Non-Linear and Symbolic Data Dependence Testing *

William Blume
Hewlett-Packard, California
bblume@hpclopt.cup.hp.com

Rudolf Eigenmann
Purdue University, Indiana
eigenman@ecn.purdue.edu

August 25, 1998

Abstract

One of the most crucial qualities of an optimizing compiler is its ability to detect when different data references access the same storage location. Such references are said to be data-dependent and they impose constraints on the amount of program modifications the compiler can apply for improving the program's performance. For parallelizing compilers the most important program constructs to investigate are loops and the array references they contain. In previous work we have found a serious limitation of current data dependence tests to be that they cannot handle loop bounds or array subscripts that are symbolic, nonlinear expressions. 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 developed at the University of Illinois. We will present measurements of the Range Test's performance and compare it with state-of-the-art tests.

1 Introduction

Parallelizing compilers are necessary to allow programs written in standard sequential languages to run efficiently on parallel machines. In order to achieve good performance, these compilers must be able to identify the important loops whose iterations can be run concurrently, and transform these loops into parallel ones [3]. 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, 20, 27, 30, 33]. Modern 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*. An obviously nonlinear expression is a term i^2 or i^3. Less obvious is the fact that a term $c \cdot i$, where c is not a known, constant integer value, is also considered nonlinear by current tests. We have found both of these patterns in important parts of our application programs.

Because nonlinear expressions prevent the application of dependence tests, parallelizing compilers perform several analyses and optimizations to eliminate nonlinear expressions. For example, constant

*This work was supported by contract DABT63-95-C-0097 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.

```
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

propagation and induction variable substitution are used to remove unknown or non-constant variables. Constant propagation finds the values assigned to program variables and replaces occurrences of these variables with their values. Induction variable substitution replaces loop-variant (i.e., non-constant) expressions by terms that are closer to the affine forms understood by the dependence tests. Other techniques have also been developed to handle additive, loop-invariant, symbolic terms or to eliminate unwanted operations such as divisions [20, 27, 30].

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 [17, 18]. 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 [9]. 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 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, 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. The Range Test has been introduced in [6]. Since then, we have evaluated the test extensively and compared it with other data dependence tests.

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 – a prerequisite for the Range Test, which allows us to compare symbolic expressions. Examples of important loop nests that the Range

```
L1 : DO i1 = P1, Q1, R1
...    ...
Ln :     DO in = Pn, Qn, Rn
S1 :       A(f(i1, ..., in)) = ...
S2 :       ... = A(g(i1, ..., in))
         ENDDO
       ...
     ENDDO
```

Figure 2: Model of loop nest for dependence testing.

Test can parallelize but current tests cannot are given in section 5. Section 6 will present performance measurements. We will then compare our work with other symbolic data dependence tests in Section 7. Section 8 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, 33].

To ease the presentation of the Range Test, we will assume that we have a perfectly nested 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), the loop's lower and upper bounds (P_i and Q_i), and the loop's stride (R_i) may be arbitrary symbolic expressions made up of loop-invariant variables and loop indices (i.e. i_x) of enclosing loops. We will also assume that all loop strides (R_i) are positive. It is not difficult to extend our test to handle imperfectly nested loops, negative strides, multidimensional arrays, and loop-variant variables. We will discuss these generalizations in Section 3.6.

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,

$$\begin{aligned}\mathcal{R}_j = \{(\alpha_1, \ldots, \alpha_j) \quad : \quad & P_1 \leq \alpha_1 \leq Q_1, \ldots, P_j \leq \alpha_j \leq Q_j, \\ & (\alpha_1 - P_1) \bmod R_1 = 0, \ldots, (\alpha_j - P_j) \bmod R_j = 0\}\end{aligned}$$

The conditions $(\alpha_i - P_i) \bmod R_i = 0$ are required to make sure that each α_i can only take on values that are some multiple of the loop's stride from the loop's initial value. 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:

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

(This definition of direction vector is true only for loops with positive strides. For a loop L_i with a negative stride, the $<$ and $>$ cases of definition of d_i are swapped.) 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 the statements 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 $d_i =$ '<'. 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: in parallel loops different iterations usually 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., they represent array sections that are carved out of a large array, with start and length of the sections 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, so 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.

At a high level the Range Test works as follows: For a given iteration i of a loop L we consider the accessed array subscript range, $range(i)$, as a symbolic expression. Then, (somewhat simplified) if we can prove that this range does not overlap with the range accessed in the next iteration, $i+1$, then there is no cross-iteration dependence for L. The two ranges do not overlap if $max(range(i)) < min(range(i+1))$. In order to perform this test we need to be able to evaluate and compare minimum and maximum values of a (symbolic) range expression for a given loop L. Furthermore, the above test is only correct if this expression is monotonically increasing with i. For decreasing subscripts we test $max(range(i+1)) < min(range(i))$. Determining monotonicity is of further importance for computing the min and max functions themselves. $min(range)$ with respect to L is determined by substituting in the range expression all index variables of inner loops by their values that cause $range$ to be minimal. For example, if $range$ is monotonically increasing with index variable j then this variable is replaced by its lower bound. Determining monotonicity is simple for linear subscripts. However, it can also be done for many nonlinear expressions by testing whether the difference between two consecutive values is always positive or always negative. All these comparisons make use of symbolic expression manipulation capabilities. They also use information about the values assumed by the program variables, with is made known by the Range Propagation algorithm. Together, these capabilities give the Range Test its ability to test nonlinear and symbolic array subscripts. The following section describes the test formally.

3.1 Disproving dependence between symbolic expressions

An important basic capability of the Range Test is to determine the minimum and maximum value that an array index function can assume with respect to a particular loop in a nest. The formal definition of these minimum and maximum values are given below. Section 3.4 describes how to compute these values.

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

$$\begin{aligned} f_j^{\min}(i_1, \ldots, i_j) &\leq \min\left\{ f(\vec{i}') : \vec{i}' \in \mathcal{R}, i'_1 = i_1, \ldots, i'_j = i_j \right\} \\ f_j^{\max}(i_1, \ldots, i_j) &\geq \max\left\{ f(\vec{i}') : \vec{i}' \in \mathcal{R}, i'_1 = i_1, \ldots, i'_j = i_j \right\} \end{aligned}$$

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

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 a nest. 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 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 $\mathtt{A}(f(\vec{i}))$ and $\mathtt{A}(g(\vec{i}\,'))$ with a direction vector $\vec{d}$ of the form d_1 = '=', ..., d_j = '='.*

Proof. Suppose that such a dependence exists, (i.e., $f(\vec{i}) = g(\vec{i}\,')$ with direction vector $\vec{d}$). By Definition 1, we have $f(\vec{i}) \le f_j^{\max}(i_1,\ldots,i_j)$ and $g_j^{min}(i'_1,\ldots,i'_j) \le g(\vec{i}\,')$. 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 $f(\vec{i}) < g(\vec{i}\,')$. Contradiction. □

Theorem 1 currently can only disprove dependences, with direction vectors of the form $(=,\ldots,=,*,\ldots,*)$. With a small modification, it can be enhanced to disprove direction vectors of the form $(=,\ldots,=,<,*,\ldots,*)$. That is, a direction vector $\vec{d}$ of the form d_1 = '=', ..., d_j = '=' and d_{j+1} = '<'. This can be done by tightening the bounds on index i_{j+1} to be $P_{j+1} \le i_{j+1} \le Q_{j+1} - R_{j+1}$ when computing $f_j^{\max}(i_1,\ldots,i_j)$ and tightening the bounds on i'_{j+1} to be $P_{j+1} + R_{j+1} \le i'_{j+1} \le Q_{j+1}$ when computing $g_j^{min}(i_1,\ldots,i_j)$. Remember that we assume that the loop stride R_{j+1} is positive. Such an optimization is useful for disproving loop-carried dependences for ranges of array accesses that do not overlap except for the very first or very last iteration of the loop. In our experience, such ranges do occur in real programs.

Theorem 1 proves that there are no carried dependences between $\mathtt{A}(f(\vec{i}))$ and $\mathtt{A}(g(\vec{i}\,'))$ 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 the accesses are non-overlapping. 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 (i.e., within a single iteration the accesses are contiguous). 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.)

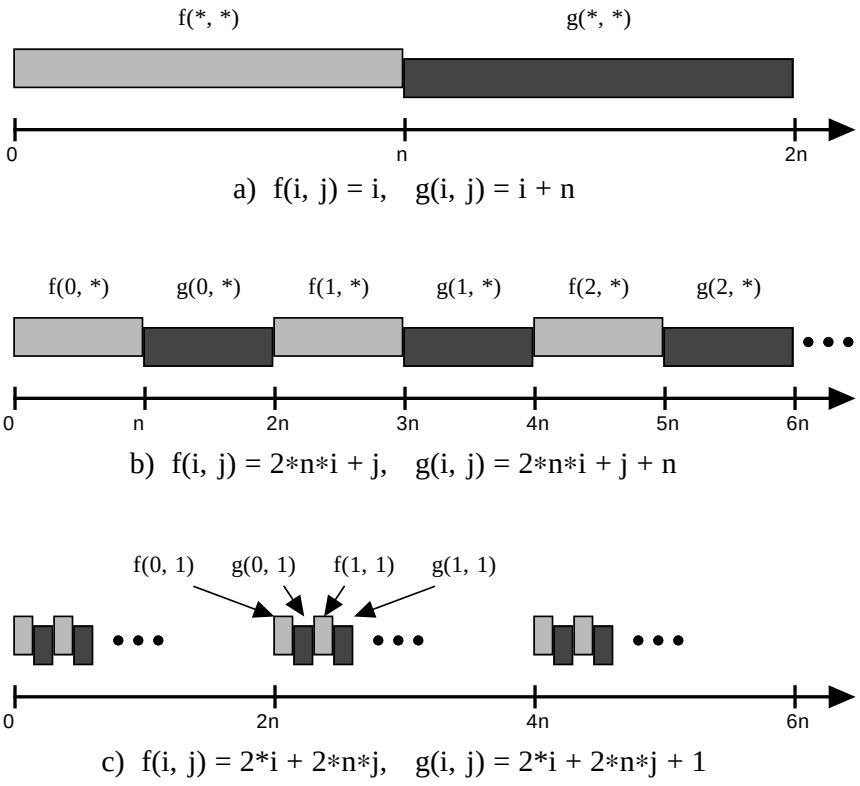

Figure 3: Examples of how array accesses can be interleaved in respect to a particular loop (loop with index i for these examples.) All examples assume that $0 \leq i, j < n$.

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

Similarly, a function $f(\vec{\imath})$ is monotonically non-increasing *for index i_j iff $f(i_1, \ldots, \alpha_j, \ldots, i_n) \geq f(i_1, \ldots, \beta_j, \ldots, i_n)$ whenever $P_j \leq \alpha_j \leq \beta_j \leq Q_j$.*

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 contiguous 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 $f_j^{\max}(i_1, \ldots, i_j) < g_j^{min}(i_1, \ldots, i_j + R_j)$ for all $(i_1, \ldots, i_j) \in \mathcal{R}_j$ and with $P_j \leq i_j \leq Q_j - R_j$, 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 $f(\vec{\imath}) \leq f_j^{\max}(i_1, \ldots, i_j)$ and $g_j^{min}(i'_1, \ldots, i'_j) \leq g(\vec{\imath}')$. Because of the direction vector $\vec{d}$ and because $g_j^{min}(i_1, \ldots, i_j)$ is monotonically non-decreasing for index i_j, $g_j^{min}(i'_1, \ldots, i'_j) \geq g_j^{min}(i_1, \ldots, i_j + R_j)$. (Remember that we assumed that R_j is always positive.) Since $f_j^{\max}(i_1, \ldots, i_j) < g_j^{min}(i_1, \ldots, i_j + R_j)$, 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 $f_j^{\max}(i_1, \ldots, i_j) < g_j^{min}(i_1, \ldots, i_j - R_j)$, for all $(i_1, \ldots, i_j) \in \mathcal{R}_j$ and with $P_j + R_j \leq i_j \leq Q_j$, 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 prove 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 for which we are attempting to disprove dependence are the innermost loops with indices i_j to i_n. The following subsection will show how we can change our notation such that these definitions and theorems still hold for arbitrary subsets of loops in a nest.

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 (as in Fig. 3a), and the tests from Theorems 2 and 3 can be used to prove independence when the values of f and g are interleaved, but the ranges are contiguous within each iteration (as in Fig. 3b). For more complex interleavings, the tests from all three Theorems would fail. Figure 3c gives an example of one of these more complex interleavings. The important observation that helps us deal with these situations nevertheless is, that these interleavings could be

made contiguous by permuting the loops. For example, if loop L_1, with index i, and loop L_2, with index j were "swapped" so that L_2 becomes the outermost loop, we would be able to use Theorem 1 to prove that there are no carried dependences in the now inner L_1 loop, and use Theorem 2 to prove that there are no carried dependences in the now outer L_2 loop.

Using this observation, 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 nest. The Range Test does not physically permute the loops; it is done logically during the analysis. One issue in doing so is to find the legal permutations. Ideally, we would consider all permutations of those loops that do not carry dependences in their actual position. However, because our test proves independence on the permuted loop nests, we use a more conservative criterion: We consider a loop as parallel if we can prove that independence of the permuted loop implies independence at the loop's actual position. In the terminology of Banerjee's Test, our conservative method means that we do not try the full tree of direction vectors. We have found our method to work very well in practice.

The advantage of our permutation method is that it allows us to use simple, contiguous representations of array accesses. Where the accesses are non-contiguous, we try to make them so during the analysis, using logical loop permutations. Alternatively, we could use an array access representation that can fully express non-contiguous reference patterns. While this would complicate the symbolic comparisons of the access ranges, it could obviate the need for the logical permutations. An effort to do so is underway in a related project [28].

Our algorithm determines a legal loop permutation by recursively finding a legal permutation of the inner loops, then finding a location where it may safely insert the next outer loop in this ordering. The final location of the outer loop is found by repeatedly moving inwards by one until it reaches a location where the test can prove that it carries no dependences, or the loop just inside this location either carries a dependence or would carry a dependence if the candidate loop was moved inside of it.

We will show that this method generates a legal permutation using Lemma 1, for whose proof we refer to Banerjee [2].

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.*

Intuitively, because our heuristic only moves a loop l inwards across loops L that don't carry dependences, we can always reverse this permutation. The reversal can be thought of as moving the L loops back inside l, which is correct according to Lemma 1.

We will show this more formally 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 j loops. For a nest of $j+1$ loops, the heuristic first recursively finds a permutation of the innermost j loops, then finds a location for the $(j+1)$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 $(j+1)$th loop inwards, all loops between the original and final positions of the $(j+1)$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 $(j+1)$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 $j+1$ 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_j 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 loop L_j carries any dependences. Loops proven not to carry dependences are added to the set D. The outer **for** loop

INPUT: Normalized, perfectly nested loops $(L_1, \ldots, L_n)$ and array access functions f and g.
OUTPUT: Set of loops D that do not have carried dependences between f and g.

$P \leftarrow ()$ $(*$ P is an ordered list representing the permuted loop nest $*)$
$D \leftarrow \emptyset$
for $j \leftarrow n$ **downto** 1 **do**
  placed $\leftarrow$ **false**
  $k \leftarrow 0$
  while not *placed* **do**
    $k \leftarrow k + 1$
S_1: **if** RTEST1$(f, g, \{L_j, P_k, \ldots, P_{|P|}\})$ **or** RTEST2$(f, g, L_j, \{P_k, \ldots, P_{|P|}\})$ **then**
      $(*$ Loop L_j does not carry dependences at this point. $*)$
      $D \leftarrow D \cup \{L_j\}$
      placed $\leftarrow$ **true**
S_2: **else if** $k = |P| + 1$ **or** $P_k \notin D$ **or not** RTEST2$(f, g, P_k, \{L_j, P_{k+1}, \ldots, P_{|P|}\})$ **then**
      $(*$ Loop L_j cannot be safely permuted any deeper. $*)$
      placed $\leftarrow$ **true**
    end if
  end while
S_3: $P \leftarrow (P_1, \ldots, P_{k-1}, L_j, P_k, \ldots, P_{|P|})$
end for

Figure 4: The Range Test algorithm.

visits each loop L_j while the inner **while** loop determines whether the current L_j carries any dependence and where to insert it in the list of permuted inner loops P. Statement S_1 tests if L_j 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_j 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_j 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_j be inserted inside of it.) Statement S_3 performs the actual insertion of L_j into P.

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 MAX$(f, \{L_{j+1}, \ldots, L_n\})$. The expression $f_j^{\min}$ is similar.) The function 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 $f_j^{\max}$

An array index function f is a symbolic expression involving several loop variables. Finding its maximum with respect to a subset of the loop variables is not straightforward. However, if we know that f is monotonically non-decreasing for some loop variable i, we can replace i with its upper bound to obtain

```
      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 + R_j
        else
          t ← t with i_j substituted by i_j − R_j
        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".

the maximum of f with respect to i.

Figure 6 shows how the Range Test generalizes this observation for computing the maximum of an expression for a given set of loops. The algorithm for computing the 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 $i_{j+1}, \ldots, i_n$.

The algorithm in Figure 6 includes a mechanism that speeds up the computation significantly. A naive implementation would not include the statements $x_1..x_7$ and instead compute the monotonicity of the expression y at statement s in each iteration of the for loop. This can be inefficient because each monotonicity computation for y requires a possibly expensive symbolic expression comparison. To avoid the cost of frequent recomputations of the monotonicity, the algorithm attempts to determine the monotonicity M from the precomputed monotonicity of the original array index expression f and the loop bound expressions. (The monotonicity states of all array accesses M_f and all loop bounds are computed only once, at the beginning of dependence testing of the program.) More specifically, the algorithm initially sets M equal to M_f (in statement x_1) then updates M after each substitution of a loop variable (i_k) with its bound (P_k, or Q_k, resp.), using the monotonicity information in M_{P_k} and M_{Q_k} of the substituted loop bound expressions. The basic idea of the update is this: Consider a subscript expression y_{before}, in which the index variable i_k is substituted by $bound$, resulting in y_{after}. Assume that y_{before} increases (monotonically) with respect to i_k. Then, y_{after} is certain to increase

```
    expression function MAX(f, L)
        y ← f
x1 :    M ← M_f
        for each L_k ∈ L from innermost to outermost loops do
x2 :        m ← M(k)
x3 :        if (m = UNKNOWN) then
s :             {compute the value of m using the symbolic comparison algorithm}
x4 :        endif
            case m
                NON-DEC:   y ← y, with Q_k substituted for i_k
x5 :                       M = M * M_{Q_k}
                NON-INC:   y ← y, with P_k substituted for i_k
x6 :                       M = M*flip-mono(M_{P_k})
                CONST:     {do nothing}
                NON-MONO:  y ← +∞
                           exit (for loop)
            end case
x7 :        M(k) =CONST
        end for
        return y
    end function
```

Used variables and operators:	
m	Monotonicity state (MS). The range of values is (NON-DEC,NON-INC,CONST,NON-MONO,UNKNOWN).
M	Vector of monotonicity states. $M(k)$ is the MS with respect to loop index k.
$M_1 * M_2$	Pairwise tests "compatibility" of the elements of M_1 and M_2: for each pair the result is equal to one element if the other element is the same or CONST, otherwise the result is UNKNOWN.
flip-mono()	Changes NON-DEC to NON-INC and vice-versa. It does no other changes.

Figure 6: Algorithm for calculating the maximum value of function f over all iterations of the loop set $\mathcal{L}$. ($\mathcal{L}$ is a subset of the loops in a given nest. Index values of loops $\notin \mathcal{L}$ remain fixed.)

with respect to some other index i_j, if both y_{before} and *bound* increase with respect to i_j. Several cases of monotonicity states for i_k, i_j, and y_{before} have to be considered, as expressed by the algorithm in statements x_5 and x_6. Statement x_7 finally sets the new monotonicity state with respect to the now eliminated variable to CONST. We have found this optimization to be very effective in practice. For many array accesses, the monotonicity never needs 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 ($A(f(\vec{i}))$ and $A(g(\vec{i}'))$). 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.

3.6 Generalizing the Range Test

In our description of the Range Test, we made some assumptions about the form of loop-nests to ease its presentation. More specifically, we assumed that all array accesses are one-dimensional, the loops have a positive stride, none of the array accesses nor loop bounds contain loop-variant variables that are not loop indices, and that the enclosing loops between the two accesses being tested are perfectly nested. Most of these assumptions can be removed with simple modifications of the Range Test algorithm described so far. Our implementation includes these extensions.

Multidimensional arrays are handled by applying the Range Test to each dimension of the array subscript, then intersecting all the sets of loops that we found to carry dependences.

Negative strides are dealt with through a modification of Theorems 2 and 3 and the functions MIN and MAX. In Figure 6 we swap the substitutions of upper bounds (Q_x) with the substitutions of lower bounds (P_x) when loop L_k has an always negative stride. This is necessary because the loop limit Q_x is less than the starting value P_x for negative strides.

For the Theorems 2 and 3, one needs to swap the terms "monotonically non-decreasing" and "monotonically non-increasing" in the theorems when loop L_j has an always negative stride. This swapping of terms is necessary because of the definition of direction vectors for loops with negative strides. That is, a dependence with a dependence direction $d_j =$ '$<$', where loop L_j has a negative stride, means that there is a dependence between two iterations i_j and i'_j of L_j, where $i_j > i'_j$. To ensure that the theorems stay correct, (see the proof of Theorem 2), one must invert the monotonicity condition on $g_j^{min}(i_1, \ldots, i_j)$ for index i_j.

For strides that cannot be proven to be always positive or always negative, our implementation of the Range Test is very conservative and marks the loops with these strides as loops that carry dependences.

Loop-variant variables are variables whose value may change within a loop and that are not loop indices of enclosing loops. We can modify the functions MIN and MAX to eliminate from their results all variables that are loop-variant for the loops in $\mathcal{L}$. A loop-variant variable can be eliminated by substituting it with its range computed by the Range Propagation facility. Function RTEST2X must also be modified to eliminate all loop-variant variables for loop L_j from expressions s and t, (see Figure 5).

Loops that aren't perfectly nested include a set of loops that only enclose the access $A(f(\vec{\imath}))$ and/or a set of loops that only enclose the access $A(g(\vec{\imath}))$. Let $\mathcal{L}_{only\_f}$ and $\mathcal{L}_{only\_g}$ be the name of these sets, and let $\mathcal{L}_{both}$ be the loop nest enclosing both accesses.

The two accesses cause a data dependence between iteration $\vec{\imath}_{both}$ and $\vec{\imath}'_{both}$ if the total range accessed by $\mathcal{L}_{only\_f}$ in iteration $\vec{\imath}_{both}$ overlaps with the range accessed by $\mathcal{L}_{only\_g}$ in iteration $\vec{\imath}'_{both}$. In order to factor in these ranges we need just modify the functions $\text{MIN}(f, \mathcal{L})$ and $\text{MAX}(g, \mathcal{L})$ to always substitute the indices of the loops in $\mathcal{L}_{only\_f}$ and $\mathcal{L}_{only\_g}$. This will ensure that any computation of these functions also includes the maximum (or minimum) of all the values that the access expression can take for all iterations of $\mathcal{L}_{only\_f}$ and $\mathcal{L}_{only\_g}$, respectively.

4 Symbolic range propagation

To provide a facility for comparing symbolic expressions, we have developed a technique called *range propagation*. We will only give a brief sketch of this technique and refer the interested reader to [7].

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 [13] 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 effects of the statements encountered along these paths, until a fixed point is reached.

Each FORTRAN statement updates the set of ranges 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. For example, in the body of the IF-statement `IF (a<100) THEN <BODY> ENDIF` the range of the variable `a` is known to be less than 100. This is done by determining the smallest upper bound and largest lower bound of the old ranges and the conditional's test. Similarly, at merge points of the 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 from the merging control flow paths. To guarantee that the algorithm eventually reaches a fixed point and halts, a *widening operator* [13] 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, 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 $[y : 10]$

```
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*

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

Table 1: Trace of Range Test for loop nest FTRVMT/109 shown in Figure 7

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 $x * y + 1 > y$.

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 * i2k * 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 $f(\texttt{jl}, \texttt{jj}, \texttt{mm}) = \texttt{258} * \texttt{i2k} * \texttt{jj} + \texttt{129} * \texttt{jl} + \texttt{mm} + \texttt{1}$ and $g(\texttt{jl}, \texttt{jj}, \texttt{mm}) = \texttt{258} * \texttt{i2k} * \texttt{jj} + \texttt{129} * \texttt{jl} + \texttt{mm} + \texttt{129} * \texttt{i2k} + \texttt{1}$. 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 constraints on variables determined by the range

```
      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*

propagation algorithm, which are $\texttt{i2k} \geq 1$, $0 \leq \texttt{jj} \leq \texttt{x(jl)}$, $0 \leq \texttt{jl} \leq \texttt{i2k} - 1$, and $0 \leq \texttt{mm} \leq 128$. 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. The key steps that helped the Range Test succeed in this loop were to (1) symbolically express the array access range of the loop body with respect to each of the enclosing loops, (2) for each enclosing loop, factoring in the symbolic bounds and finding the first and last access of the range, (3) proving non-overlap by comparing these access boundaries symbolically in many possible loop permutations.

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 version 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. The key property on the Range Test in this example is that it can deal with truly nonlinear subscripts. It can do so as long as it can determine (symbolic) lower and upper bounds of the involved array access ranges.

6 Measurements

To measure the effectiveness and speed of the Range Test, we compared its results with the Omega Test [30]. We chose this comparison because the Omega test is among the most accurate data-dependence tests available. In terms of efficiency, the Omega test may not be the ideal comparison because, as others have pointed out, simpler and faster tests are equally powerful in practice [29]. Nevertheless, the Omega test serves as an interesting reference point.

Roughly, the Omega Test is a variant of integer Fourier-Motzkin analysis [16, 31] with optimizations to make the common cases fast. For affine array subscripts and loop bounds, the Omega Test is an exact data dependence test. The Omega Test handles non-affine expressions using uninterpreted function symbols. Our implementation of the Omega Test uses the Omega Library version 0.91 [25].

Since uninterpreted function symbols are the Omega Test's solution to non-affine expressions, the functionality of uninterpreted function symbols needs some further explanation. An uninterpreted function symbol is simply a variable with one or more arguments, (e.g., $f(i,j)$), representing a side-effect-free function. The Omega Test accepts affine expressions extended to also contain uninterpreted function symbols, (e.g., $i + 2 * f(i)$). The current implementation of the Omega Test only allows the loop indices of enclosing loops to be the arguments of uninterpreted function symbols. Two identical uninterpreted function symbols can be cancelled out or combined together if all their arguments are equal, (e.g., $f(i) - f(i') = 0$ if $i = i'$). Because our implementation of our interface to the Omega Test adds no constraints on the values that these uninterpreted function symbols can take, the Omega Test can apply no other kind of simplification on uninterpreted function symbols.

Our interface to the Omega Test handles non-affine expressions by translating them into uninterpreted function symbols. For example, to set up a dependence test between $A(n*i+j)$ and $A(n*i+j+1)$, the interface would translate the non-affine expression $n * i$ into the uninterpreted function symbol $f(i)$. Thus, the interface would feed the constraint $f(i) + j = f(i') + j' + 1$ to the Omega Test. We also use uninterpreted function symbols to handle loop-variant variables. That is, variables whose values change for some loop but are not loop indices.

6.1 Effectiveness

To measure the effectiveness of the Range Test and Omega Test, we counted the number of loops found parallel by these techniques as well as the number of loop-carried dependences eliminated. The results of these measurements is shown in Table 2. These results were run on a subset of the Perfect Benchmarks, two National Center of Supercomputing Applications (NCSA) codes, and most of the Fortran codes in the Spec92 benchmarks. Before running either the Range or Omega tests we detected obvious dependences with some very simple data dependence tests. The most important of these simple tests were the GCD test and a simple test that eliminated dependences between $A(i)$ and $A(i)$ for a loop with index i. We counted an eliminated loop-carried dependence arc multiple times if that arc carried dependences for multiple loops.

We broke our results into three categories. The *Both tests* category displays the number of loops that were found parallel and the number of loop-carried dependences eliminated by both the Range and Omega tests. The *Range only* category displays the number of loops found parallel and loop-carried dependences eliminated by the Range Test but not the Omega Test. Similarly, the *Omega only* category displays the number of loops found parallel and loop-carried dependences eliminated by the Omega Test but not the Range Test.

All of the advanced restructuring techniques developed and implemented in Polaris were used before dependence testing. These techniques include partial inlining, interprocedural symbolic constant propagation with procedure cloning, array privatization, generalized induction variable substitution, and reduction recognition. Because of this, dependence arcs from reductions, induction variables, and private arrays and scalars have already been eliminated when the Range and Omega Tests were executed. Details of these advanced techniques can be found in [11, 10].

From Table 2, we can see that there are cases where the Range Test does better, and cases where the Omega Test does better. This should not be surprising, because the Omega Test has difficulties with non-affine expressions while the Range Test was designed to handle such cases. On the other hand, the Omega test is exact for affine expressions while the Range Test is not.

To get a better understanding why one test was more successful than the other for some cases, we examined every loop-carried dependence eliminated by only the Range Test or only the Omega Test. For the Range Test, almost every dependence arc that only it eliminated were dependences between non-affine array accesses. Most of these cases were due to the linearization of arrays from partial inlining or from induction variable substitution. The only cases where additional dependence arcs eliminated by the Range Test were not from non-affine array accesses were all the additional loop-carried dependences eliminated for WAVE5. For these cases, the Range Test used the constraint $1252 \leq n \leq 50080$, which was

		detected by		(additional) detections by only one test			
	Number	*Both tests*		*Range Test only*		*Omega Test only*	
Code	*of lines*	*Par. Loops*	*L.C. Deps.*	*Par. Loops*	*L.C. Deps.*	*Par. Loops*	*L.C. Deps.*
ARC2D	4694	2	216	0	0	0	0
BDNA	4887	29	462	0	0	6	46
FLO52	2368	8	37	1	4	0	0
MDG	1430	13	128	6	23	1	9
OCEAN	3285	136	1725	76	884	0	0
TRFD	634	6	60	6	35	0	21
CLOUD3D	14438	1	200	0	12	0	0
CMHOG	11286	11	17	16	16	0	0
DODUC	5334	2	68	0	0	0	0
FPPPP	2718	4	176	0	0	0	0
HYDRO2D	4461	3	6	3	6	0	0
MDLJDP2	4136	0	9	0	6	1	3
MDLJSP2	3885	0	9	0	6	1	3
NASA7	1204	17	108	0	39	0	2
ORA	453	2	51	0	0	0	9
SU2COR	2514	44	1059	0	0	2	568
SWM256	487	12	24	0	0	0	0
TOMCATV	195	3	13	0	0	0	5
WAVE5	7628	124	864	15	174	0	2

Table 2: Number of parallel loops and eliminated loop-carried dependences detected by the Range Test and the Omega Test. (The numbers include non-obvious dependences only).

generated by Range Propagation from a conditional statement just before the loop, to break dependences between accessed pairs such as $A(i)$ and $A(i+n)$ or $A(i+n)$ and $A(i+51332)$, where $1 \leq i \leq 1252$. Although it does not exist in our implementation, an interface to the Omega Test could be developed that makes available such additional constraints as well.

The Omega Test sometimes did better than the Range Test for several reasons. The most common reason is coupled subscripts. Coupled subscripts are dependences between multidimensional array accesses where one can disprove dependences by examining all the dimensions together, but can't disprove dependences by testing each dimension, one by one. Almost all of the coupled subscripts that we've seen were one of the access pairs below:

- Between $A(i,j)$ and $A(j,j)$ where $i < j$,
- Between $A(i,j)$ and $A(j,i)$ where $i < j$,
- Between $A(i,i+j)$ and $A(i+j,i)$,
- Between $A(i,c)$ and $A(c,i)$ where c is an integer.

Other coupled subscripts were simple variants of the above four types. Coupled subscripts accounted for about half of the additional loop-carried dependences eliminated by only the Omega Test for codes BDNA and SU2COR, and all of the additional loop-carried dependences for codes MDG, NASA7, and TOMCATV.

The other case where the Omega Test sometimes did better than the Range Test were for those cases where the ranges of two array accesses overlapped, but this overlap is solely due to a dependence between the two accesses for the same loop iteration. This case occurred for all the additional loop-carried dependences eliminated by only the Omega Test for ORA, MDLJDP2, and MDLJSP2. One example, taken from ORA, is a dependence between $A(i+400)$ and $A(i+50*j+300)$, where $1 \leq i \leq 19$. A dependence exists between these two accesses only when $j = 2$ and $i = i'$, where i' is the value

of the i index for the second access. Another example, taken from MDLJDP2 amd MDLJSP2, is a dependence between $A(l)$ and $A(4*f(i,j,k)+l)$, where $1 \leq l \leq 4$, and $f(i,j,k)$ is actually a complicated non-affine expression. Now the Range Test is powerful to analyze this non-affine expression, which is $f(i,j,k) = n^2 * i + n * j + k - n^2 - n$, to determine that $f(i,j,k) \geq 0$. However, it is not smart enough to see that the coefficient 4 on the term $4 * f(i,j,k)$ guarantees that there can only be a dependence between iteration l' and l'' of the loop with index l if and only if $l' = l''$, since $1 \leq l \leq 4$. On the other hand, the Omega Test can prove this, even though it does not know the values that f can take.

Another reason for why the Omega Test sometimes did better than the Range Test was that our interface to the Omega Test was able to replace complex loop-invariant expressions with symbolic constants but our interface to the Range Test did not. For example, for the code BDNA, the dependence tests need to disprove dependences between $A(i)$ and $A(i+2*x(1))$, where $1 \leq i \leq x(1)$. Our interface to the Omega Test replaces the loop-invariant $x(1)$ term with a symbolic constant t, getting $A(i+2*t)$, before feeding it to the Omega Test. Since these transformed accesses are affine, the Omega Test disproves the dependence between the two accesses. However, the Range Test receives these accesses in their raw form. Since the current implementation of Range Propagation cannot determine constraints on the $x(1)$ term, it cannot prove that $i \leq x(1) < 1 + 2 * x(1) \leq i + 2 * x(1)$ since it doesn't know that $x(1)$ is always positive. Thus, the Range Test fails for this pair of accesses. However, if some pre-processing pass was written to replace all loop-invariant expressions with symbolic constants, (e.g., replace all $x(1)$s with t), the Range Test would also succeed. Because of this, we do not consider this case to be a shortcoming to the Range Test, just its interface. This case occurred for half of additional loop-carried dependences eliminated by only the Omega Test for BDNA and SU2COR. (The other half were coupled subscripts, described above.)

The final reason why the Omega Test sometimes did better than the Range Test for our measurements was because the Omega test can always break cross-iteration dependences for loops with only one iteration while the Range Test cannot. Being that there is no benefit in parallelizing single iteration loops, we do not consider this to be a weakness of the Range Test for our purpose. This case occurred for the cross-iteration dependences eliminated only by the Omega Test for TRFD and WAVE5.

Overall, the Range Test was able to determine that more loops were parallel than the Omega Test. Additionally, we found that most of the loops that were identified as parallel by only the Range Test were loops that take up a significant fraction of the program's execution time, while the loops identified as parallel by only the Omega Test were all insignificant. Thus, for our test suite, the Range Test has a greater impact than the Omega Test in identifying significant amounts of loop-level parallelism in real programs.

6.2 Speed

One of the biggest arguments made by compiler developers against symbolic data dependence tests such as the Range Test is that they are too slow. Although the Range Test is slower than other dependence tests such as the GCD or Banerjee's Inequalities test, since it manipulates symbolic expressions rather than integers, we believe that the Range Test is efficient enough for use in commercial parallelizing compilers.

To support this assertion, we have measured the execution times taken by the Range and Omega tests to perform the dependence testing for the measurements displayed in Table 2 and discussed in the previous section. These timings are displayed in Table 3. These timings were collected on a Sparc 10. Our measurements were collected from Polaris, which was compiled with g++ 2.6.3 with the -O flag. The columns *Range Test* and *Omega test* show the times taken by the Range and Omega Tests to perform the dependence tests in the previous experiment.

To give the reader an idea of the significance of these timings compared to the rest of the compiler, we also included timings for all the preprocessing performed before dependence testing and timings of the rest of the dependence testing pass. The preprocessing phase includes the time to parse the Fortran codes as well as several restructuring techniques, including the advanced techniques described

Code	*Preprocessing*	*Range test*	*Omega test*	*Rest of dd testing*
ARC2D	300	20	49	160
BDNA	620	41	110	280
FLO52	250	17	18	83
MDG	200	43	74	140
OCEAN	930	400	300	100
TRFD	170	46	69	24
CLOUD3D	735	52	140	760
CMHOG	1200	180	250	3800
DODUC	600	8	39	450
FPPPP	1000	7	1800	1000
HYDRO2D	100	0.8	3	15
MDLJDP2	110	3	8	18
MDLJSP2	110	3	10	18
NASA7	110	31	33	100
ORA	13	0.9	0.3	2
SU2COR	1200	350	4800	2700
SWM256	23	0.7	0.4	5
TOMCATV	18	0.6	0.3	5
WAVE5	1300	48	71	540

Table 3: Seconds of time taken by the Range and Omega tests on real programs. Timings for the rest of dependence testing as well as all the analyses performed before dependence testing are also included.

in the previous subsection. Because some of these advanced techniques may significantly increase code size, (e.g., partial inlining and interprocedural constant propagation with procedure cloning), the time spent applying these advanced techniques and dependence testing may be much greater than in other parallelizing compilers. The timings of the rest of dependence testing pass includes timings of the simple dependence tests described in the previous subsection, as well as timings of the functions that determine and iterate over all possible dependences that need to be tested in a program unit, that create a dependence graph, and that identify parallel loops.

From Table 3, one can see that although the Range Test does take a significant amount of time in a parallelizing compiler, it does not dominate that execution time. In the worst case, it only took about a third of the execution time of Polaris. In the average case, it took much less. Additionally, it was on average about twice as fast as the Omega Test. In a few cases, it was much faster. Thus, we feel confident in claiming that the Range Test is efficient enough to be incorporated in commercial parallelizing compilers.

7 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 [17, 24, 9]. 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 [5]. 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, 32], which is one of the most effective state-of-the-art tests. (However, our approach and implementation are substantially different.) 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 this approach 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. The same holds for the Omega test, which also can test all direction vectors.

Haghighat and Polychronopoulos, presented a dependence test to handle nonlinear, symbolic expressions [21]. 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 important 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 constraints on variables 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 constraints on variables and perform symbolic inequality tests, although we use different techniques. We will compare these two methods later in this section.

In a separate paper, the same authors [22] 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 [26] presents an alternate way to handle symbolic, nonlinear 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 de-linearizes 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 nonlinear expressions into linear ones, allowing exact data tests like the Omega Test [30] to be applied. Because of this, there are situations where Maslov's algorithm succeeds whereas the Range Test does not, such as the array references $A(n * i + j)$ and $A(i + n * j)$, where $1 \leq i \leq j \leq n$. However, the de-linearization 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 constraints on variables. 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 [23, 12]. These algorithms, however, only propagate integer ranges. Cousot and Halbwachs [14] offer a powerful algorithm for determining symbolic linear constraints between variables. (Their algorithm is used by Haghighat's symbolic dependence test to determine constraints on variables.) 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.

8 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 nonlinear 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 [19, 10]. 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 the Cedar multiprocessor, a parallel research 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 [10], 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 [27], 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 have developed several techniques to improve its efficiency, such as using Static Single Assignment form [15], propagating ranges derived only from control flow, or propagating ranges only on demand [8].

References

[1] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer, Boston, Mass., 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):211–243, 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 Evaluation of Supercomputers. *Int'l. Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[5] 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. Parallel Programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.

[6] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proceedings of Supercomputing '94, Washington D.C.*, pages 528–537, November 1994.

[7] William Blume and Rudolf Eigenmann. Symbolic Range Propagation. *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, April 1995.

[8] William Blume and Rudolf Eigenmann. Demand-Driven, Symbolic Range Propagation. *Lecture Notes in Computer Science, 1033: Languages and Compilers for Parallel Computing*, pages 141–160, 1996.

[9] William Blume and Rudolf Eigenmann. An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. *Proceedings of the 1994 International Conference on Parallel Processing*, pages II233 – II238, August, 1994.

[10] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the Effectiveness of Parallelizing Compilers. *Lecture Notes in Computer Science, 892: Languages and Compilers for Parallel Computing*, pages 141–154, 1995.

[11] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel and Distributed Technology*, 2(3):37–47, Fall 1994.

[12] François 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.

[13] 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.

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

[15] 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.

[16] R. J. Duffin. On Fourier's Analysis of Linear Inequality Systems. *Mathematical Programming Study 1*, pages 71–95, 1974.

[17] 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*, pages 65–83, August 1991.

[18] Rudolf Eigenmann, Jay Hoeflinger, and David Padua. On the Automatic Parallelization of the Perfect Benchmarks. *IEEE Trans. Parallel Distributed Syst.*, 9(1), January 1998.

[19] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The Polaris Internal Representation. *International Journal of Parallel Programming*, 22(5):553–586, October 1994.

[20] 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.

[21] 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, Mass.*, pages 310–330, 1991.

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

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

[24] 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.

[25] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library (version 0.91) Interface Guide. Technical report, University of Maryland, February 1995.

[26] 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.

[27] D. Maydan, J. Hennessy, and M. Lam. Efficient and Exact Data Dependence Analysis. In *SIGPLAN NOTICES: Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 26-28*, pages 1–14. ACM Press, 1991.

[28] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of Array Access Patterns for Compiler Optimizations. In *Proceedings of the ACM SIGPLAN 98 Conference on Programming Language Design and Implementation*, June 1998.

[29] Paul M. Petersen and David A. Padua. Static and Dynamic Evaluation of Data Dependence Analysis. *Proceeding of the Int'l Conference on Supercomputing, ICS'93, Tokyo, Japan*, pages 107–116, June 1993.

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

[31] H. P. Williams. Fourier's method of Linear Programming and its Dual. *The American Mathematical Monthly*, 93(9):681–695, November 1986.

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

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