Global optimization

Honors Compilers

April 16th 2002

Data flow analysis

To generate better code, need to examine definitions and uses of variables beyond basic bloc

With use-definition information, various optimizing transformations can be performed:

- Common subexpression elimination
- Loop-invariant code motion
- Constant folding
- Reduction in strength
- Dead code elimination

Basic tool: iterative algorithms over graphs

The flow graph

Nodes are basic blocks

Edges are transfers (conditional/unconditional jumps)

For every node B (basic block) we define the sets

- Pred (B) and succ (B) which describe the graph

Within a basic block we can easily single pass) compute local information, typically a set

- Variables that are assigned a value : def (B)
- Variables that are operands: use (B)

Global information reaching B is computed from th information on all Pred (B) (forward propagation) o Succ (B) (backwards propagation)

Example: live variable analysis

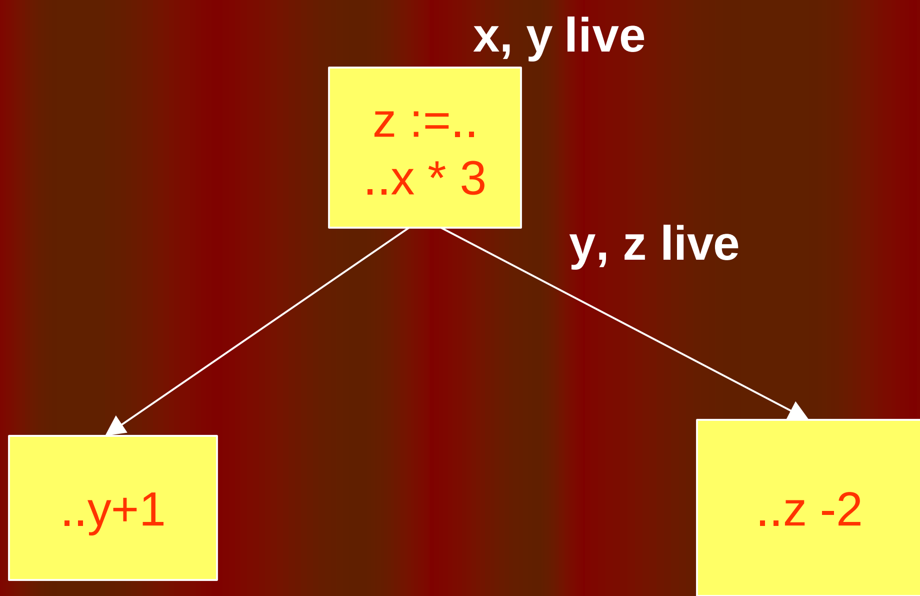
Definition: a variable is a live if its current value is used subsequently in the computation

Use: if a variable is not live on exit from a block, it does not need to be saved (stored in memory)

Livein (B) and Liveout (B) are the sets of variables live on entry/entry from block B.

- Liveout (B) = È livein (n) over all n e succ (B)
- A variable is live on exit from B if it is live in any successor of B
- Livein (B) = liveout (B) È use (B) – defs (B)
- A variable is live on entrance if it is live on exit or used within B
- Live (B_{exit}) = f
- On exit nothing is live

Liveness conditions



Example: reaching definitions

Definition: the set of computations (quadruples) that may be used at a point

Use: compute use-definition relations.

In (B) = ∪ out (p) for all p e pred (B)

- A computation is reaches the entrance to a block if it reached the exit of a predecessor

Out (B) = in (B) + gen (B) – kill (B)

- A computation reaches the exit if it is reaches the entrar and is not recomputed in the block, or if it is computed locally

In (B_{entry}) = f

- Nothing reaches the entry to the program

Iterative solution

Note that the equations are monotonic: if out (B) increases,
(B ') increases for some successor.

General approach: start from lower bound, iterate until nothing changes.

```
Initially in (b) = f  for all b, out (b) = gen (b)
change := true;
while change loop
        change := false;
        forall b e blocks loop
            in (b) = ∪ out (p),  forall p e pred (b);
            oldout := out (b);
            out (b) := gen (b) ∪ in (b) – kill (b);
            if oldout /= out (b) then change := true; end if;
      end loop;
end loop;
```

Workpile algorithm

Instead of recomputing all blocks, keep a queue of nodes that may have changed. Iterate until queue empty:

```
while not empty (queue) loop
    dequeue (b);
    recompute (b);
    if b has changed, enqueue all its successors;
end loop;
```

Better algorithms use node orderings.

Example: available expressions

Definition: computation (triple, e.g. x+y) that may be available at a point because previously compute

Use: common subexpression elimination

Local information:

- exp_gen (b) is set of expressions computed in b
- exp_kill (b) is the set of expressions whose operands are evaluated in b

in (b) = n out(p) for all p e pred (b)

- Computation is available on entry if it is available on exit from all predecessors

out (b) = exp_gen (b) ∪ in (b) – exp_kill (b)

Iterative solution

Equations are monotonic: if out (b) decreases, in (
can only decrease, for all successors of b.

Initially

in (b_{entry}) = f , out (b_{entry}) = e_gen (b_{entry})

For other blocks, let U be the set of all expresions,
then

out (b) = U- e_kill (b)

Iterate until no changes: in (b) can only decrease.
Final value is at most the empty set, so convergen
is guaranteed in a fixed number of steps.

Use-definition chaining

The closure of available expressions: map each occurrence (operand in a quadruple) to the quadruple that may have generated the value.

ud (o): set of quadruples that may have computed the value of o

Inverse map: du (q) : set of occurrences that may use the value computed at q.

finding loops in flow-graph

A node n1 dominates n2 if all execution paths that reach n2 go through n1 first.

The entry point of the program dominates all node in the program

The entry to a loop dominates all nodes in the loop

A loop is identified by the presence of a (back) edg from a node n to a dominator of n

Data-flow equation:

dom (b) = n dom (p) forall p e b

a dominator of a node dominates all its predecessors

Loop optimization

A computation (x op y) is invariant within a loop if

- x and y are constant
- ud (x) and ud (y) are all outside the loop
- There is one computation of x and y within the loop, and that computation is invariant

A quadruple Q that is loop invariant can be moved to the pre-header of the loop iff:

- Q dominates all exits from the loop
- Q is the only assignment to the target variable in the loo
- There is no use of the target variable that has another definition.

An exception may now be raised before the loop

Strength reduction

Specialized loop optimization: formal differentiation

An induction variable in a loop takes values that form an arithmetic series: $k = j * c_0 + c_1$

Where j is the loop variable j = 0, 1, ... , c and k a constants. J is a basic induction variable.

Can compute $k := k + c_0$, replacing multiplication with addition

If j increments by d, k increments by $d * c_0$

Generalization to polynomials in j: all multiplication can be removed.

Important for loops over multidimensional arrays

Induction variables

For every induction variable, establish a triple (var, incr, init)

The loop variable *v* is (*v*, 1, v_0)

Any variable that has a single assignment of the form $k := j * c_0 + c_1$ is an induction variable with (j, $c_0 * incr_j$, $c_1 + c_0 j_0$)

Note that $c_0 * incr_j$ is a static constant.

Insert in loop pre-header: $k := c_0 * j_0 + c_1$

Insert after incrementing j: $k := k + c_0 * incr_j$

Remove original assignment to k

Global constant propagation

- Domain is set of values, not bit-vector.
- Status of a variable is (c, non-const, unknown)
- Like common subexpression elimination, but instead of intersection, define a merge operation:
  - Merge (c, unknown) = c
  - Merge (non-const, anything) = non-const
  - Merge (c1, c2) = if c1 = c2 then c1 else non-const

    In (b) = Merge { out (p) } forall p e pred (b)
- Initially all variables are unknown, except for explicit constant assignments