkl[Φ(ijkl)]=xkl[Φ(l)]
      ENDDO                             ENDDO
    ENDDO                             ENDDO
    ijkl=ijkl+ij+left
  ENDDO                             ENDDO
ENDDO                             ENDDO
```

(a) *Original* (b) *GIV Analyzed*

Figure 4: TRFD Code Segment

EXAMPLE 2. *Consider Figure 4(a) depicting a segment of the original TRFD code. The characteristic functions of* ijkl *and l as shown in the analyzed code in Figure 4(b) are*

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

and $\Phi(l) = \{1,+,1\}_l$ respectively. The closed form of $\Phi(\mathsf{ijkl})$ is

$$\begin{aligned}\mathcal{CR}^{-1}(\Phi(\mathsf{ijkl})) = l &+ i*(k+(\mathsf{m}+\mathsf{m}^2+2*\mathsf{left}+6)/4) \\ &+ j*(\mathsf{left}+(\mathsf{m}+\mathsf{m}^2)/2) \\ &+ ((i*\mathsf{m})^2+\mathsf{m}*i^2)/4 \\ &+ (k^2+3*k+i^2*(\mathsf{left}+1))/2+2\end{aligned}$$

For IVS, the closed form replaces the ijkl *variable in the* xijkl[ijkl] *array index.* ◇

For this paper, we extended the GIV recognition algorithm to handle pointers and conditionally updated variables, which will be presented in the next sections. The extended CR-based GIV analysis constitutes the first phase in our compiler implementation to analyze GIVs, pointers, and conditionally updated variables.

2.3 Pointer Arithmetic and Array Recovery

We extended our GIV recognition algorithm to analyze pointer-based array accesses. The modified algorithm exploits the fact that the analysis of pointer updates in a loop nest can be viewed as a special form of induction variable recognition. To this end, we introduced the notion of *Pointer Access Descriptions* (PADs) [55] which are canonical CR forms that capture the pointer-based access functions defined over the iteration space of the loop nest.

DSP codes commonly implement filtering operations. The DSP implementations of these algorithms exhibit nonlinear induction variables and pointer updates. Our pointer analysis for array recovery can handle these specialized algorithms, including radix-2 FFTs.

EXAMPLE 3. *Consider the code segment of the* Lsp_Az *routine of the GSM Enhanced Full Rate speech codec [22] shown in Figure 5(a). Because the loop nest is triangular and involves a nonlinear data-dependent pointer update, Franke and O'Boyle's array recovery method cannot be applied. In contrast, our pointer analysis algorithm annotates the example code with PADs shown in Figure 5(b). The transformation of PADs to closed-form array accesses results in the array recovered code shown in Figure 5(c).* ◇

2.4 Stepping Functions and Monotonicity

To analyze the monotonic properties of GIVs and induction expressions, 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 4. *The* initial value $V\Phi_i$ *of a CR form* Φ_i *is*

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

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 5. *The* step *function* $\Delta\Phi_i$ *of a CR form* Φ_i *is*

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

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

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

We use the step functions to determine the monotonic properties of GIVs and induction expressions. The monotonic properties are determined by the stepping functions to find the direction of the growth of the GIVs and induction expressions in a loop iteration space.

EXAMPLE 4. *Consider the* ijkl *induction variable of the TRFD code shown in Figure 4. Its CR form* $\Phi(\mathsf{ijkl})$ *has the following four step functions in the i, j, k, and l direction, respectively:*

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

Note that the step functions in the k and l directions are non-negative, because the CR coefficients are non-negative (this can also be verified using the closed form $\mathcal{CR}^{-1}\Delta_k\Phi(\mathsf{ijkl}) = 2+i+k > 0$, with $i \geq 0$ and $k \geq 0$ in the loop nest). Therefore, the growth of the ijkl *induction variable in the k, l direction of the index space is non-negative and the addressing of the* xijkl[ijkl] *is strictly monotonically increasing in the inner k, l loop nest, allowing the loop nest to be parallelized. Also note that the growth of* ijkl *in the entire i, j, k, l index space is non-negative if $\mathsf{left} > \mathsf{m}(\mathsf{m}{+}1)/2$ holds. The closed-form coefficients of the step functions can be used to implement runtime tests to implement two versions of the code: one that is fully parallelized and one version that is partially parallelized.* ◇

Stepping functions of PADs are used for coalescing load/store operations to subsequent memory addresses [18]. When the value of a PAD computed for one or more memory accesses is aligned, and its stepping function is a multiple of the memory word (or vector) size, the memory accesses can be coalesced. Step information on PADs is also used to optimize spatial locality. Due to the spatial constraints of this paper, these aspects will not be further discussed.

3. ANALYSIS ALGORITHMS

This section presents our analysis algorithms based on the uniform representation of symbolic expressions with CR forms. An algorithm is presented to analyze conditionally updated variables. The algorithm is used for value range analysis, array region analysis, and dependence testing.

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

(a) *Original*

```
int *f = ..., *lsp = ...;
. . .
for (i = 0; i <= 3; i++) {
  *({f+2,+,1}_i) = *({f,+,1}_i);
  for (j = 0; j <= {0,+,1}_i; j++)
    *({{f+2,+,1}_i,+,-1}_j) += *({{f,+,1}_i,+,-1}_j)
      - 2 *(*{lsp+2,+,2}_i)*(*{{f,+,1}_i,+,-1}_j);
  *(f+1) -= 2*{lsp+2,+,2}_i;
}
```

(b) *Analyzed*

```
int *f = ..., *lsp = ...;
. . .
for (i = 0; i <= 3; i++) {
  f[i+2] = f[i];
  for (j = 0; j <= i; j++)
    f[i-j+2] += f[i-j] - 2*lsp[2*i+2]*f[i-j+1];
  f[1] -= 2*lsp[2*i+2];
}
```

(c) *Transformed*

Figure 5: ETSI `Lsp_Az` **Code Segment**

3.1 Analyzing Conditionally Updated Variables

We modified and extended our analysis algorithms to handle conditionally updated variables that may or may not have closed forms. In the modified GIV recognition algorithm, we formulate the multivariate CR forms for each non-aliased scalar integer and 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 [54]. These CR forms and PADs describe sequences of possible values for the conditionally updated variables in a loop. We note that none of the CR forms describe the variable values exactly, since the actual flow of control is unknown at compile time. 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 6. *Let $\{\Phi_i^1, \ldots, \Phi_i^n\}$ be a set of n multivariate polynomial CR forms over the same index variable 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. Some adjustments are required to cover CR forms of geometric series, which involves the use of value range analysis discussed in Section 3.2.

The extended and modified GIV recognition algorithm for modeling conditionally updated variables proceeds as follows:

1. For each path p in the body of a (nested) loop, find the set A_p of variable-update pairs $\langle v, e\rangle$ from the set of assignments of expressions e to v using our single-path CR-based GIV recognition algorithm [54].

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 variable in expression e with its previously computed CR form. The $\prec$ relation defines a topological order on the pairs in $\bigcup A_p$ 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 (multivariate) characteristic function of a GIV (to enable the application of IVS).

5. Compute the V and Δ functions for all CR forms of array index expressions and pointer accesses.

This extended algorithm is capable of analyzing conditionally updated variables and pointers to determine array access information for accurate value range analysis, array region and bounds analysis, and dependence testing.

EXAMPLE 5. *Figure 6(a) depicts an example code segment with conditional variable updates. First, the CR forms are calculated. The CR forms for the updates in the first path A_{p_1} and second path A_{p_2} are shown in Figure 6(b). Then, by combining the CR forms, we compute the* min *and* max *bounding functions as shown in Figure 6(c). For sake of exposition, the closed-forms of these expressions are shown in Figure 6(d). Because the* min *and* max *bounding functions of pointer* q *are identical, the array access made by* q *forms a GIV with closed-form array access* B$[(i^2{-}i)/2]$.

Next, array index stride information is collected from the CR forms, see Figure 6(e), consisting of the set of all possible stride values of the array index expression through the iterations of the loop. ◇

3.2 Value Range Analysis

It is well known that the tightness of a range analyzer is dependent on how the analyzed expression is formed [2] and if monotonicity can be exploited [12]. For example, $i(i{-}1)$ and $2^i{-}i$ are monotonic and non-negative for $0 \leq i \leq n$. Without using monotonic properties, bounds can be very loose, e.g. $[0,n]([0,n]{-}1) = [0,n][-1,n{-}1] = [-n,n^2{-}n]$ and $2^{[0,n]} - [0,n] = [1{-}n,2^n]$, respectively. Techniques such as symbolic forward differencing can be used to detect monotonicity for improving range analysis [12, 30]. However, difference methods are symbolic and iterative and must be applied to every separate (index) expression to be bounded. In contrast, our unified approach generates normal forms and is straightforward to integrate in a compiler framework. First, our GIV recognition algorithm uses the $\mathcal{CR}$ operator and CR algebra to calculate CR normal forms for expressions over an index space

```
p = A; q = B; k = m = 0;
for (i=0; i<=n; i++)
  if (C[k+2]) {   // Path 1
    for (j=0; j<i; j++)
      *p++ = *q++;
    k += 2;
  } else {        // Path 2
    *p++ = 0;
    q += i;
    k += m;
    m += 2;
  }
```

(a) *Example Loop Nest*

A_{p_1} *(Path 1)*
p $= \{A,+,0,+,1\}_i$
q $= \{B,+,0,+,1\}_i$
k $= \{0,+,2\}_i$
m $= 0$

A_{p_2} *(Path 2)*
p $= \{A,+,1\}_i$
q $= \{B,+,0,+,1\}_i$
k $= 0$
k $= \{0,+,0,+,2\}_i$
m $= \{0,+,2\}_i$

(b) *CR Forms*

Minimum
p $\geq$ A
q $\geq \{B,+,0,+,1\}_i$
k ≥ 0
m ≥ 0

Maximum
p $\leq \{A,+,1,+,1\}_i$
q $\leq \{B,+,0,+,1\}_i$
k $\leq \{0,+,2,+,2\}_i$
m $\leq \{0,+,2\}_i$

(c) *Min/Max CR Forms*

Minimum
p $\geq$ A
q $\geq$ &B$[(i^2{-}i)/2]$
k ≥ 0
m ≥ 0

Maximum
p $\leq$ &A$[(i^2{+}i)/2]$
q $\leq$ &B$[(i^2{-}i)/2]$
k $\leq i^2{+}i$
m $\leq 2i$

(d) *Closed Min/Max*

$\Delta_i\Phi_i$ *Strides*
A[] $= 0, \{0,+,1\}_i$
B[] $= \{0,+,1\}_i$
C[] $= 0, \{0,+,2\}_i$

(e) *Array Strides*

$L\Phi_i..U\Phi_i$ *Bounds*
A$[0..(n^2{+}n/2)]$
B$[0..(n^2{-}n/2)]$
C$[2..n^2{+}n+2]$

(f) *Array Bounds*

Figure 6: Analysis of an Example Code Segment Containing Conditional Updates

by reusing the CR forms of GIVs. Then, we use the stepping functions defined in Section 2.4 to test monotonicity of expressions for value range analysis. The lower and upper bound of a CR form are defined as follows.

DEFINITION 7. *The* lower bound $L\Phi_i$ *of a multivariate 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 multivariate 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*

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

Monotonicity of the CR forms of characteristic functions (polynomials and geometric series) is determined by the auxiliary function M. The bounds can be formed for polynomial and geometric series (and combinations thereof), such as for GIVs, affine expressions, polynomial expressions, and shift operations (e.g. 1<<i and N>>i).

EXAMPLE 6. *Consider* $k = (i^2{-}i)/2$ *with* $0 \leq i \leq n$*. The range of* k *is* $[L\{0,+,0,+,1\}_i, U\{0,+,0,+,1\}_i] = [0, n^2{-}n]$*, where* $\Phi(k) = \{0,+,0,+,1\}_i$*, which is exact, in contrast to the range* $[0,n]^2 - [0,n] = [-n, n^2]$ *derived using "traditional" methods [12].* ◇

EXAMPLE 7. *Consider expression* $n*j+i+2*k+1$ *with CR form* $\{\{1,+,n\}_j,+,1,+,2\}_i$ *from Example 1, where* $0 \leq i \leq n$*,* $0 \leq j \leq m$*, and* $k = (i^2{-}i)/2$*. The range is*

$$\begin{aligned} &[\, L\{\{1,+,n\}_j,+,1,+,2\}_i \;,\; U\{\{1,+,n\}_j,+,1,+,2\}_i \,] \\ &= [\, L\{1,+,n\}_j \;,\; U(\{1,+,n\}_j + n^2) \,] \\ &= [\, L\{1,+,n\}_j \;,\; U(\{n^2+1,+,n\}_j) \,] \\ &= [\, 1 \;,\; n^2 + m\,n + 1 \,] \end{aligned}$$

which is exact. ◇

After bounding the CR forms, standard lower and upper bound relations [12, 25, 30] are applied on the resulting expression using (symbolic) bounds on loop-invariant variables.

3.3 Array Region Analysis

By statically determining the bounds of accesses to an array region many code optimizations can be applied, such as eliminating dynamic array bounds checking, locality optimizations, prefetching, and blocking. Array region analysis is also used by methods for dependence testing.

Our array region analysis method exploits the L and U bounds to determine the range of values of affine and nonlinear expressions in rectangular or non-rectangular loop nests by applying the bounds to the set of CR forms of conditionally updated (pointer) variables. This is illustrated with two examples.

EXAMPLE 8. *Consider the* `Lsp_Az` *code segment shown in Figure 5(a). The array index bounds of the* f[] *array stores in the innermost loop of the triangular loop nest are determined as follows:*

$$\begin{aligned} &L\{\{\mathsf{f}{+}2,+,1\}_i,+,-1\}_j \\ &\quad = L\,\mathcal{CR}_j^{-1}(\{\{\mathsf{f}{+}2,+,1\}_i,+,-1\}_j)[j \leftarrow \{0,+,1\}_i] \\ &\quad = L(\{\mathsf{f}{+}2,+,1\}_i - \{0,+,1\}_i) \\ &\quad = \mathsf{f}{+}2 \\ &U\{\{\mathsf{f}+2,+,1\}_i,+,-1\}_j \\ &\quad = U\{\mathsf{f}{+}2,+,1\}_i \\ &\quad = U\,\mathcal{CR}_i^{-1}\{\mathsf{f}{+}2,+,1\}_i[i \leftarrow 3] \\ &\quad = \mathsf{f}{+}5 \end{aligned}$$

Therefore, it can be determined that the array accesses are within the bounds f$[2..5]$*. Note that the iteration space is triangular and that the bound on the inner loop variable* $j \leq i$ *is given by the CR form* $\{0,+,1\}_i$ *used in the derivation of the lower bound.* ◇

EXAMPLE 9. *Consider the example shown in Figure 6. Conventional value range analysis fails to determine accurate bounds for this example, due to conditional flow, nonlinear operations, and pointer arithmetic. In contrast, our array region analysis determines the bounds of the array accesses shown in Figure 6(f).* ◇

3.4 Nonlinear Dependence Testing

Our set of dependence tests do not require the application of IVS or array recovery to disprove dependence in a loop nest. The dependence tests are directly applied to the PADs of pointer dereferences and the CR forms of the array accesses constructed by the GIV recognition algorithm presented in Section 3. After constructing the CR forms and PADs, dependence testing can be applied to the CR forms and PADs, because these forms can be manipulated with the CR algebra in the derivation of the lower L and upper U bounds to determine (symbolic) dependence directions. The dependence tests are applicable to loops with conditionally updated variables and pointers by constructing a dependence system utilizing the *min* and *max* bounding functions over the set of CR forms of a conditionally updated variable.

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

Figure 7: Example Loop with Conditional Pointer Update

Our nonlinear dependence tests are described below.

3.4.1 Nonlinear GCD Test

Because polynomial GIVs are divisible using an algorithm that efficiently operates on the CR forms of polynomial GIVs [61], we implemented a nonlinear GCD test based on polynomial division. This test is also applicable to testing with affine expressions. Because affine expressions have very simple CR forms, the performance of the algorithm is similar to the performance of the conventional GCD test. The nonlinear GCD test is not applicable for array dependence problems involving geometric GIVs and conditionally updated variables. However, since this is an inexpensive test, it should be applied first.

3.4.2 Nonlinear Value Range Test

This nonlinear range test is similar to Blume and Eigenmann's range test algorithm [11], which verifies whether array reads and writes are performed on non-overlapping array regions. However, our approach with CR forms permits dependence testing in the presence of conditional control flow and pointer arithmetic as is demonstrated in the following example.

EXAMPLE 10. *Consider the example loop shown in Figure 7. The bounding functions of the PAD of pointer* p *are*

$$\mathsf{A} \le \mathsf{p} \le \{\mathsf{A}, +, 1\}_i$$

Therefore, the difference between the memory locations accessed by p *and* q *can be determined by*

$$L(\{\mathsf{A}+2\mathsf{n}, +, -1\}_i - \{\mathsf{A}, +, 1\}_i) = L(\{2\mathsf{n}, +, -2\}_i) = 2 > 0$$

where $\{\mathsf{A}, +, 1\}_i$ *is the PAD* $\{\mathsf{A}, +, 1\}_i = max(\mathsf{A}, \{\mathsf{A}, +, 1\}_i)$ *of* p *derived from the conditional flow in the loop and* $\{\mathsf{A}+2\mathsf{n}, +, -1\}_i$ *is the PAD of* q*. Because the difference is positive, the array access are non-overlapping and there is no dependence.* ◇

3.4.3 Nonlinear EVT

This nonlinear dependence test is based on the Banerjee bounds test [8], also known as the extreme value test (EVT). The test computes direction vector hierarchy information by performing symbolic subscript-by-subscript testing for multidimensional loops. The test builds the direction vector hierarchy by solving a set of dependence equations. Our nonlinear EVT can determine absence of dependence for a larger set of dependence problems compared to standard EVT, by including common nonlinear forms. The standard EVT and most other tests require affine closed forms, while our test can handle conditional induction variable updates, pointer arithmetic, and polynomial GIVs.

EXAMPLE 11. *Consider the example triangular loop nest depicted in Figure 8(a) with dependence equations shown in Figure 8(b). Note that pointers* p *and* q *read and write to the same array* A*. The equations are computed using the PAD forms for the* p *and* q *pointer updates using our extended GIV recognition algorithm. The normalized dependence equation can be rewritten in multivariate CR normal form*

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

```
p = q = A;
for (i=0; i<n; i++) {
  for (j=0; j<=i; j++)
    ∗q += ∗++p;
  q++;
}
```

(a) *Loop Nest*

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

(b) *Dependence Equations*

Figure 8: Example Dependence System

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

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

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

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

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

Since zero does no lie between $-\infty$ *and* -2*, our nonlinear extreme value test determines that there is no flow dependence.* ◇

3.5 Evaluation

We compared the capabilities of several dependence tests with respect to array-based testing on affine and nonlinear subscripts, whether closed-form subscripts and loop bounds are required (by applying IVS, i.e. conditionally updated variables are not permitted), and if dependence testing can be applied to pointer arithmetic. The results are shown in Table 1. The table is not meant to be exhaustive or complete. Section 4 gives an extensive overview of related work on dependence testing, GIVs, value range analysis, pointer analysis, and array region analysis.

	Affine subscripts	*Nonlinear subscripts*	*Closed forms*	*Pointer arithmetic*
EV	inexact	no	required	no
VR	inexact	yes	required	no
FM	exact	no*	required	no
ME	inexact	yes	no	no
CR	inexact	yes	optional	yes

EV = Extreme Value test [8]
VR = Value Range test (Polaris) [10]
FM = Fourier-Motzkin (e.g. Omega test [45])
ME = Monotonic Evolution [59]
CR = CR-based dependence testing

Note*: Omega test incorporates work on nonlinear testing

Table 1: Dependence Tests

4. RELATED WORK

GIVs and IVS:

Most compiler analysis methods are capable of recognizing linear induction variables [1, 3, 39, 57, 64]. Generalized induction variables (GIVs) [5, 20, 21, 29, 56] are more general in that they form polynomial and geometric progressions through loop iterations. The demand-driven sequence classification method by Gerlek et al. [27] and Haghighat's symbolic differencing method [29, 30] are powerful GIV recognition methods. However, both methods require extensive symbolic manipulation. In contrast, our approach is not computationally expensive. In addition, symbolic differencing is not safe when the order of the polynomial GIVs cannot be determined in advance [54]. Haghighat [31] also shows how conditionally updated GIV can be detected when the net effect of the updates is identical for all paths through the loop body. However, it is not clear how conditionally updated variables can be analyzed to enable value range analysis and dependence testing.

Dependence Testing:

Previous work on array-based dependence analysis by Banerjee [8], Wolfe [57], Goff, Kennedy, and Tseng [28], and Kong and Psarris [35, 42], have introduced tests that analyze the equality constraints of a dependence system. Compared to methods for testing inequality constraints, these test are more efficient and typically serve preliminary roles in a dependence analysis system. A linear programming method for dependence testing was introduced by Dantzig in [17] based on Fourier-Motzkin variable elimination. The technique is later improved by Pugh in [45] to an integer programming method. These tests work by systematically eliminating variables from a system of inequalities in order to determine if dependence is possible. The Omega test also implements a variation of a nonlinear test using Presburger formulas [46, 47]. Other nonlinear tests that combine the analysis of both equality and inequality constraints are introduced by Maydan, Hennessy, and Lam [38] and Wolfe [58]. In [11] and later extended in [10, 13] Blume and Eigenmann describe a dependence test based on symbolic analysis that disproves carried dependences by evaluating ranges of permuted loops nest. Collard et al. describe a method for flow-sensitive array data-flow analysis [16]. Mateev et al. [37] describe a technique for dependence analysis that verifies the legality of a program transformation. The technique however is only as powerful as the underlying symbolic engine it uses to determine the legality of the transformation. Wu et al. [59] developed a method for dependence testing that does not require the computation of closed forms for GIVs and IVS can be delayed until after dependence testing, as in our method. However, their method cannot handle cases that our method can handle, where induction variable step sizes are relevant, such as in TRFD. In addition, our method enables the conversion to closed forms directly without requiring the application of an additional IVS algorithm.

Pointer Analysis:

Allen and Johnson [4] 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 [39] mentions the regeneration of array indexes from pointer-based array traversal, but no explicit details are given. Franke and O'Boyle [26] developed a compiler transformation to convert pointer-based accesses to explicit array accesses for array recovery. Array recovery makes the code amenable to data flow analysis [19], loop optimizations [39, 57, 64], loop scheduling for power reduction [60], and DSP architecture-specific optimizations that require explicit array references, e.g. [9, 15]. 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. Furthermore, pointer assignments, apart from initializations to some start element of the array to be traversed, are not permitted. In contrast, our algorithm can handle non-rectangular loops, more general pointer initializations, and the most common types of data dependent and independent pointer updates.

Value Range Analysis:

In [13, 12] Blume and Eigenmann describe techniques for computing variable ranges and propagating those computed values based on control flow information. In [24] Fahringer, and later in [23, 25] 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, our method adopts the uniform CR-based representation to propagate bounding functions on expressions rather than fixed symbolic bounds. As a result, our value range information is more accurate and faster to compute.

Array Region Analysis:

Pugh and Wonnacott [46] use a set of constraints to describe linear array data flow problems and describe how to solve them by the Fourier-Motzkin variable elimination method. Gu et al. [34] improved array region analysis by analyzing conditional array regions. They also describe array regions by a set of convex regions. Rugina and Rinard [48] developed a method for symbolic bounds analysis of accessed memory regions. The method builds a symbolic constraint system over the lower and upper bounds of pointers, array indices, and accessed memory regions, which are represented by polynomials over the input parameters of a procedure. The method then solves this symbolic constraint system using ILP. While the method is quite powerful, the method is limited to the analysis of non-negative variables and parameters and is also restricted to the analysis of linear induction variables.

5. CONCLUSIONS

In this paper we presented a new unified framework for compiler analysis based on the chains of recurrences algebra. We have shown that the algebraic approach provides a powerful framework for developing analysis algorithms for restructuring compilers. The analysis of induction variables and pointers does not require the computation of closed forms, which enables the application of data dependence testing and value range analysis to loops with conditionally updated variables and pointer arithmetic.

Because the chains of recurrences representation of multivariate indexing functions and induction variables easily facilitates monotonic analysis, exact symbolic value range bounds can be derived without requiring expensive symbolic differencing techniques. The determination of exact value range bounds of symbolic expressions and pointer arithmetic has many applications in compiler analysis. The method presented is complementary to current interprocedural value range propagation algorithms and array-based data dependence algorithms. Therefore, we believe that the unified approach presents new avenues for developing more powerful compiler analysis algorithms, thus enabling more aggressive optimizations on loops with more common forms of conditional control flow and loops incorporating pointer arithmetic commonly found in DSP codes.

Part of this project is to implement the CR-based algorithms for nonlinear data dependence testing and symbolic analysis in SUIF and Polaris. Our current implementation in SUIF includes the GIV recognition algorithm and our nonlinear data dependence tests. The software for SUIF and Polaris will be made available from the site: `http://www.cs.fsu.edu/~birch/research/crDemo3.php` The site also includes an interactive demonstration of the capabilities of the CR-based analysis methods.

6. REFERENCES

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

[2] 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.

[3] ALLEN, F., COCKE, J., AND KENNEDY, K. Reduction of operator strength. In *Program Flow Analysis* (New-Jersey, 1981), S. Muchnick and N. Jones, Eds., Prentice-Hall, pp. 79–101.

[4] 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.

[5] AMMERGUALLAT, Z., AND HARRISON III, W. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation* (White Plains, NY, 1990), pp. 283–295.

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

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

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

[9] BASU, A., LEUPERS, R., AND MARWEDEL, P. Array index allocation under register constraints in DSP programs. In *12th Int. Conf. on VLSI Design* (Goa, India, 1999).

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

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

[12] 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.

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

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

[15] CINTRA, M., AND ARAUJO, G. Array reference allocation using SSA-form and live range growth. In *Proc. of the ACM SIGPLAN 2000 LCTES* (Vancouver, June 2000), pp. 26–33.

[16] COLLARD, J.-F., BARTHOU, D., AND FEAUTRIER, P. Fuzzy array dataflow analysis. In *In proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming* (1995), pp. 92–101.

[17] DANTZIG, G. B., AND EAVES, B. C. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory 14*, 14 (1973), 288–297.

[18] DAVIDSON, W., AND JINTURKAR, S. Memory access coalescing: A technique for eliminating redundant memory accesses. In *proceedings of the SIGPLAN'94 Symposium on Programming Language Design and Implementation* (1994), pp. 186–195.

[19] DUESTERWALD, E., GUPTA, R., AND SOFFA, M. A practical data flow framework for array reference analysis and its use in optimizations. In *Proc. of the SIGPLAN Conference on Programming Languages Design and Implementation* (Albuquerque, New Mexico, 1993), pp. 67–77.

[20] EIGENMANN, R., HOEFLINGER, J., JAXON, G., LI, Z., AND PADUA, D. Restructuring Fortran programs for Cedar. In *Proc. of ICPP'91* (St. Charles, Illinois, 1991), vol. 1, pp. 57–66.

[21] EIGENMANN, R., HOEFLINGER, J., LI, Z., AND PADUA, D. Experience in the automatic parallelization of four perfect-benchmark programs. In 4^{th} *Annual Workshop on Languages and Compilers for Parallel Computing, LNCS 589* (Santa Clara, CA, 1991), Springer Verlag, pp. 65–83.

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

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

[24] FAHRINGER, T., AND SCHOLZ, B. Symbolic evaluation for parallelizing compilers. In 11^{th} *ACM International Conference on Supercomputing* (New York, 1997), ACM Press, pp. 261–268.

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

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

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

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

[29] HAGHIGHAT, M., AND POLYCHRONOPOULOS, C. Symbolic program analysis and optimization for parallelizing compilers. In 5^{th} *Annual Workshop on Languages and Compilers for Parallel Computing, LNCS 757* (New Haven, Connecticut, 1992), Springer Verlag, pp. 538–562.

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

[31] 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.

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

[33] HAVLAK, P., AND KENNEDY, K. Experience with interprocedural analysis of array side effects. *Supercomputing* (1990), 952–961.

[34] J. GU, Z. L., AND LEE, G. Experience with efficient array data flow analysis for array privatization. In *Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (June 1997), pp. 157–167.

[35] KONG, X., KLAPPHOLZ, D., AND PSARRIS, K. The I Test: An improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems 2*, 3 (1991), 342–349.

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

[37] MATEEV, N., MENON, V., AND PINGALI, K. Fractal symbolic analysis. In *proceedings of the International Conference on Supercomputing (ICS)* (2001), pp. 38–49.

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

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

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

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

[42] PSARRIS, K., KONG, X., AND KLAPPHOLZ, D. The direction vector I Test. *IEEE Transactions on Parallel and Distributed Systems 4*, 11 (November 1993), 1280–1290.

[43] 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).

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

[45] PUGH, W. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing* (1991), pp. 4–13.

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

[47] PUGH, W., AND WONNACOTT, D. Nonlinear array dependence analysis. In *In 3rd Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* (1994).

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

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

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

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

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

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

[54] 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.

[55] 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.

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

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

[58] WOLFE, M., AND TSENG, C.-W. The power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems 3*, 5 (1992), 591–601.

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

[60] YU, T. Z., CHEN, F., AND SHA, E. H.-M. Loop scheduling algorithms for power reduction. In *Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (Seattle, Washington, May 1998).

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

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

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

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