Global optimization

Honors Compilers
April 16th 2002
Data flow analysis

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

- $\text{Pred}(B)$ and $\text{succ}(B)$ which describe the graph

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

- Variables that are assigned a value: $\text{def}(B)$
- Variables that are operands: $\text{use}(B)$

Global information reaching $B$ is computed from the information on all $\text{Pred}(B)$ (forward propagation) and $\text{Succ}(B)$ (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.

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

\[ z := \ldots \\
\ldots x \times 3 \\
\ldots y + 1 \\
\ldots z - 2 \]

- \( x, y \) live
- \( y, z \) live
Example: reaching definitions

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

**Use**: compute use-definition relations.

\[
\text{In} (B) = \bigcup \text{out} (p) \text{ for all } p \in \text{pred} (B)
\]

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

\[
\text{Out} (B) = \text{in} (B) + \text{gen} (B) - \text{kill} (B)
\]

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

\[
\text{In} (B_{\text{entry}}) = f
\]

- 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 in blocks loop
        in (b) = \cup out (p), forall p in pred (b);
        oldout := out (b);
        out (b) := gen (b) \cup 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:

```plaintext
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 computed.

**Use:** common subexpression elimination

**Local information:**
- \(\text{exp}_\text{gen}(b)\) is set of expressions computed in \(b\)
- \(\text{exp}_\text{kill}(b)\) is the set of expressions whose operands are evaluated in \(b\)

\[
in(b) = \forall p \in \text{pred}(b) \quad \text{out}(p)
\]

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

\[
\text{out}(b) = \text{exp}_\text{gen}(b) \cup \text{in}(b) - \text{exp}_\text{kill}(b)
\]
Iterative solution

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

Initially
\[ \text{in (} b_{\text{entry}} \text{)} = f \ , \ \text{out (} b_{\text{entry}} \text{)} = e_{\text{gen}} (b_{\text{entry}}) \]

For other blocks, let U be the set of all expressions, then
\[ \text{out (} b \text{)} = U - e_{\text{kill}} (b) \]

Iterate until no changes: in (b) can only decrease.

Final value is at most the empty set, so convergence 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 $n_1$ dominates $n_2$ if all execution paths that reach $n_2$ go through $n_1$ first.

The entry point of the program dominates all nodes in the program.

The entry to a loop dominates all nodes in the loop.

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

Data-flow equation:

$$\text{dom}(b) = n \quad \text{dom}(p) \quad \forall p \in b$$

A dominator of a node dominates all its predecessors.
A computation \((x \text{ op } y)\) 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 loop
- 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:

\[ k = j \times c_0 + c_1 \]

Where \( j \) is the loop variable \( j = 0, 1, \ldots \), \( c \) and \( k \) are 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 \( d \times c_0 \).

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 \( k := j \cdot c_0 + c_1 \) is an induction variable with \( (j, c_0 \cdot incr_j, c_1 + c_0 j_0) \)
Note that \( c_0 \cdot incr_j \) is a static constant.
Insert in loop pre-header: \( k := c_0 \cdot j_0 + c_1 \)
Insert after incrementing \( j \): \( k := k + c_0 \cdot incr_j \)
Remove original assignment to \( k \)
Global constant propagation

Domain is set of values, not bit-vector.

Status of a variable is \( (c, \text{non-const, unknown}) \).

Like common subexpression elimination, but instead of intersection, define a merge operation:

- \( \text{Merge} (c, \text{unknown}) = c \)
- \( \text{Merge} (\text{non-const}, \text{anything}) = \text{non-const} \)
- \( \text{Merge} (c_1, c_2) = \begin{cases} c_1 & \text{if } c_1 = c_2 \\ \text{non-const} & \text{else} \end{cases} \)

Let \( \text{In} (b) = \text{Merge} \{ \text{out} (p) \} \text{forall } p \in \text{pred} (b) \).

Initially all variables are unknown, except for explicit constant assignments.