From Last Lecture

• What is a Loop?

• Dominator Tree

• Natural Loops

• Back Edges
Finding Loops: Summary

• Define loops in graph theoretic terms
• Definitions and algorithms for:
  – Dominators
  – Back edges
  – Natural loops
Finding Back Edges

- **Depth-first spanning tree**
  - Edges traversed in a depth-first search of the flow graph form a depth-first spanning tree

- **Categorizing edges in graph**
  - Advancing (A) edges: from ancestor to proper descendant
  - Cross (C) edges: from right to left
  - Retreating (R) edges: from descendant to ancestor (not necessarily proper)
Back Edges

• Definition
  – **Back edge**: t->h, h dominates t

• Relationships between graph edges and back edges

• Algorithm
  – Perform a depth first search
  – For each retreating edge t->h, check if h is in t’s dominator list

• Most programs (all structured code, and most GOTO programs) have **reducible** flow graphs
  – retreating edges = back edges
Examples

All theretreating edges are back edges
Constructing Natural Loops

- The **natural loop of a back edge** is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.

- **Algorithm**
  - delete $h$ from the flow graph
  - find those nodes that can reach $t$
    (those nodes plus $h$ form the natural loop of $t \rightarrow h$)
Inner Loops

• If two loops do not have the same header:
  – they are either disjoint, or
  – one is entirely contained (nested within) the other
    • inner loop: one that contains no other loop.

• If two loops share the same header:
  – Hard to tell which is the inner loop
  – Combine as one
Preheader

• Optimizations often require code to be executed once before the loop
• Create a preheader basic block for every loop
CSC D70: Compiler Optimization
Static Single Assignment (SSA)

Prof. Gennady Pekhimenko
University of Toronto
Winter 2020

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons
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

- 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

\[
\begin{align*}
X_1 &= A + 1 \\
Y &= X_1 + B \\
F &= 2 \\
F &= 3 \\
X_2 &= F + 7 \\
C &= X_2 + D
\end{align*}
\]
Definition-Use and Use-Definition Chains

- **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 $X$?
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, 
N def
M uses
⇒ O(NM) space and time

One solution: limit each variable to ONE definition site
DU and UD Chains Can Be Expensive (2)

foo(int i, int j) {
    ...
    switch (i) {
    case 0: x=3; break;
    case 1: x=1; break;
    case 2: x=6;
    case 3: x=7;
    default: x = 11;
    }
    \textit{x1 is one of the above x’s}
    switch (j) {
    case 0: y=x1+7;
    case 1: y=x1+4;
    case 2: y=x1-2;
    case 3: y=x1+1;
    default: y=x1+9;
    }
    ...
    \textbf{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 (reminiscent of Value Numbering):
  - Visit each instruction in program order:
    - LHS: assign to a *fresh version* of the variable
    - RHS: use the *most recent version* of each variable

```
a ← x + y
b ← a + x
a ← b + 2
c ← y + 1
a ← c + a
```

```
\[ a_1 \leftarrow x + y \]
\[ b_1 \leftarrow a_1 + x \]
\[ a_2 \leftarrow b_1 + 2 \]
\[ c_1 \leftarrow y + 1 \]
\[ a_3 \leftarrow c_1 + a_2 \]
```
What about Joins in the CFG?

c ← 12
if (i) {
    a ← x + y
    b ← a + x
} else {
    a ← b + 2
    c ← y + 1
}
a ← c + a

Use a notational fiction: a Φ function
Merging at Joins: the $\Phi$ function

- $c_1 \leftarrow 12$
- if (i)

- $a_1 \leftarrow x + y$
- $b_1 \leftarrow a_1 + x$
- $a_2 \leftarrow b + 2$
- $c_2 \leftarrow y + 1$

- $a_3 \leftarrow \Phi(a_1, a_2)$
- $c_3 \leftarrow \Phi(c_1, c_2)$
- $b_2 \leftarrow \Phi(b_1, ?)$
- $a_4 \leftarrow c_3 + a_3$
The Φ function

- Φ merges multiple definitions along multiple control paths into a single definition.

- At a basic block with \( p \) predecessors, there are \( p \) arguments to the Φ function.

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

- How do we choose which \( x_i \) to use?
  - We don’t really care!
  - If we care, use moves on each incoming edge
“Implementing” Φ

\[ c_1 \leftarrow 12 \]

\[
\text{if (i)}
\]

\[
a_1 \leftarrow x + y
\]
\[
b_1 \leftarrow a_1 + x
\]
\[
a_3 \leftarrow a_1
\]
\[
c_3 \leftarrow c_1
\]
\[
a_2 \leftarrow b + 2
\]
\[
c_2 \leftarrow y + 1
\]
\[
a_3 \leftarrow a_2
\]
\[
c_3 \leftarrow c_2
\]
\[
a_4 \leftarrow c_3 + a_3
\]

\[
\Phi(a_1, a_2)
\]
\[
\Phi(c_1, c_2)
\]
Trivial SSA

• Each assignment generates a fresh variable.
• At each join point insert Φ functions for all live variables.

Too many Φ functions inserted.
Minimal SSA

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

```
x ← 1
y ← x
y ← 2
z ← y + x

x₁ ← 1
y₁ ← x₁
y₂ ← 2
y₃ ← Φ(y₁, y₂)
z₁ ← y₃ + x₁
```
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_3 + 1
c_2 ← c_3 + b_2
a_2 ← b_2 * 2
if a_2 < N

return c_2

Notice use of $c_1$
When Do We Insert \( \Phi \)?

If there is a def of \( a \) in block 5, which nodes need a \( \Phi() \)?
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 is only common block along paths)
    - $z \notin P_{xq}$ or $z \notin P_{yr}$ where $P_{xz} = P_{xq} \rightarrow z$ and $P_{yz} = P_{yr} \rightarrow z$
      (at least one path reaches $Z$ for first time)
  - Entry block contains an implicit def of all vars
- Note: $v = \Phi(\ldots)$ is a def of $v$
Dominance Property of SSA

• In SSA, definitions dominate uses.
  – If $x_i$ is used in $x \leftarrow \Phi(..., x_i, ...)$, then $BB(x_i)$ dominates $i^{th}$ predecessor of $BB(PHI)$
  – If $x$ is used in $y \leftarrow ... x ...$, then $BB(x)$ dominates $BB(y)$

• We can use this for an efficient algorithm to convert to SSA
Dominance

x strictly dominates w (x sdom w) iff x dom w AND x ≠ w

If there is a def of a in block 5, which nodes need a Φ()?
Dominance Frontier

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

$x$ strictly dominates $w$ ($x \text{ sdom } w$) iff $x \text{ dom } w$ AND $x \neq w$
Dominance Frontier and Path Convergence

If there is a def of a in block 5, nodes in DF(5) need a $\Phi()$ for a
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 $\text{DominanceFrontier}($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 $\Phi()$ neccessary
Using Dominance Frontier to Place Φ()

foreach node n {
    foreach variable v defined in n {
        orig[n] U= {v}
        defsites[v] U= {n}
    }
}

foreach variable v {
    W = defsites[v]
    while W not empty {
        n = remove node from W
        foreach y in DF[n]
            if y \notin PHI[v] {
                insert “v ← Φ(v,v,...)” at top of y
                PHI[v] = PHI[v] U {y}
                if v \notin orig[y]: W = W U {y}
            }
    }
}
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

i ← 1
j ← 1
k ← 0

k < 100?

j < 20?

j ← i
k ← k + 1

j ← k
k ← k + 2

D-tree
Compute Dominance Frontiers

```
i ← 1
j ← 1
k ← 0
```

```
k < 100?
```

```
j < 20?
```

```
j ← i
k ← k + 1
```

```
j ← k
k ← k + 2
```

```
return j
```

DFs:
1. \{\}
2. \{2\}
3. \{2\}
4. \{\}
5. \{7\}
6. \{7\}
7. \{2\}
Insert $\Phi()$

### Code

```plaintext
i ← 1
j ← 1
k ← 0

k < 100?
  j < 20?
    j ← i
    k ← k + 1
  return j
  j ← k
  k ← k + 2
```

### Data Flow Sinks (DFs)

<table>
<thead>
<tr>
<th>i</th>
<th>DF{1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>DF{1,5,6}</td>
</tr>
<tr>
<td>k</td>
<td>DF{1,5,6}</td>
</tr>
</tbody>
</table>

### Original Definitions (orig[n])

<table>
<thead>
<tr>
<th>i</th>
<th>{1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>{1,5,6}</td>
</tr>
<tr>
<td>k</td>
<td>{1,5,6}</td>
</tr>
</tbody>
</table>

### Defining Sites (defsites[v])

<table>
<thead>
<tr>
<th>i</th>
<th>{1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>{1,5,6}</td>
</tr>
<tr>
<td>k</td>
<td>{1,5,6}</td>
</tr>
</tbody>
</table>
Insert $\Phi()$

```
i ← 1
j ← 1
k ← 0

k < 100?
j < 20?
return j

j ← i
k ← k + 1
j ← k
k ← k + 2

j ← $\Phi(j,j)$
```

DFs

<table>
<thead>
<tr>
<th>DFs</th>
<th>orig[n]</th>
<th>defsites[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 { }</td>
<td>1 { i,j,k}</td>
<td></td>
</tr>
<tr>
<td>2 {2}</td>
<td>2 { }</td>
<td></td>
</tr>
<tr>
<td>3 {2}</td>
<td>3 { }</td>
<td></td>
</tr>
<tr>
<td>4 { }</td>
<td>4 { }</td>
<td></td>
</tr>
<tr>
<td>5 {7}</td>
<td>5 {j,k}</td>
<td></td>
</tr>
<tr>
<td>6 {7}</td>
<td>6 {j,k}</td>
<td></td>
</tr>
<tr>
<td>7 {2}</td>
<td>7 { }</td>
<td></td>
</tr>
</tbody>
</table>

var j: W={1,5,6}

DF{1}    DF{5}
\begin{align*}
i &\leftarrow 1 \\
j &\leftarrow 1 \\
k &\leftarrow 0
\end{align*}

\begin{align*}
j &\leftarrow \phi(j,j) \\
k &< 100? \\
j &< 20? \\
\text{return } j
\end{align*}

\begin{align*}
j &\leftarrow \phi(j,j) \\
j &\leftarrow i \\
k &\leftarrow k + 1 \\
j &\leftarrow k \\
k &\leftarrow k + 2 \\
j &\leftarrow \phi(j,j)
\end{align*}

\begin{align*}
\text{DFs} &:
\begin{array}{ll}
1 & \{\}
2 & \{2\}
3 & \{2\}
4 & \{\}
5 & \{7\}
6 & \{7\}
7 & \{2\}
\end{array}
\end{align*}

\begin{align*}
\text{orig}[n] &:
\begin{array}{ll}
1 & \{i,j,k\}
2 & \{\}
3 & \{\}
4 & \{\}
5 & \{j,k\}
6 & \{j,k\}
7 & \{\}
\end{array}
\end{align*}

\begin{align*}
\text{defsites[v]} &:
\begin{array}{ll}
i & \{1\}
\end{align*}

\text{var } j: \text{W=\{1,5,6,7\}}

\begin{align*}
\text{DF\{1\}} & \quad \text{DF\{5\}} & \quad \text{DF\{7\}}
\end{align*}
\[ \begin{align*}
  i & \leftarrow 1 \\
  j & \leftarrow 1 \\
  k & \leftarrow 0
\end{align*} \]

\[ j \leftarrow \Phi(j, j) \]

\[ k < 100? \]

\[ j < 20? \]

\[ \text{return } j \]

\[ j \leftarrow i \]

\[ k \leftarrow k + 1 \]

\[ j \leftarrow k \]

\[ k \leftarrow k + 2 \]

\[ j \leftarrow \Phi(j, j) \]

\[ \text{DFs} \]

\[ \text{orig[n]} \]

\[ \text{dfsites[v]} \]

\[ \text{var } j: W = \{1, 5, 6, 7\} \]

\[ \text{DF\{1\} } \quad \text{DF\{5\} } \quad \text{DF\{7\} } \quad \text{DF\{6\} } \]
\[ i \leftarrow 1 \\
j \leftarrow 1 \\
k \leftarrow 0 \]

\[ j \leftarrow \Phi(j, j) \\
k \leftarrow \Phi(k, k) \]

\[ k < 100? \]

\[ j < 20? \]

\[ j \leftarrow i \\
k \leftarrow k + 1 \]

\[ j \leftarrow k \\
k \leftarrow k + 2 \]

\[ j \leftarrow \Phi(j, j) \\
k \leftarrow \Phi(k, k) \]

DFs

<table>
<thead>
<tr>
<th>orig[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Def sites[v]

| i     | \{1\}     |
| j     | \{1,5,6\} |
| k     | \{1,5,6\} |

var k: W={1,5,6}
i_1 \leftarrow 1
j_1 \leftarrow 1
k \leftarrow 0

j_2 \leftarrow \Phi(j, j_1)
k \leftarrow \Phi(k, k)
k < 100?

j < 20?

j \leftarrow i_1
k \leftarrow k + 1

j \leftarrow k
k \leftarrow k + 2

j \leftarrow \Phi(j, j)
k \leftarrow \Phi(k, k)
Rename Vars

1. \( i_1 \leftarrow 1 \)
   \( j_1 \leftarrow 1 \)
   \( k_1 \leftarrow 0 \)

2. \( j_2 \leftarrow \phi(j_4, j_1) \)
   \( k_2 \leftarrow \phi(k_4, k_1) \)
   \( k_2 < 100? \)

3. \( j_2 < 20? \)

4. return \( j_2 \)

5. \( j_3 \leftarrow i_1 \)
   \( k_3 \leftarrow k_2 + 1 \)

6. \( j_5 \leftarrow k_2 \)
   \( k_5 \leftarrow k_2 + 2 \)

7. \( j_4 \leftarrow \phi(j_3, j_5) \)
   \( k_4 \leftarrow \phi(k_3, k_5) \)
Computing $DF(n)$

Diagram:

- $n$ is the root node.
- $n$ is in the domain of $a$.
- $n$ is in the domain of $b$.
- $n$ is not in the domain of $c$.

$n \text{ dom } a$
$n \text{ dom } b$
!$n \text{ dom } c$
Computing $DF(n)$

$n \text{ dom } a$
$n \text{ dom } b$
$n \text{ dom } c$

DF(a)
DF(b)
Computing the Dominance Frontier

compute-DF(n)

S = {} 

foreach node y in succ[n]
    if idom(y) ≠ n 
        S = S ∪ { y } 

dominance Frontier

foreach child of n, c, in D-tree
compute-DF(c)

foreach w in DF[c]
    if !n dom w 
        S = S ∪ { w }

DF[n] = S

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

• Only 1 assignment per variable

• Definitions dominate uses
Constant Propagation

- If “v ∈ c”, replace all uses of v with c
- If “v ∈ Φ(c,c,c)” (each input is the same constant), replace all uses of v with c

W □ list of all defs
while !W.isEmpty {
    Stmt S □ W.removeOne
    if ((S has form “v ∈ c”) || (S has form “v ∈ Φ(c,...,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(y,y,y)” and replace all x with y
  – delete “x ∈ y” and replace all x with y

• **Constant Folding**
  – (Also, constant conditions too!)
Constant Propagation

Convert to SSA Form

1
i ← 1
j ← 1
k ← 0

2
k < 100?

3
j < 20?

4
return j

5
j ← i
k ← k + 1

6
j ← k
k ← k + 2

7


1
i₁ ← 1
j₁ ← 1
k₁ ← 0

2
j₂ ← Φ(j₄, j₁)
k₂ ← Φ(k₄, k₁)
k₂ < 100?

3
j₃ ← i₁
k₃ ← k₂ + 1

4
return j₂

5
j₅ ← k₂
k₅ ← k₂ + 2

6
j₄ ← Φ(j₃, j₅)
k₄ ← Φ(k₃, k₅)
\begin{align*}
    &i_1 \leftarrow 1 \\
    &j_1 \leftarrow 1 \\
    &k_1 \leftarrow 0 \\
    &j_2 \leftarrow \Phi(j_4, j_1) \\
    &k_2 \leftarrow \Phi(k_4, k_1) \\
    &k_2 < 100? \\
    &j_2 < 20? \\
    &\text{return } j_2 \\
    &j_3 \leftarrow i_1 \\
    &k_3 \leftarrow k_2 + 1 \\
    &j_4 \leftarrow \Phi(j_3, j_5) \\
    &k_4 \leftarrow \Phi(k_3, k_5) \\
    &j_5 \leftarrow k_2 \\
    &k_5 \leftarrow k_2 + 2
\end{align*}
\[
\begin{align*}
&i_1 \leftarrow 1 \\
&j_1 \leftarrow 1 \\
&k_1 \leftarrow 0 \\
&j_2 \leftarrow \Phi(j_4, 1) \\
&k_2 \leftarrow \Phi(k_4, 0) \\
&k_2 < 100? \\
&j_2 < 20? \\
&j_3 \leftarrow 1 \\
&k_3 \leftarrow k_2 + 1 \\
&j_5 \leftarrow k_2 \\
&k_5 \leftarrow k_2 + 2 \\
&j_4 \leftarrow \Phi(j_3, j_5) \\
&k_4 \leftarrow \Phi(k_3, k_5)
\end{align*}
\]
Not a very exciting result (yet)
Conditional Constant Propagation

• Does block 6 ever execute?
• Simple Constant Propagation can’t tell
• But “Conditional Const. Prop.” can tell:
  • Assumes blocks don’t execute until proven otherwise
  • Assumes values are constants until proven otherwise

```
1
i_1 ← 1
j_1 ← 1
k_1 ← 0

2
j_2 ← \Phi(j_4, 1)
k_2 ← \Phi(k_4, 1)
k_2 < 100?

3
j_2 < 20?

4
return j_2

5
j_3 ← 1
k_3 ← k_2 + 1

6
j_5 ← k_2
k_5 ← k_2 + 2

7
j_4 ← \Phi(1, j_5)
k_4 ← \Phi(k_3, k_5)
```
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:**

  ![Lattice Diagram]

  - 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. $i_1 \leftarrow 1$
   $j_1 \leftarrow 1$
   $k_1 \leftarrow 0$

2. $j_2 \leftarrow \Phi(j_4, 1)$
   $k_2 \leftarrow \Phi(k_4, 0)$
   $k_2 < 100?$

3. $j_2 < 20?$

4. return $j_2$

5. $j_3 \leftarrow 1$
   $k_3 \leftarrow k_2 + 1$

6. $j_5 \leftarrow k_2$
   $k_5 \leftarrow k_2 + 2$

7. $j_4 \leftarrow \Phi(1, j_5)$
   $k_4 \leftarrow \Phi(k_3, k_5)$
\[ i_1 \leftarrow 1 \]
\[ j_1 \leftarrow 1 \]
\[ k_1 \leftarrow 0 \]

\[ j_2 \leftarrow \Phi(j_4, 1) \]
\[ k_2 \leftarrow \Phi(k_4, 0) \]
\[ k_2 < 100? \]

\[ j_2 < 20? \]

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

\[ j_4 \leftarrow \Phi(1, j_5) \]
\[ k_4 \leftarrow \Phi(k_3, k_5) \]

\[ j_5 \leftarrow k_2 \]
\[ k_5 \leftarrow k_2 + 2 \]

\[ \text{return } j_2 \]
Conditional Constant Propagation

\[
\begin{align*}
&i_1 \leftarrow 1 \\
&j_1 \leftarrow 1 \\
&k_1 \leftarrow 0 \\
&j_2 \leftarrow \Phi(j_4, 1) \\
&k_2 \leftarrow \Phi(k_4, 0) \\
&k_2 < 100? \\
&j_2 < 20? \\
&j_3 \leftarrow 1 \\
&k_3 \leftarrow k_2 + 1 \\
&j_4 \leftarrow \Phi(1, j_5) \\
&k_4 \leftarrow \Phi(k_3, k_5) \\
&j_5 \leftarrow k_2 \\
&k_5 \leftarrow k_2 + 2 \\
&k_2 \leftarrow \Phi(k_3, 0) \\
&k_2 < 100? \\
&j_2 < 20? \\
&k_3 \leftarrow k_2 + 1 \\
&\text{return 1}
\end{align*}
\]
CSC D70: Compiler Optimization
Static Single Assignment (SSA)

Prof. Gennady Pekhimenko
University of Toronto
Winter 2020

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons
Backup Slides