Lecture 14
SSA-Style Optimizations

(Slide content courtesy of Seth Goldstein.)

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

\[
\begin{align*}
\text{x} & \leftarrow 1 \\
\text{y} & \leftarrow \text{x} \\
\text{y} & \leftarrow 2 \\
\text{z} & \leftarrow \text{y} + \text{x} \\
\text{y}_2 & \leftarrow 2 \\
\text{y}_3 & \leftarrow \Phi(\text{y}_1, \text{y}_2) \\
\text{x}_1 & \leftarrow 1 \\
\text{y}_1 & \leftarrow \text{x}_1 \\
\text{z}_1 & \leftarrow \text{y}_3 + \text{x}_1
\end{align*}
\]

Review: Dominance Frontier and Path Convergence

Constant Propagation
- If "$v \leftarrow c$", replace all uses of $v$ with $c$
- If "$v \leftarrow \Phi(c, c, c)$", replace all uses of $v$ with $c$

$W$ $\leftarrow$ list of all defs
while !W.isEmpty {
  Stmt $S \leftarrow W$.removeOne
  if ({($S$ has form "$v \leftarrow c$") \|}$
    $S$ has form "$v \leftarrow \Phi(c, \ldots, c)$") then {
      delete $S$
      foreach stmt $U$ that uses $v$ {
        replace $v$ with $c$ in $U$
        $W$.add($U$)
      }
    }
  }
}
Other Optimizations with SSA

- Copy propagation
  - delete "x ← Φ(v,x,y)" and replace all x with y
  - delete "x ← y" and replace all x with y
- Constant Folding
  - (Also, constant conditions too!)

Constant Propagation

1. i ← 1
2. j ← 1
3. k ← 0

1. i₁ ← 1
2. j₁ ← 1
3. k₁ ← 0

2. k < 100?
3. j < 20?
4. return j
5. j ← i
6. k ← k + 1
7. j ← k
8. k ← k + 2

2. j₂ < 20?
3. j₂ < 100?
4. return j₂
5. j₂ ← i₂
6. k₂ ← k₂ + 1
7. j₂ ← k₂
8. k₂ ← k₂ + 2

Constant Propagation

1. i₁ ← 1
2. j₁ ← 1
3. k₁ ← 0

1. i₁ ← 1
2. j₁ ← 1
3. k₁ ← 0

2. j₂ < 20?
3. j₂ < 100?
4. return j₂
5. j₂ ← i₂
6. k₂ ← k₂ + 1
7. j₂ ← k₂
8. k₂ ← k₂ + 2

2. j₂ < 20?
3. j₂ < 100?
4. return j₂
Constant Propagation

\[ j_1 \leftarrow 1 \\
  j_2 \leftarrow 1 \\
  k_4 \leftarrow 0 \]

\[ j_2 \leftarrow \Phi(j_4, i) \\
  k_3 \leftarrow \Phi(k_4, i) \\
  k_3 \leftarrow 100? \]

\[ j_2 < 20? \text{ return } j_2 \]

\[ j_3 \leftarrow 1 \\
  k_3 \leftarrow k_2 + 1 \]

\[ j_4 \leftarrow \Phi(l, j_3) \\
  k_4 \leftarrow \Phi(k_3, k_5) \]

But, so what?

Conditional Constant Propagation

\[ j_1 \leftarrow 1 \\
  j_1 \leftarrow 1 \\
  k_1 \leftarrow 0 \]

\[ j_2 \leftarrow \Phi(j_4, 1) \\
  k_2 \leftarrow \Phi(k_4, 1) \\
  k_2 \leftarrow 100? \]

\[ j_2 < 20? \text{ return } j_2 \]

\[ j_3 \leftarrow 1 \\
  k_3 \leftarrow k_2 + 1 \\
  j_5 \leftarrow k_2 \\
  k_5 \leftarrow k_5 + 2 \]

\[ j_4 \leftarrow \Phi(l, j_5) \\
  k_4 \leftarrow \Phi(k_3, k_5) \]

Does block 6 ever execute?

- Simple CP can’t tell
- Conditional CP can tell:
  - Assumes blocks don’t execute until proven otherwise
  - Assumes values are constants until proven otherwise

Conditional Constant Propagation Algorithm

Keeps track of:

- Blocks
  - Assume unexecuted until proven otherwise
- Variables
  - Assume not executed (only with proof of assignments of a non-constant value do we assume not constant)

Lattice for representing variables:

```
    1 0 1 2
   / \ / \ / \\
 1   2 1   0   1
```

- not executed
- we have seen evidence that the variable has been assigned a constant with the value
- we have seen evidence that the variable can hold different values at different times
Conditional Constant Propagation

1. $i_1 \leftarrow 1$
2. $j_1 \leftarrow 1$
3. $k_1 \leftarrow 0$
4. $j_2 \leftarrow \Phi(j_1, 1)$
5. $k_2 \leftarrow \Phi(k_1, 0)$
6. $k_2 < 100?$
7. $j_2 < 20?$
8. return $j_2$
9. $j_3 \leftarrow 1$
10. $k_3 \leftarrow k_2 + 1$
11. $j_4 \leftarrow \Phi(1, j_3)$
12. $k_4 \leftarrow \Phi(k_3, k_5)$
13. $k_5 \leftarrow k_2 + 2$

Dead Code Elimination

$W \leftarrow \text{list of all defs}$
while !$W$.isEmpty {
    Stmt $S \leftarrow W$.removeOne
    if $|S$.users| != 0 then continue
    if $S$.hasSideEffects() then continue
    foreach def in $S$.operands.definers {
        def.users \leftarrow def.users - {$S$}
        if |def.users| == 0 then
            $W \leftarrow W$.UNION {def}
    }
    delete $S$
}

Example DCE

B0 $i \leftarrow 0$
1. $j \leftarrow 0$
2. $i \leftarrow i*2$
3. $j \leftarrow j+1$
4. $j < 10?$
5. B2 return $j$

Standard DCE leaves Zombies!
**Aggressive Dead Code Elimination**

Assume a statement is dead until proven otherwise.

`init:`

- mark as `live` all stmts that have `side-effects`:
  - `I/O`
  - stores into memory
  - returns
  - calls a function that MIGHT have side-effects

As we mark S live, insert S.operands.definers into W

while (|W| > 0) {
  S <− W.removeOne()
  if (S is live) continue;
  mark S live, insert S.operands.definers into W
}

**Example DCE**

```
B0  i₀ ← 0
    j₀ ← 0
B1  j₁ ← Φ(3₁, 2₁)
    i₁ ← Φ(1₀, 1₁)
    i₂ ← i₁ + 2
    j₂ ← j₁ + 1
    j₂ < 10?

B2  return j₁
```

**Problem!**

```
B0  i₀ ← 0
    j₀ ← 0
B1  j₁ ← Φ(3₀, 2₀)
    i₁ ← Φ(1₀, 1₁)
    i₂ ← i₁ + 2
    j₂ ← j₁ + 1
    j₂ < 10?

B2  return j₂
```

**Earlier CCP Example Revisited**

```
1  i₁ ← 1
   j₁ ← 1
   k₂ ← 0

2  j₂ ← Φ(3₁, 1)
   k₃ ← Φ(4₁, 1)
   k₄ < 100?

3  j₂ < 20?

5  j₃ ← 1
   k₃ ← k₂ + 1

6  j₃ ← k₂ + 2

7  j₄ ← Φ(1₂, 2₁)
   k₄ ← Φ(3₄, 2₄)
```

**Applying Dead Code Elimination to the Result of CCP**

```
After CCP

k₂ ← Φ(3₂, 0)
   k₂ < 100?

k₃ < k₂ + 1
return 1
```

```
After DCE

k₃ < k₂ + 1
return 1
```

Small problem.
Fixing DCE

if $S$ is live, then
  if $T$ determines if $S$ can execute, $T$ should be live

Aggressive Dead Code Elimination (Fixed Version)

Assume a statement is dead until proven otherwise.

init:
  mark as live all stmts that have side-effects:
  - I/O
  - stores into memory
  - returns
  - calls a function that MIGHT have side-effects
As we mark $S$ live, insert:
  - $S$.operands.definers into $W$
  - $S$.CD$^{-1}$ into $W$
while ($|W| > 0$) {
  $S$ <- W.removeOne()
  if ($S$ is live) continue;
  mark $S$ live, insert:
  - $S$.operands.definers into $W$
  - $S$.CD$^{-1}$ into $W$
}

Example DCE

Control Dependence

$Y$ is control-dependent on $X$ if
  • $X$ branches to $u$ and $v$
  • $\exists$ a path $u$→exit which does not go through $Y$
  • $\forall$ paths $v$→exit go through $Y$

i.e. $X$ can determine whether or not $Y$ is executed.
Finding the Control Dependence Graph

Y is control-dependent on X if
- X branches to u and v
- \exists a path u→exit which does not go through Y
- ∀ paths v→exit go through Y

i.e. X can determine whether or not Y is executed.

Example DCE

```
B0  i_0 \leftarrow 0
    j_0 \leftarrow 0
B1  j_1 \leftarrow \Phi(i_0, j_2)
    i_2 \leftarrow i_1 + 2
    j_2 \leftarrow j_1 + 1
    j_2 < 10?
B2  return j_2

B0  j_0 \leftarrow 0
B1  j_1 \leftarrow \Phi(i_0, j_2)
    j_2 \leftarrow j_1 + 1
    j_2 < 10?
B2  return j_2
```

Finding the Control Dependence Graph

Y is control-dependent on X if
- X branches to u and v
- \exists a path u→exit which does not go through Y
- ∀ paths v→exit go through Y

i.e. X can determine whether or not Y is executed.

Any ideas?

Dominance Frontier and Path Convergence

```
START
```

Finding the Control Dependence Graph

Y is control-dependent on X if
- X branches to u and v
- \exists a path u→exit which does not go through Y
- ∀ paths v→exit go through Y

i.e. X can determine whether or not Y is executed.
Finding the CDG

- Construct CFG
- Add entry node and exit node
- Add (entry,exit)
- Create $G'$, the reverse CFG
- Compute D-tree in $G'$ (post-dominators of $G$)
- Compute $DF_{G'}(y)$ for all $y \in G'$ (post-DF of $G$)
- Add $(x,y) \in G$ to CDG if $x \in DF_{G'}(y)$

CDG of example

exit

entry

exit

2

1

0

entry

2

1

0

entry: