Demand-driven, Symbolic Range Propagation*

William Blume
Hewlett Packard, Cupertino, California

Rudolf Eigenmann
Purdue University, West Lafayette, Indiana

Abstract. To effectively parallelize real programs, parallelizing compilers need powerful symbolic analysis techniques [13, 6]. In previous work we have introduced an algorithm called range propagation [4] that provides such capabilities. Range propagation has been implemented in Polaris, a parallelizing Fortran compiler being developed at the University of Illinois [7]. Because the algorithm is relatively expensive and several compilation passes make use of it, we have studied opportunities for increasing its efficiency. In this paper we present an algorithm that provides range analysis on-demand. We have implemented this algorithm in Polaris as well, and have measured its effectiveness.

1 Introduction

Range propagation was developed for the Polaris parallelizing compiler in response to our observed need of compilation passes to determine and reason about the values and expressions that program variables can take on.

The algorithm centers upon the computation and manipulation of *ranges*, which are lower and upper bound expressions for the value of a given variable at a given program statement.

Range propagation inludes two algorithms that depend on each other: one that can compare expressions and one that can derive upper and lower bounds of variables from the program text. The latter uses abstract interpretation [10] for analyzing the program.

As an example, suppose the data-dependence test wishes to compare the expression $x * y - 1$ with $-y$. Upper and lower bound analysis has determined that $x = [y : 10]$ and $y = [1 : \infty]$. It did this by analyzing all assignments to these variables and factoring in additional contraints imposed by control statements such as IF and DO statements. The goal of the comparison algorithm is to prove that the difference $d = expression_1 - expression_2 = [x * y + y - 1 : x * y + y - 1]$ is either always positive or always negative and, hence, either $expression_1 > expression_2$ or $expression_1 < expression_2$ holds. First, it replaces x with $[y : 10]$, getting $d = [[y : 10] * y + y - 1 : [y : 10] * y + y - 1]$. Simplifying leads to $d = [y^2 + y - 1 : 11 * y - 1]$. At this point it cannot yet determine whether the range of d is always positive or negative. The next step replaces y with $[1 : \infty]$, getting $d = [[1 : \infty]^2 + [1 : \infty] - 1 : 11 * [1 : \infty] - 1]$. After simplifying, $d = [1 : \infty]$. Now, the lower bound of d is an integer greater than zero, and hence $x * y - 1 > -y$. The techniques used to determine the substitution order of variables and to simplify the substituted ranges are described in [4].

One issue is that the compiler usually modifies the program between different applications of range propagation, requiring repeated recomputations of the program's ranges. Because of this, a significant fraction of a compiler's execution time can be spent performing range propagation.

To lower these costs, we have developed a demand-driven algorithm for performing range propagation. It can compute the range for a particular variable when that range is requested by the compilation pass, as opposed to a conventional data-flow algorithm that computes all ranges at once. Since many restructuring techniques only need to know a small subset of the ranges of all variables, a demand-driven algorithm should greatly reduce the costs of range propagation.

* This work was done at the University of Illinois at Urbana-Champaign under support by Army contract DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the U.S. Army or the government.

```
function get_range(s : statement, v : variable) : range
    if (R(s, v) has not been defined) then
        c ← get_control_range(s, v)
        d ← get_data_range(v)
        R(s, v) ← c ∩ d
    end if
    return R(s, v)
end function
```

Fig. 1. The demand-driven range propagation algorithm

2 Determining upper and lower bounds of variable values

The range propagation algorithm computes the range of each variable at each point of the program. A range is simply a symbolic lower bound and a symbolic upper bound on the values that a variable may take.

Since one cannot always statically compute the exact range that all variables may take in a program, range propagation computes a conservative approximation of the range of a variable. The lower bound of this approximation is always guaranteed to be smaller than or equal to the actual lower bound while the upper bound of this approximation is always guaranteed to be larger than or equal to the actual upper bound.

We break up the problem of computing the ranges of a variable at a particular statement into two sub-problems: the computation of the *control ranges* of the variable, and the computation of the *data ranges* of the variable. The final range for the variable is simply the intersection of its control and data ranges. The control ranges of a variable are those ranges computed from the constraints imposed by the control flow of the program, such as from `IF` or `DO` statements. The data ranges of a variable are those ranges computed from the assignments to that variable. We compute the control and data ranges separately because control ranges are much cheaper to compute.

The top-level function `get_range` for the demand-driven range propagation algorithm is shown in Figure 1. It stores its results in a global structure named R, so that future invocations of this function can reuse these results rather than recomputing. This storing and reusing values to avoid recomputation is called **memoization** [17, 1]. The algorithm simply checks whether the range for the given variable and statement already exists in R, computes and stores the range if it does not, then returns the range. The range is computed by intersecting the data and control ranges returned by `get_control_range` and `get_data_range`, two functions that will be described in Sections 3 and 4, respectively.

Notation used in this paper: A **control-flow graph** (CFG) of a program is a directed graph. Each vertex in the CFG corresponds to a statement in the program and an edge between two vertices indicates that the second statement may be immediately executed after the first statment. An edge is said to be a **back-edge** if the order of the source of the edge is larger than the order of the sink of the edge, under a depth first ordering of the CFG.[2] Vertex u **dominates** vertex v if and only if every path from *start* to v pass through vertex u. Vertex u **strictly dominates** vertex v if and only if u dominates v and u does not equal v. Vertex u is the **immediate dominator** of vertex v if and only if u strictly dominates v and there is no vertex w such that u strictly dominates w and w strictly dominates v. See Aho, Sethi, and Ullman [2] for more details on these definitions.

Similar dominance relationships can be defined for the control-flow edges in the program. For example, a control-flow edge dominates a statement if all paths from *start* to that statement pass through that

[2] Our definition of back-edges is wider than the definition of back-edges given in other papers, (i.e., we define more edges to be back-edges than they do). We have defined back-edges differently so that if one deleted all the back-edges from a graph, the graph is guaranteed to be acyclic, even if the original graph is irreducible.

control-flow edge. In this paper, we would say that a control-flow edge is an **immediate dominating control-flow edge** (or `icdom`(s)) of a statement (s) if that edge is the immediate dominator of the statement.

Our demand-driven range propagation algorithm assumes that programs are in **Static Single Assignment** (SSA) form. A program is in SSA form when every variable within it has at most one defining statement. Programs are translated into SSA form by inserting *ϕ-functions* and renaming variables. A **ϕ-function**, denoted as $v \leftarrow \phi(w_1, w_2, \ldots, w_n)$, is a special assignment to a variable v that is inserted at a join in the control flow where at least two definitions of v reach this join. The ϕ-function has an argument for each entering control-flow edge of this join. The ith argument (w_i) corresponds to the value that the variable assigned by the ϕ-function (v) would take if the control-flow of the program took the ith control-flow edge to reach the join node, (i.e., $v = w_i$). An efficient algorithm to translate programs into SSA form is described in [12]. In this paper, we will assume that the function **def**(v) would return the single statement in the program that defines v.

3 Computing control ranges

3.1 Needed functionality

Our demand-driven control range propagation algorithm assumes that the immediate dominating control-flow edge is known for each statement and that the program is in SSA form. The function `icdom`(s) will represent the immediate dominating control-flow edge of the statement s. A linear-time algorithm for computing dominators has been developed by Harel [14]. Alternatively, one can approximate the dominating control-flow edges from the statement dominators, which must be computed when translating into SSA form.

To compute the control ranges of a program, we will assume that there exists a function `get_local_control_ranges`(e, v), which computes and returns the range of a given variable v at a given control-flow edge e computed from the control constraints imposed by the source statement of that edge. For example, `get_local_control_ranges`(e, v) would return $[a : \infty]$, if e is the exiting control flow edge for the `then` case of the statement `IF (V .GE. A) THEN`. If there are no control flow constraints imposed on that variable for that control-flow edge, then the function returns the unconstrained range $[-\infty : \infty]$.

3.2 Algorithm

The algorithm for demand-driven control range propagation is shown in Figure 2. This algorithm is composed of two mutually recursive functions: a function that computes the control range that holds for the entry of a given statement, and a function that computes the control range that holds after taking a given control-flow edge. Intuitively, these functions compute the control range of a given variable and statement (or control-flow edge) by intersecting the ranges that hold for the variable for all the dominating control-flow edges of the given statement (or control-flow edge). The control range of a statement is simply the control range that holds after passing through that statement's immediately dominating control-flow edge, (i.e., `icdom`(s)). As discussed in [8], this is more efficient but slightly more conservative than intersecting the control ranges of all incoming control flow edges. If the statement does not have an immediately dominating control-flow edge, then its result is the unconstrained range $[-\infty : \infty]$. The control range for a control-flow edge is the control range that is imposed by the edge, (i.e., the result of `get_local_control_range`), intersected with the control range for the source statement of that edge. The result is stored in the data structure C so as to avoid needless recomputation, (i.e., it memoizes).

Although recursive, the functions in Figure 2 are guaranteed to terminate. By definition of immediately dominating control-flow edges, the source of `icdom`(s) must strictly dominate statement s. Since the graphical representation of the dominator relationship is a tree, the algorithm will eventually reach a statement that does not have an immediately dominating control-flow edge.

```
function get_control_range(s : statement, v : variable) : range
    if (icdom(s) is not defined) then
      return [−∞ : ∞]
    else
      return get_control_range1(icdom(s), v)
    end if
end function

function get_control_range1(e : control-flow edge, v : variable) : range
    if (C(e, v) has not been defined) then
      c ← get_local_control_range(e, v)
      p ← get_control_range(source(e), v)
      C(e, v) ← c ∩ p
    end if
    return C(e, v)
end function
```

Fig. 2. The demand-driven control range propagation algorithm

3.3 Time complexity

The worst case time taken by the algorithm in Figure 2 is bounded by $O(c|S|)$, where c is the time taken to perform an intersection, (which equals the time taken to perform a constant number of symbolic expression comparisons), and $|S|$ is the number of statements in the program. However, from the extensive use of memoization, (i.e., storing computed values into C and reusing them), the worst case time taken to compute the range for every variable at every statement is $O(c|S||V|)$, where $|V|$ is the number of scalar variables in the program. Since a non-demand-driven algorithm would also take at least $O(c|S||V|)$ time, (since such an algorithm would have to visit each (statement, variable) pair in the program), the demand-driven algorithm is at most as expensive as a non-demand-driven algorithm, ignoring a constant factor.

3.4 Optimizations

By design of the algorithm, the time taken to compute a single control range is dependent upon the number of control-flow edges that dominate the given statement s. The number of dominating control-flow edges of a statement can be very large (O($|S|$)). However, only a few of these edges add new constraints, (e.g., edges exiting `IF` or `DO` statements or from `ASSERT` directives). Because of this, our algorithm creates and uses a sparse form of the `icdom` function, where this sparse form returns the most immediate dominating control-flow edge that adds at least one range to one variable.

We have implemented an algorithm for computing the sparse immediate dominating control-flow edge of a statement. It simply traces back through all the statement's dominating control flow edges, using the `icdom` relationship, until it finds an edge that adds a control range to at least one variable. A global structure is used to memoize the result of this computation so that the algorithm would not recompute it in future calls.

4 Computing data ranges

The algorithm for computing ranges originating from the program's data flow is much more complex than the algorithm for computing ranges originating from constraints imposed by the control flow. This additional complexity arises from the need to iterate to a fixed point, (i.e., perform data-flow analysis), to compute the data ranges.

```
type node_ptr = pointer to d_range_node
type node = structure
    var : variable
    value : range              = ⊤ (assignment stmt.); [−∞ : ∞] (otherwise)
    old_value : range          = ⊤
    committed : boolean        = false (assignment stmt.); true (otherwise)
    prev : set of node_ptr     = ∅
    next : set of node_ptr     = ∅
end structure
```

Fig. 3. Fields and initial values of a node structure

4.1 Data-flow graph

To allow the algorithm to cleanly and efficiently perform data flow analysis on a program in a demand-driven manner, we create and iterate over a data-flow graph that contains only the information needed to compute the desired range. Each node in this data-flow graph represents a variable and its data range. An edge exists from the node for variable x to the node for variable y if and only if the computation of the range of x depends upon the range of y. One node in this graph, denoted as *root*, is the node for the variable of the requested data range. All other nodes in the graph that we need to iterate over are reachable from *root*.

The fields of a single node of this graph are shown in Figure 3. *value, old_value, prev*, and *next* are working fields for the node. Once the final value has been determined, *committed* is set to true, which "freezes" the node's value and, in this way, memoizes the node's range.

With the exception of the initialization of the *value* field, all the initializations are straightforward. The initial range assigned to the *value* field is determined from the single definition point of the variable, which is given by the program's SSA representation. If the variable's definition is not an assignment statement, (e.g., the variable is a formal parameter, an argument to a procedure call, or an I/O statement), then its range is set to the unconstrained range $[-\infty : \infty]$ and committed. Otherwise, its value is set to the undefined range denoted $\top$.

4.2 Algorithm

The top-level of the algorithm for computing data ranges is shown in Figure 4. This algorithm simply calls the function `get_node` to build and iterate over a data-flow graph whose root is the node for the given variable, then returns the computed range stored in this root node.

The function `get_node` has two responsibilities. One of these responsibilities is to build a data-flow graph. More specifically, it creates a node for the given variable as well as data-flow subgraphs for the variables that the given variable's range may depend on. The function `create_node`, creates the node for the given variable, as shown in Figure 3. The function `add_children_to_node`, which is shown in Figure 5, creates nodes for the variables that the current node depends upon and adds edges between this current node and the newly created nodes by updating their *next* and *prev* fields. These other nodes are created by recursive calls to `get_node`. The global array D is used to memoize created nodes for future reuse.

The other responsibility of function `get_node` is to compute and commit the data ranges for the data-flow subgraph under construction. It does this eagerly, whenever the subgraph contains all nodes that it reaches (and hence it contains all variables needed for the analysis of this subgraph). We determine this state of the subgraph using Tarjan's *strong-connect* algorithm.

This eagerness to compute and commit data ranges as early as possible is one of the inherent properties of the algorithm. The reason for this eager computation of ranges is that it maximizes the number of committed ranges that a particular range may depend on, which improves the accuracy of the ranges

```
function get_data_range(v : variable) : range
    root ← get_node(v)
    return root.value
end function

function get_node(v : variable) : node_ptr
    if (D(v) has not been defined) then
      root ← create_node(v)
      D(v) ← root
      call add_children_to_node(root)
      // Node root is now fully-initialized
      if (all uncommitted nodes reachable from root have been fully initialized) then
        compute_data_ranges(root)
        commit_data_ranges(root)
      end if
    end if
    return D(v)
end function
```

Fig. 4. The demand-driven data range propagation algorithm

```
procedure add_children_to_node(x : node_ptr)
    if (def(v) is an assignment statement) then
      for each variable w in rhs of def(v) do
        y ← get_node(w)
        x.next ← x.next ∪ {y}
        y.prev ← y.prev ∪ {x}
      end for
    end if
end procedure
```

Fig. 5. Algorithm to create children for a data-flow graph node.

computed by function `compute_data_ranges_phase`. Computing and committing data ranges early also minimizes the number of *poisoned* ranges generated. Poisoned ranges will be discussed in Section 5.

The point in algorithm `get_node` where the subgraph under construction is ready for computing the data ranges can be thought of as where "all uncommitted nodes reachable from *root* have been fully initialized", whereby the nodes are marked "initialized" as indicated by the comment line. The function `get_node` computes and commits the ranges of this subgraph by calling functions `compute_data_ranges` and `commit_data_ranges`. The implementation of `compute_data_ranges` will be described in the next subsection. The implementation of `commit_data_ranges` is shown in Figure 6. This algorithm sets the *committed* field to true for all (uncommitted) nodes in the subgraph rooted at *root*. By committing these nodes, it memoizes the ranges contained in these nodes, so that future invocations of `get_data_range` will return the ranges in these nodes rather than recomputing them.

```
procedure commit_data_ranges(root : node_ptr)
    for each uncommitted node x reachable from root do
      x.committed ← true
    end for
end procedure
```

Fig. 6. Algorithm to commit data ranges in the data-flow subgraph rooted at *root*

```
procedure compute_data_ranges(root : node_ptr)
    compute_data_ranges_phase(root, WIDENING_PHASE)
    compute_data_ranges_phase(root, NARROWING_PHASE)
end procedure
```

Fig. 7. Algorithm to compute data ranges from the given graph.

```
procedure compute_data_ranges_phase(root : node_ptr, phase : phase_type)
    work_list ← all uncommitted nodes that are reachable from root
    while (work_list is not empty) do
      x ← dequeue(work_list)
      v ← x.var
      if (rhs of def(v) is a φ-function) then
        r ← ⊤
        for each edge e entering def(v) do
          y ← node in x.next associated with edge e
          s ← y.value ∩ get_control_range1(e, y.var)
          r ← r ∪ s
        end for
        if (def(v) has an entering back edge in CFG and x.old_value ≠ ⊤) then
          if (phase = WIDENING_PHASE) then
            r ← x.value ▽ r
          else   // (phase = NARROWING_PHASE)
            r ← x.value △ r
          end if
        end if
      else
        b ← rhs of def(v)
        r ← [b : b]
      end if
      for each node y ∈ x.next such that y.committed = false do
        s ← y.value ∩ get_control_range(def(v), y.var)
        r ← r with all occurrences of variable y.var replaced with s
      end for
      x.old_value ← x.value
      x.value ← r
      if (x.old_value ≠ x.value) then
        work_list ← work_list ∪ x.prev
      end if
    end while
end procedure
```

Fig. 8. Algorithm to compute data ranges for nodes whose definitions are ϕ-functions.

4.3 Computing data ranges from a data-flow graph

The main function for computing data ranges is shown in Figure 7. It calls the function `compute_data_ranges_phase` twice in order to compute the ranges of ϕ-functions in two phases: the *widening phase* and the *narrowing phase*. These two phases are discussed in [8]. Briefly, the widening phase applies some conservative operations in order to guarantee termination of the algorithm in loop situations. The narrowing phase follows for regaining some of the accuracy lost by the widening operations. In Section 4.4 we will illustrate this mechanism in an example.

The implementation of function `compute_data_ranges_phase` is shown in Figure 8. It performs an iterative data-flow analysis upon all uncommitted nodes in the data-flow graph. More specifically, it initially inserts all uncommitted nodes on the priority queue *work_list*. It then repeatedly removes a

$$[a:b] \cup [c:d] \Rightarrow [\min(a,c) : \max(b,d)] \quad (1)$$

$$[a:b] \cap [c:d] \Rightarrow [\max(a,c) : \min(b,d)] \quad (2)$$

$$[a:b] \bigtriangledown [c:d] \Rightarrow [\textbf{if } a = c \textbf{ then } a \textbf{ else } -\infty : \textbf{if } b = d \textbf{ then } b \textbf{ else } \infty] \quad (3)$$

$$[a:b] \bigtriangleup [c:d] \Rightarrow [\textbf{if } a \neq -\infty \textbf{ then } a \textbf{ else } c : \textbf{if } b \neq \infty \textbf{ then } b \textbf{ else } d] \quad (4)$$

Table 1. Basic operations used by the range propagation algorithm.

node from *work_list*, updates that node's data range, then adds all nodes to *work_list* that depend upon its data range, (i.e., the nodes in $x.prev$), if its data range has changed. The algorithm quits only when the *work_list* becomes empty. To minimize the number of updates performed upon the graph's nodes, the nodes in *work_list* should be ordered by a topological order of the data-flow graph, ignoring any back-edges, (that is, in rPOSTORDER, as described in [16]).

The data range (r) of a node x, whose variable's (v) definition contains a ϕ-function, is computed by unioning ($\cup$) the ranges of the arguments of its ϕ-function. The semantics of the union operator is given in Table 1. This union results in a range whose lower bound is the minimum of the lower bounds of the argument's ranges and whose upper bound is a maximum of the upper bounds of the arguments' ranges. An argument's range (s) is the intersection ($\cap$) of the argument's current data range and the control range holding for the argument's control-flow edge.[3]

A node whose variable's definition is not a ϕ-function, is initially assigned a single-element data range whose bounds are equal to the right-hand-side of the variable's definition.

For both types of nodes the algorithm then replaces all variables of uncommitted nodes in their range with the ranges that these variables assume at the given statement. As a result of this replacement step, only committed range values appear in the ranges being computed. This way, ranges being computed cannot be self-referential, which would introduce difficulties in the algorithm. A self-referential range is a range assigned to a variable v, where after repeated replacements of variables in the range with those variables' ranges, its symbolic value contains variable v. An example of a directly self-referential range is $x = [1 : x + 1]$. An example of indirectly self-referential ranges is the pair of ranges $x = [1 : y]$ and $y = [1 : x + 1]$. Such self-referential ranges are not very useful because self-referential bounds typically add no constraint information to variables. For example, in the range $x = [1 : x + 1]$ the upper bound adds no information, since all it says is $x \leq x + 1$, which is always true. Hence, this range is equivalent to the simpler range $x = [1 : \infty]$. Another problem with directly or indirectly self-referential ranges is that the expression comparison algorithm, which is briefly described in the introduction, has difficulties determining a good order to substitute variables with ranges, resulting in more variable substitutions and less accurate results.

Furthermore, the replacement mechanism simplifies the termination test of the data-flow algorithm. The algorithm terminates if all nodes are processed without changing their range value. Such a change is easy to detect if the range includes only variables with fixed (i.e., committed) values.

The replacement scheme comes at the cost of some inaccuracy. We can regain some of this accuracy by "undoing" the replacement at the end of the range computation. This is further discussed in [8].

4.4 Example

[3] By construction of ϕ-functions, each argument of a ϕ-function corresponds to one of the entering control-flow edges of the ϕ-function's basic block.

```
c       Assert: n ≥ 8
        i1 = n
100     CONTINUE
          i2 = φ(i1, i4)
          IF (x(i2) > 0) THEN
            i3 = i2/2
          ENDIF
          i4 = φ(i3, i2)
        IF (i4 ≥ 2) THEN GOTO 100
```

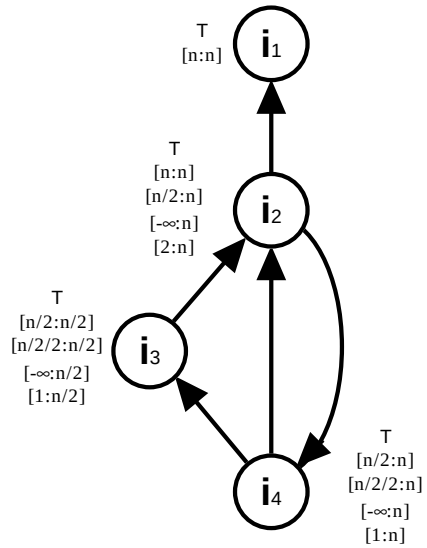

Fig. 9. Example program fragment in SSA form and its data-flow graph

As an example of how function `get_data_range` works, suppose that we wish to compute the data range for variable i_4 in the Fortran code fragment shown in Figure 9. For this example, the function call `get_data_range`(i_4) creates and iterates over the data-flow graph shown in figure 9. Function `get_range_data` invokes `compute_data_ranges` (and `commit_data_ranges`) twice in this example; once to compute the range of i_1, and once to compute the ranges for i_2, i_3, and i_4. This is because $\{i_1\}$ and $\{i_2, i_3, i_4\}$ are strongly-connected components of the data flow graph, and function `get_node` computes and commits all nodes in a created subgraph as soon as it is fully formed, (i.e., is a strongly-connected component).

The ranges listed alongside each node in the graph represent the values that the node's *value* field take while function `compute_data_ranges_phase` iterates to a fixed point. To give the reader an understanding of how these ranges are computed, we will describe how the data range was computed for the node for variable i_2. Initially, on entry to `compute_data_ranges`, the range of i_2 is the undefined range $\top$. In the widening phase, the first visit to i_2[4] sets its value to the union of the ranges of i_1 and i_4, both intersected with the control range of their incoming control-flow edge. The control range of i_1 is the unconstrained range $[-\infty : \infty]$ since there is no control-flow constraints on i_1 coming from the statement before the `CONTINUE` statement. The control range for i_4 comes from the `IF` statement, and is $[2 : \infty]$ because of the condition $(i_4 \geq 2)$ in the `IF` statement. That leads to the formula for the range of i_2:

$$i_2 = (i_1 \cap [-\infty : \infty]) \cup (i_4 \cap [2 : \infty]) = i_1 \cup (i_4 \cap [2 : \infty])$$

So the value of the range of i_2 on the first visit to its node is $[n : n] \cup (\top \cap [2 : \infty]) = [n : n]$. Since the range of i_2 has changed, predecessors of node i_2 (i.e., i_1 and i_4), are placed in the work list and the algorithm continues. When the algorithm returns to i_2, the range of i_3 would have been updated to $[i_2/2 : i_2/2] = [[n : n]/2 : [n : n]/2] = [n/2 : n/2]$, and i_4 would have been updated to $[n/2 : n/2] \cup [n : n] = [n/2 : n]$. At this point, the new range for i_2 is $[n : n] \cup ([n/2 : n] \cap [2 : \infty]) = [n/2 : \infty]$. On the third visit to i_2, the algorithm determines its range to be $[n : n] \cup ([n/2/2 : n] \cap [2 : \infty]) = [n/2/2 : n]$. Now, since i_2 is a loop header node, the algorithm also applies the widening operator ($\bigtriangledown$) to this result,

[4] In this example, we will use a variable's name to represent both the variable and the data-flow node for that variable.

getting $[n/2 : n] \bigtriangledown [n/2/2 : n] = [-\infty : n]$. On the fourth visit to i_2, the algorithm would find that the range of i_2 is $[-\infty : n] \bigtriangledown ([n : n] \cup ([-\infty : n] \cap [2 : \infty])) = [-\infty : n] \bigtriangledown [2 : n] = [-\infty : n]$. Since the range of i_2 has not changed, the algorithm will not put i_2's successors on the work list, causing the widening phase to stop. Note, that the widening operator, which is applied after two visits to a node, has caused the value to become stable. However, it has introduced an overly conservative lower bound. Function `compute_data_range` would then enter the narrowing phase. In the narrowing phase, the narrowing operator ($\bigtriangleup$) would be applied to the range of i_2 instead of the widening operator. Thus the narrowing phase, on its first visit to i_2, would compute the range of i_2 to be $[-\infty : n] \bigtriangleup ([n : n] \cup ([-\infty : n] \cap [2 : \infty])) = [-\infty : n] \bigtriangleup [2 : n] = [2 : n]$. On its second visit of i_2, the computed range of i_2 would also be $[2 : n]$, causing the narrowing phase to stop.

When function `get_data_range`(i_4) completes, it will have computed and committed the data ranges $i_1 = [n : n]$, $i_2 = [2 : n]$, $i_3 = [i_2/2 : i_2/2]$ [5], and $i_4 = [1 : n]$. So, the function `get_data_range` will return $[1 : n]$.

5 Handling union and intersection operations

In our presentation of the algorithms to compute control and data ranges, we have not addressed the complexities associated with making the union or intersection of two ranges. Both of these operations form the bounds of the resulting range by taking the minimum or maximum of the bounds of their two arguments. Simplifying these minimum and maximum expressions typically requires symbolic expression comparisons, which in turn can perform several `get_range` operations.

5.1 Control ranges

Handling this recursion in the control range computation algorithm in Figure 2 is not difficult. One only needs to initialize $C(e, v)$, (i.e., the memoized control range of variable v at control-flow edge e), to the unconstrained range $[-\infty : \infty]$ and allow the intersection operation to use only control ranges to compare bounds when simplifying the range. Initially setting the control range of the current variable and statement to $[-\infty : \infty]$ ensures termination, since this assignment guarantees that any recursive invocation of `get_control_range` will not attempt to compute the control range for this variable, statement pair. Because the intersection operator can generate many recursive calls to `get_control_range`, the worst case time complexity of the algorithm in Figure 2 would be $O(c|S||V|)$.

5.2 Data ranges

A simple way to handle the recursion caused by union and intersection operations performed in the algorithm `compute_data_ranges_phase` in Figure 8 is to allow these operations to use only control ranges to simplify their results. Since the computation of control ranges will never invoke `get_data_range`, the algorithm is guaranteed to terminate. The worst case time complexity of the algorithm remains unchanged.

However, by using only control ranges to perform unions and intersections when computing data ranges, the resulting data ranges may lead to overly conservative results. This is because the widening operator may replace partially unsimplified range bounds with $\pm\infty$. Thus, it is desirable to be able to use data ranges in these simplifications as well. Unfortunately, avoiding infinite recursions is complex. We handle this problem by assigning a variable's node a timestamp when we create it in `create_node`. This timestamp, which is associated with a particular invocation of `get_data_range`, is used to identify when a node belongs to an older invocation of `get_data_range`. The functions `compute_data_ranges`, `compute_data_ranges_phase` and `commit_data_ranges` are allowed to only visit nodes created by the current invocation of `get_data_range`. Additionally, if the function `compute_data_ranges_phase` attempts

[5] The range $i_3 = [i_2/2 : i_2/2]$ was created in the loop at the end of function `compute_data_ranges`.

to access the data range of a uncommitted node created by a previous invocation of `get_data_range`, it would use the range $[-\infty : \infty]$ for that node's data range. Also, any node that uses the range of a node created by an older invocation of `get_data_range` is marked as *poisoned*. Poisoned nodes are nodes that cannot be memoized, (i.e., their values will not be stored in the global data structure), nor committed, since their data ranges may be overly conservative. Instead, they are deleted from the data-flow graph on the exit of `get_data_range`. Any node that uses the data range of a poisoned node is itself poisoned.

6 Performance

The time complexities of the algorithms are discussed in [8]. In this paper we show that our demand-driven range propagation algorithm is efficient for real programs, even when called many times. All optimizations described in the previous sections have been implemented in these algorithms. These times taken to compute all control and data ranges in a program are displayed in Table 2. The *code* column displays the name of each Fortran code examined. These codes were taken from the Perfect Benchmarks, which is a suite of Fortran 77 programs representing applications in a number of areas in engineering and scientific computing [3]. The *Number of lines* column displays the number of lines in each code after being converted into SSA form. The *Computing control ranges* and *Computing data ranges* columns give the total times taken to compute every control and data range respectively in each of the codes. The *Merging data and control ranges* column shows the time taken to intersect all the control and data ranges in the program, as done by the top-level algorithm described in Section 2. The total time to compute all the ranges in the program is just the sum of these columns. All timings are user times measured on a Sparc 10, using g++ 2.6.3 with the flag -O.

Code	*Number of lines*	*Computing data ranges (s)*	*Computing control ranges (s)*	*Merging data and control ranges (s)*
ARC2D	3573	2.4	1.3	1.4
BDNA	5960	3.8	5.4	4.3
FLO52	3348	3.9	4.9	2.7
MDG	1487	0.7	1.9	1.1
OCEAN	3142	8.0	4.6	25.3
TRFD	965	1.7	4.9	17.6

Table 2. Time taken in seconds to compute all data and control ranges, and to merge all data and control ranges on a Sparc 10 workstation.

As Table 2 has shown, computing all data and control ranges is very efficient. However, merging them can be expensive. This cost of merging them together arises from the intersection operation in the function `get_range`. In our experience, much of this time in computing and merging control and data ranges is spent computing the intersection and union of ranges. This cost arises from the fact that the symbolic expression comparison algorithm is used to simplify the intersection and union of ranges, and this algorithm can be expensive. However, unions and intersections are very cheap if one of their arguments is $[-\infty : \infty]$

Table 2 has shown that the demand-driven range propagation algorithm is efficient for computing all ranges in a program. However, it says little about the costs of computing a single range. The demand-driven range propagator may need to compute the values of other control and data ranges to determine the value of a certain range. In the worst case, every data and control range may need to be computed.

To determine the efficiency of the demand-driven range propagation algorithm for computing a single range, we have measured the number of control and data ranges computed by the algorithm to determine this range. Since the running time of the algorithm is proportional to the number of ranges it needs to

compute, the fraction of computed control and data ranges computed out of the set of all control and data ranges should roughly indicate what fraction of the execution times shown in Table 2 that a typical range computation takes.

We have collected both the average and the maximum number of data and control ranges computed for a single invocation of `get_control_range` from Figure 2 and `get_data_range` from Figure 4. Tables 3 and 4 displays these results. These numbers were computed by requesting each control or data range in a program, then counting the number of control and data ranges created by each request. All memoized ranges, (i.e., the ranges stored in R, C, and D, from functions `get_range`, `get_control_range`, `get_data_range`), were cleared before each request for a control or data range. Ideally, the average and maximum number of control ranges created should be one when a single control range is requested. Also, the ideal average and maximum number of data ranges created should be one and the ideal average and maximum number of control and poisoned ranges created should be zero when a data range is requested.

Code	*No. control ranges*	*No. computed*	
		Avg.	*Max.*
ARC2D	10227	1.9	5
BDNA	27285	2.7	8
FLO52	35094	2.6	11
MDG	4152	2.4	7
OCEAN	94819	2.4	37
TRFD	9994	3.1	21

Table 3. Average and maximum number of control ranges computed when computing a single control range.

Examining Table 3, one can see that typically only two or three control ranges need to be computed to determine the value of a single control range. This low number is mostly due to the use of the sparse `icdom` optimization, as described in Section 3. This sparse `icdom` relationship usually causes the algorithm in Figure 2 to compute the control ranges for only the control-flow statements, (e.g., `IF` and `DO` statements) that enclose the current statement. Since our test programs do not have deeply nested control flow, it is not surprising that the average number of computed control ranges is small.

One can roughly determine the cost of computing a control range by dividing the total time taken by `get_control_range`, as shown in Table 2, by the total number of control ranges in the program, as shown in column *No. control ranges* in Table 3, then multiplying this result by the number of ranges computed for that control range. Doing this, we find that the average control range computation takes about a few hundred microseconds, and the longest control range computation takes about a few milliseconds. Thus, we feel confident to claim that computing a single control range using a demand-driven algorithm is very efficient.

Table 4 displays the average and maximum number of control, data, and poisoned ranges computed per data range.[6] The overall average and maximum cost of computing a data range can be approximated by adding these averages and maximums respectively. This table shows that computing a data range typically causes the algorithm to compute several data and control ranges, possibly many data and control ranges in the worst case. Additionally, the large discrepancies between the averages and the maximums indicate that the costs of computing a data range may vary greatly. Despite the potentially large number of data ranges computed, the average and maximum data and control ranges computed is still a small

[6] Poisoned ranges are overly conservative data ranges computed by a recursive call to `get_data_range`. Unlike control and data ranges, poisoned ranges are not memoized, so they may be repeatedly recomputed when computing a data range. We count such ranges multiple times, once per computation, in the table. See Section 4 for more details.

Code	No. data ranges	Control Avg.	Control Max.	Data Avg.	Data Max.	Poisoned Avg.	Poisoned Max.
ARC2D	1002	2.9	31	2.4	20	0.1	24
BDNA	1431	5.5	47	3.9	29	1.3	40
FLO52	1142	22.9	243	9.8	97	2.8	54
MDG	436	4.0	20	2.9	11	1.1	49
OCEAN	1529	10.5	201	6.6	64	0.4	30
TRFD	521	8.4	68	4.4	21	0.2	6

Table 4. Average and maximum number of control ranges, data ranges, and poisoned data ranges computed when computing a single data range.

fraction of the total number of control and data ranges in the program. Thus, a demand-driven data range computation algorithm is still more efficient than its non-demand-driven counterpart.

We can determine the rough cost of computing a data range by dividing the total time taken by `get_data_range` by the total number of data ranges in the program, then multiplying by the number of data and poisoned ranges ranges computed for that data range. Doing this, one can determine that the average time taken to compute a data range is a few tens of milliseconds, and the worst case time is a few hundreds of milliseconds for real codes.

If one finds the cost of computing a single data range to be too expensive, one can sometimes compute and use only control ranges. We have found that using only control ranges, coupled with symbolic constant propagation and induction variable substitution, provides sufficient information for applications of range propagation by parallelizing compilers on some Fortran programs. This is because such transformations transform most expressions in a program into expressions made up of only symbolic constants and enclosing loop indices, and one only needs to know the constraints imposed upon these loop indices and symbolic constants to compare or compute the ranges of such expressions.

7 Related work

The idea for representing program constraints as ranges was first proposed by Harrison [15] for array bounds checking and program verification. Bourdoncle [9] greatly improves the accuracy of the integer range propagation algorithm by Harrison, through the use of abstract interpretation [10]. However, Bourdoncle's algorithm does not generate symbolic ranges. Neither Harrison's nor Bourdoncle's algorithms are demand-driven, nor do they use a sparse data-flow representation of a program, such as SSA form or definition-use chains.

Cousot and Halbwachs [11] present 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 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.

They are more accurate in the computation and propagation of affine variable constraints than our algorithm. However, they cannot handle non-affine variable constraints, such as $a < b * c$. Additionally, by using a convex hull representation to compute variable constraints, their algorithm cannot benefit from a sparse data-flow representation of a program. Because of this, their algorithm can be much less efficient than ours. Also, their convex hull representation prevents one from creating a demand-driven version of their algorithm that is not overly complex.

Tu and Padua [18] also present a demand-driven, symbolic expression comparison and constraint propagation technique, based on an extension of SSA called gated SSA form. 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. Variants of ϕ-functions, which can be substituted in other expressions, are used to represent ranges of values. (These variants of ϕ-functions are simply ϕ-functions extended to contain conditional predicates that indicate which of their arguments should be their result.) Rewrite rules are used to simplify expressions containing such ϕ-functions.

The differences between our algorithm and theirs are mainly due to the two different applications. Their algorithm was designed to compare the bounds of array sections for array privatization [19]. Because conditional array definitions and uses occur in a significant fraction of important loop nests, flow-sensitive analysis is essential to successfully perform such comparisons. This capability was included in their algorithm. On the other hand, the bounds being compared are usually very similar to each other, requiring the substitution of only a few variables with their constant values to make the two bounds equal each other, except for a constant offset. On the other hand, range propagation was orignally designed for dependence testing. Our symbolic data dependence test, called the Range Test [5], often needs to compare expressions that are more complicated and dissimilar to each other than the expressions compared for array privatization. Because of this, more constraint information and a more powerful expression comparator is needed to compare such expressions. Also, dependence testing requests many more constraints, making memoization much more important. Therefore a memoization mechanism was included in our algorithm while not in Tu's. In our experience, conditionally defined constants and constraints do not significantly improve the effectiveness of dependence testing. Because of these differences, Polaris includes implementations of both range propagation and Tu's and Padua's gated-SSA demand-driven analysis. In future work, we will attempt to merge the two algorithms.

8 Conclusions

We have developed a demand-driven range propagation algorithm and have shown it to be efficient for computing a single range as well as many ranges. Because of its efficiency, it is feasible to use range propagation at many points in a compiler without a serious degradation of the compiler's performance, even though the program may be modified between these points. Since these ranges can be used to perform symbolic expression comparisons and range computation of expressions, and these operations enable powerful symbolic analyses, demand-driven range propagation can significantly increase the effectiveness of parallelizing and optimizing compilers.

References

1. H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.
3. M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evalution of Supercomputers. *Int'l. Journal of Supercomputer Applications, Fall 1989*, 3(3):5–40, Fall 1989.
4. William Blume and Rudolf Eigenmann. Symbolic Range Propagation. *Proceedings of the 9th International Parallel Processing Symposium, April 1995.*
5. William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proceedings of Supercomputing '94, Washington D.C., November 1994*, pages 528–537.
6. 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.

7. 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. *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing, Ithaca, New York; also: Lecture Notes in Computer Science 892, Springer-Verlag*, pages 141–154, August 1994.
8. William Joseph Blume. *Symbolic Analysis Techniques for Effective Automatice Parallelization.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., June 1995.
9. Francqis 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. D. Harel. A linear time algorithm for finding dominators in a flow graph and related problems. *Proceedings of the 17th ACM Symposium of Theory of Computing*, pages 185–194, May 1985.
15. William H. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.
16. Matthew S. Hecht and Jeffrey D. Ullman. A Simple Algorithm for Global Data Flow Analysis Problems. *SIAM Journal on Computing*, 4(4):519–532, December 1975.
17. 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.
18. Peng Tu and David Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain, July 1995.*
19. Peng Tu and David Padua. Automatic Array Privatization. In Utpal BanerjeeDavid GelernterAlex NicolauDavid Padua, editor, *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, volume 768, pages 500–521, August 12-14, 1993.

This article was processed using the LaTeX macro package with LLNCS style