Symbolic Range Propagation *

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

September 20, 1994

Abstract

Many analyses and transformations in a parallelizing compiler can benefit from the ability to compare arbitrary symbolic expressions. In this paper, we describe how one can compare expressions by using symbolic ranges of variables. A range is a lower and upper bound on a variable. We will also describe how these ranges can be efficiently computed from the program text. Symbolic range propagation has been implemented in Polaris, a parallelizing compiler being developed at the University of Illinois, and is used for symbolic dependence testing, detection of zero-trip loops, determining array sections possibly referenced by an access, and loop iteration-count estimation.

1 Introduction

To effectively parallelize real programs, parallelizing compilers need powerful symbolic analysis techniques [11, 3]. One of most useful of these techniques is the ability to compare arbitrary symbolic expressions, using constraint information derived from the program [3]. Many transformations in a parallelizing compiler can benefit from such an ability. Examples are symbolic dependence testing, detection of zero-trip loops, dead-code elimination, determination of array sections referenced by an array access, and loop trip-count estimation. We will examine two of these examples, symbolic dependence testing and detection of zero-trip loops, in detail.

The ability to compare symbolic expressions is probably most useful for symbolic data dependence tests. In fact, two symbolic data dependence tests, the authors' Range Test [4] and the Symbolic Banerjee's Inequalities Test as suggested by Haghighat and Polychronopoulos [11], are built upon this ability.

The Range Test proves that two array accesses do not have a loop-carried dependence by proving that the range of possible values referenced by one access does not overlap with the range of possible values for the other access. It proves this by determining that the symbolic upper bound of one of these ranges is less than the symbolic lower bound of the other range. This requires the ability to compare symbolic expressions.

The real strength of the Range Test is that it can prove independence for non-affine (nonlinear) array references, which most other dependence tests cannot do. For example, the Range Test can prove that all the loops in Figure 1 are parallel but, to our knowledge, no other dependence test can do so. This loop nest was taken from TRFD, a code in the Perfect Benchmarks, and accounts for 28% of the code's serial execution time. For the Range Test to prove that there are no dependences for the writes to array

*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.

```
    DO i = 0, n - 1
      DO j = 0, i
        DO l = 0, i - j
S1 :      x = ((i^2 + i + 2 * j) * (n^2 + n + 2))/4 + l + 1
          A(x) = ...
        ENDDO
        DO k = 0, n - i - 2
          do l = 0, k + i + 1
S2 :        x = ((i^2 + i + 2 * j) * (n^2 + n + 2))/4
                +i - j + k * (i + 1) + (k^2 + k)/2 + l + 2
            A(x) = ...
          ENDDO
        ENDDO
      ENDDO
    ENDDO
```

Figure 1: A loop nest, extracted from TRFD, that contains non-affine array references. Loop peeling and induction variable substitution were performed to place the loop nest into this form.

A, it needs to compare non-affine symbolic range bounds derived from statements $S1$ and $S2$, under the constraints imposed by the enclosing loops, (e.g., $0 \leq i \leq n-1$, $n \geq 1$). For example, one of the comparisons performed by the Range Test for this example was to compare $((i+1)*(n^2+n))/2+i+k+2$ to 0. We have seen several other important loop nests from real applications that needed the Range Test with range propagation to effectively parallelize them.

The ability to compare symbolic expressions is also needed to prove that a loop is not a zero-trip loop. By knowing that a loop is not a zero-trip loop, one can peel off an iteration of a loop or generate a last-value assignment for an induction variable eliminated by induction variable substitution without enclosing this iteration or last-value assignment with a conditional statement that tests whether the loop was a zero-trip loop. Eliminating this conditional is desirable because conditional statements often weaken the results of global analyses, (e.g., constant propagation and induction variable substitution). For example, both loop peeling and induction variable substitution was needed to get the loop nest in Figure 1 into the form shown. Without the ability to detect zero-trip loops, these transformations would have inserted conditional statements that would have prevented the outermost loops from being parallelized.

In response to this need for comparing arbitrary symbolic expressions under constraints derived from the program unit, we have designed the range propagation algorithm. Range propagation centers upon the computation and manipulation of *ranges*. A range, denoted $[a : b]$, consists of a symbolic lower bound a and a symbolic upper bound b. In this paper, we will always assume that $a \leq b$ and that x does not occur in either a or b. Constraints are represented by a mapping from program variables to ranges. For example, assigning the range $[a : b]$ to a variable x denotes the constraint $a \leq x \leq b$. This mapping from variables to ranges will be called a *range dictionary*.

Range propagation consists of two major parts: an expression comparison algorithm and a range propagation algorithm. The expression comparison algorithm determines which inequality relationships (e.g., $<, =, \geq$) hold between two expressions, given a set of constraints in a range dictionary. The range propagation algorithm determines the constraints that hold at each point in a program. It does this by computing the ranges that initially hold for each variable, then propagating these ranges through the program unit. Since the range propagation algorithm needs to compare symbolic expressions to simplify ranges, we will discuss the expression comparison algorithm first.

Section 2 will describe how one can compare two expressions, using constraints represented by a range dictionary. Section 3 will then describe how these range dictionaries can be computed from a program unit. Related work will then be discussed in Section 4. Section 5 will give our conclusions and

plans for future work.

2 Comparing expressions using symbolic ranges

In this section, we will first give an overall sketch of our expression comparison algorithm, which determines the relationship between two arbitrary symbolic expressions under the constraints given by a range dictionary. We will then discuss how the two most important functions of the algorithm are implemented. A cost analysis of this algorithm will then be performed.

2.1 Algorithm

This algorithm assumes that the compiler can manipulate and simplify arbitrary symbolic expressions. Efficient symbolic analysis techniques for parallelizing compilers can be found in [11, 12, 14].

Roughly, we determine the inequality relationship between two symbolic expressions p and q, in respect to a range dictionary R, by forming the difference d between p and q, then repeatedly replacing (substituting) each variable x in d with the range of x, until we reach a point where the lower bound of d is a non-negative integer or the upper bound of d is a non-positive integer. We can then determine the relationship between p and q from the bounds of this final value of d. For example, $p > q$ if the lower bound of the final value of d is an integer greater than zero.

The expression comparison algorithm, which implements the intuitive description above, is shown in Figure 2. Statement $S1$ determines the difference d between p and q. (It is assumed that d has been simplified down to its simplest form.) Statement $S2$ determines the best order to replace variables in d with their ranges. This order is stored in an ordered list S. The ordered list S contains all variables that may eventually need to be replaced in d, including variables that are not initially in d. Statement $S3$ is a while loop that repeatedly removes a variable x from the list S and replaces x in d with its range $R(x)$, until we run out of variables in S or d is comparable. (We say that a range d is comparable when its lower bound is a positive integer or its upper bound is a negative integer.) Statement $S4$ handles the case where the while loop ran out of variables to replace before it could make d be comparable. This loop simply replaces all remaining variables in d with the unconstrained range $[-\infty : +\infty]$. (We do this rather than failing because there are cases where d would be comparable if the variables were eliminated, (e.g., $d = [\max(1, x) : +\infty]$).) Statement $S5$ returns the inequality relationship between p and q by examining the lower and upper bounds of d. If it cannot determine any relationships, it returns the *unknown* inequality relationship.

For example, suppose we wish to compare $x * y + 1$ with y, where $x = [y : 10]$ and $y = [1 : \infty]$. First, we calculate the difference, which is $d = x * y - y + 1$. Then, we replace x with $[y : 10]$, getting $d = [y^2 - y + 1 : 9 * y + 1]$, (we'll describe how we computed this new value for d in the next subsection). Since d is still not comparable, we replace y with $[1 : \infty]$, getting $d = [1 : \infty]$. Since the lower bound of d is greater than zero, we have the relationship $x * y + 1 > y$.

To implement the algorithm in Figure 2, one must implement two major functions: `replacement_order()` and `replace_var()`. The function `replacement_order()` attempts to determine the best order to replace variables with their ranges, while the function `replace_var()` performs the actual replacements. The next two subsections will describe these functions in detail.

2.2 Replacing a variable with its range

The most important and most complex part of the expression comparison algorithm is the replacement of variables with their ranges, (i.e., `replace_var()`). There are several methods that such a replacement can be performed. In this section, we will describe two methods: a faster but less accurate algorithm, which substitutes each occurrence of the variable with its range, and a slower but more accurate algorithm, which determines the new range of the expression from its monotonicity.

INPUT: Symbolic expressions p and q and range dictionary R,
which maps each variable x to a range $R(x) = [a : b]$.
OUTPUT: Inequality relationship between p and q
(i.e, $<$, $\leq$, $=$, $>$, $\geq$, or ? (unknown)).

$S1$: $d \leftarrow [q - p : q - p]$
$S2$: $S \leftarrow$ `replacement_order`(d, R)
$S3$: **while** (d is not comparable **and** S is not empty) **do**
  $x \leftarrow S$.head
  $S \leftarrow S$.tail
  $d \leftarrow$ `replace_var`(d, x, $R(x)$, R)
  end while
  if (d is not comparable) **then**
$S4$: **foreach** variable x in d **do**
    $d \leftarrow$ `replace_var`(d, x, $[-\infty : +\infty]$, R)
    end foreach
  endif
$S5$: **return** inequality relationship for d

Figure 2: The symbolic expression comparison algorithm

Range substitution method

The most intuitive method to replace a variable with its range is to physically substitute each occurrence of the variable in the given expression with the variable's range, then simplify the resulting expression. This simplification transforms the resulting expression into a range whose bounds do not contain ranges. As an example, suppose we wish to replace the range $[1 : y]$ for variable x in the difference range $d = [x^2 - x : \infty]$. The resulting range can be computed by first substituting $[1 : y]$ for x in d, getting $[[1 : y]^2 - [1 : y] : \infty]$, then simplifying this range down to $[1 - y : \infty]$, (assuming that $y \geq 1$).

By far the most complicated part of this method is the simplification of the substituted expression. We perform this simplification by working from the innermost to the outermost operations of the expression, applying rewrite rules at each point. These rewrite rules transform a subexpression whose arguments are range expressions into a range subexpression whose arguments do not contain ranges. The rewrite rules are displayed in Table 1. All these rewrite rules assume that we are working with integer-valued symbolic expressions and that all ranges' lower bounds are less than or equal to their upper bounds. The rest of this paper will also make these assumptions.

As an example, consider the simplification of $[[1 : y]^2 - [1 : y] : \infty]$, taken from the previous example. The steps to simplify this expression are:

$$\begin{array}{rcll}
[[1 : y]^2 - [1 : y] : \infty] & \Rightarrow & [[1^2 : y^2] - [1 : y] : \infty] & \text{by rule (6)} \\
 & \Rightarrow & [[1^2 : y^2] + [-y : -1] : \infty] & \text{by rule (4)} \\
 & \Rightarrow & [[1^2 - y : y^2 - 1] : \infty] & \text{by rule (3)} \\
 & \Rightarrow & [1^2 - y : \infty] & \text{by rule (1)} \\
 & \Rightarrow & [1 - y : \infty] &
\end{array}$$

Note that some of the rewrite rules in Table 1, particularly those for multiplication, division, and exponentiation, require the comparison of symbolic expressions to zero. For example, to simplify expression $x * [1 : 10]$, one must determine the sign of x. This can be easily done by recursively calling the expression comparison algorithm in Figure 2. The costs of performing these recursive expression comparisons can be greatly decreased by caching the signs of variables and using these cached signs whenever possible.

$$
\begin{array}{rcll}
[[a:b]:c] & \Rightarrow & [a:c] & (1)\\
[a:[b:c]] & \Rightarrow & [a:c] & (2)\\
[a:b]+c & \Rightarrow & [a+c:b+c] & (3)\\
[a:b]*c & \Rightarrow & \begin{cases} [a*c:b*c] & \text{if } c \geq 0 \\ [b*c:a*c] & \text{if } c \leq 0 \\ [-\infty:\infty] & \text{otherwise} \end{cases} & (4)\\
[a:b]/c & \Rightarrow & \begin{cases} [a/c:b/c] & \text{if } c > 0 \\ [b/c:a/c] & \text{if } c < 0 \\ [-\infty:\infty] & \text{otherwise} \end{cases} & (5)\\
[a:b]^i & \Rightarrow & \begin{cases} [a^i:b^i] & \text{if } i \text{ is even and } a \geq 0 \\ [b^i:a^i] & \text{if } i \text{ is even and } b \leq 0 \\ [a^i:b^i] & \text{if } i \text{ is odd} \\ [-\infty:\infty] & \text{otherwise} \end{cases} & (6)\\
\min([a:b],c) & \Rightarrow & [\min(a,c):\min(b,c)] & (7)\\
\max([a:b],c) & \Rightarrow & [\max(a,c):\max(b,c)] & (8)
\end{array}
$$

Table 1: Rewrite rules for simplifying expressions containing ranges. Variables a, b, and c are assumed to be arbitrary symbolic integer-valued expressions. The variable i is an integer.

Because this method may make recursive calls to the expression comparison algorithm, it could conceivably go into an infinite loop. For example, in the course of determining whether $x \geq 0$, the algorithm may need to prove that $x \geq 0$. To guarantee termination, we do not allow the range for variable x to be substituted in any recursive calls to the expression comparison algorithm while the expression comparison algorithm is trying to replace x in some expression. Instead, we substitute the unconstrained range $[-\infty:\infty]$ for x. This effectively prevents any more recursive expression comparisons, since the simplification of ranges containing $[-\infty:\infty]$ does not require them. (We did not include the rewrite rules for expressions containing $[-\infty:\infty]$, since they are easy to determine.) Since there are only a finite number of variables, there can only be a finite number of invocations of the expression comparison algorithm on the stack at one time.

It is often desirable to have MIN and MAX expressions in the range expressions for variables. To uncover more opportunities for simplification, these MIN and MAX subexpressions should be moved outward so that they enclose the entire expression, (although they shouldn't be pulled out of ranges). Thus, when we substitute a range containing MINs and MAXs into an expression, we must pull these MINs and MAXs outward to the top levels of the expression. This can be done by using rewrite rules similar to those given earlier for simplifying range expressions. These rules are displayed in Table 2.

In addition to the range, MIN, or MAX specific simplifications given earlier, we apply conventional symbolic simplification techniques to the expression. These techniques include constant folding, distribution of products-of-sums, and the combination and cancellation of common symbolic terms [6, 11, 12, 14]. We also use advanced techniques to simplify expressions containing integer divisions, which were developed by Haghighat [12]. In addition to Haghighat's exact techniques, we use an approximate technique that handles divisions by multiplying them out from the difference range d from Figure 2. For example, we convert the expression $d = a/b + c$ to the range $d' = [a - b + 1 + b * c : a + b * c]$, assuming that $a > 0$ and $b > 0$. (One can see that this range is correct, since $b * (a/b) = b * ((a - a \bmod b))/b) = a - a \bmod b = a - [0 : b - 1] = [a - b + 1 : a]$.)

$$\min(\min(a,b),c) \Rightarrow \min(a,b,c) \tag{9}$$

$$\min(a,b)+c \Rightarrow \min(a+c,b+c) \tag{10}$$

$$\min(a,b)*c \Rightarrow \begin{cases} \min(a*c,b*c) & \text{if } c \geq 0 \\ \max(a*c,b*c) & \text{if } c \leq 0 \end{cases} \tag{11}$$

$$\min(a,b)/c \Rightarrow \begin{cases} \min(a/c,b/c) & \text{if } c > 0 \\ \max(a/c,b/c) & \text{if } c < 0 \end{cases} \tag{12}$$

$$\min(a,b)^i \Rightarrow \begin{cases} \min(a^i,b^i) & \text{if } i \text{ is even and } \min(a,b) \geq 0 \\ \max(a^i,b^i) & \text{if } i \text{ is even and } \min(a,b) \leq 0 \\ \min(a^i,b^i) & \text{if } i \text{ is odd} \end{cases} \tag{13}$$

Table 2: Rewrite rules for simplifying min expressions. The rewrite rules for max expressions is similar. Variables a, b, and c are assumed to be arbitrary symbolic integer expressions. Variable i is an integer.

Monotonicity replacement method

One problem with the range substitution method from the previous subsection is that it can generate overly conservative lower or upper bounds when it replaces a variable in an expression that has multiple occurrences of that variable. This occurs because both bounds of the range of the variable to be replaced are used to compute the lower or upper bound of the resulting range expression, although it is impossible for a variable to be equal to both its lower and upper bounds, (assuming that its lower bound does not equal its upper bound). For example, when $[1:y]$ was substituted for x in $x^2 - x$ in the example from the previous subsection, the final lower bound of the resulting expression $(1-y)$ was formed by taking the lower bound of $[1:y]$ for the x^2 term while taking the upper bound of $[1:y]$ for the $-x$ term. Since there is no legal value of x in the range $[1:y]$ where $x^2 - x$ would take on the value $1-y$, this bound is overly conservative.

In this subsection, we will present a variable replacement method that can determine the exact lower and upper bounds for an expression, if it succeeds. (If it fails, we can still use the variable substitution method.) We will assume that we are replacing variable x in expression $f(x)$ with range $[a:b]$.

The exact lower and bounds for $f(x)$ after replacing x are the minimum and maximum values that $f(x)$ can take for any value of x in the range $[a:b]$. Determining these minimum and maximum values are simple if we can prove that $f(x)$ is *monotonic* for x. The expression f is *monotonically non-decreasing* for x within range $[a:b]$, if $f(x) \geq f(x')$ whenever $x \geq x'$ for $x, x' \in [a:b]$. Similarly, f is *monotonically non-increasing* for x within range $[a:b]$, if $f(x) \leq f(x')$ whenever $x \geq x'$ for $x, x' \in [a:b]$. Now, if $f(x)$ is monotonically non-decreasing for x, then the range of values taken by $f(x)$ is $[f(a):f(b)]$. Similarly, if $f(x)$ is monotonically non-increasing for x, then the range of values taken by $f(x)$ is $[f(b):f(a)]$.

Determining whether $f(x)$ is monotonically non-decreasing or monotonically non-increasing is not difficult. One can prove that $f(x)$ is monotonically non-decreasing for x by proving that $f(x+1)-f(x) \geq 0$. This inequality can be easily proven by recursively calling the expression comparison algorithm in Figure 2. Similarly, one can prove that $f(x)$ is monotonically non-increasing for x by proving that $f(x+1)-f(x) \leq 0$.

The monotonicity replacement method, when used by itself, can be very expensive. (A back-of-the-envelope calculation estimated it to be at least $O(v!)$, where v is the number of variables in the program.) Because of this, we do not use the monotonicity replacement method to determine the sign of the forward difference expression $f(x+1)-f(x)$. Instead, we use the range substitution method given in the previous subsection. This greatly improves the worst case performance of this technique while preserving much of its accuracy. This also guarantees that this method will eventually halt, since the range substitution method eventually halts. Detailed analysis of its time complexity will be done later in this section.

As an example, suppose we wish to replace the range $[1 : y]$ for variable x in the difference range $d = [x^2 - x : \infty]$. To calculate the new lower bound of d, we must first determine whether the lower bound is monotonically non-increasing or monotonically non-decreasing for x. This is done by calculating the forward difference $(x+1)^2 - (x+1) - (x^2 - x) = 2 * x$, then determining whether $2 * x$ is greater-equal or less-equal to zero by a recursive call to the expression comparison algorithm. Now, $2 * x$ is greater than zero, since $2 * x \Rightarrow 2 * [1 : y] \Rightarrow [2 : 2 * y]$ has a positive lower bound. Hence, $x^2 - x$ must be monotonically non-decreasing and the range of $x^2 - x$ is $[1^2 - 1 : y^2 - y] = [0 : y^2 - y]$. After doing a similar computation for the upper bound of d, we determine the new range of d to be $[0 : \infty]$. Note that this lower bound is better than the lower bound $1 - y$ computed from the range substitution method in the previous subsection

One complication in using monotonicity for replacing ranges for variables is that either the range we are replacing with or the given expression may contain MIN or MAX expressions. Replacing a variable in a MIN or MAX expression causes problems because the monotonicity of the arguments of the MIN or MAX may differ. Having MINs or MAXs in the range we are replacing for a variable causes problems because we wish to have all MINs and MAXs in the resulting expression to be the outermost terms so to aid simplification.

With a few additional rules, these MIN or MAX expressions can be easily handled. To replace a range for variable x in $\min(f(x), g(x))$, one simply performs the monotonicity replacement method for x on $f(x)$ to get $f(a)$, the same method for x on $g(x)$ to get $g(b)$, then return the expression $\min(f(a), g(b))$. Pulling MIN and MAX expressions out of an resulting expression is also simple. If $f(x)$ is monotonically non-decreasing, then $f(\min(a, b))$ is transformed into $\min(f(a), f(b))$. Similarly, if $f(x)$ is monotonically non-increasing, then $f(\min(a, b))$ is transformed into $\max(f(a), f(b))$. The rules for handling expressions with MAXs are similar.

As an example, suppose we wish to replace x with $[1 : \min(10, y)]$ in $d = [\max(x^2 - x, -x), \infty]$. Since the previous example has shown that $x^2 - x$ is monotonically non-decreasing, its lower bound is $1^2 - 1 = 0$. Similarly, since $-x$ is monotonically non-increasing, its lower bound is $-\min(10, y)$, which can be rewritten to $\max(-10, -y)$. Thus, the new value of d is $[\max(0, \max(-10, -y)) : \infty] = [\max(0, -y) : \infty]$.

2.3 Determining a replacement order

The order in which one replaces ranges for variables is very important. A poorly chosen replacement order may require more replacements and result in less accurate ranges than a well chosen replacement order. For example, suppose we wish to compute the relationship between x and y, where $x = [1 : y]$ and $y = [1 : \infty]$. If we replace x then replace y in $d = x - y$, we get:

$$\begin{aligned} d &= x - y \\ &= [1 : y] - y \\ &= [1 - y : 0] \end{aligned}$$

and we can determine that $x \leq y$, since the upper bound of d is zero. On the other hand, if we replace y first, we get:

$$\begin{aligned} d &= x - y \\ &= x - [1 : \infty] \\ &= [-\infty : x - 1] \\ &= [-\infty : [1 : y] - 1] \\ &= [-\infty : y - 1] \\ &= [-\infty : [1 : \infty] - 1] \\ &= [-\infty : \infty] \end{aligned}$$

In this case, we would not be able to determine any relationship between x and y.

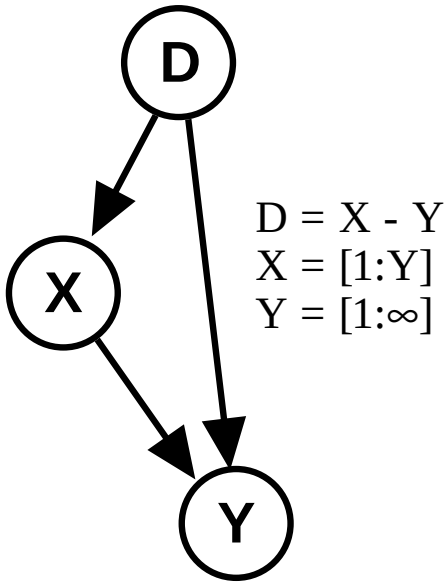

Figure 3: An example of a Range Dependence Graph (RDG).

To determine a good order to replace variables, we create a Range Dependence Graph (RDG), then determine an ordering for its vertices. Each vertex in the RDG represents a variable that may need be replaced at some point to convert the difference range d into a range that is comparable by our definition. The RDG also contains a vertex for the initial value of d. An edge exists between vertices x and y if and only if the range for variable x contains the variable y. An example of an RDG is shown in Figure 3. Note that RDGs may contain cycles.

We determine a good replacement order by determining the strongly-connected components[1] (SCCs) in the RDG, then topologically sorting[2] these SCCs[3]. This topological ordering of the SCCs gives an good overall replacement order of the variables between SCCs in the RDG, but does not specify any replacement order for variables within a SCC in respect to each other. To find a replacement order for variables within a SCC, we perform a topological sort of each SCC ignoring back-edges.[4] We use an arbitrary vertex with an edge originating from outside the SCC as the root (start) node for this internal topological sort. Efficient algorithms for finding SCCs and back-edges, and performing topological sorts can be found in [7].

As an example, we will compute the replacement order for the RDG graph shown in Figure 4. The algorithm first computes the strongly connected components of the graph, then topologically sorts them into the order (SCC1, SCC2, SCC3, SCC4). The algorithm then topologically sorts the vertices in each SCC, ignoring back-edges. The orders for SCC1 and SCC4 are trivial, ((t) and (z) respectively). The order for SCC2 is computed by choosing an arbitrary vertex with an incoming edge, in this case the vertex u, then topologically sorting SCC2 ignoring the back edge $w \rightarrow u$, getting the order (u, v, w). The

[1] A strongly-connected component of a graph is a maximal subgraph of that graph where each vertex in the subgraph can reach all other vertices in the subgraph.

[2] A topological ordering of a directed acyclic graph is an ordering such that if there is a path from vertex u to vertex v then u occurs before v in this ordering.

[3] By definition of SCCs, one can always topologically sort the SCCs of a graph in respect to each other.

[4] Back-edges are edges in a graph, which if deleted would result in an acyclic graph.

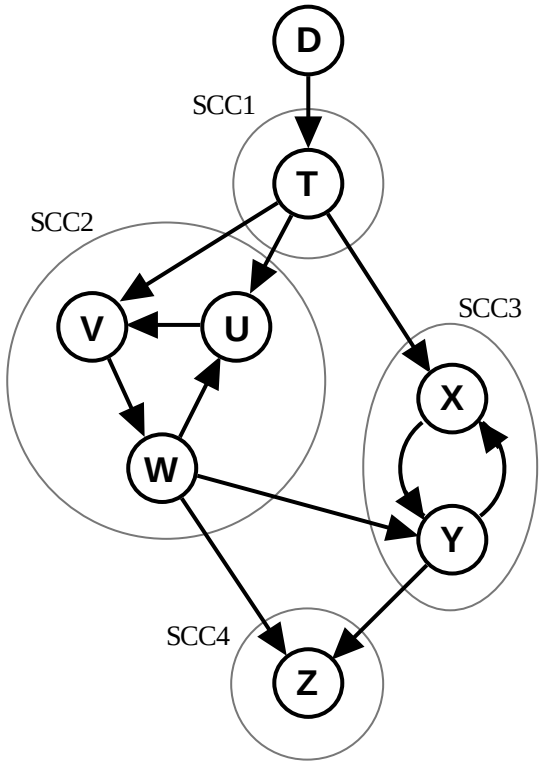

Figure 4: An example of a cyclic Range Dependence Graph.

ordering for SCC3, which is (x, y), is computed similarly. The final replacement order is then formed by concatenating these individual orderings in the order of the SCC ordering. This results in the order (t, u, v, w, x, y, z).

For acyclic RDGs, it can be seen that the computed replacement order is optimal in the number of replacements. However, it may not be optimal for cyclic RDGs. This may be because we chose the wrong root node to start the topological sort in the SSC, or may be that the optimal replacement order may require a variable to be visited more than once. To partially overcome this problem, we repeat the replacement order of each multiple vertex SCC in the final replacement order. The SCC's variable order and its duplicate variable order are adjacent to each other in the final replacement order. For example, the replacement order for Figure 4 would be $(t, u, v, w, u, v, w, x, y, x, y, z)$. We have found this heuristic to be effective for many of the cyclic RDGs we have seen in practice.

2.4 Time complexity

In our experience, nearly all the time spent by the expression comparison algorithm in Figure 2 is in performing variable replacements. Thus we feel justified to characterize the performance of the algorithm by the number of variable replacements (i.e., calls to `replace_var()`) it performs. Computing this number of replacements is a bit more complicated than it looks, since each variable replacement may make a recursive call to the expression comparison algorithm, which can perform several variable replacements itself.

To ease the computation of the number of variable replacements performed by the expression comparison algorithm, we will assume that the only recursive comparisons performed by the range substitution

method will be those to determine the uncached signs of variables. We feel justified to make such an assumption because it (nearly) holds for all expressions that are polynomials in a fully simplified, sums-of-products form possibly divided by an integer and enclosed by one or more MINs or MAXs;[5] nearly all expressions derived from real programs are of this form. Alternately, one may modify the algorithm to guarantee that the assumption holds.

If we use the range substitution method to replacing variables with ranges, no recursive calls to the expression comparison algorithm would be needed if the the signs of all variables in a program is known and stored in the sign cache. In this case, only $O(v)$ replacements would need to be made, where v is the number of integer variables in the program, or more accurately, the number of variables in the RDG. However, if none of the variable signs are cached, then the algorithm may perform up to $O(v)$ recursive calls to the expression comparison algorithm, (each performing $O(v)$ replacements), to determine the signs of the $O(v)$ variables. Thus, the expression comparison algorithm may perform at most $O(v^2)$ variable replacements.

For the monotonicity replacement method, as many as $O(v)$ recursive calls to the expression comparison algorithm would be performed to determine the monotonicity of the $O(v)$ variables. By design of this method, each of these recursive calls can only invoke the range substitution method. Thus, this method will perform at most $O(v^2)$ variable replacements if all the sign caches were already filled. If none of the sign caches were filled, it would still make at most $O(v^2)$ replacements, since the filling of the sign caches only add $O(v^2)$ additional replacements.

In practice, the average number of variable replacements required by the expression comparison algorithm is very small. In our experience with the Range Test, which performs many symbolic comparisons, about one-third to one-half of the invocations of the expression comparison algorithm do not perform any substitutions. Ignoring these cases, the average number of variable replacements performed varied from 2 to 5, depending upon the program being tested.

Determining the cost of a single variable replacement is a bit more difficult. Its cost equals the cost of performing the physical substitution of the variable's range into the original expression plus the cost of simplifying the resulting expression down. The costs of both of these transformations are functions of the sizes of the expression and ranges involved. Unfortunately, each transformation can cause a multiplicative growth in the size of the final expression. This makes the worst-case time complexity of the expression comparison algorithm to be exponential on the sizes of of the expressions being compared and the ranges in the range dictionary. In practice, however, little growth occurs in expression size, with one major exception. This little growth occurs because constant folding or the cancellation of common terms. The one exception mentioned are expressions that contain MINs and MAXs. We will discuss heuristics in the next section that prevent many of these MINs and MAXs from occurring in the ranges in the range dictionary, hence limiting the growth in expression size. We have found these heuristics to be effective at preventing exponential expression growth in practice.

3 Propagating ranges

The range propagation algorithm centers on the collection and propagation of 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 its control flow paths, updating the current ranges to reflect the side effects of the statements encountered along these paths, until a fixed point is reached. This section will describe how to compute ranges for FORTRAN 77 programs. Similar algorithms can be designed for other languages.

[5] This assumption may not hold for exponentiation terms. (See rewrite rules (6) and (13) from Tables 1 and 2.) However it does hold for most of the cases we've seen.

$$
\begin{aligned}
[a:b] \cup [c:d] &\Rightarrow [\min(a,c) : \max(b,d)] && (14)\\
[a:b] \cap [c:d] &\Rightarrow [\max(a,c) : \min(b,d)] && (15)\\
[a:b] \bigtriangledown [c:d] &\Rightarrow [\textbf{if } a = c \textbf{ then } a \textbf{ else } -\infty : \\
&\quad\ \textbf{if } b = d \textbf{ then } b \textbf{ else } \infty] && (16)\\
[a:b] \bigtriangleup [c:d] &\Rightarrow [\textbf{if } a \neq -\infty \textbf{ then } a \textbf{ else } c : \\
&\quad\ \textbf{if } b \neq \infty \textbf{ then } b \textbf{ else } d] && (17)
\end{aligned}
$$

Table 3: Basic operations used by the range propagation algorithm.

3.1 Basic operations

To compute the ranges for each statement in the program unit, some basic operations to merge or join ranges is required. These basic operations, which are displayed in Table 3, are the union ($\cup$), intersection ($\cap$), widening ($\bigtriangledown$), and narrowing ($\bigtriangleup$) operations. The union operator is used to merge ranges coming from multiple points in the control flow of the program. The intersection operator is used to add new constraint information to a range. This new constraint information typically originates from conditional tests or loop bounds. The widening operator, which is essentially an overly conservative union operator, is used at selected points of the program unit to guarantee termination. The narrowing operator is used to regain some of the information lost by the widening operator. Our definitions of the narrowing and widening operators were influenced by the operators given by Bourdoncle [5].

The range propagation algorithm uses a special range, denoted as $\top$, which represents an undefined value. The union operator, when applied upon $\top$ and a range x, has the following identity.

$$x \cup \top = \top \cup x = x$$

The application of the other operators on $\top$ and x have the same identity.

As mentioned in Section 2, the expression comparison algorithm can be very expensive when the variables' ranges contain MINs and MAXs. Because of this, we attempt to eliminate the new MINs and MAXs introduced by the union and intersection operators in Table 3 by using the expression comparison algorithm to determine the MIN (or MAX) argument with the smallest (or greatest) value. Also, in response to an observation that expression comparison is often slowest for range bounds that contain both MINs and MAXs, we disallow MIN expressions to occur in range lower bounds and MAX expressions to occur in range upper bounds. More specifically, if the union operator is unable to eliminate the MIN (or MAX) from its lower (or upper) bound, it replaces the bound with $-\infty$ (or $+\infty$).

3.2 Algorithm

The ranges for a program unit are computed in two very-similar phases: a widening phase then a narrowing phase. For each of these phases, mappings of variables to their ranges, (i.e., range dictionaries), are associated with each statement and each control-flow edge of the program unit. In the widening phase, iterative data-flow analysis [1] is used to compute the ranges for each statement and control flow-edge of the program. The ranges for all variables for all program entry points is initially defined to be $[-\infty : \infty]$. All other statements and control flow-edges are initially assigned a range dictionary whose ranges are undefined ($\top$). The range dictionary for a statement is the union of the ranges of all the entering control-flow edges for that statement. The range dictionaries for the exiting control-flow edges of a statement are computed by modifying a copy of the statement's range dictionary with the side effects of the statement. The modifications made for each kind of statement is as follows: 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 intersects the entering range dictionary with ranges derived from

the conditional's test. A similar intersection operation is done for the bounds of `DO` loops. To guarantee that the algorithm eventually reaches a fixed point and halts, a widening operator is also applied on the statement's range dictionary at all loop headers. The arguments to the widening operator is ranges from statement's new and old range dictionaries. This widening operator is only applied for such nodes when the ranges for all entering control-flow edges were defined for both the new and old range dictionaries for the statement when they computed. (This causes the algorithm to visit the loop-header node twice before it applies the widening operator.) The narrowing phase is identical to the widening phase, except that its initial ranges are the ranges computed from the widening phase, and the narrowing operator is applied at loop-header nodes instead of the widening operator.

Variable modifications by a statement must also be taken into account when computing the ranges for the statement's exiting control-flow edges. We take such variable modifications into account by eliminating all occurrences of modified variables in the exiting control-flow edges' range dictionaries. (We do this because the ranges are in terms of these variables before they are modified, and will thus be incorrect after this statement.) A simple way to perform this is to replace all occurrences of that variable with the variable's range. However, a more accurate result can be achieved for variable modifications caused by a special class of assignment statement called an *invertible assignment* [9]. Invertible assignments are assignment statements where the variable on the left-hand-side also occurs on the right-hand-side, and where this assignment can be rewritten so that the old value of this variable equals some function of the new value of this variable; that is, if $x_{new} = f(x_{old})$ then one can determine some inverse f' of f such that $x_{old} = f'(x_{new})$. For example, $x = x + 1$ is an invertible assignment, where the inverse of $x + 1$ equals $x - 1$. For variable modifications caused by invertible assignments, one can replace all occurrences of the variable with the inverse of the assignment's right-hand-side. For example, if $y = [1 : x]$ before the invertible assignment $x = x + 1$, then the new range for y is $[1 : x - 1]$.

3.3 Example

An example of the ranges generated by the range propagation algorithm for a small code fragment is shown in Figure 5. Only the ranges for variable x are shown. We will discuss how the ranges for x were computed for only a few statements in the code fragment: The range after statement S_5 is determined by simply computing the range of $x + 1$, which is $[1 : n] + 1 = [2 : n + 1]$. (Remember that the range of x is not allowed to contain x.) The range inside the `THEN` part of the `IF` statement at S_3 was calculated by intersecting the old range $[1 : 2 * n]$ with the range $[-\infty : n]$, resulting in $[\max(1, -\infty) : \min(2 * n, n)] = [1 : n]$. The range after the `ENDIF` statement at S_8 was calculated by calculating the union of the two ranges ($[2 : n + 1]$ and $[2 : 2 * n]$) entering the statement, forming $[\min(2, 2) : \max(n + 1, 2 * n)] = [2 : 2 * n]$. Finally, the range at the top of the loop at S_2 was computed with the help of the widening and narrowing operators. Without a widening operator, the values of the range after S_2 would take on the successive values: $[1 : 1]$, $[1 : 2]$, $[1 : 4]$, ..., preventing the algorithm from terminating. The widening operator prevents this by assigning x to the conservative value $[1 : 2] \bigtriangledown [1 : 4] = [1 : \infty]$. However, one can see that the upper bound of this range is overly conservative, since the upper bound from $S1$ (1) and the upper bound from $S9$ ($2 * n$) are both less than ∞. This overly conservative value is corrected in the narrowing phase, where the old value of the range after $S2$ is narrowed with the range formed by the union of the ranges from $S1$ and $S9$, getting $[1 : \infty] \bigtriangleup ([1 : 1] \cup [2 : 2 * n]) = [1 : 2 * n]$. (Remember that the narrowing operator replaces all infinite bounds of the old range with the bounds of the new range, but otherwise leaves the range alone.)

3.4 Time Complexity

The time taken to perform a union or intersection of ranges for a particular statement would be the time taken to perform a symbolic expression comparison ($O(v^2)$ replacements) times the number of variables in the range dictionary $O(v)$, or $O(rv^3)$, where r is the average cost to perform a single variable replacement and v is the number of variables. Similarly, the cost for handling variable modifications is also $O(rv^3)$.

S_1 :	$x = 1$	$x = [-\infty : \infty]$
S_2 :	`CONTINUE`	$x = [1 : 1]$
S_3 :	`IF` $(x \leq n)$ `THEN`	$x = [1 : 2 * n]$
S_4 :	`IF` $(A(x) \geq 1.0)$ `THEN`	$x = [1 : n]$
S_5 :	$x = x + 1$	$x = [1 : n]$
S_6 :	`ELSE`	$x = [2 : n + 1]$
S_7 :	$x = 2 * x$	$x = [1 : n]$
S_8 :	`ENDIF`	$x = [2 : 2 * n]$
S_9 :	`GOTO` S_2	$x = [2 : 2 * n]$
S_{10} :	`ENDIF`	$x = \top$
		$x = [1 : \infty]$

Figure 5: Computation of ranges for an small code segment

Widening and narrowing operators only cost $O(v)$ time. Thus the time spent to compute the range for a particular statement or control flow edge is $O(rv^3)$.

Because of the widening operators, each range in the program unit can only be modified a constant number of times. Thus the time taken by the range propagation algorithm is $O((s + e)rv^3)$, where s is the number of statements in the program and e is the number of control-flow edges in the program. Since $e \geq s$, we can simplify this down to $O(erv^3)$.

4 Related work

The idea for representing program constraints as ranges was first proposed by Harrison [13] for array bounds checking and program verification. In his paper, Harrison describes how one can compute the range of integer values that variables can take in a program unit, using data-flow analysis. Although he does propose simple techniques to handle symbolic ranges, our symbolic analysis techniques are superior. (He restricts the bounds of his symbolic ranges to the form $<$ `variable` $> + <$ `constant` $>$.)

Bourdoncle [5] greatly improves the accuracy of the integer range propagation algorithm by Harrison, through the use of abstract interpretation [8]. Our use of the narrowing operator was influenced by his algorithm. Bourdoncle's algorithm is unable to generate symbolic ranges. The accuracy of the ranges generated by his (and Harrison's) technique can be improved with our monotonicity replacement method in Section 2.

Cousot and Halbwachs [9] presents a different method to compute and propagate constraints through a program. In their technique, sets of constraints between variables are represented as a convex polyhedron in the n-space of variable values. Because of this representation, all constraints are restricted to be in the form of affine inequality relationships, (e.g., $5 * x + 2 * y \leq 2$). Abstract interpretation is used to compute the convex polyhedron of variable constraints for each statement and each control flow edge of the program. Two affine symbolic expressions can be compared, in respect to constraints given as a convex polyhedron, by determining what side the convex polyhedron falls on the hyperplane formed by the difference of the two expressions.

Although our range propagation algorithm was heavily influenced by their abstract interpretation algorithm, our representation of variable constraints is quite different. They are more accurate in the computation and propagation of affine variable constraints. However, they cannot handle non-affine variable constraints, such as $a < b * c$. Another strength in our representation is that our technique can use a sparse data-flow form, such as definition-use chains [1] or Static Single Assignment [10], while theirs cannot. Performing range propagation on such a sparse form should greatly increase its efficiency.

Tu and Padua [15] also present a symbolic expression comparison and constraint propagation technique, based on an extension of Static Single Assignment form [10]. Their technique compares expressions

by repeatedly substituting variables with their constant symbolic values until the two expressions differ by only an integer constant. The values to substitute are determined by a demand-driven analysis of the program. Phi (ϕ) expressions, which are allowed to be substituted in these expressions, are used to represent ranges of values. Rewrite rules similar to ours are used to simplify expressions containing phi expressions.

Our variable replacement and simplification techniques are more powerful than theirs. However, their constraint propagation methods are more efficient since they are demand-driven and use Static Single Assignment form. Because of this, we plan on incorporating their demand-driven techniques into the range propagation algorithm. Additionally, their constraint propagation algorithm is capable of performing flow-sensitive analyses.

5 Conclusions

We have presented a powerful, but efficient, technique to compute the symbolic ranges for a program unit, and to use these ranges to compare arbitrary, possibly non-affine, symbolic expressions. We have also shown that this technique can benefit several passes in a parallelizing compiler.

We have implemented range propagation in Polaris, a parallelizing compiler being developed at the University of Illinois [2]. Range propagation is currently being used for symbolic data dependence testing by the Range Test [4], detection of zero-trip loops, computation of symbolic range of values that may possibly be referenced by an array access, and estimation of iteration counts of loops. Range propagation has enabled the Range Test to effectively parallelize two codes in the Perfect Benchmarks, TRFD and OCEAN, with the aid of other restructuring techniques. Because of this, we were able to achieve speedups close to the hand-parallelized versions, which were 43 for TRFD and 16 for OCEAN on Cedar, a machine with 32 vector processors, where commercial parallelizing compilers could only achieve a speedup of at most 2. Thus, we feel safe to conclude that ability to compare symbolic expressions, as provided by range propagation, can significantly improve the effectiveness of parallelizing compilers.

Our future work will be in improving the efficiency of the range propagation algorithm from Section 3. We plan on converting the algorithm so that it uses Static Single Assignment form. By using such a form, we should be able to improve its time complexity to $O(erv^2)$, as well as eliminate the practical costs of handling side effects and duplicating ranges. We will also look into demand-driven analysis [15], which should be greatly beneficial for range propagation since several of the compiler transformations which call it need only a few ranges.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. Reading, MA., 1986.

[2] 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.

[3] 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.

[4] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *To appear in Supercomputing '94*, 1994.

[5] 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.

[6] Thomas E. Cheatham Jr., Glenn H. Holloway, and Judy A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Transactions on Software Engineering*, SE-5(4):402–417, July 1979.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[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] 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.

[12] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic Program Analysis and Optimization for Parallelizing Compilers. *Presented at the 5th Annual Workshop on Languages and Compilers for Parallel Computing, New Haven, CT*, August 3-5, 1992.

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

[14] Paul Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Rice University, May 1994.

[15] 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.