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 to optimize the code.

Data Flow Analysis

Basics tool: iterative algorithms over graphs

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

A. Pnueli
Nodes are basic blocks.

**The Flow Graph**

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

- Edges are transfers (conditional/unconditional jumps)
- Nodes are basic blocks

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

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

Informations reaching $B$ is computed from the information on all $\text{Pred}(B)$ (forward propagation) or all $\text{Succ}(B)$ (backward propagation)

Global informations are computed:

- Honors Compilers, NYU, Fall, 2007

A. Pnueli
Example: Live Variables Analysis

Definition: A variable is alive at a location if its current value is used subsequently in the computation.

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

Definiton: A variable is alive on exit from a block if it is live on exit or used within it.

- $\phi = (\text{Liveout}(B)_{\text{exit}})^{\text{def}}$
- $\text{Livein}(B) = (\text{Liveout}(B) \cup \text{use}(B)) - \text{defs}(B)$
- $\text{Livein}(B) = \text{Liveout}(B)_{\text{entry}}$ and $\text{Liveout}(B) = \text{Livein}(B)_{\text{exit}}$

On entry to or exit from block $B$, the sets of variables live on entry to or exit from block $B$ are the sets of variables which are defined before they are used.

We exclude variables which are defined before they are used.

On exit, nothing is alive.
$\exists x \cdot \lnot \exists y :: z = x + y + 1$

Liveness Conditions
Example: Reaching Definitions

- Nothing reaches the entry of the program.
  \[ \emptyset = \text{entry}(B) \cap \text{init}(B) \]

- In the block, or if it is computed locally:
  - A computation reaches the exit if it reaches the entry and is not recomputed.
    \[ \text{Out}(B) = \text{In}(B) \cap (\text{gen}(B) \cup \text{kill}(B)) \]

- A computation reaches the entry of a block if it reaches the exit of a predecessor.

- Use: compute use-definition relations.

- At a location:
  - Definition: the set of computations (quadruples) that may be used.

A. Pnueli
Note that the equations are monotonic: if $O(B)$ increases, $I(B)^{un}$ increases; if $O(B)$ decreases for some successor, $I(B)$ decreases.

Iterative Solution

General approach: Start from lower bound (e.g. empty sets), iterate until nothing changes.

Initially $\forall q, \forall B \in \text{blocks}, \text{change} := \text{false};$

while $\text{change} := \text{true};$

for all $q, \forall B \in \emptyset = (q)\text{out}$

end loop;

end loop;

end loop;
Work-pile Algorithm

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

```
while not empty(queue):
    dequeue \( b \);
    recompute \( \text{out}(b) \);
    if \( \text{out}(b) \) has changed then enqueue all of \( b \)'s successors;

Better algorithms use node orderings.
```
Example: Available Expressions

Definition: Computation (triple, e.g., (x + y) that may be previously computed.

Use: common subexpressions elimination

Local Information:

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

\( \text{Out}(b) = \text{In}(b) \land \text{ExpGen}(b) \land \text{ExpKill}(b) \)

\( \text{In} = \bigcup_{d \in \text{Pred}(b)} \text{Out}(d) \cap \text{ExpGen}(b) \setminus \text{ExpKill}(b) \)
Iterative Solution

Note that the equations are monotonic: if $O(B)'$ increases,

Iterate until nothing changes.

Approach as above: start from lower bound (e.g. empty sets),

Initially $\forall q, \emptyset = (q)\text{out}$

\[ (q)^p \text{kill} = (q)^{\text{exp-gen}} \bigcup (q)^{\text{exp-kill}} \]

\[ \text{if oldout} \neq \text{out} \text{ then change } := \text{true; end it; if out} \text{ else } \]

$\text{out} := \text{false; end loop; end loop; end loop; end loop;}$

\[ \forall q, \forall B \in \text{blocks} \]

\[ \text{change } := \text{false; while change} \]

\[ \text{change } := \text{true; for all } q, \emptyset = (q)\text{out} \]

\[ (q)^p \text{kill} = (q)^{\text{exp-gen}} \bigcup (q)^{\text{exp-kill}} \]

\[ \text{if oldout} \neq \text{out} \text{ then change } := \text{true; endif; } \]

Approach as above: start from lower bound (e.g. empty sets),

Iterate until nothing changes.
Use-definition Chaining

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

- Value of an operand: \((b)_{np}\)
- Inverse map: \((o)_{pn}\)
- Use-denition chaining
Finding Loops in a Flow-Graph

A.Pnueli

Lecture 10: Global Optimization

Honors Compilers, NYU, Fall, 2007

A node \( n \) dominates all the predecessors of \( n \).

A dominator of a node \( n \) which is different from \( n \), must dominate all the predecessors of \( n \).

For all \( B \neq B_{entry} \) and \( \emptyset \):

\[
\{ u, \ldots, i \} = (B_{dom})_{B_{entry}}
\]

Initial conditions should be:

\[
(d)_{B_{dom}}(q)_{pred} \bigcup \{ q \} = (q)_{dom}
\]

The entry point of the program dominates all other nodes.

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

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

A dominator of a node \( n \), which is different from \( n \), must dominate all the predecessors of \( n \).

\( n \) to a dominator of \( n \).

The entry point of the program dominates all other nodes.

A node \( n \) dominates \( u \) if, either \( u = n \) or all execution paths that reach \( u \) must go through \( n \) first.
\{0, 1, 3, 4, 7, 8, 10\} = \{8, 7, 3, 4, 1\} \cap \{10\} = (8)D \cap \{10\} = (10)D

\{0, 1, 3, 4, 7, 8\} = \{8, 7, 3, 4, 1\} \cap \{6\} = (8)D \cap \{6\} = (6)D

\{8, 7, 3, 4, 1\} = \{7, 4, 3, 1\} \cap \{8\} = (7)D \cap \{8\} = (8)D

\{7, 4, 3, 1\} = (\{10, \ldots, 2, 1, 3, 4, 6, 1\} \cup \{5, 4, 3, 1\}) \cap \{7\} = (10)D \cup (6)D \cup (5)D \cap \{7\} = (7)D

\{4, 3, 1\} = \{4, 3, 1\} \cap \{6\} = (4)D \cap \{6\} = (6)D

\{4, 3, 5, 1\} = \{4, 3, 1\} \cap \{5\} = (4)D \cap \{5\} = (5)D

\{4, 3\} = (\{10, \ldots, 2, 1, 3, 4\}) \cap \{4\} = (7)D \cup (3)D \cap \{4\} = (4)D

\{0, 1, 3, 4, 7, 8\} = (8)D \cup (4)D \cup (7)D \cup (1)D \cap \{3\} = (3)D

\{1\} = \emptyset \cap \{1\} = (1)D
The Dominators Tree
This is a loop induced by a back edge $u$. This is a loop induced by a back edge $u$ in which node $p$ dominates node $u$. The scope of the loop induced by the back edge $u$ is constructed as follows: $p \leftarrow u$. Node $p$ will be the header of the loop. Mark node $p$ as visited.

Perform a depth-first search on the reverse control-flow graph, starting with $u$. Insert all the nodes visited in this search into loop.

Natural Loop
A. Pnueli

Hions Compilers, NYU, Fall, 2007

Lecture 10: Global Optimization

Loops in Example

\[
\begin{align*}
\{0, 1, 2, \ldots, 10\} & = (i \leftarrow 9) \text{loop} \\
\{3, 4, 5, 6, 7, 8, 10\} & = (i \leftarrow 8) \text{loop} \\
\{4, 5, 6, 7, 8, 10\} & = (i \leftarrow 7) \text{loop} \\
\{7, 8, 10\} & = (i \leftarrow 6) \text{loop}
\end{align*}
\]
A computation (x op y) is invariant within a loop if

- An exception may now be raised before the loop
- There is no use of the target variable that has another definition
- There is the only assignment to the target variable in the loop, and
  - Q dominates all exits from the loop, and
  - Q is the only assignment to the target variable in the loop, and
- A quadruple Q that is loop invariant can be moved to the pre-header of the loop iff
  - A quadruple Q that is loop invariant can be moved to the pre-header of the loop iff
    - The quadruple Q that is loop invariant can be moved to the pre-header of the loop iff
      - These computations are invariant
        - There is at most one computation of x and y within the loop, and
        - (f)pn and (x)pn are all outside the loop, or
        - x and y are constant, or
      - A computation (x op y) is invariant within a loop if
A. Pnueli

### Lecture 10: Global Optimization

**Strength Reduction**

**Specialized Loop Optimization:**

- **Formal Differentiation**

### Generalization to Polynomials in $j$

- All multiplications can be removed.

### Incrementation by $p$

- $k$ increments by $p \cdot c_0$.

### Additional Important for Loops over Multi-dimensional Arrays

- Can be removed.

### Constants

- $k$ is a basic induction variable.

### Arithmetic Series

- An arithmetic series: $k = \sum c_i \cdot j = 0, 1, \ldots, c_0$ and $c_1$ are

### Loop Over Multidimensional Arrays

- Important for loops over multi-dimensional arrays.

### Honors Compilers, NYU, Fall, 2007

273
Induction Variables

For every induction variable, establish a triple

\( (\text{var}, \text{incr}, \text{init}) \)

Any variable that has a single assignment of the form

\( k := j \ c_0 + c_1 \)

is an induction variable with triple

\( (k, c_0 \ \text{incr}, c_0 \ \text{init} + c_1) \)

Note that \( c_0 \ \text{incr} \) is a static constant.

Insert in loop pre-header:

\( k := c_0 \ \text{init} + c_1 \)

Insert after incrementing \( j \):

\( k := k + c_0 \ \text{incr} j \)

Remove original assignment to \( k \).
Constant assignments
Initially all variables are unknown, except for explicit

\[(d) \in O \quad \text{merge} \quad \in \quad (q) \nu I\]
\[\text{merge}(c_1,c_2) = \text{if } c_1 = c_2 \text{ then } c_1 \text{ else non-const}\]
\[\text{merge}(\text{non-const, anything}) = \text{non-const}\]
\[\text{merge}(c,\text{unknown}) = c\]

Domain is set of values (not bit-vector).

Global Constant Propagation