A Propagation Engine for GCC

Diego Novillo

Red Hat Canada

GCC Developers' Summit – Jun 23, 2005
Ottawa, Canada

Introduction

- Several transformations can be expressed in terms of propagating values or attributes through the IL
  - Constants
  - Copies
  - Ranges
  - Type attributes
- Engine is a generalization of propagation code in SSA-CCP
- Propagation is done through simulation
  - Assignments generate new values
  - Values are stored in a value array indexed by SSA number
  - Simulation keeps track of def-use and control edges

Propagation Engine Overview

- Simulates execution of statements that produce "*interesting*" values.
- Flow of control and data are simulated with work lists.
  - CFG work list → control flow edges.
  - SSA work list → def-use edges.
- Values produced by an expression are associated to the SSA name on the LHS of the expression.
- User deals with values produced by statements and PHI nodes.
- Engine deals with all the mechanics of visits and iteration.

Propagation Engine Details – 1

- In CCP, a_4 `= 13` is represented with `const_val[4] = 13`
- After visiting that statement, all statements that use a_4 are added to the SSA work list.
- If a conditional jump uses a_4, and the predicate can be computed at compile-time, only the edges over which the predicate is true are added to the CFG work list.
- Usage

```
ssa_propagate (visit_stmt, visit_phi)
```

Propagation Engine Details – 2

- Mark all edges not-executable and seed CFG work list with starting basic block.
- Take block **B**. Evaluate every statement **S** by calling **`visit_stmt`**:
  a) SSA_PROP_INTERESTING: **S** produces an interesting value.
     - Regular statement, user returns SSA name N_i where value has been stored. Def-use edges out of N_i are added to SSA work list.
     - If **S** is a conditional jump, user code returns edge that will always be taken.

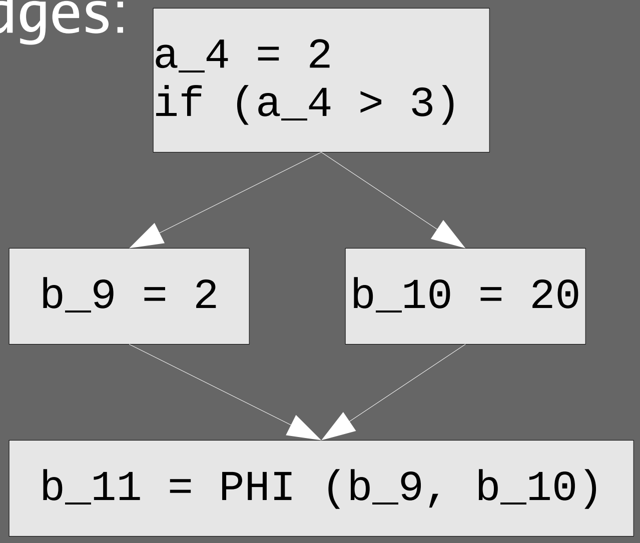
  b) SSA_PROP_NOT_INTERESTING: No edges added. **S** may be visited again.

  c) SSA_PROP_VARYING: Edges added. **S** will *not* be visited again.

redhat

Propagation Engine Details – 3

- Once all statements have been visited, they are not visited again unless their operands change and they have not been marked *varying.*
- If **B** has PHI nodes, call `visit_phi`.
  - PHI nodes are *always* simulated.
  - User code may choose to only visit arguments flowing through executable edges:



Propagation Engine Details – 4

  - Return values from `visit_phi` have same semantics as `visit_stmt`.
  - PHI nodes are merging points, so they need to "intersect" all the incoming arguments.
- Simulation terminates when both SSA and CFG work lists are drained.
- Values should be kept in an array indexed by SSA version number.
- After propagation, call `substitute_and_fold` to do final replacement in IL.

Propagating Memory Operations

- For memory store/load expressions, propagated values are associated with memory expression.
- Final substitution will replace loads with propagated values if the associated memory expression matches the load expression.

A_3 associated with
<13, A[i_9]>

```
# A_3 = V_MAY_DEF <A_2>
A[i_9] = 13
[ ... ]
# VUSE <A_3>
x_3 = A[i_9]
```

Load from A_3 uses
same memory expression
as the store

redhat

Value Range Propagation – 1

- Based on Patterson's range propagation for jump prediction
  - No branch probabilities (only taken/not-taken)
  - Only a single range per SSA name.
- Goal is to reduce bound checking code generated by compiler (Java, mudflap, etc).

```
for (int i = 0; i < a->len; i++)
  {
    if (i < 0 || i >= a->len)
      throw 5;
    call (a->data[i]);
  }
```

- Conditional inside the loop is unnecessary.

Value Range Propagation – 2

- Two main phases
  - **Range Assertions**. When a conditional executes, the taken branch indicates what values will the SSA name(s) in the predicate take:

```
if (a_3 > 10)
  a_4 = ASSERT_EXPR <a_3, a_3 > 10>
  ...
else
  a_5 = ASSERT_EXPR <a_4, a_4 <= 10>
```

    Now we can associate a range value to a_4 and a_5.

  - **Range propagation**. Value ranges derived from assertions and other expressions are propagated using the propagation engine.

Value Range Propagation – 3

- Why are ASSERT_EXPR necessary?

```
p_4 = p_3 + 1
if (p_4 == 0)
   ...
x_10 = *p_4
...
if (p_4 == 0)
    ...
```

We can't tell if p_4 is 0 here

p_4 can't possibly be 0 here

- We cannot associate a known range value to p_4.
- An ASSERT_EXPR after `x_10` will create a new version `p_5 = ASSERT_EXPR <p_4, p_4 != 0>` to which we can pin the non-NULL range.
- A new version guarantees that the range is associated in the right area of the code.

Value Range Propagation – 4

- Two range representations
  - Range [MIN, MAX] → MIN <= N <= MAX
  - Anti-range ~[MIN, MAX] → N < MIN or N > MAX
- Lattice has 4 states

- No upward transitions
- Infinite values are represented using `TYPE_MIN_VALUE` and `TYPE_MAX_VALUE`

Value Range Propagation – 5

- Statements are evaluated by `vrp_visit_stmt`.
- Expression evaluation is a bit more involved than CCP.
- There is some limited symbolic processing (mostly taken out of predicates involving more than one SSA name).
- Equivalences between names are also propagated. Multiple ranges per name.

```
if (p_4)
  if (q_3 == p_4)
    if (q_3)     /* Redundant.  */
```

- If an expression cannot resolve into a range, it tries to derive an anti-range before giving up.
- Scalar evolutions are used to refine ranges for statements inside loops.

Value Range Propagation – 6

- PHI nodes are evaluated by `vrp_visit_phi`.
- When two ranges VR0 and VR1 have a non-empty intersection, it merges into VR0 U VR1.
- It also tries to derive an anti-range before giving up (e.g., PHI <~[0, 0], [10, 20]> is ~[0, 0]).
- Once propagation is complete
  - Single valued ranges are stored in a value vector.
  - Call `substitute_and_fold` to fold superfluous predicates, simplify statements using range information and do constant/copy replacement and folding with the single valued ranges.

redhat

Conclusions

- Propagation algorithm in SSA-CCP can be abstracted out and re-used in several other propagation problems.
- Three basic elements
  - A lattice to control state transitions.
  - Implement a statement visit function.
    - Return 3 indicators: interesting, not interesting, varying.
  - Implement a PHI visit function.
    - Same 3 indicators.
    - Merge values from executable edges.
- To do
  - More than a single SSA name returned from a statement visit.
  - More than one edge taken from a conditional jump visit.