An Empirical Evaluation of Chains of Recurrences for Array Dependence Testing

J. Birch   R.A. van Engelen[*]   K.A. Gallivan   Y. Shou
Department of Computer Science and School of Computational Science
Florida State University
Tallahassee, FL 32306-4530
{birch,engelen,gallivan,shou}@cs.fsu.edu

ABSTRACT

Code restructuring compilers rely heavily on program analysis techniques to automatically detect data dependences between program statements. Dependences between statement instances in the iteration space of a loop nest impose ordering constraints that must be preserved in order to produce valid optimized, vectorized, and parallelized loop nests. This paper evaluates a new approach for fast and accurate nonlinear array dependence testing using Chains of Recurrences (CRs). A flow-sensitive loop analysis algorithm is presented for constructing the CR forms of array index expressions. Unlike other approaches, the CR forms are directly integrated into a standard dependence test to solve nonlinear CR-based dependence equations. To study the coverage and performance of the proposed CR-based enhancements of a standard test, we chose the inexact Banerjee test. We implemented a new CR-based Banerjee test in the Polaris compiler and compared the results to the Omega test and Range test on a set of SPEC and LAPACK Benchmark programs. The experimental results suggest that a CR enhancement can dramatically increase the effectiveness of a dependence test without a significant cost increase. More surprisingly, the findings indicate that the enhanced test exceeds the capabilities of the Omega and Range tests for many nonlinear dependence relations detected in the PERFECT Club and LAPACK Benchmark programs.

Categories and Subject Descriptors

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

General Terms

Compiler Algorithms, Theory, Experimentation

Keywords

Dependence Testing, Loop Optimization, Chains of Recurrences

1. INTRODUCTION

With the increasing demand on general-purpose processors with multimedia extensions and the advent of newer generations of multi-core processor systems entering the personal computing market, automatic vectorizing and parallelizing compilers are no longer relegated to specialized super-computing systems. They are now a basic necessity to develop software that harnesses the power of general computing systems.

Consequently, advanced compiler techniques for vectorization and parallelization that were traditionally targeting vector processors and massively parallel systems have now become mainstream and can be found in open source and proprietary compilers.

An optimizing compiler relies heavily on data dependence analysis techniques to vectorize and parallelize program codes automatically and effectively, and to apply other type of loop optimizations to increase instruction-level parallelism and improve cache locality. Data dependences between statements restrict the reordering and optimization of performance-critical code fragments, such as loops. Thus, data dependence analysis is the discovery of the partial ordering of program statements at compile time. The true dependences between reordered statements must be preserved in order to produce semantically valid program optimizations.

Unfortunately, solving an affine dependence problem is equivalent to finding the solution to a set of Diophantine equations, which is known to be NP-complete [14]. However, many effective and practical dependence solvers are in use today, often trading off accuracy for speed of the analysis [17]. Lower accuracy results in a larger set of assumed dependences, thereby restricting the applicability of optimizing transformations. In addition, non-affine array index expressions require more specialized nonlinear dependence analysis techniques [6, 20] and array dependences cannot always be effectively analyzed in pointer-based C code due to assumed pointer aliases [12, 17].

Recently, there has been a series of research contributions proposing the use of the Chains of Recurrences (CRs) in compiler analysis [4, 5, 26, 27, 29, 30, 31], using variations of the CR-based induction variable (IV) recognition algorithm proposed by Van Engelen [28] based on the CR algebra [34]. Specifically data dependence analysis has been identified as an area of potential impact for CR-based enhancement [30].

The CR-based enhanced approaches have several advantages. Firstly, *induction variable classification* [13] is not required to analyze IV expressions. Secondly, source-to-source normalizing code changes, such as *induction variable substitution* (IVS) [16] and *array recovery* [11, 12] are not required. The simplified CR forms of IV expressions such as array index expressions and pointer arith-

```
        DO 109 JL = 1, I2K
          DO 103 JJ = JL, NPTS, I2KP
            DO 103 MM = 1, MTRN
              JS2 = (JJ-1)*NSKIP
              JS = JS2 + (MM-1)*MSKIP+1
              H = DATA(JS) - DATA(JS+I2KS)
              DATA(JS) = DATA(JS) + DATA(JS+I2KS)
              DATA(JS+I2KS) = H
103        CONTINUE
109    CONTINUE
```

Figure 1: Ocean Benchmark Fragment: OCSI

metic [31] are normal forms [27]. This makes the approach more amenable to the analysis of lower-level code representations in compilers. Thirdly, a CR-based approach applies flow-sensitive IV analysis [26, 30] with monotonic IV progression analysis that is used to determine accurate value bounds on nonlinear IV expressions [5]. These are essential properties to enhance the accuracy of nonlinear array dependence testing. Finally, the analysis is applied directly to array index expressions containing (nonlinear) IV recurrences, instead of the closed-form index expressions of the IV recurrences obtained after IVS, which enables dependence testing on a class of loops with IV recurrences that do not have closed forms. This class of loop was previously deemed impossible to analyze.

This paper evaluates the implementation of a new CR-based enhancement of the Banerjee test [3, 32]. This test is also known as the Extreme Value test and handles only affine pair-wise dependence problems. Therefore, it is known to be less powerful than exact tests on coupled subscripts. Because of its limitations it is a good candidate to demonstrate the impact of CR-based enhancements. We compared the CR-enhanced test to the Omega test and Range test on a set of benchmark programs. The experimental results suggest that a CR enhancement can dramatically increase the effectiveness of a dependence test without a significant cost increase. More surprisingly, the findings indicate that the enhanced test exceeds the capabilities of the Omega and Range tests for many nonlinear dependence relations detected in the PERFECT Club and LAPACK Benchmark programs.

The remainder of this paper is organized as follows. In Section 2 we present motivating examples and compare our approach to related work. Section 3 describes the CR-based enhancements of the Banerjee test to solve linear and nonlinear dependence equations.In Section 4 we present our results. Section 5 discusses the results and outlines possible future research directions. Finally, Section 6 concludes with a summary of our findings.

2. COMPARISON TO RELATED WORK

Popular data dependence tests, including GCD [32], Banerjee [3, 32], and Fourier Motzkin [10], restrict their analyses to array subscripts that are expressed as *affine*, i.e. linear, combinations of loop indices [17]. If only affine expressions are permissible, dependence analysis is restricted and code optimizations are unnecessarily limited.

Figure 1 shows an example code from OCEAN of the Perfect Club Benchmarks that would not be handled by these tests. Firstly, array dependence testing requires the elimination of IVs in the array index expression through IVS. Secondly, after applying IVS to the subscripts of DATA reveals nonlinear expressions induced by the non-constant lower bound of JJ. While this code is parallelizable, accounting for as much as 44% of the code's sequential execution time [8], it cannot be automatically parallelized due to assumed dependences.

```
        DO 100 I = 1, M
          DO 90 J = 1, I
            IJ=IJ+1
            IJKL=IJKL+I-J+1
            DO 80 K = I+1, M
              DO 70 L = 1, K
                IJKL=IJKL+1
                XIJKL(IJKL)=XKL(L)
70          CONTINUE
80        CONTINUE
          IJKL=IJKL+IJ+LEFT
90      CONTINUE
100  CONTINUE
```

Figure 2: TRFD Benchmark Fragment: OLDA

There are many examples in scientific codes of nonlinear expressions [19]. To handle these cases a compiler could employ a symbolic analyzer, where the analyzer would discover at compile time the properties of variables known otherwise only at run time. In [16] Haghighat and Polychronopoulos describe such a system where the symbolic analysis focuses on the recognition of *generalized induction variables* (GIVs), variables characterized by the polynomial and geometric progressions in the loop iteration space. The technique however, requires extensive compiler support for symbolic manipulation and an algorithm for dependence testing is not provided, only suggested. In [21] Menon et al. describe a technique for dependence analysis that verifies the legality of program transformation, basing program ordering constraints on program semantics. The authors, however, point out that the technique is only as powerful as the underlying symbolic engine it uses. In [8], and later extended in [6, 9], Blume and Eigenmann describe an implementation of a dependence test based on symbolic analysis that disproves loop-carried dependences through the evaluation of subscript expression ranges. Unlike the Banerjee test [3], which also evaluates ranges, the test is able to filter dependences where separate array ranges overlap but memory accesses are actually interleaved. The test filters many nonlinear cases and is much more powerful than the Banerjee test, but requires non-trivial analyses of expression monotonicity and compiler support.

One of the most popular dependence test used in practice is the Omega test [24, 25] due to its ability to provide exact answers for linear dependence problems and its ability to handle some nonlinear problems. Because the test does not require extensive compiler support and is exact for linear dependence problems it is considered quite adequate. However the Omega test, which has exponential worst-case complexity, may be prohibitively slow depending on the application compiled [23]. Moreover, its analysis of nonlinear expressions is approximate; the test attempts to prove dependence impossible while avoiding analysis of the nonlinear terms. In the example code shown in Figure 2 from TRFD of the Perfect Club Benchmarks parallelization could not take place, even after IVS, due to the presence of a third-order multivariate polynomial subscript in XIJKL, which prevents the discovery of the non-existence of the assumed output dependence on XIJKL. Note that if a dependence test could independently determine the monotonicity of the XIJKL memory writes in the loop iteration space, it would allow the compiler to automatically parallelize the code in Figure 2.

The dependence test based on *monotonic evolutions* [33] considers monotonic progressions of array index expressions to disprove dependence. However, their specialized dependence test cannot disprove dependence when the direction of the monotonicity alternates between inner and outer loop iterations, as is the case with OLDA in Figure 2.

The CR algebra was introduced by Eugene Zima to optimize function evaluations on unit-distance grids [34]. In [28] van Engelen extended the algebra with new rules for IV analysis and presented an efficient CR-based algorithm to analyze coupled IV recurrences with linear, polynomial, and geometric progressions in loops. This work was further extended in [30] with a flow-sensitive IV analysis framework for CR-enhanced dependence testing methods. In this latter work a CR form is essentially an abbreviated form for the recurrence of an IV. The expressiveness of the CR form, the extended CR algebra rules, and CR value range analysis [5] provides an economical framework for analyzing and manipulating both linear and nonlinear IVs that is independent of the underlying low-level code representation.

More specifically, in a loop under CR-based IV analysis, a CR form represents the recurrence relation of an IV with respect to the loop index or loop label i. The notation of a basic recurrence of a function for the one-dimensional case is:

$$\Phi_i = \{\iota_0, \odot_1, h_1\}_i \qquad , \qquad (1)$$

where $\odot_1 \in \{+, *\}$ and ι_0 is the initial value of the recurrence and h_1 is an update function, i.e. a stride. When the stride is a function that can be expressed as a basic recurrence, whose stride is also expressed as a recurrence and so on, a chain of recurrences is formed:

$$\Phi_i = \{\iota_0, \odot_1, \{\iota_1, \odot_2, \ldots, \{\iota_{k-1}, \odot_k, h_k\}_i\}_i\}_i \qquad , \qquad (2)$$

where $\odot_j \in \{+, *\}$ and ι_j are initial values (possibly symbolic loop-invariant expressions). The nested CR form (2) is usually written in flattened form:

$$\Phi_i = \{\iota_0, \odot_1, \iota_1, \odot_2, \ldots, \iota_{k-1}, \odot_k, h_k\}_i \qquad . \qquad (3)$$

When $\odot_j = +$, for all $0 \leq j \leq k$, the closed form function of the CR is a polynomial of order k. The semantics of the one-dimensional k-order CR form is defined by the following pseudo-code loop template to compute the value sequence $f[i]$ of the CR form for $i = 0, \ldots, n$ using variables r_j initialized with the (symbolic) CR coefficients:

$$
\begin{aligned}
r_0 &:= \iota_0 \\
r_1 &:= \iota_1 \\
&\;\;\vdots \\
r_{k-1} &:= \iota_{k-1} \\
\end{aligned}
$$
for $i = 0$ **to** n **do**
$$
\begin{aligned}
f[i] &:= r_0 \\
r_0 &:= r_0 \odot_1 r_1 \\
r_1 &:= r_1 \odot_2 r_2 \\
&\;\;\vdots \\
r_{k-1} &:= r_{k-1} \odot_k h_k \\
\end{aligned}
$$
end for

For example, the value sequence of the CR form $\{1, *, 1, +, 1\}$, where $r_0 = r_1 = 2 = h_2 = 1$, is computed for $n = 5$ as follows:

i	0	1	2	3	4	5
r_0	1	1	2	6	24	120
r_1	1	2	3	4	5	6
h_2	1	1	1	1	1	1

During each iteration i, the value at $f[i]$ is set to r_0, r_0 is set to $r_0 * r_1$, and r_1 is set to $r_1 + 1$, thereby producing the well-known factorial sequence $f[i] = i!$. Multi-variate CR forms (MCR) are defined similarly, using multi-dimensional loops to compute their value sequences [2].
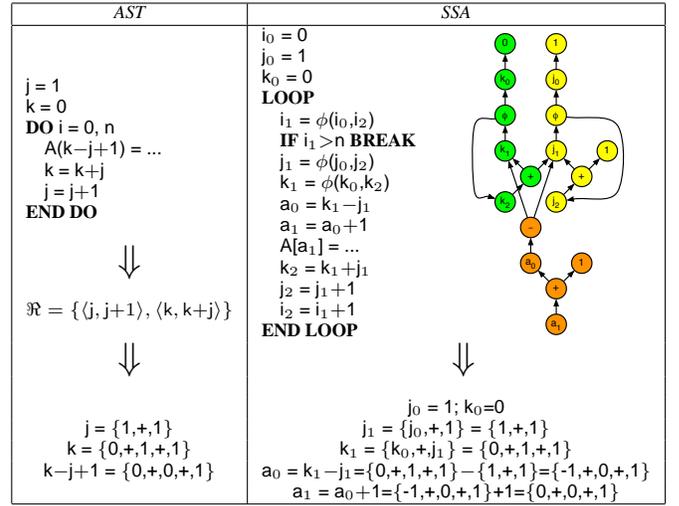

Figure 3: Methods for CR-Based IV Analysis on AST and SSA Forms for an Example Loop

The power of the CR algebra is that it allows CR forms to be combined, simplified, and expressed as closed-forms using a collection of rewrite rules. For loop analysis, the loop template is effectively used in reverse: IV recurrences are detected and rewritten in CR form. The CR algebra is then applied to simplify recurrence expressions and find closed forms, when required.

The CR algebra is defined by a list of rewrite rules [31] applied to nested recurrences, a selection of which is shown below:

Expression		Rewrite
$-\{\iota_0, +, h_1\}_i$	$\Rightarrow$	$\{-\iota_0, +, -h_1\}_i$
$c * \{\iota_0, +, h_1\}_i$	$\Rightarrow$	$\{c * \iota_0, +, c * h_1\}_i$
$\{\iota_0, +, h_1\}_i \pm c$	$\Rightarrow$	$\{\iota_0 \pm c, +, h_1\}_i$
$\{\iota_0, +, h_1\}_i \pm \{\kappa_0, +, g_1\}_i$	$\Rightarrow$	$\{\iota_0 \pm \kappa_0, +, h_1 \pm g_1\}_i$.

Consider the example loop shown in Figure 3. IV analysis on *abstract syntax trees* (ASTs) is similar to IV analysis on *single static assignment* (SSA) forms and proceeds in two stages: detection of the recurrence relations ($\Re$ in case of AST [28] and the operator graph for SSA [4, 26, 32]) followed by CR simplification of the recurrence forms using the CR algebra. Note that the CR form for j is $\{1, +, 1\}$, which is the linear function $j[i] = i + 1$. The CR form for k is $\{0, +, 1, +, 1\}$, which is a quadratic polynomial $k[i] = (i^2 + i)/2$. The A(k−j+1) subscript is simplified to

$$
\begin{aligned}
k - j + 1 &= \{0, +, 1, +, 1\} - \{1, +, 1\} + 1 \\
&= \{0, +, \{1, +, 1\}\} - \{1, +, 1\} + 1 \\
&= \{-1, +, \{1, +, 1\} - 1\} + 1 \\
&= \{0, +, \{0, +, 1\}\} \\
&= \{0, +, 0, +, 1\} \qquad .
\end{aligned}
$$

Because the CR algebra is closed under addition and multiplication of polynomial and geometric functions, our method is based on the computation of CR forms for array index expressions to solve CR-based nonlinear dependence equations. In contrast, other approaches apply IV recognition and IVS (e.g. with CR forms as in [4]) but do not use recurrences to enhance dependence testing.

When considering nonlinear dependence testing, it is important to compare the efficacy of value range bounding techniques. The reason is that most nonlinear dependence tests solve dependence equations by computing the interval of the equation's solution space to verify that a solution cannot exist (note that for disproving dependence an exact solution is not required, just the absence of a solu-

tion). For example, consider the loop nest and dependence equation from [30] shown in Figure 5.

Testing for flow dependence direction $(<, <)$ between the definition of $*q$ and the use of $*++p$, where p and q are represented as recurrences [31] requiring both to array A, requires the solution of

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

Computing the solution interval of this nonlinear equation (it has a second-order CR form polynomial), under the given loop constraints and dependence direction constraints, gives $0 \notin [-\infty, -2]$, thereby disproving flow dependence in the loop nest.

When value range bounds cannot be accurately determined for nonlinear expressions, a dependence equation solver will fail or provide inadequate coverage. Figure 4 compares our CR-based bounds analysis to other nonlinear value range bounding techniques. The figure shows different approaches proposed in the literature and used by compilers to compute the lower and upper bound of nonlinear expressions.

Consider for example the task of computing the value range of index expression $E = \mathsf{k}-\mathsf{j}+1$ of A in the example loop in Figure 3. A popular method for value range analysis is *interval arithmetic*, which is applied to closed-form expressions, e.g. obtained after IVS. The corresponding path is $(\Re, E) \overset{IVS}{\to} (\chi, E') \overset{simp}{\to} E'' \overset{bound}{\to} I'$. However, this method is known to be inaccurate for nonlinear expressions with coupled IVs. With the closed-form $E'' = (\mathsf{i}^2-\mathsf{i})/2$ evaluated on $\mathsf{i} = 0, \ldots, \mathsf{n}$ the interval is computed as $([0, \mathsf{n}]^2 - [0, \mathsf{n}])/2 = [-\mathsf{n}/2, \mathsf{n}^2/2]$ which is overly conservative compared to the actual range $[0, (\mathsf{n}^2-\mathsf{n})/2]$.

In [20] *symbolic differentiation* is proposed to increase the accuracy of interval arithmetic when an expression is monotonic in each direction of the iteration space. Monotonicity in the continuous domain implies monotonicity in the discrete domain. However, certain functions are monotonic in the discrete domain but not in the continuous domain. For example, consider $(x^2-x)/2$ where i in $E'' = (\mathsf{i}^2-\mathsf{i})/2$ is replaced with a continuous variable $0 \le x \le \mathsf{n}$. Then, $\frac{\partial}{\partial x}(x^2-x)/2 = x-\frac{1}{2}$, which is ≥ 0 for $x \ge \frac{1}{2}$. Thus, the expression is monotonic in the continuous domain only for $\mathsf{i} \ge 1$. Note that E'' is monotonic in the *discrete* domain for $\mathsf{i} = 0, \ldots, \mathsf{n}$.

Others [9, 15] proposed methods based on *symbolic differencing* in the discrete domain. A symbolic expression is iteratively differenced to determine monotonic properties. The corresponding path in the diagram is $(\Re, E) \overset{IVS}{\to} (\chi, E') \overset{simp}{\to} E'' \overset{df\text{-}bound}{\to} I$. For example, the symbolic difference series of $E'' = (\mathsf{i}^2-\mathsf{i})/2$ is i, $\mathsf{i}+1$, $\mathsf{i}+2$, etc., which is non-negative for $\mathsf{i} \ge 0$. In fact, differencing methods construct Newton series, which are known to be identical to the CR form Φ in Figure 4. Therefore, the *df-bound* arc's target interval I is identical to the range computations on Φ described in [5], but the path to reach I with CR forms is more direct.
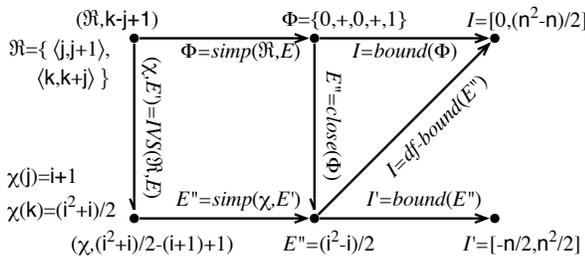

Figure 4: Comparison of Bounds Analysis Techniques

```
p = q = A;                    {0, +, 1}_{i^d} = {{1, +, 1, +, 1}_{i^u}, +, 1}_{j^u}
for (i=0; i<n; i++) {         0 ≤ i^d ≤ n − 1
   for (j=0; j<=i; j++)       0 ≤ i^u ≤ n − 1
      *q += *++p;             0 ≤ j^d ≤ i^d
   q++;                       0 ≤ j^u ≤ i^u
}
```
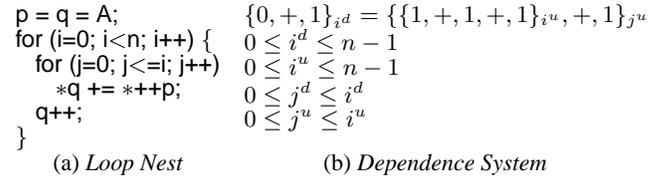
    (a) *Loop Nest*   (b) *Dependence System*

Figure 5: Example Dependence System

Once CR forms are obtained for IVs and index expressions, dependence testing based on CR value range bounding techniques appears to be a natural extension of the CR framework that does not require any ad-hoc code normalization, closed-form expression simplifications, or compute-intensive symbolic bounding methods.

3. A CR-ENHANCED BANERJEE TEST

To evaluate the CR enhancements on dependence testing, we implemented a CR-enhanced Banerjee test in the Polaris compiler. This section first discusses the techniques we use to construct CR forms in Polaris and then presents the dependence test implementation.

3.1 CR Construction in Polaris

The technique for constructing CR forms for IVs and IV expressions from ASTs was first published in [28] and later improved in [5, 29, 30] with more powerful flow-sensitive approaches to analyze larger classes of conditionally updated IVs.

The implementation of CR construction in Polaris is based on this earlier work [28]. It applies a depth first search and backward substitution technique on the body of a loop to determine the recurrence forms of coupled IVs with complexity $O(n)$, where n is the number of loop-body statements. The CR forms with *discordant* initial values and updates are then constructed using matching techniques and simplified using the CR simplification rules as described in the literature.

In the flow-sensitive approach applied to ASTs, multiple control-flow paths in a loop nest are recognized by program keywords, e.g. IF, ELSE, ENDIF in the case of FORTRAN, as described in [30]. When control-flow path forks are encountered, at least one, but possibly several new paths are created by duplicating previously created paths. The number of paths to duplicate is dependent on both the nesting level of the region and the number of *inclusive* traversable paths, where the term *inclusive* refers to paths that may still execute if the current path in question is executed. During this process the algorithm also keeps tracks of the paths that reach the specific array index expressions for flow-sensitive dependence analysis. Consider the FORTRAN code fragment:

```
DO J = 1, N
   IF (...)
      A(K) = ...
      K = K + 1
   ELSE
      T = T + 1
   ENDIF
ENDDO
```

Our approach ensures that K is determined to be strictly monotonic on the path through $A(\mathsf{K})$ even though it is not strictly monotonic in the loop iteration space. Note that this consideration bears some resemblance with *monotonic evolutions* [33], but differs in that we keep the IV stride information that is critical for the success of a

dependence test for more complicated cases. Consider for example the loop:

```
DO I = 0, N
    R(A+B) = . . . R(I+B) . . .
    IF (. . . )
        A = A + 2
    ELSE IF (. . . )
        A = A − 1
    ENDIF
    A = A + 2
    B = B + I
ENDDO
```

The I-loop body has three paths on which recurrence updates are found:

P_0:A = A + 2 P_1:A = A + 1 P_2:A = A + 4
     B = B + I   B = B + I   B = B + I

With I having a starting value of 0 and a constant stride of 1, the first step of CR construction is to replace the loop index I in P_0, P_1, and P_2 with its CR form $\{0,+,1\}_I$:

P_0:A = A + 2 P_1:A = A + 1 P_2:A = A + 4
    B = B + $\{0,+,1\}_I$ B = B + $\{0,+,1\}_I$ B = B + $\{0,+,1\}_I$

Next the IV's found in P_0, P_1, and P_2 are updated to their final CR forms:

P_0:A = $\{A_0, +, 2\}_I$ P_1:A = $\{A_0, +, 1\}_I$ P_2:A = $\{A_0, +, 4\}_I$
   B = $\{B_0,+,0,+,1\}_I$ B = $\{B_0,+,0,+,1\}_I$ B = $\{B_0,+,0,+,1\}_I$

Finally, because the potential update to A is not the same for each path, the multiple CRs created need to be replaced with minimum and maximum CR forms.

Min:A = $\{A_0, +, 1\}_I$ *Max*:A = $\{A_0, +, 4\}_I$
     B = $\{B_0,+,0,+,1\}_I$ B = $\{B_0,+,0,+,1\}_I$

After CR construction, occurrences of IVs and loop indices found in the array index of R are replaced with their CR forms and simplified. Subsequent passes for CR formation take place for outer loops until all loops have been analyzed. When all loops have been processed the array subscripts involved in the dependence are passed to the CR-enhanced Banerjee test.

3.2   CR-Enhanced Banerjee Test Design and Implementation

The Banerjee test [3, 32] is a well-known dependence test that disproves dependences based on the *Intermediate Value Theorem*: a continuous function over a closed interval can take every value between values defined by the endpoints. When the range of possible values for two array subscript expressions overlap, the test must assume the two expressions will take the same value, thus assuming a dependence.

Consider the general form of an affine dependence equation for an n-dimensional loop nest:

$$a_1 x_1 + a_2 x_2 + a_3 x_3 + \ldots + a_n x_n + b = 0 \quad ,$$

for loop indices x_i with coefficient a_i determined from the array index expression pair to be tested. The test computes a lower and upper bound for the left side of the dependence equation and then checks whether zero lies in the interval. If the interval does not include zero, no dependence exists.

The key idea in our CR-enhanced Banerjee test is to replace the terms in the dependence equation with the CR forms of each loop

Steps for Banerjee test using CR forms

1. If self-output dependence, test monotonicity of inner-most loop directly from the CR form of the index expression.
2. Apply bounds analysis to inner-most CR form.
3. If all CR forms have been tested, goto step 5
4. Compute bounds for CR form, only computing final bounds for the CR form if the bound is not dependent on an outer loop index. Goto step 1
5. Compute the final bounds of all CR forms.

Figure 6: Applying the Banerjee test on CR-Forms

index:

$$\Phi_1 + \Phi_2 + \Phi_3 + \ldots + \Phi_n + b = 0 \quad ,$$

and then apply CR bounds analysis to determine whether zero is included in the solution interval. In this way, we can apply dependence testing directly on the CR forms of array index expressions, for more details see [29, 30].

We have implemented an algorithm in Polaris [7] for computing the dependence direction vector hierarchy based on the CR-enhanced Banerjee test. In this implementation, we partition the CR forms of the dependence equation by loop index similar to the partitioning of the *learning functions* for trapezoidal loops [1, 3, 17]. This ensures that cancellation is promoted and separate but unnecessary direction vector analysis can be avoided. This requires handling the CRs of IVs in an order corresponding to the innermost to outermost loop. After replacing a CR with a bounds corresponding to an outer-loop CR form, we halt bounds analysis and promote the leftover term to its appropriate partition.

During this process we avoid unnecessary direction vector analysis for dependences. In certain cases, when the non-star direction vector elements correspond only to indices without a CR form, analysis of that direction vector may be avoided. The steps for applying the Banerjee test to CR forms is given in Figure 6.

We illustrate the process with an example. Consider the following simplified code fragment from the SDSI benchmark:

```
        DO 20 I = 1, 10
            DO 20 J = 1, N
                A(K) = . . .
                K = K + 2
20      CONTINUE
```

The self-output dependence for array A has CR form:

$$\{K_0, +, 2\}_{J_d} - \{K_0, +, 2\}_{J_u} \quad .$$

The partitioned CR dependence equation is:

Loop	*Source*		*Sink*
(I):		$\varnothing$	
(J):	$\{K_0, +, 2\}_{J_d}$	=	$\{K_0, +, 2\}_{J_u}$.

The monotonicity of the equation is checked for loop J. After that, only the direction vectors $(*, =), (<, =), (=, =),$ and $(>, =)$ need to be checked.

In addition to the CR-enhanced Banerjee test, we also apply a stronger output dependence test based on monotonic CR analysis, which comes almost for free with CR forms. Note that *self-output dependences* are output dependences of the same array. If a dependence test can prove the expression of the dependence is monotonic, then the self output dependence can be disproved.

Monotonic analysis of CR-forms is straightforward and amounts to the discovery of inconsistent step directions of normalized polynomial CR Forms. Consider the following example:

```
K = . . .
DO 11 I = 0, M
    A(K) = . . .
    IF (. . . )
        K = K + 1
    ELSE
        K = K + 2
    END IF
ENDDO
```

From this we see that the CR forms for K are $\{K_0, +, 1\}_I$ and $\{K_0, +, 2\}_I$ which are both strictly monotonically increasing functions, because the strides of both CRs are always positive. Therefore, $A(K)$ has no output dependence.

4. RESULTS

In this section we compare the dependence analysis coverage and performance of the CR-enhanced Banerjee test to the Omega test and Range test using the Polaris compiler release 1.65 compiled with GCC 2.95. The Polaris compiler includes implementations of both the Omega test and the Range test. The tests were conducted on a Dual Intel Xeon 2.8 GHz running Redhat Linux 9. We performed our evaluations using the SPEC Benchmarks and LAPACK. All results are with loop normalization and constant propagation having been performed first. The GCD [3] and Simple Subscript [14] tests are run preliminary to all dependence testing.

4.1 Filtered Dependences

First, we study the coverage of dependences filtered by the tests. Specifically, we are concerned with identifying independences discovered by the CR-enhanced Banerjee test, that were not discovered by either the Omega test or the Range test. In Figure 7, the columns labeled 'Filtered LC' show the number of false loop-carried dependences filtered out by the CR-enhanced Banerjee test *in addition* to the filtering done by the dependence test indicated above the column label. The columns labeled '% increase' show the percentage of additional false loop-carried dependences removed.

The false dependences filtered by the original Banerjee test is a subset of those filtered by both the Omega test and the Range test, thus the additional increase in filtered dependences can be attributed to the CR enhancement. Specifically, the increase in the number of independences proved ranged from 0% to as much as 49% in the case of the Range test and 57% in the case of the Omega test. The cases where the percent increase in additional false loop-carried dependence filtered was small occurred in codes that had fewer dependences to analysis and/or involved mostly linear dependence equations with unknown symbolic loop bounds. When the Range and Omega tests where both applied, one after the other, the results were generally unchanged with as much as a 49% increase in the number of additional dependences filtered. In Figure 8 we show a similar set of results but this time the IVS routine built into Polaris is run first before applying the Omega the and the Range test. A comparison of Figure 7 and 8 show that while IVS certainly improved results for both the Omega test and the Range test, it did not account for a large percentage of the additional independences filtered by the CR-enhanced Banerjee test for some benchmarks.

Figure 9 shows the total number of false dependences filtered by each test. The column labeled 'Parallel Loops', indicate the additional parallel loops discovered by Polaris due to dependences filtered by the particular test, while the column labeled 'Filtered LC'

show the number of loop-carried dependences filtered out by each test. For all benchmarks the CR enhancement allows the Banerjee test to filter a competitive number of loop-carried dependences, enabling a comparable number of loops to be identified as parallel. Still, for benchmarks such as QCD and MDG, the CR-enhanced Banerjee test filters more loop-carried dependences than either the Range or Omega tests.

A seemingly contradictory observation of Figure 9 is that the CR-enhanced Banerjee test may filter out more false loop-carried dependences than either the Range or the Omega test for a specific benchmark, while the Range and Omega test afford the discovery of more parallel loops for the benchmark. However, an increase in false loop-carried dependences filtered does not guarantee more parallelism will be exploited. Specifically, the CR-enhanced Banerjee test was often able to filter false self-output loop-carried dependence not filtered by either the Omega or the Range test and in a loop where other loop-carried dependences remained. As indicated by Figure 7, many of the dependences filtered by the CR-enhanced Banerjee test were not the same dependences filtered by the Omega or the Range test. For example, for benchmark MDG there were more than 300 additional false dependences filtered compared to the Omega and Range test, but when comparing this result to the increase in total false dependences filtered, shown in Figure 9, it is clear that most dependences filtered were not the same. In cases where codes were dominated by dependence pairs where dependence statements involved symbolic values that canceled, the CR-enhanced Banerjee test did support the discovery of a comparable number of parallel loops.

Results from Figures 7 and 9 show a CR enhancement complements dependence testing. Despite the Banerjee test being subsumable by both the Omega and Range tests, the CR enhancement was able to provide a benefit that resulted in the removal of false loop-carried dependences that the Omega test and the Range test were not able to discover. Moreover, Figure 8 shows improvements occurred irregardless of whether or not IVS was performed first. These results show there is the possibility of filtering additional dependences when a dependence test is enhanced by utilizing the CR-form.

4.2 Analysis Impact

Recall that the construction of the CR forms for loop IVs allows the analysis of expressions that are nonlinear and the analysis of expressions that have no closed-forms. In the SPEC FORTRAN benchmarks, including those found of the Perfect Club suite, we found relatively few cases of nonlinear dependence equations. In total, less than 1% of the total number of CR-based dependence equations passed to the Banerjee Test were nonlinear polynomials. A larger, but still relatively small percentage, $\approx 5\%$, of the dependence equations analyzed were affected by conditional flow and thus had no closed-forms.

While relatively few dependence pairs with nonlinear or min/max CR forms, were passed to the CR-enhanced Banerjee test, of the dependence pairs that were, many were found to be independences. Table 1 gives a breakdown of the dependence equations encountered that involved non-linear expressions, were affected by conditional flow, or used CR-based monotonic analysis for the removal of self-output dependences. The column labeled '% Encountered' is the percentage of this type dependence equation encountered compared to the other dependence equation types listed in the table. The column labeled '% False Dependence' shows the percentage of these encountered equations that contained at least one false loop-carried dependence that was filtered. Because such a high percentage of these identified dependence equations were shown to

Bench	CR-Banerjee after Range		CR-Banerjee after Omega		CR-Banerjee After Both Tests	
	Filtered LC	% Increase	Filtered LC	% Increase	Filtered LC	% Increase
PERFECT						
DYFESM	35	9.14%	32	8.08%	32	8.08%
MDG	199	49.38%	199	57.02%	199	49.38%
OCEAN	7	1.24%	7	1.32%	7	1.24%
QCD	47	30.52%	47	30.52%	47	30.52%
TRFD	2	2.74%	2	2.70%	2	2.70%
LAPACK						
GEP	5	125.00%	3	50.00%	3	50.00%
NEP	5	16.67%	3	6.82%	3	6.82%
SEP	13	16.88%	17	23.29%	13	16.88%

Figure 7: The number of loop-carried dependence filtered out by the CR-enhanced Banerjee test in addition to other dependence tests along with percent increase to total number of loop-carried dependences filtered.

Bench	CR-Banerjee after Omega		CR-Banerjee after Range		CR-Banerjee After Both Tests	
	Filtered LC	% Increase	Filtered LC	% Increase	Filtered LC	% Increase
PERFECT						
DYFESM	27	6.77%	24	5.87%	24	5.87%
MDG	165	36.11%	163	42.12%	163	35.44%
OCEAN	1	0.20%	1	0.20%	1	0.20%
QCD	4	1.92%	4	1.92%	4	1.92%
LAPACK						
GEP	4	100.00%	2	33.33%	2	33.33%
NEP	4	13.33%	2	4.55%	3	4.55%

Figure 8: Dependence testing with induction variable substitution having been applied first, including only those benchmarks that saw benefit for the Omega or Range tests. The number of loop-carried dependence filtered out by the CR-enhanced Banerjee test in addition to other dependence tests along with percent increase to total number of loop-carried dependences filtered

Expression Type	% Encountered	% False Dependence
Non-linear	4 %	100%
Conditional	12 %	19.75%
Monotonic	84 %	76.15%

Table 1: CR-enhanced Banerjee impact on the Perfect Club Benchmarks.

contain at least one false dependence, these results suggest that for program codes that contain more nonlinear and/or non-closed form expressions the impact of the CR-enhanced Banerjee test would be even greater.

4.3   Analysis Efficiency

Figure 10 shows the timing results of our tests. The bar-chart on the right shows the total time taken by each dependence test for each benchmark while the pie-chart on the left provides a break-down of where the CR-enhanced Banerjee test is spending its time. For all benchmarks, excluding OCEAN, the CR-enhanced Banerjee test takes less time to complete than the Omega test, but requires more time than the Range test. The fact that our implementation of the CR-enhanced Banerjee was not faster than the Range test is not surprising considering our implementation is experimental and not optimally integrated with Polaris. There are currently several instances of internal expression cloning and/or simplification that could be reduced or eliminated with better sharing of CR forms and expressions throughout the implementation. In addition, if a dependence is not filtered early on in the hierarchal analysis of the

direction vectors associated with a dependence, dependence testing grows exponentially, thus reducing efficiency.

The pie-chart of Figure 10 shows the percent time spent doing recurrence recognition, CR construction, direction vector analysis, and CR value range analysis. The CR-enhanced Banerjee test spent most of its time setting up and creating dependence equations for specific direction vectors, a process that takes place for the Banerjee test regardless of the enhancement of the CR-form. This process accounted for over half, $\approx 54\%$, of the total time spent doing dependence testing. This helps to explain the non-typical timing performance seen on the Benchmark OCEAN which has many dependences involving arrays that contain several loop indices. 31% of the time was spent simplifying and analyzing CR-forms and CR-forms converted to closed-form expressions. 13% of the time was spent finding and duplicating recurrence pairs (a process that includes preventable cloning) and the remaining 1% was spent constructing the CR forms.

Overall, the timing performance of the the CR-enhanced Banerjee test compared well versus the Range and Omega test, and even outperformed the Omega test on all but one of the Perfect Benchmarks. We expect to further improve the performance by eliminating the need for internal expression cloning and by identifying and avoiding unnecessary direction vector analysis.

5.   DISCUSSION

The Banerjee test relies on bounds analysis of variables of linear expressions to determine if the lower and upper bounds of array subscript expressions overlap. Because the constraints used by the test are a subset of those used by both the Range and Omega tests, the filtered dependences of both tests subsume those of the Banerjee test. The CR enhancement to the Banerjee test adds analysis of

Bench	Range Test		Omega Test		CR-Banerjee Test	
	Parallel Loops	Filtered LC	Parallel Loops	Filtered LC	Parallel Loops	Filtered LC
PERFECT						
DYFESM	52	383	55	396	43	387
MDG	10	403	10	349	9	447
OCEAN	53	563	53	530	42	400
QCD	27	154	27	154	26	178
TRFD	17	73	17	74	17	71
LAPACK						
GEP	1	4	1	6	1	9
NEP	10	30	10	44	9	31
SEP	13	77	13	73	10	84

Figure 9: A comparisons of dependence tests without induction variable substitution having been applied.
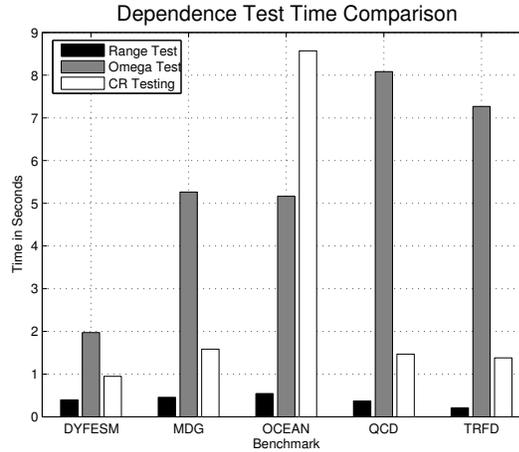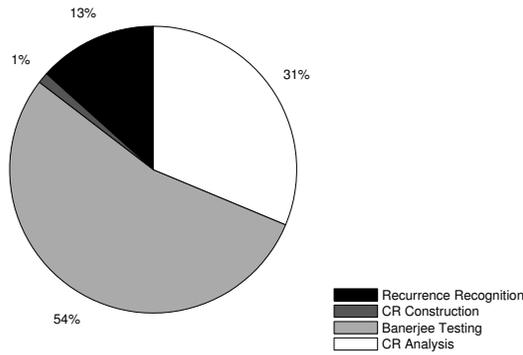

Figure 10: Timing analysis shows a comparisons of the Range, Omega, and CR-enhanced Banerjee test applied to PERFECT Club. Also a breakdown of where time is spent for the CR-enhanced Banerjee test

non-linear and conditional-flow influenced expressions. It also provides the ability to discover the monotonicity of expressions with respect to specific loop indices. This aids in both bounds analysis and in the filtering of self-output dependences. These enhancements can also be useful to other more powerful dependence tests such as the I-test [18, 22] and the Range test. The impact of the CR-based enhancement on the Banerjee test is highlighted by the additional filtering of false dependences as shown in Figures 7 and 8 of the previous section. In addition, CR analysis is shown in Table 1 to be effective at handling problems traditionally not handled by dependence tests. The example below, taken from OLDA of TISI.f:360, shows an output dependence not filtered by either the Omega test or the Range test:

```
          NRS = (NUM * (NUM+1))/2
          MRSIJ0 = 0
          DO 100 MRS = 1, NRS
             MRSIJ = MRSIJ0
             DO 90 MI = 1, MORB
                IJ=IJ+1
                IJKL=IJKL+I-J+1
                DO 80 MJ = 1, MI
                   MRSIJ=MRSIJ+1
80                 XRSIJ[MRSIJ]=XIJ[MJ]
90           CONTINUE
             MRSIJ0=MRSIJ0+NRS
100       CONTINUE
```

In the code, the output dependence to XRSIJ(MRSIJ) has the following partitioned CR-forms:

Loop	*CR-Form*
(MJ):	$\{0,+,1\}_{MJ}$
(MI):	$\{0,+,1,+,1\}_{MI}$
(MRS):	$\{(1+MRSIJ0),+,(NUM + NUM**2)/2\}_{MRS}$

Analysis of CR-forms for loops MJ and MI determine both forms to be monotonically increasing. What are not known are the upper-bounds of the CR forms for any of the loops or the monotonicity of the CR-form for loop MRS. Based on what is known, the CR-enhanced Banerjee test is able to reduce the direction vector $(*,*,*)$ to $\{(<>,*,=),(=,<>,=)\}$, thus filtering a loop-carried dependence (loop MJ) for a non-linear dependence equation.

Results from Figures 7 and 9 show a significant increase in additional false loop-carried dependences filtered. The additional filtering, though, did not translate into an increase in the number of parallel loops automatically identified and so there was no obvious potential for speedup based on parallelization (this, however, does not exclude the possibility of potential speedup from improvements in the application of optimizations). The lack of additional parallel loops were due to loop-carried array based and IV based dependences that remained after filtering. Traditionally the IV based dependences are removed with a separate pass of IVS, suggesting the optimization pass is necessary regardless of it not being useful to a CR-enhanced dependence test. This, however, is not the case as most CR-forms can also convert directly to equivalent closed-

form expressions [28]. This conversion can take place selectively, only when arrays of the loop are known to not carry dependences, and thus the removal of the IV is necessary. This avoids doing IVS unnecessarily and it also avoids adding potentially more expensive computations of closed-form expressions to the loops nest. We illustrate the process with the code below:

```
        DO I = 1, N
            A(K) = . . .
            K = K + I
        ENDDO
```

Here, variable K is non-linear and has the CR form $\{K_0, +, 1, +, 1\}_I$. Application of CR-inverse rules will produce the closed form for K: $\{K_0, +, 1, +, 1\}_I \rightarrow K_0 + ((i^2 + i)/2)$. If this closed form is placed at the start of the loop replacing the K in A(K) through forward substitution, we kill the cross iteration loop dependence of K as shown below:

```
        K_0 = K
        DO I = 1, N
  C         loop-carried dependence on K is killed
            K = K_0 + ((i**2 + i)/2)
            A(K) = . . .
            K = K + I
        ENDDO
```

In cases where an IV or an array expression does not have a closed-form, dependence testing may still lead to loop parallelization for any inner loops where the IV is not updated. In addition, dependence testing may also benefit the optimization and vectorization of codes. For instance in the following example:

Vectorization:

```
  J = 0                           J = 0
  DO I = 0, N                     A[I:N] = . . .
      A[I] = . . .                DO I = 0, N
      . . . = A[J]                    . . . = A[J]
      IF (. . .)          ⟶         IF (. . .)
          J = J + 1                      J = J + 1
      ENDIF                         ENDIF
  ENDDO                           ENDDO
```

In the code above there is a flow dependence for array A that allows loop fission to take place followed by loop vectorization. These steps are not possible if a dependence test must rely on IVS to discover the flow dependence.

Loop Fusion:

```
  DO I = 0, N
      A[I] = . . .                 J = 0
  ENDDO                            DO I = 0, N
                                       A[I] = . . .
  J = 0                                . . . = A[J]
  DO I = 0, N           ⟶             IF (. . .)
      . . . = A[J]                        J = J + 1
      IF (. . .)                       ENDIF
          J = J + 1                 ENDDO
      ENDIF
  ENDDO
```

In the code above J does not have a closed-form, but its maximum CR form $\{0, +, 1\}_J$ can still be used to show the flow dependence between statement A[I] and A[J] remain after loop fusion. After loop fusion this code can be better scheduled to exploit instruction level parallelism.

6. CONCLUSION

In this paper we proposed a new approach to dependence testing that rephrases the dependence problem in terms of Chains of Recurrences (CRs). The focus of our work is the enhancement of an existing dependence test to increase the coverage of problems tested and improve the ability to filter independences. We presented an algorithm for using CR-forms in dependence equations for the Banerjee test and provided results comparing the CR-enhanced Banerjee test with the Range and Omega tests. Results showed an increase in the coverage of dependence problems analyzed, but more significantly, an increase was seen in the number of independences discovered. When comparing results to the Omega and Range tests, the CR enhancement increased the number of dependences disproved in the Perfect Club Benchmark Suite by as much as 62% for the Omega test and 54% in the case of the Range test. These numbers are even more impressive when you consider the inherent weakness of the original Banerjee test which is normally subsumed by the abilities of the other tests. Timing results show the CR enhancement adds little costs to the dependence filtering process, with the construction of CR forms amounting to only $\approx 1\%$ of the entire process.

Acknowledgments

7. REFERENCES

[1] J. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):491–542, 1987.

[2] O. Bachmann. Chains of recurrences for functions of two variables and their applicat ion to surface plotting. In N. Kajler, editor, *Human Interaction for Symbolic Computation*. Springer-Verlag, 1996.

[3] U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

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

[5] J. Birch, R. A. van Engelen, and K. A. Gallivan. Value range analysis of conditionally updated variables and pointers. In *In Proceedings of Compilers for Parallel Computing (CPC '04)*, pages 265–276, 2004.

[6] W. Blume. Symbolic analysis techniques for effective automatic parallelization. Technical Report UIUCDCS-R-95-1913, University of Illinois, 1995.

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

[8] W. Blume and R. Eigenmann. The range test: a dependence test for symbolic, non-linear expressions. In *Proceedings of the 1994 conference on Supercomputing (SC '94)*, pages 528–537. IEEE Computer Society Press, 1994.

[9] W. Blume and R. Eigenmann. Nonlinear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1180–1194, 1998.

[10] G. B. Dantzig and B. C. Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14(3):288–297, 1973.

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

[12] B. Franke and M. F. P. O'Boyle. Compiler transformation of pointers to explicit array accesses in DSP applications. In *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*, pages 69–85. Springer-Verlag, 2001.

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

[14] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*, volume 26, pages 15–29. ACM Press, 1991.

[15] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing (LCPC '93)*, volume 768 of *Lecture Notes in Computer Science (LNCS)*, pages 567–585. Springer-Verlag, 1994.

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

[17] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[18] X. Kong, D. Klappholz, and K. Psarris. The i test: An improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):342–349, 1991.

[19] K. Kyriakopoulos and K. Psarris. Addressing the issues in data dependence analysis. In *Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems (PDCS '05)*, 2005.

[20] K. Kyriakopoulos and K. Psarris. Efficient techniques for advanced data dependence analysis. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*, pages 143–156. IEEE Computer Society Press, 2005.

[21] V. Menon, K. Pingali, and N. Mateev. Fractal symbolic analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6):776–813, 2003.

[22] K. Psarris, X. Kong, and D. Klappholz. The directoin vector i test. *IEEE Transactions on Parallel and Distributed Systems*, 4(11):1280–1290, 1993.

[23] K. Psarris and K. Kyriakopoulos. Data dependence testing in practice. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 264–273. IEEE Computer Society Press, 1999.

[24] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing (SC '91)*, pages 4–13. ACM Press, 1991.

[25] W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation (PLDI '92)*, pages 140–151. ACM Press, 1992.

[26] Y. Shou, R. A. van Engelen, J. Birch, and K. A. Gallivan. Toward efficient flow-sensitive induction variable analysis and dependence testing for loop optimization. In *Proceedings of the ACM SouthEast Conference (ACMSE '06)*, pages 1–6, 2006.

[27] R. A. van Engelen. Symbolic evaluation of chains of recurrences for loop optimization. Technical Report TR-000102, Florida State University, 2000.

[28] R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*, pages 118–132. Springer-Verlag, 2001.

[29] R. A. van Engelen, J. Birch, and K. A. Gallivan. Array data dependence testing with the chains of recurrences algebra. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA '04)*, pages 70–81. IEEE Computer Society Press, 2004.

[30] R. A. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. A. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the 18th annual international conference on Supercomputing (ICS '04)*, pages 106–115. ACM Press, 2004.

[31] R. A. van Engelen and K. A. Gallivan. An efficient algorithm for pointer-to-array access conversion for compiling and optimizing DSP applications. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA '01)*, pages 80–89. IEEE Computer Society Press, 2001.

[32] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, Redwood City, CA, 1996.

[33] P. Wu, A. Cohen, J. Hoeflinger, and D. Padua. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *Proceedings of the ACM International Conference on Supercomputing (ICS '01)*, pages 78–91. ACM Press, 2001.

[34] E. V. Zima. Simplification and optimization transformations of chains of recurrences. In *Proceedings of the International Symposium on Symbolic and Algebraic Computing (ISSAC '95)*, pages 42–50. ACM Press, 1995.