The Range Test: A Dependence Test for Symbolic, Non-linear Expressions *

William Blume
blume@csrd.uiuc.edu

Rudolf Eigenmann
eigenman@csrd.uiuc.edu

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, Illinois 61801-2307

Abstract

Most current data dependence tests cannot handle loop bounds or array subscripts that are symbolic, nonlinear expressions (e.g. $A(n*i+j)$, where $0 \leq j \leq n$). In this paper, we describe a dependence test, called the range test, that can handle such expressions. Briefly, the range test proves independence by determining whether certain symbolic inequalities hold for a permutation of the loop nest. Powerful symbolic analyses and constraint propagation techniques were developed to prove such inequalities. The range test has been implemented in Polaris, a parallelizing compiler being developed at the University of Illinois.

1 Introduction

To allow sequential programs to run efficiently on parallel machines, parallelizing compilers were developed to transform these programs into parallel ones [3]. These compilers must be able to identify the important loops that can be run in parallel if they are to achieve decent speedups. Powerful dependence tests are needed to effectively exploit the inherent parallelism in these sequential programs.

There has been much research in the area of data dependence analysis [1, 14, 20, 21, 24]. Modern day data dependence tests have become very accurate and efficient. However, most of these tests require the loop bounds and array subscripts to be represented as a linear (affine) function of loop index variables. That is, the expressions must be in the form $c_0 + \sum_{j=1}^{n} c_j i_j$ where c_j are integer constants and i_j are loop index variables. Expressions not of this form are called *nonlinear*.

```
k = 0
DO j = 1, m            DO j = 1, m
  DO i = 1, n            DO i = 1, n
    k = k + 1
    A(k) = ···               A(i + n*(j-1)) = ···
  ENDDO                  ENDDO
ENDDO                  ENDDO
```

Figure 1: Before and after induction variable substitution

Because nonlinear expressions prevent the application of dependence tests, parallelizing compilers perform several analyses and optimizations to eliminate nonlinear expressions. For example, constant propagation and induction variable substitution are used to remove loop-variant variables. Other techniques have also been developed to handle additive, loop-invariant, symbolic terms or to eliminate unwanted operations such as divisions [14, 20, 21].

Unfortunately, not all nonlinear expressions can be removed. It was believed that this would not be a problem for dependence testing real programs since nonlinear expressions would be rare. However, our manual parallelization effort of the Perfect Benchmarks have shown us that this is not the case [12, 11]. For example, a parallelizing compiler could achieve a speedup of at most two for the codes *OCEAN* and *TRFD* from the Perfect Benchmarks if it could not parallelize loops with nonlinear array subscripts [6]. For some of these loops, nonlinear expressions occurred in the original program text. For other loops, nonlinear expressions were introduced by the compiler.

Two common compiler passes can introduce nonlinearities into array subscript expressions: induction variable

*This work was supported by contract DABT63-92-C-0033 from the Advanced Research Project Agency. This work is not necessarily representative of the positions or policies of the U.S. Army or the government.

ISSN 1063–9535. Copyright ©1994 IEEE. All rights reserved.

Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE. For information on obtaining permission, send a blank email message to info.pub.permission@ieee.org.

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

substitution and array linearization. Induction variable substitution replaces variables that are incremented by a constant value for each loop iteration with a closed-form expression made up of only loop invariants and loop indices. However, when induction variable substitution is performed upon multiply nested loops, the resulting closed-form expression may be nonlinear. For example, performing induction variable substitution on the loop nest in Figure 1 introduces a nonlinear expression into the subscript of array `A`. (Remember that, by our definition of the term nonlinear terms like `n*j` are considered to be nonlinear, even though the variable `n` is loop-invariant.)

Array linearization transforms two or more dimensions of an array into a single dimension. Array linearization is needed by subroutine inlining or interprocedural analysis when an array is dimensioned differently across procedure boundaries. If the declared dimensions of a multidimensional array are symbolic expressions, the resulting linearized array may be nonlinear. For example, if the array `A`, which was originally dimensioned as `A(n,m)`, was linearized, its declaration will be changed to `A(n*m)`, and a reference `A(i,j)` will be changed to `A(i + n*(j-1))`.

In this paper, we will present the range test, a dependence test that can handle symbolic, nonlinear array subscripts and loop bounds. In the range test, we mark a loop as parallel if we can prove that the range of elements accessed by an iteration of that loop do not overlap with the range of elements accessed by other iterations. We prove this by determining whether certain symbolic inequality relationships hold. Powerful variable constraint propagation and symbolic simplification techniques were developed to determine such inequality relationships. To maximize the number of loops found parallel using the range test, we examine the loops in the loop nest in a permuted order.

Section 2 briefly defines data dependences and direction vectors. Section 3 then describes the range test. Section 4 gives an overview of the range propagation algorithm, which allows us to compare symbolic expressions. Real-life examples of important loop nests that the range test can parallelize but current tests cannot are given in section 5. We will then compare our work with other symbolic data dependence tests in Section 6. Section 7 presents our conclusions and plans for future work.

2 Data dependence

In this section we will give a brief definition of data dependences. For a more thorough description of data dependence and dependence analysis, see Banerjee et al [3, 1, 24].

To ease the presentation of the range test, we will assume that we have a perfectly nested, normalized, FORTRAN-77 loop nest as shown in Figure 2. We will also assume that the tested array `A` has only one dimension. The array access functions (f and g) and loop upper bounds (N_i) may be arbitrary symbolic expressions made up of loop-invariant variables and loop indices (i.e. i_x) of enclosing loops. It is not difficult to extend our test to handle imperfectly nested loops, symbolic lower bounds, non-unit strides, multidimensional arrays, and loop-variant variables. In fact, our implementation includes these extensions.

```
L1 : DO i1 = 0, N1
...     ...
Ln :      DO in = 0, Nn
S1 :         A(f(i1, ..., in)) = ...
S2 :         ... = A(g(i1, ..., in))
          ENDDO
        ...
     ENDDO
```

Figure 2: Model of loop nest for dependence testing

2.1 Data dependence

We define an index subspace $\mathcal{R}_j$, where $1 \leq j \leq n$, to be the set of all loop index vectors $(\alpha_1, \ldots, \alpha_j)$ that fall within loop bounds of the outermost j loops. More formally,

$$\mathcal{R}_j = \{(\alpha_1, \ldots, \alpha_j) : 0 \leq \alpha_1 \leq N_1, \ldots, 0 \leq \alpha_j \leq N_j\}.$$

The index space $\mathcal{R}$ is defined to be equal to the index subspace $\mathcal{R}_n$.

A *data dependence* exists between array accesses $A(f(\vec{\alpha}))$ and $A(g(\vec{\beta}))$ if and only if at least one of the two accesses is a write, $f(\vec{\alpha}) = g(\vec{\beta})$, and $\vec{\alpha}, \vec{\beta} \in \mathcal{R}$.

2.2 Direction vectors

Suppose that a dependence exists between $A(f(\vec{\alpha}))$ and $A(g(\vec{\beta}))$. Then, the *direction vector* $\vec{d} = (d_1, \ldots, d_n)$ for this dependence is defined as:

$$d_i = \begin{cases} < & \text{if } \alpha_i < \beta_i \\ = & \text{if } \alpha_i = \beta_i \\ > & \text{if } \alpha_i > \beta_i \end{cases}$$

Since there may be more than one pair of integer vectors $\vec{\alpha}$ and $\vec{\beta}$ that satisfy the dependence equation, there may be more than one direction vector between S_1 and S_2.

A dependence is *carried* by loop L_i (or loop at level i) if and only if there exists a dependence vector $\vec{d}$ where $d_1 =$ '=', ..., $d_{i-1} =$ '=' and $d_i \neq$ '='. If a loop does not carry any dependences, then that loop may be run in parallel without synchronization.

3 The range test

The range test grew out of a simple observation in our hand analysis of real programs: most parallel loop iterations access adjacent array ranges. These ranges can be

very regular (e.g., an inner loop accesses a fixed-length array section and the outer loop strides over this section), they can be increasing or decreasing (e.g., if the two loops are triangular); or, they can be irregular (e.g., if they represent array sections that are carved out of a large array; start and length of the sections are typically stored in index arrays)[1]. With one additional observation we can describe the majority of all access patterns: the loops visiting these ranges may be interchanged, such that the access patterns appear "interleaved". Now, if we managed to prove that such adjacent array ranges do not overlap – possibly "looking through interchanged loops" – we could tell that the loops are parallel. The following section describes such a test formally.

3.1 Disproving dependence between symbolic expressions

Essentially, the range test disproves carried dependences between $A(f(\vec{\imath}))$ and $A(g(\vec{\imath}'))$ for a loop at level j, by proving that the range of elements taken by f and g do not overlap for adjacent iterations of the loop at level j. It determines whether these ranges overlap by comparing the minimum and maximum values of these ranges. The formal definition of these minimum and maximum values are defined below. Section 3.4 describes how to compute these minimum and maximum values

Definition 1 *Let $f_j^{\min}(i_1,\ldots,i_j)$ and $f_j^{\max}(i_1,\ldots,i_j)$ be functions that obey the following constraints:*

$$f_j^{\min}(i_1,\ldots,i_j) \leq \min\left\{f(\vec{\imath}') : \vec{\imath}' \in \mathcal{R}, i_1' = i_1,\ldots,i_j' = i_j\right\}$$
$$f_j^{\max}(i_1,\ldots,i_j) \geq \max\left\{f(\vec{\imath}') : \vec{\imath}' \in \mathcal{R}, i_1' = i_1,\ldots,i_j' = i_j\right\}$$

Intuitively, $f_j^{\min}(i_1,\ldots,i_j)$ and $f_j^{\max}(i_1,\ldots,i_j)$ are functions that return the minimum and maximum values that f may take for a particular iteration of the outermost j loops. In our implementation of the range test, these functions are represented as symbolic expressions made up of loop indices $i_1,\ldots,i_j$ and loop-invariant variables.

The ability to determine the minimum or maximum of f or g in respect to some set of loops leads us to our first dependence test. If the maximum of f is less than the minimum of g in respect to some subset of loops, then these loops cannot carry any dependences. The theorem below states this formally.

Theorem 1 *If $f_j^{\max}(i_1,\ldots,i_j) < g_j^{min}(i_1,\ldots,i_j)$ for all $(i_1,\ldots,i_j) \in \mathcal{R}_j$, then there can be no dependences between $\mathtt{A}(f(\vec{\imath}))$ and $\mathtt{A}(g(\vec{\imath}'))$ with a direction vector $\vec{d}$ of the form $d_1 =$ '$=$', ..., $d_j =$ '$=$'.*

[1] The range test does not yet handle such subscripted subscript patterns (e.g., $A(X(i))$)

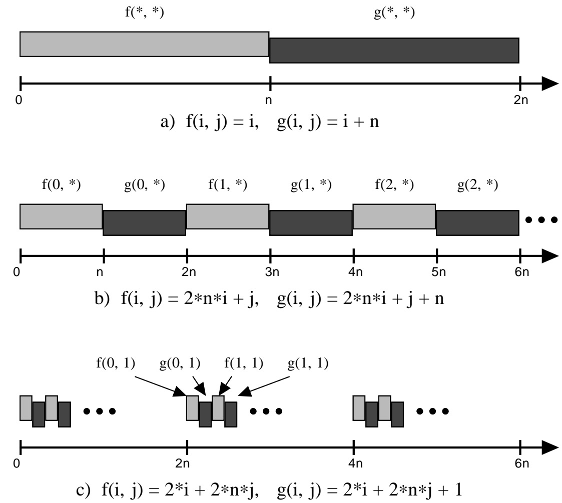

Figure 3: Examples of how array accesses can be interleaved in respect to a particular loop (loop with index i for these examples.) All examples assume that $0 \leq i, j < n$.

Proof. Suppose that such a dependence exists, (i.e., $f(\vec{\imath}) = g(\vec{\imath}')$ with direction vector $\vec{d}$). By Definition 1, we have $f(\vec{\imath}) \leq f_j^{\max}(i_1,\ldots,i_j)$ and $g_j^{min}(i_1',\ldots,i_j') \leq g(\vec{\imath}')$. Because of the direction vector $\vec{d}$, $g_j^{min}(i_1',\ldots,i_j') = g_j^{min}(i_1,\ldots,i_j)$. Since $f_j^{\max}(i_1,\ldots,i_j) < g_j^{min}(i_1,\ldots,i_j)$, it must hold that $f(\vec{\imath}) < g(\vec{\imath}')$. Contradiction. □

Theorem 1 proves that there are no carried dependences between $\mathtt{A}(f(\vec{\imath}))$ and $\mathtt{A}(g(\vec{\imath}'))$ for loops with indices $i_{j+1},\ldots,i_n$, if the range of possible values taken by f for these loops does not overlap with the range of possible values taken by g. However, it cannot prove that there are no carried dependences for a certain loop if the possible values taken by f and g are interleaved for that loop. Figure 3 shows some examples of how array accesses can be interleaved for a particular loop nest. We have found such examples do occur often in practice. All these examples assume that we have a loop nest of the form

```
L1 : DO i = 0, n-1
L2 :   DO j = 0, n-1
S1 :     A(f(i,j)) = ···
S2 :     ··· = A(g(i,j))
       ENDDO
     ENDDO
```

and that the range test is currently attempting to prove that S_1 and S_2 do not carry dependences for loop L_1. For figure 3a, the range of accesses made by $\mathtt{A}(f(i,j))$ and

$\mathtt{A}(g(i,j))$ never overlap, so Theorem 1 can prove that loop L_1 does not carry a dependence for this access pair. However, the set of accesses made by $\mathtt{A}(f(i,j))$ and $\mathtt{A}(g(i,j))$ are interleaved in figures 3b and 3c, causing the test from Theorem 1 to fail, even though there isn't a carried dependence. We will present a second dependence test that can disprove carried dependences for a special case of these interleavings, where the possible values taken by f and g for a single iteration are not interleaved with the possible values taken by other iterations of f and g. Figure 3b shows an example of this case. However, before we describe this test, we must define the property of *monotonicity* for a particular loop index. (We will deal with Figure 3c in Section 3.2.)

Definition 2 *A function* $f(\vec{\imath})$ *is* monotonically non-decreasing *for index* i_j *iff* $f(i_1,\ldots,\alpha_j,\ldots,i_n) \le f(i_1,\ldots,\beta_j,\ldots,i_n)$ *whenever* $0 \le \alpha_j \le \beta_j \le N_j$.

Similarly, a function $f(\vec{\imath})$ *is* monotonically non-increasing *for index* i_j *iff* $f(i_1,\ldots,\alpha_j,\ldots,i_n) \ge f(i_1,\ldots,\beta_j,\ldots,i_n)$ *whenever* $0 \le \alpha_j \le \beta_j \le N_j$.

We can prove whether an expression is monotonically non-decreasing for a loop level j by proving that the difference $f(i_1,\ldots,i_j+1,\ldots,i_n) - f(i_1,\ldots,i_j,\ldots,i_n)$ is always greater than or equal to zero, using the techniques described in Section 4. Similarly, we can prove whether an expression is monotonically non-increasing for a loop level j by proving that the difference is always less than or equal to zero.

Using this definition, we will now show how one can disprove dependences carried at level j when the possible values taken by f and g are not interleaved for a single iteration of the loop at level j.

Theorem 2 *If* $g_j^{min}(i_1,\ldots,i_j)$ *is monotonically non-decreasing for* i_j *and if* $f_j^{\max}(i_1,\ldots,i_j) < g_j^{min}(i_1,\ldots,i_j+1)$ *forall* $(i_1,\ldots,i_j) \in \mathcal{R}_j$ *and for* $0 \le i_j \le N_j - 1$*, then there can be no dependences from* $\mathtt{A}(f(\vec{\imath}))$ *to* $\mathtt{A}(g(\vec{\imath}'))$ *with a direction vector* $\vec{d}$ *of the form* $d_1 =$ '$=$', ..., $d_{j-1} =$ '$=$', $d_j =$ '$<$'.

Proof. Suppose that such a dependence exists, (i.e., $f(\vec{\imath}) = g(\vec{\imath}')$ with direction vector $\vec{d}$). By Definition 1, we have $f(\vec{\imath}) \le f_j^{\max}(i_1,\ldots,i_j)$ and $g_j^{min}(i'_1,\ldots,i'_j) \le g(\vec{\imath}')$. Because of the direction vector $\vec{d}$ and because $g_j^{min}(i_1,\ldots,i_j)$ is monotonically non-decreasing for index i_j, $g_j^{min}(i'_1,\ldots,i'_j) \ge g_j^{min}(i_1,\ldots,i_j+1)$. Since $f_j^{\max}(i_1,\ldots,i_j) < g_j^{min}(i_1,\ldots,i_j+1)$, it must hold that $f(\vec{\imath}) < g(\vec{\imath}')$. Contradiction. □

Theorem 3 *If* $g_j^{min}(i_1,\ldots,i_j)$ *is monotonically non-increasing for* i_j *and if* $f_j^{\max}(i_1,\ldots,i_j) < g_j^{min}(i_1,\ldots,i_j-1)$*, forall* $(i_1,\ldots,i_j) \in \mathcal{R}_j$ *and for* $1 \le i_j \le N_j$*, then there can be no dependences from* $\mathtt{A}(g(\vec{\imath}'))$ *to* $\mathtt{A}(f(\vec{\imath}))$ *with a direction vector* $\vec{d}$ *of the form* $d_1 =$ '$=$', ..., $d_{j-1} =$ '$=$', $d_j =$ '$<$'.

Proof. Similar to proof of Theorem 2.

By definition of loop-carried dependences, the test from Theorem 2 (or Theorem 3) must be applied twice to prove that a pair of access functions f and g do not carry dependences for the loop with index i_j: once to disprove a dependence with direction vector $\vec{d}$ from f to g, and once to disprove a dependence with direction vector $\vec{d}$ from g to f. Also note that the tests from these two theorems can disprove a dependence direction $\vec{d}$ from f to g and from g to f, and thus disprove that the loop with index i_j does not carry a dependence, only if both f and g are monotonically non-decreasing or monotonically non-increasing for i_j.

In the previous definitions and theorems, we have assumed that the subset of loops that we are attempting to disprove dependence for are the innermost loops with indices i_j to i_n. With some minor changes in notation, these definitions and theorems still hold for arbitrary subsets of loops. The following subsection will exploit this property.

3.2 Permuting loops for dependence testing

As described earlier, the test from Theorem 1 can be used to prove independence when the values of access functions f and g are not interleaved, and the tests from Theorems 2 and 3 can be used to prove independence when the values of f and g are interleave, but the values taken by f and g for a loop iteration are not interleaved with values taken by other iterations. For more complex interleavings, the tests from all three Theorems would fail. Figure 3c gives an example of one of these more complex interleavings. Fortunately, we have observed that most of these interleavings can be eliminated by permuting the order in which we test the loops. For example, if loop L_1, with index i, and loop L_2, with index j were "swapped" so that L_2 is now treated as the outermost loop, we would be able to use Theorem 1 to prove that there are no carried dependences in the "inner" L_1 loop, and use Theorem 2 to prove that there are no carried dependences in the "outer" L_2 loop.

So, the range test attempts to maximize the number of loops that it can identify as not carrying dependences by applying its tests upon a permuted ordering of the loops in the loop nest. The range test does not physically permute the loops; it is done logically and temporarily by the test. Also, the range test only tries those permutations such that all loops identified as not carrying any dependences for the permuted loop nest also will not carry any dependences in the original nest.

The range test uses a heuristic to find a valid logical permutation of a loop nest, which seems to be quite acceptable in practice. This heuristic determines a valid permutation of a loop nest by recursively finding a valid permutation of the inner loops of the loop nest, then finding a location where it may safely insert the outermost loop in this permutation. The final location of the outermost loop is

found by repeatedly moving inwards by one until it reaches a location where it can be proven that it carries no dependences, or the loop just inside this location either carries a dependence or would carry a dependence if outermost loop was inserted inside of it.

To prove that all loops not carrying dependences in the permuted loop nest generated by the heuristic above also do not carry dependences in the original loop nest, we will need the following lemma. Due to lack of space, no proof is given, but techniques similar to Banerjee's [2] can be used to do so.

Lemma 1 *A loop that does not carry a dependence can be legally moved deeper into the loop nest and all loops that didn't carry a dependence beforehand would still not do so.*

We will prove that the heuristic generates a legal permutation by induction. For the base case, where the loop nest is a single loop, the permutation is trivially legal. For the inductive step, assume that the heuristic generates legal permutations for loop nests of i loops. For a nest of $i+1$ loops, the heuristic first recursively finds a permutation of the innermost i loops, then finds a location for the $(i+1)$th loop in this permutation. By the inductive hypothesis, the recursive first step results in a legal permutation. For the second step, which moves the $(i+1)$th loop inwards, all loops between the original and final positions of the $(i+1)$th loop do not carry dependences, by definition of the heuristic. Thus, we can undo this second step by moving all these loops back inside the $(i+1)$th loop; and, by Lemma 1, all loops not carrying dependences still do not so. Therefore, the heuristic generates a legal permutation for a nest of $i+1$ loops.

3.3 Algorithm

The algorithm for the Range Test, which implements the permutation heuristic described previously, is displayed in Figure 4. It generates the permuted loop nest, represented by the ordered list P, by visiting each loop L_i in the original loop nest, from innermost to outermost, and finding its proper location in the set of inner permuted loops P. Simultaneously, the algorithm determines whether each L_i carries any dependences. Loops proven not to carry dependences are added to the set D. The outer **for** loop visits each loop L_i while the inner **while** loop determines whether the current L_i carries any dependence and where to insert it in the list of permuted inner loops P. Statement S_1 tests if L_i does not carry a dependence at a particular location in P. If it doesn't carry a dependence, it is added to the set of parallel loops D and inserted into P at this location. Statement S_2 tests if it is legal to move the current location of loop L_i in P inward by one. If not, it is inserted into P at this location. (It is not legal to move the current location of L_i in P inwards by one if the current location is the innermost location in P, if the next inner loop in P carries a dependence, or if the next inner loop in P would become carry a dependence should L_i be inserted inside of it.) Statement S_3 performs the actual insertion of L_i into P.

```
INPUT: Normalized, perfectly nested loops
  (L_1, ..., L_n) and array access functions f and g.
OUTPUT: Set of loops D that do not have carried
  dependences between f and g.
  P ← ()   (* P is an ordered list representing
              the permuted loop nest *)
  D ← ∅
  for i ← n downto 1 do
    placed ← false ; j ← 0
    while not placed do
      j ← j + 1
S_1 : if RTEST1(f, g, {L_i, P_j, ..., P_|P|})
        or RTEST2(f, g, L_i, {P_j, ..., P_|P|}) then
          D ← D ∪ {L_i} ; placed ← true
S_2 : else if j = |P| + 1 or P_j ∉ D
        or not RTEST2(f, g, P_j,
                  {L_i, P_{j+1}, ..., P_|P|}) then
          placed ← true
      end if
    end while
S_3 : P ← (P_1, ..., P_{j-1}, L_i, P_j, ..., P_|P|)
  end for
```

Figure 4: The range test algorithm

Functions RTEST1 and RTEST2 are displayed in Figure 5. Function RTEST1, which applies Theorem 1, returns true if and only if it can prove that the loops in $\mathcal{L}$ do not carry any dependences for access functions f and g. Function RTEST2, which calls RTEST2x that implements Theorems 2 and 3, returns true if and only if it can prove that loop L_j carries no dependences for access functions f and g and the inner permuted loops $\mathcal{L}$. The functions MIN and MAX represent the $f_j^{\min}$ and the g_j^{max} functions described earlier. (The expression $f_j^{\max}$, used in Section 3, can be computed by calling $\mathrm{MAX}(f, \{L_{j+1}, \ldots, L_n\})$. The expression $f_j^{\min}$ is similar.) The function $\mathrm{MAX}(f, \mathcal{L})$ returns the maximum value that function f can take for the indices of the loops in $\mathcal{L}$. This maximum value is a symbolic expression made up of loop-invariant variables and indices of loops not in $\mathcal{L}$. The function MIN is similar. The implementations of MIN and MAX will be described in the next subsection.

3.4 Computing $f_j^{\min}$ and $f_j^{\max}$

There are several methods in which one can compute the minimum or maximum of an expression for a subset of loop indices. One simple but powerful method is to substitute the lower or upper bound of each index, depending on the index's monotonicity. Figure 6 shows how the range test computes the maximum of an expression for a given set of loops. The algorithm for computing the

```
    boolean function RTEST1(f, g, 𝓛)
      (* Apply Theorem 1 *)
R1 :  return MAX(f, 𝓛) < MIN(g, 𝓛)
R2 :    or MAX(g, 𝓛) < MIN(f, 𝓛)
    end function

    boolean function RTEST2(f, g, L_j, 𝓛)
      return f and g are both mono. non-decreasing
          or mono. non-increasing for L_j
R3 :    and RTEST2x(f, g, L_j, 𝓛)
R4 :    and RTEST2x(g, f, L_j, 𝓛)
    end function

    boolean function RTEST2x(f, g, L_j, 𝓛)
      (* Apply Theorem 2 or 3 *)
      s ← MAX(f, 𝓛)
      t ← MIN(g, 𝓛)
      if t is mono. non-decreasing for L_j then
        t ← t with i_j substituted by i_j + 1
      else
        t ← t with i_j substituted by i_j − 1
      endif
      return (s < t)
    end function
```

Figure 5: Algorithm for disproving carried dependences for loop L_j in respect to loops $\mathcal{L}$. Loop L_j is assumed to have index i_j. The word *monotonically* has been abbreviated to "mono."

minimum is very similar; simply switch the monotonically non-decreasing and monotonically non-increasing cases. It can be proven that the result of these functions meets Definition 1; that is, they are the minimum or maximum of f in the subspace spanned by indices $i_{j+1}, \ldots, i_n$.

Although the algorithm for computing the maximum in Figure 6 is powerful, a naive implementation of it, which computes the monotonicity of the intermediate maximum expression y for each index, can be inefficient. This is because each monotonicity computation for y requires a symbolic expression comparison, which can be quite expensive. To avoid these costs, the range test attempts to determine the monotonicity states of y from the precomputed monotonicity states of the input access expression f and the the loop bounds. (The monotonicity states of all array accesses f and all loop bounds are computed only once, at the beginning of dependence testing of the program.) More specifically, the range test initially sets the monotonicity states of y to the monotonicity states of f, then update y's monotonicity states after each substitution, using the monotonicity states of the substituted loop bound. For those cases where the range test couldn't determine the new monotonicity of y for index i from the old monotonicity of y for i and the monotonicity of the substituted loop bound for i, it marks the monotonicity of

```
expression function MAX(f, 𝓛)
  y ← f
  for each L_k ∈ 𝓛 from
        innermost to outermost loops do
    if y is mono. non-decreasing for i_k then
      y ← y with i_k substituted with N_k
    else if y is mono. non-increasing for i_k then
      y ← y with i_k substituted with 0
    else
      y ← +∞
    end if
  end for
  return y
end function
```

Figure 6: Algorithm for calculating the maximum value of function f for fixed values of indices i_j of loops L_j, where $L_j \notin \mathcal{L}$.

y for i as unknown. Later, when it finds the monotonicity of y for i to be marked as unknown when it is substituting for i, it computes the monotonicity for i using an expensive symbolic expression comparison. We have found this optimization to be very effective in practice. For many array accesses, the monotonicity states of y never need to be computed with symbolic comparisons.

3.5 Time complexity

Since the range test spends nearly all of its time performing symbolic expression comparisons, its time complexity can be characterized by the number of symbolic comparisons performed. These comparisons occur explicitly in the functions RTEST1 and RTEST2X and implicitly in the monotonicity tests of MIN and MAX. Since the range test may call RTEST1 and RTEST2 as many as $O(n^2)$ times, where n is the loop nest depth, and RTEST1 and RTEST2 call MIN and MAX, which performs at most $O(n)$ symbolic comparisons to determine monotonicity for each index, the range test performs at most $O(n^3)$ symbolic comparisons for one pair of array accesses $(A(f(\vec{\imath}))$ and $A(g(\vec{\imath}')))$. In practice, only a few permutations are examined and at most a constant number of symbolic comparisons are done by the monotonicity tests of MIN and MAX. So, the average number of symbolic comparisons done by the the range test is near $O(n)$.

Unfortunately, determining the costs of symbolic expression comparison is much more difficult. The worst case performance of symbolic comparisons is exponential on the size of the expressions compared and upon the number of variables in the program. However, the average case performance is much better.

4 Symbolic range propagation

To provide a facility for comparing symbolic expressions, we have developed a technique called *range propagation*. Due to space constraints, we will only give a brief sketch of this technique.

Range propagation consists of two parts: the range propagation algorithm and an expression comparison facility. The range propagation algorithm collects and propagates variable constraints through a program. The expression comparison facility uses these variable constraints to determine arithmetic relationships between two symbolic expressions.

The range propagation algorithm centers on the collection and propagation of symbolic lower and upper bounds on variables, called ranges, through a program unit. Abstract interpretation [8] is used to compute the ranges for variables at each point of a program unit. That is, the algorithm "executes" the program by following the control flow paths of the program, updating the current ranges to reflect the side effects of the statements encountered along these paths, until a fixed point is reached.

Each FORTRAN statement modifies the set of ranges entering it in the following way. An assignment statement sets the range for the left-hand side variable to the range computed from the right-hand side expression. A conditional statement constrains the entering ranges by the conditional's test. That is, the new ranges are calculated by determining the smallest upper bound and largest lower bound of the old ranges and the test. Similarly, at merge points of control flow, such as `ENDIF` statements, the ranges of these paths are merged. These merged ranges are computed by taking the largest upper bound and smallest lower bound of the ranges belonging to each of the entering control flow paths. To guarantee that the algorithm eventually reaches a fixed point and halts, a *widening operator* [8] is also applied to merge points that are loop headers. This widening operator sets a range to a conservative value if the range has changed too often during the course of computation.

Now, we will describe how the information collected by the range propagation algorithm can be used to compare symbolic expressions. We compare two expressions by calculating the integer range spanned by their difference, then determining whether this range is always positive or always negative. This integer range is calculated by repeatedly substituting ranges for variables in the difference expression then simplifying the expression down, until all variables are eliminated. Often, the simplification of expressions containing ranges needs to determine inequality relationships of its subexpressions, typically the subexpression's sign.

For example, suppose we wish to compare $x*y+1$ with y, where $x=[y:10]$, (meaning $y \leq x \leq 10$), and $y=[1:\infty]$. First, we calculate the difference, which is $x*y-y+1$. Then, we substitute $[y:10]$ for x in $x*y-y+1$, getting $[y:10]*y-y+1$. Simplifying this expression down, we get the range $[y*(y-1)+1:9*y+1]$. Since the simplified range still contains variables, we substitute $[1:\infty]$ for y, getting $[[1:\infty]*([1:\infty]-1)+1:9*[1:\infty]+1]$. After simplification, this becomes $[1:\infty]$. From this range, we can now see that $x*y+1>y$.

```
DO jl = 0, i2k - 1
  exj = ···
  DO jj = 0, x(jl)
    DO mm = 0, 128
      js = 258*i2k*jj + 129*jl + mm + 1
      js2 = js + 129*i2k
      h = data(js) - data(js2)
      data(js) = data(js) + data(js2)
      data(js2) = h * exj
    ENDDO
  ENDDO
ENDDO
```

Figure 7: Simplified version of loop nest FTRVMT/109 from *OCEAN*

5 Examples

In this section, we will provide examples of important loop nests, taken from the Perfect Benchmarks [4], that the range test can determine to be parallel but which conventional data dependence tests cannot.

One example is a loop nest taken from subroutine FTRVMT from the code *OCEAN*. This loop nest accounts for 44% of the code's sequential execution time on an Alliant FX/80. A simplified version of this loop is shown in Figure 7. Conventional data dependence tests cannot prove that these loops do not carry any dependences because of the $258*\mathtt{i2k}*\mathtt{jj}$ term in the subscripts for array `data`. The range test, on the other hand, can do so.

Table 1 shows a trace of the range test for proving that there are no loop-carried dependences for array access functions $f(\mathtt{jl},\mathtt{jj},\mathtt{mm}) = 258*\mathtt{i2k}*\mathtt{jj}+129*\mathtt{jl}+\mathtt{mm}+1$ and $g(\mathtt{jl},\mathtt{jj},\mathtt{mm}) = 258*\mathtt{i2k}*\mathtt{jj}+129*\mathtt{jl}+\mathtt{mm}+129*\mathtt{i2k}+1$. The final column of this table shows the symbolic expressions compared at the given statement of the range test algorithm (S_i) and the RTEST functions (R_j). The results of these comparisons were calculated from the variable constraints determined by the range propagation algorithm, which are $\mathtt{i2k} \geq 1$, $0 \leq \mathtt{jj} \leq \mathtt{x(jl)}$, $0 \leq \mathtt{jl} \leq \mathtt{i2k}-1$, and $0 \leq \mathtt{mm} \leq 128$. Since the upper bound `x(jl)` of loop `jj` is not monotonic, the test used $+\infty$ as an approximation of this bound. For this pair of access functions, the range test had to use Theorems 1, 2, and 3 and permute the `jl` loop inside the `jj` loop to prove that there are no loop-carried dependences.

Another important loop nest, which needs a dependence test for symbolic, nonlinear expressions, can be found in subroutine OLDA from the code *TRFD*. A simplified ver-

L_i	P_j	Stmt	Test	Comparison results		
mm		S_1	R_1	$258*\texttt{i2k}*\texttt{jj}+129*\texttt{jl}+129$	$<$	$258*\texttt{i2k}*\texttt{jj}+129*\texttt{jl}+129*\texttt{i2k}+1$
jj	mm	S_1	R_1	$+\infty$	$\not<$	$129*\texttt{jl}+129*\texttt{i2k}+1$
jj	mm	S_1	R_2	$+\infty$	$\not<$	$129*\texttt{jl}+1$
jj	mm	S_1	R_3	$258*\texttt{i2k}*\texttt{jj}+129*\texttt{jl}+129$	$<$	$258*\texttt{i2k}*\texttt{jj}+129*\texttt{jl}+387*\texttt{i2k}+1$
jj	mm	S_1	R_4	$258*\texttt{i2k}*\texttt{jj}+129*\texttt{jl}+129*\texttt{i2k}+129$	$<$	$258*\texttt{i2k}*\texttt{jj}+129*\texttt{jl}+258*\texttt{i2k}+1$
jl	jj	S_1	R_1	$+\infty$	$\not<$	$129*\texttt{i2k}+1$
jl	jj	S_1	R_2	$+\infty$	$\not<$	1
jl	jj	S_1	R_3	$+\infty$	$\not<$	$129*\texttt{jl}+129*\texttt{i2k}+130$
jl	jj	S_2	R_3	$258*\texttt{i2k}*\texttt{jj}+129*\texttt{i2k}$	$<$	$258*\texttt{i2k}*\texttt{jj}+387*\texttt{i2k}+1$
jl	jj	S_2	R_4	$258*\texttt{i2k}*\texttt{jj}+258*\texttt{i2k}$	$<$	$258*\texttt{i2k}*\texttt{jj}+258*\texttt{i2k}+1$
jl	mm	S_1	R_1	$258*\texttt{i2k}*\texttt{jj}+129*\texttt{i2k}$	$<$	$258*\texttt{i2k}*\texttt{jj}+129*\texttt{i2k}+1$

Table 1: Trace of range test for loop nest FTRVMT/109 shown in Figure 7

```
      mrsij0 = 0
      DO mrs = 0, (num*num+num)/2 - 1
        mrsij=mrsij0
        DO mi = 0, num - 1
          DO mj = 0, mi - 1
S1 :        mrsij = mrsij + 1
S2 :        xrsij(mrsij) = xij(mj)
          ENDDO
        ENDDO
        mrsij0=mrsij0+(num*num+num)/2
      ENDDO
```

Figure 8: Simplified version of loop nest OLDA/100 from *TRFD*

sion of this loop nest is shown in Figure 8. This loop nest accounts for 69% of the code's sequential execution time on an Alliant FX/80.

To parallelize this loop nest, induction variable substitution must be used to replace the induction variable `mrsij` at statement S_1 with the statement:

$$\texttt{mrsij} = (\texttt{mi}**2 - \texttt{mi} + \texttt{mrs}*(\texttt{num}**2+\texttt{num}))/2 + \texttt{mj} + 1.$$

After this substitution, conventional data dependence tests cannot prove that there are no self-dependences for `xrsij` at S_2 because of the nonlinear array subscript (after forward-substituting the value of `mrsij`). The range test, on the other hand, would have no difficulties in proving that this array has no self-dependences.

6 Related work

The range test was developed, independent of other dependence tests, to handle the symbolic array subscripts we encountered in actual programs. Early ideas of such a test were described in [12, 18, 6]. The most distinguished feature of the test may be the fact that it is now available in an actual compiler, which has proven to parallelize important programs to an unprecedented degree.

The following discussion compares our test to one of the most effective state-of-the-art tests and points out related ideas of other projects.

Mathematically, the range test can be thought of as an extension of a symbolic version of the Triangular Banerjee's Inequalities test with dependence direction vectors [1, 23], although our implementation differs. The only drawback of our test, compared to the Triangular Banerjee's test with directions, is that it cannot test arbitrary direction vectors, particularly those containing more than one '$<$' or '$>$' (e.g., $(<,<)$). The permutation of loop indices partially overcomes this drawback. (These permutations can be thought of as permutations of the dependence direction vectors tested.) We have found that this limited set of direction vectors, along with the permutation of loop indices, was sufficient to parallelize all of the relevant loop nests in our test suite. One advantage of the range test is that the worst case of the number of direction vectors tested is better than Banerjee's Inequalities with directions, since we test at most $O(n^2)$ direction vectors while Banerjee's Inequalities with directions may test as many as $O(3^n)$ direction vectors.

Haghighat and Polychronopoulos, presented a dependence test to handle nonlinear, symbolic expressions [15]. Their algorithm is essentially a symbolic version of Banerjee's Inequalities test. However, their test did not include the extensions to Banerjee's Inequalities to test dependence direction vectors and to handle triangular loops, nor does it include our extension to handle nonlinear expressions containing i^c terms, as in Figure 8 after induction variable substitution, where i is a loop index and c is an integer constant greater than 1. We have seen several examples in the Perfect Benchmarks that need all these extensions to be identified as parallel. The same authors presented ideas to calculate the set of variable constraints holding for each statement of the program unit, then to use these constraints to prove or disprove symbolic inequalities for dependence testing. We also determine variable constraints and perform symbolic inequality tests, although we use different techniques. We will compare these two methods later in this section.

In a separate paper, Haghighat and Polychronopoulos [16] describe a technique to prove that a symbolic expression is strictly increasing or decreasing. By using this

technique, self-dependences for an array reference can be eliminated. Their example can prove that all the loops in Figure 8, after induction variable substitution, are parallel. However, as described, the test only handles self-dependences. The subroutine OLDA in *TRFD* has other important loop nests that has multiple array accesses with nonlinear subscript expressions similar to the subscripts from Figure 8.

Maslov [19] presents an alternate way to handle symbolic, non-linear expressions. Instead of testing these expressions directly, his algorithm partitions the expression into several independent subexpressions, then tests these partitions using conventional data dependence tests. Essentially, it delinearizes array references. For example, it converts an array reference $A(n*i+j)$, where $1 \leq j \leq n$, into a two-dimensional array $A(j,i)$. The greatest strength of this technique is that it can convert non-linear expressions into linear ones, allowing exact data tests like the Omega Test [21] to be applied. Because of this, there are situations where Maslov's algorithm whereas we cannot, such as the array references $A(n*i+j)$ and $A(i+n*j)$, where $1 \leq i \leq j \leq n$. However, the delinearization algorithm cannot handle expressions containing terms of the form i^c, as in Figure 8 after induction variable substitution. Furthermore, the algorithm requires some additional symbolic capabilities; the compiler must be able to calculate symbolic gcd's and modulos, and the compiler must be able to sort the set of symbolic coefficients (c_j's). Performing this symbolic sort can be particularly difficult, since one may be unable to determine that some of the coefficients are less than others (i.e., the c_j's may not have a total ordering).

There has been some work in the determination of variable constraints. Much work has been done in determining the possible range, or interval, of values that variables can take, for the purpose of array bounds checking or program verification [17, 7]. These algorithms, however, only propagate integer ranges. Cousot and Halbwachs [9] offer a powerful algorithm for determining symbolic linear constraints between variables. (Their algorithm is used by Haghighat's symbolic dependence test to determine variable constraints.) Their algorithm is based upon the calculation, intersection, and merging of convex polyhedrons in the n-space of variable values. Although their algorithm is more accurate at calculating linear constraints than ours, their algorithm cannot handle nonlinear constraints such as $a < b*c$. Although not too common, we have seen cases where nonlinear bounds must be propagated or expressions with nonlinear differences must be compared.

7 Conclusions

We have developed a symbolic data dependence test, called the range test, that can identify parallel loops in the presence of nonlinear array subscripts and loop bounds. We have shown that the range test can prove that two very important loop nests in the Perfect Benchmarks are parallel, whereas conventional data dependence tests cannot. In our experiments, we have found that the range test can prove independence for many of the other parallel loops that contain symbolic non-linear array subscript expressions.

We have implemented the range test together with a symbolic range propagation algorithm in Polaris, a parallelizing compiler being developed at the University of Illinois [13, 5]. Currently, the range test is the only data dependence test implemented in Polaris. To determine its effectiveness, we have run it through an initial compiler test suite, which consists of half of the codes of the Perfect Benchmarks plus other applications gathered from users of high performance machines at the University of Illinois. We have found that in all cases, Polaris is able to parallelize these codes nearly as well as the hand-parallelized versions. For two of the codes, *TRFD* and *OCEAN*, current commercial parallelizing compilers can only achieve a speedup of at most 2 on Cedar, a parallel machine with 32 vector processors, due to false dependences seen for nonlinear array accesses. However, with the range test, along with other advanced techniques mentioned in [5], we are able to optimize the codes close to the hand parallelized versions, which reached a speedup of 43 for *TRFD* and 16 for *OCEAN*.

With the aid of memoization [20], or the caching of already tested array subscript pairs, we have found the execution time of the range test to be acceptable, even when applied as the only test. In future versions of Polaris, we will only invoke this test when other dependence tests fail due to nonlinear expressions. In these versions, the range test should not significantly increase the compiler's execution time. The range propagation algorithm can be somewhat costly, although not prohibitively so. Because of this, we will look into several techniques to improve its efficiency, such as using Static Single Assignment form [10], propagating ranges derived only from control flow, or propagating ranges only on demand [22].

References

[1] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer. Boston, MA., 1988.

[2] Utpal Banerjee. A Theory of Loop Permutations. In A. Nicolau D. Gelernter and D. Padua, editors, *Languages and Compilers for Parallel Processing*, pages 54–74. MIT Press, 1990.

[3] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2), February 1993.

[4] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker,

C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evalution of Supercomputers. *Int'l. Journal of Supercomputer Applications, Fall 1989*, 3(3):5–40, Fall 1989.

[5] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The Next Generation in Parallelizing Compilers. *Proceedings of the Seventh Annual Languages and Compilers for Parallelism Workshop, Portland, Oregon*, August 1994.

[6] William Blume and Rudolf Eigenmann. An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. *Proceedings of ICPP'94, St. Charles, IL*, 1994.

[7] Francois Bourdoncle. Abstract Debugging of Higher-Order Imperative Languages. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 46–55, June 1993.

[8] Partrick Cousot and Radhia Cousot. Abstract Interpretation: A unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[9] Partrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[11] Rudolf Eigenmann, Jay Hoeflinger, G. Jaxon, and David Padua. The Cedar Fortran Project. Technical Report 1262, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res, & Dev., April 1992.

[12] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

[13] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The Polaris Internal Representation. Technical Report 1317, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. and Dev., October 1993.

[14] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical Dependence Testing. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.

[15] Mohammad Haghighat and Constantine Polychronopoulos. Symbolic Dependence Analysis for High-Performance Parallelizing Compilers. *Parallel and Distributed Computing: Advances in Languages and Compilers for Parallel Processing, MIT Press, Cambridge, MA*, pages 310–330, 1991.

[16] Mohammad Haghighat and Constantine Polychronopoulos. Symbolic Analysis: A Basis for Paralleliziation, Optimization, and Scheduling of Programs. *Proceedings of the Sixth Annual Languages and Compilers for Parallelism Workshop, Portland, Oregon*, August 1993.

[17] William H. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[18] Jay Hoeflinger. Run-Time Dependence Testing by Integer Sequence Analysis. Technical Report 1194, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1992.

[19] Vadim Maslov. Delinearization: An Efficient Way to Break Multiloop Dependence Equations. *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 152–161, June 1992.

[20] D. Maydan, J. Hennessy, and M. Lam. Efficient and Exact Data Dependence Analysis. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[21] William Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[22] Peng Tu and David Padua. Demand-Driven Symbolic Analysis. Technical Report 1336, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., Febraury 1994.

[23] Michael Wolfe. Triangular Banerjee's Inequalities with Directions. Technical report, Oregon Graduate Institute of Science and Technology, June 1992. CS/E 92–013.

[24] Michael Wolfe and Utpal Banerjee. Data Dependence and its Application to Parallel Processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.