Lecture 13
Introduction to Static Single Assignment (SSA)

(Slide content courtesy of Seth Goldstein.)

Recurring Theme: Where Is a Variable Defined or Used?

• Example: Loop-Invariant Code Motion
  – Are B, C, and D only defined outside the loop?
  – Other definitions of A inside the loop?
  – Uses of A inside the loop?

• Example: Copy Propagation
  – For a given use of X:
    • Are all reaching definitions of X:
      – copies from same variable: e.g., X = Y
    • Where Y is not redefined since that copy?
  – If so, substitute use of X with use of Y

• It would be nice if we could traverse directly between related uses and def's
  – this would enable a form of sparse code analysis (skip over “don’t care” cases)

Appearances of Same Variable Name May Be Unrelated

X₁ = A + 1  Y = X₁ + B
F = 2  F = 3
X₂ = F + 7  C = X₂ + D

• The values in reused storage locations may be provably independent
  – in which case the compiler can optimize them as separate values
• Compiler could use renaming to make these different versions more explicit

Definition-Use and Use-Definition Chains

X₁ = A + 1  Y = X₁ + B
F = 2  F = 3
X₂ = F + 7  C = X₂ + D

• Use-Definition (UD) Chains:
  – for a given definition of a variable X, what are all of its uses?
• Definition-Use (DU) Chains:
  – for a given use of a variable X, what are all of the reaching definitions of X?
Unfortunately DU and UD Chains Can Be Expensive

foo(int i, int j) {
    switch (i) {
        case 0: x=3; break;
        case 1: x=1; break;
        case 2: x=6; break;
        case 3: x=7; break;
        default: x = 11;
    }
    switch (j) {
        case 0: y=x+7; break;
        case 1: y=x+4; break;
        case 2: y=x-2; break;
        case 3: y=x+1; break;
        default: y=x+9;
    }
}

In general, Ndefs \( \Rightarrow O(NM) \) space and time

One solution: limit each variable to ONE definition site

Static Single Assignment (SSA)

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - each use uses the most recently defined var.
  - (Similar to Value Numbering)

Straight-line SSA

\[
\begin{align*}
a & \leftarrow x + y \\
b & \leftarrow a + x \\
a & \leftarrow b + 2 \\
c & \leftarrow y + 1 \\
a & \leftarrow c + a \\
a_1 & \leftarrow x + y \\
b_1 & \leftarrow a_1 + x \\
a_2 & \leftarrow b_1 + 2 \\
c_1 & \leftarrow y + 1 \\
a_3 & \leftarrow c_1 + a_2
\end{align*}
\]
**Static Single Assignment (SSA)**

- Static single assignment is an IR where every variable is assigned a value at most once in the program text.
- Easy for a basic block:
  - Assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)
- What about at joins in the CFG?

**Merging at Joins**

```
if (i) {
  a ← x + y
  b ← a + x
} else {
  a ← b + 2
  c ← y + 1
}
```

```
a ← c + a
```

```
c ← 12
if (i)
```

```
a1 ← x + y
b1 ← a1 + x
```

```
a2 ← b + 2
```

```
c2 ← y + 1
```

```
a3 ← Φ(a1, a2)
c3 ← Φ(c1, c2)
b2 ← Φ(b1, ?)
a4 ← c2 + a3
```

```
a4 ← c3 + a3
```

```
c4 ← 12
```

```
a1 ← x + y
b1 ← a1 + x
```

```
a2 ← b + 2
```

```
c2 ← y + 1
```

```
a4 ← c2 + a3
```

```
a4 ← c3 + a3
```

```
c4 ← 12
```

```
a1 ← x + y
b1 ← a1 + x
```

```
a2 ← b + 2
```

```
c2 ← y + 1
```

```
a4 ← c2 + a3
```

```
a4 ← c3 + a3
```
The $\Phi$ function

- $\Phi$ merges multiple definitions along multiple control paths into a single definition.
- At a basic block with $p$ predecessors, there are $p$ arguments to the $\Phi$ function.

\[ x_{\text{new}} \leftarrow \Phi(x_1, x_2, x_3, \ldots, x_p) \]

- How do we choose which $x_i$ to use?
  - We don't really care!
  - If we care, use moves on each incoming edge

"Implementing" $\Phi$

```
c_1 \leftarrow c_2 \leftarrow \Phi(c_1, c_2)
d_2 \leftarrow d_3 \leftarrow \Phi(d_2, d_3)
e_3 \leftarrow e_4 \leftarrow \Phi(e_3, e_4)
```

Trivial SSA

- Each assignment generates a fresh variable.
- At each join point, insert $\Phi$ functions for all live variables.

```
x \leftarrow 1
y \leftarrow x
y \leftarrow 2
z \leftarrow y + x
```

```
x_1 \leftarrow 1
y_1 \leftarrow x_1
y_2 \leftarrow 2
x_2 \leftarrow \Phi(x_1, x_1)
y_3 \leftarrow \Phi(y_1, y_2)
z_1 \leftarrow y_3 + x_2
```

Way too many $\Phi$ functions inserted.

Minimal SSA

- Each assignment generates a fresh variable.
- At each join point, insert $\Phi$ functions for all live variables with multiple outstanding defs.

```
x \leftarrow 1
y \leftarrow x
y \leftarrow 2
z \leftarrow y + x
```

```
x_1 \leftarrow 1
y_1 \leftarrow x_1
y_2 \leftarrow 2
x_2 \leftarrow \Phi(x_1, x_1)
y_3 \leftarrow \Phi(y_1, y_2)
z_1 \leftarrow y_3 + x_2
```
Another Example

```
a ← 0
b ← a + 1
c ← c + b
a ← b * 2
if a < N
  return c
```

Notice use of \(c_1\)

```
a_1 ← 0
a_3 ← \Phi(a_1, a_2)
c_3 ← \Phi(c_1, c_2)
b_2 ← a_1 + 1
c_2 ← c_1 + b_2
a_2 ← b_2 * 2
if a_2 < N
  return c_2
```

When Do We Insert \(\Phi\)?

- If there is a def of \(a\) in block 5, which nodes need a \(\Phi()\)?

CFG

When do we insert \(\Phi\)?

- We insert a \(\Phi\) function for variable \(A\) in block \(Z\) iff:
  - \(A\) was defined more than once before
    - (i.e., \(A\) defined in \(X\) and \(Y\) AND \(X \neq Y\))
    - There exists a non-empty path from \(x\) to \(z\), \(P_{xz}\), and a non-empty path from \(y\) to \(z\), \(P_{yz}\), s.t.
      - \(P_{xz} \cap P_{yz} = \{z\}\)
      - \(z \not\in P_{xz}\) or \(z \not\in P_{yz}\) where \(P_{xz} = P_{xq} \rightarrow z\) and \(P_{yz} = P_{yr} \rightarrow z\)
  - Entry block contains an implicit def of all vars
- Note: \(A = \Phi(...)\) is a def of \(A\)

Dominance Property of SSA

- In SSA, definitions dominate uses.
  - If \(x\) is used in \(x ← \Phi(..., x, ...),\) then \(BB(x)\) dominates \(i^{th}\) predecessor of \(BB(\Phi())\)
  - If \(x\) is used in \(y ← ... x ...\), then \(BB(x)\) dominates \(BB(y)\)
- We can use this for an efficient algorithm to convert to SSA
Dominance

CFG

D-Tree

If there is a def of \( a \) in block 5, which nodes need a \( \Phi() \)?

\[
x \text{ strictly dominates } w \ (x \text{ sdom } w) \iff x \text{ dom } w \text{ AND } x \neq w
\]

Dominance Frontier

CFG

D-Tree

The Dominance Frontier of a node \( x = \{ w \mid x \text{ dom pred}(w) \text{ AND } ! (x \text{ sdom } w) \} \)

Dominance Frontier and Path Convergence

Using Dominance Frontier to Compute SSA

• place all \( \Phi() \)
• Rename all variables
Using Dominance Frontier to Place $\Phi()$

- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in $\text{DominanceFrontier}(\text{defsite})$
      - if we haven’t put $\Phi()$ in node, then put one in
      - if this node didn’t define the variable before, then add this node to the defsites
- This essentially computes the iterated Dominance Frontier on the fly, inserting the minimal number of $\Phi()$ necessary

Renaming Variables

- Algorithm:
  - Walk the D-tree, renaming variables as you go
  - Replace uses with more recent renamed def
- For straight-line code this is easy
- What if there are branches and joins?
  - use the closest def such that the def is above the use in the D-tree
- Easy implementation:
  - for each var: rename(v)
  - rename(v): replace uses with top of stack at def: push onto stack
call rename(v) on all children in D-tree for each def in this block pop from stack

Compute Dominance Tree

1. $i \leftarrow 1$
2. $j \leftarrow 1$
3. $k \leftarrow 0$
4. $k < 100?$
5. $j < 20?$
6. $j \leftarrow i$
7. $k \leftarrow k + 1$
8. $k \leftarrow k + 2$
9. return $j$
**Compute Dominance Frontiers**

- **i** ← 1
- **j** ← 1
- **k** ← 0

- **k** < 100?
  - **j** < 20?
    - **return j**
  - **j** ← **i**
  - **k** ← **k** + 1
  - **j** ← **k** + 1
  - **k** ← **k** + 2
  - **j** ← **Φ**(*j*, *j*)

**Insert Φ()**

- **i** ← 1
- **j** ← 1
- **k** ← 0

- **k** < 100?
  - **j** < 20?
    - **return j**
  - **j** ← **i**
  - **k** ← **k** + 1
  - **j** ← **k** + 1
  - **k** ← **k** + 2
  - **j** ← **Φ**(*j*, *j*)

**Insert Φ()**

- **i** ← 1
- **j** ← 1
- **k** ← 0

- **k** < 100?
  - **j** < 20?
    - **return j**
  - **j** ← **i**
  - **k** ← **k** + 1
  - **j** ← **k** + 1
  - **k** ← **k** + 2
  - **j** ← **Φ**(*j*, *j*)

**Insert Φ()**

- **i** ← 1
- **j** ← 1
- **k** ← 0

- **k** < 100?
  - **j** < 20?
    - **return j**
  - **j** ← **i**
  - **k** ← **k** + 1
  - **j** ← **k** + 1
  - **k** ← **k** + 2
  - **j** ← **Φ**(*j*, *j*)

**Insert Φ()**

- **i** ← 1
- **j** ← 1
- **k** ← 0

- **k** < 100?
  - **j** < 20?
    - **return j**
  - **j** ← **i**
  - **k** ← **k** + 1
  - **j** ← **k** + 1
  - **k** ← **k** + 2
  - **j** ← **Φ**(*j*, *j*)

**Insert Φ()**

- **i** ← 1
- **j** ← 1
- **k** ← 0

- **k** < 100?
  - **j** < 20?
    - **return j**
  - **j** ← **i**
  - **k** ← **k** + 1
  - **j** ← **k** + 1
  - **k** ← **k** + 2
  - **j** ← **Φ**(*j*, *j*)

**Insert Φ()**

- **i** ← 1
- **j** ← 1
- **k** ← 0

- **k** < 100?
  - **j** < 20?
    - **return j**
  - **j** ← **i**
  - **k** ← **k** + 1
  - **j** ← **k** + 1
  - **k** ← **k** + 2
  - **j** ← **Φ**(*j*, *j*)

**Insert Φ()**

- **i** ← 1
- **j** ← 1
- **k** ← 0

- **k** < 100?
  - **j** < 20?
    - **return j**
  - **j** ← **i**
  - **k** ← **k** + 1
  - **j** ← **k** + 1
  - **k** ← **k** + 2
  - **j** ← **Φ**(*j*, *j*)
\[ \begin{align*}
\text{Insert } \Phi() & \\
1 & i \leftarrow 1 \\
2 & j \leftarrow i \\
3 & k \leftarrow 0 \\
4 & \text{DFS } 1 \\
5 & \text{DFS } 2 \\
6 & \text{DFS } 3 \\
7 & \text{DFS } 4 \\
8 & \text{DFS } 5 \\
9 & \text{DFS } 6 \\
10 & \text{DFS } 7 \\
\end{align*} \]
Computing DF(n)

CompuMng the Dominance Frontier

The Dominance Frontier of a node x = 
\{ w | x dom pred(w) AND !(x sdom w) \}

compute-DF(n)
S = {}
foreach node y in succ(n)
  if idom(y) = n
    S = S U \{ y \}
  foreach child of n, c, in D-tree
    compute-DF(c)
    foreach w in DF(c)
      if ln dom w
        S = S U \{ w \}
       
DF(n) = S

SSA Properties

• Only 1 assignment per variable
• Definitions dominate uses