• Questions?

• *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, by Cytron, Ferrante, Rosen, Wegman, and Zadeck

• HW 2 now posted - Building & using SSA
Overview

- Quick review of SSA
- Building SSA
  - Dominance frontiers
  - Inserting $\phi$ instructions
  - Renaming variables to unique versions
- Loop invariant code motion
- Translating out of SSA
Recap

- Static single assignment form (SSA)
  - Sparse data flow representation between definitions and uses:
    - Each definition is a unique assignment
    - Each use refers to a single definition
    - $\Phi$ introduced at merge points
  - Enhances & simplifies many optimizations
Straight-line example

\[
\begin{align*}
V & \leftarrow 4 \\
& \leftarrow V + 5 \\
V & \leftarrow 6 \\
& \leftarrow V + 7 \\
V_0 & \leftarrow 4 \\
& \leftarrow V_0 + 5 \\
V_1 & \leftarrow 6 \\
& \leftarrow V_1 + 7
\end{align*}
\]
Merge points

V = 1  V = 2  V = 3

X = V  Y = V  Z = V
\[ v_1 = 1 \quad v_2 = 2 \quad v_3 = 3 \]

\[ v_4 = \Phi(v_1, v_2, v_3) \]

\[ x = v_4 \quad y = v_4 \quad z = v_4 \]
Another example
Another example

```
v1 = 0
v3 = 1
v5 = !(v1,v2)
x = v5
v4 = foo()
v6 = !(v3,v4)
v7 = !(v2,v6)
y = v7
v2 = x + 1
v8 = foo(x,v4)
v9 = foo(x,v6)
v10 = foo(x,v7)
y = v10
```

Diagram:

- A
  - B (v1 = 0)
    - C (v3 = 1)
      - D (v5 = !(v1,v2))
        - E (x = v5)
          - F (v4 = foo())
            - G (v6 = !(v3,v4))
              - H (v7 = !(v2,v6))
                - I (y = v7)
```
Constraints on SSA

1. If twononnull paths $X \rightarrow Z$ and $Y \rightarrow Z$ converge at a node $Z$, and nodes $X$ and $Y$ contain assignments to $V$ (in the original program), then a trivial $\phi$-function $V = \phi(V_i, ..., V)$ has been inserted at $Z$ (in the new program).

2. Each mention of $V$ in the original program or in an inserted $\phi$-function has been replaced by a mention of a new variable $V_i$, leaving the new program in SSA form.

3. Along any control flow path, consider any use of a variable $V$ (in the original program) and the corresponding use of $V_i$ (in the new program). Then $V$ and $V_i$ have the same value.
Minimal SSA

- SSA form is *minimal* if the number of $\Phi$ instructions is as small as possible subject to (1).

- A minimal SSA form is *pruned* if only $\Phi$ instructions with live results are inserted. This breaks (1) but is safe.

- More expensive to build the pruned form.

- Different optimizations want different forms.
How do we compute SSA?

• Where to place Φ instructions?
• Naive approach:
  • Consider every pair of assignments
  • Calculate all possible merge points
  • Insert Φ at every merge point
  • Expensive!
Observation

- SSA closely related to dominance
- Definition must dominate non-Φ use
- Φ instruction appears when a definition reaches non-dominated successor
- Suppose V is assigned in node X and there exists P and Y such that P->Y, dom(X, P), and not dom(X, Y). Then Y must contain a Φ.
Dominance frontier

- The dominance frontier of $X$ is the set of nodes $Y$ where
  - $X$ dominates a predecessor of $Y$, and
  - $X$ does not strictly dominate $Y$
- Note, if $X \rightarrow X$, then $X$ in $DF(X)$.
- Formally:
  - $DF(X) = \{ Y \mid (\exists P \in \text{Pred}(Y))(X \not\gg P \text{ and } X \gg Y) \}$
Computing the dominance frontier

- $DF(X)$ breaks into two components:
  
  $$DF(X) = DF_{local}(X) \cup \bigcup_{Z \in \text{children}(X)} DF_{up}(Z)$$

- A local part: $DF_{local}(X) \overset{\text{def}}{=} \{ Y \in \text{Succ}(X) \mid X \gg Y \}$

- An inherited part: $DF_{up}(Z) \overset{\text{def}}{=} \{ Y \in DF(Z) \mid \text{idom}(Z) \gg Y \}$

- Bottom-up traversal of dominator tree
DF algorithm

for each \( X \) in a bottom-up traversal of the dominator tree do
    \( DF(X) \leftarrow \emptyset \)
    for each \( Y \in \text{Succ}(X) \) do
        /*local*/ if \( idom(Y) \neq X \) then \( DF(X) \leftarrow DF(X) \cup \{Y\} \)
        end
        for each \( Z \in \text{Children}(X) \) do
            for each \( Y \in DF(Z) \) do
                /*up*/ if \( idom(Y) \neq X \) then \( DF(X) \leftarrow DF(X) \cup \{Y\} \)
                end
            end
        end
    end
v1 = 0

v3 = 1

v5 = (v1, v2)

x = v5

v4 = foo()

v6 = (v3, v4)

v7 = (v2, v6)

y = v7

v2 = x + 1

DF_{local}(I) = {}

DF_{local}(H) = \{ I \}

DF_{local}(G) = \{ D, I \}

DF_{local}(F) = \{ H \}

DF_{local}(E) = \{ H \}

DF_{local}(D) = {}

DF_{local}(C) = {}

DF_{local}(B) = {}

DF_{local}(A) = {}
DF(I) = \{\}
DF(H) = \{I\}
DF(G) = \{D, I\}
DF(F) = \{H\}
DF(E) = \{H\}
DF(D) = \{D, I\}
DF(C) = \{I\}
DF(B) = \{I\}
DF(A) = \{}
Φ placement

for each variable $V$
    $HasAlready ← Φ$
    $EverOnWorkList ← Φ$
    $WorkList ← Φ$
    for each node $X$ containing an assignment to $V$
        $EverOnWorkList ← EverOnWorkList ∪ \{X\}$
        $WorkList ← WorkList ∪ \{X\}$
    end for
while $WorkList ≠ Φ$
    remove $X$ from $WorkList$
    for each $Y ∈ DF(X)$
        if $Y \notin HasAlready$
            insert a Φ-node for $V$ at $Y$
            $HasAlready ← HasAlready ∪ \{Y\}$
        end if
        if $Y \notin EverOnWorkList$
            $EverOnWorkList ← EverOnWorkList ∪ \{Y\}$
            $WorkList ← WorkList ∪ \{Y\}$
        end if
    end for
end while
endfor
Renaming

Data Structures

Stacks array of stacks, one for each original variable V
  The subscript of the most recent definition of V
  Initially, Stacks[V] = EmptyStack, ∀ V

Counters an array of counters, one for each original variable
  The number of assignments to V processed
  Initially, Counters[V] = 0, ∀ V

procedure GenName(Variable V)
  i ← Counters[V]
  replace V by V_i
  Push i onto Stacks[V]
  Counters[V] ← i + 1

Rename - a recursive procedure

• Walks the dominator tree in preorder
• Initially, call Rename(entry)
procedure Rename(Block X)
    // first process ϕ-nodes
    for each ϕ-node P in X
        GenName(LHS(P))
    // then process statements in block X
    for each statement A in X
        for each variable V ∈ RHS(A)
            replace V by V_i, where i = Top(Stacks[V])
        for each variable V ∈ LHS(A)
            GenName(V)
    // then update any ϕ-functions in CFG successors of X
    for each Y ∈ SUCC(X)
        j ← position in Y’s ϕ-nodes corresponding to X
        for each ϕ-node P in Y
            replace the j^{th} operand of RHS(P) by V_i
            where i = Top(Stacks[V])
    // recursively visit children of X in dominator tree
    for each Y ∈ SUCC(X)
        Rename(Y)
    // when backing out of X, pop variables defined in X
    for each ϕ-node or statement A in X
        for each V_i ∈ LHS(A)
            Pop(Stacks[V])
Translating out of SSA

- Map back to regular variables
- Property of original SSA
  - No two versions of the original variable are live simultaneously
  - Preserved by some optimizations, not by others
  - When preserved, just restore original names
CSE violates property

<table>
<thead>
<tr>
<th>Original</th>
<th>After SSA</th>
<th>After CSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = a + b)</td>
<td>(x_1 = a + b)</td>
<td>(x_1 = a + b)</td>
</tr>
<tr>
<td>(w = x + 1)</td>
<td>(w = x_1 + 1)</td>
<td>(w = x_1 + 1)</td>
</tr>
<tr>
<td>(x = \text{foo}())</td>
<td>(x_2 = \text{foo}())</td>
<td>(x_2 = \text{foo}())</td>
</tr>
<tr>
<td>(y = a + b)</td>
<td>(y = a + b)</td>
<td>(y = x_1)</td>
</tr>
<tr>
<td>(z = x + 2)</td>
<td>(z = x_2 + 2)</td>
<td>(z = x_2 + 2)</td>
</tr>
</tbody>
</table>
General approach

• Don’t restore original variables
• Replace phi instructions with copy instructions in predecessors
• Rely on operand coalescing / register allocation to combine variables when possible
HW 2

Construct SSA

Two optimizations

• Constant propagation
• Global common subexpression elimination

Two more possibilities for extra credit

• Copy propagation
• Loop-invariant code motion

Reports
Finding loops

- Compute dominators and back edges (lecture 2)
- Back edges define loop:
  - Loop header (target of back edge)
  - Loop preheader (create empty one if necessary)
  - Loop nodes
Loop invariant code motion

- Test for statement where:
  - The statement is in a loop and all right-hand side operands are defined outside the loop
  - The statement has no side effects (e.g., a call)
  - The statement does not access memory (load / store)
- Then, the right-hand side expression may be hoisted into the preheader of the loop.
Example

\[ C_2 = \Phi(C_1, C_3) \]

\[ X = A + B \]
\[ T = A + B \]
\[ C_2 = \Phi(C_1, C_3) \]
\[ X = T \]
Questions

• Does LICM produce overlapping SSA variables?

• What order to process instructions for LICM? Does it matter?

• How do nested loops affect LICM?