Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis (in parts 2, 3, and 4)
- Fundamentals of control-flow analysis
- Algorithms for two machine-independent optimizations
- SSA form and optimizations
Edges whose heads dominate their tails are called *back edges* \((a \rightarrow b : b = \text{head}, \ a = \text{tail})\).

Given a back edge \(n \rightarrow d\):

- The *natural loop* of the edge is \(d\) plus the set of nodes that can reach \(n\) without going through \(d\).
- \(d\) is the header of the loop.
  - A single entry point to the loop that dominates all nodes in the loop.
  - At least one path back to the header exists (so that the loop can be iterated).
/* The back edge under consideration is $n \rightarrow d$ */
{ stack = empty; loop = \{$d\$\};
 /* This ensures that we do not look at predecessors of $d$ */
 insert($n$);
 while (stack is not empty) do {
   pop($m$, stack);
   for each predecessor $p$ of $m$ do insert($p$);
 }
 }

procedure insert($m$) {
 if $m \notin$ loop then {
   loop = loop $\cup$ \{$m$\};
   push($m$, stack);
 }
 }
Dominators, Back Edges, and Natural Loops

Flow Graph

Dominator Tree

Back edges and their natural loops

<table>
<thead>
<tr>
<th>Back edge</th>
<th>Natural Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 → 4</td>
<td>{4, 5, 6, 7, 8, 10}</td>
</tr>
<tr>
<td>10 → 7</td>
<td>{7, 8, 10}</td>
</tr>
<tr>
<td>4 → 3</td>
<td>{3, 4, 5, 6, 7, 8, 10}</td>
</tr>
<tr>
<td>10 → 3</td>
<td>{3, 4, 5, 6, 7, 8, 10}</td>
</tr>
<tr>
<td>11 → 1</td>
<td>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}</td>
</tr>
</tbody>
</table>
Dominators, Back Edges, and Natural Loops

Flow Graph

Dominator Tree

Back edges and their natural loops

<table>
<thead>
<tr>
<th></th>
<th>7 → 3</th>
<th>10 → 7</th>
<th>4 → 3</th>
<th>10 → 3</th>
<th>11 → 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>{3,4,5,6,7,8,10}</td>
<td>{7,8,10}</td>
<td>{3,4}</td>
<td>{3,4,5,6,7,8,10}</td>
<td>{1,2,3,4,5,6,7,8,9,10,11}</td>
</tr>
</tbody>
</table>

Adapted from the "Dragon Book", A-W, 1986
void dfs-num(int n) {
    mark node n “visited”;
    for each node s adjacent to n do {
        if s is “unvisited” {
            add edge n → s to dfs tree T;
            dfs-num(s);
        }
    }
    depth-first-num[n] = i ; i-- ;
}

// Main program
{ T = empty; mark all nodes of CFG as “unvisited”;
  i = number of nodes of CFG;
  dfs-num(n₀); // n₀ is the entry node of the CFG
}
Depth-First Numbering Example 1

Start

1

i = 0 read n

2

n <> 1

3

even(n)

4

print i

5

n = n/2

6

n = 3*n+1

7

Stop

8

i = i+1

9

B0

B1

B2

B3

B4

B5

B6

B7

B8

Y. N. Srikant  Control-Flow Analysis
Depth-First Numbering Example 2

Flow Graph

Dominator Tree

Nodes of the CFG show the DF-numbering

Adapted from the “Dragon Book”, A-W, 1986
Unless two loops have the same header, they are either disjoint or one is nested within the other.

Nesting is checked by testing whether the set of nodes of a loop A is a subset of the set of nodes of another loop B.

Similarly, two loops are disjoint if their sets of nodes are disjoint.

When two loops share a header, neither of these may hold (see next slide).

In such a case the two loops are combined and transformed as in the next slide.
Inner Loops and Loops with the same header

Adapted from the "Dragon Book", A-W, 1986

<table>
<thead>
<tr>
<th>C→A</th>
<th>D→A</th>
<th>E→A</th>
</tr>
</thead>
<tbody>
<tr>
<td>{A,B,C}</td>
<td>{A,B,D}</td>
<td>{A,B,C, D,E}</td>
</tr>
</tbody>
</table>

E is a dummy node

Back edges and their natural loops

<table>
<thead>
<tr>
<th>7→3</th>
<th>10→7</th>
<th>4→3</th>
<th>10→3</th>
<th>11→1</th>
</tr>
</thead>
<tbody>
<tr>
<td>{3,4,5,6, 7,8,10}</td>
<td>{7,8,10}</td>
<td>{3,4}</td>
<td>{3,4,5,6, 7,8,10}</td>
<td>{1,2,3,4,5, 6,7,8,9,10,11}</td>
</tr>
</tbody>
</table>
Preheader

Y.N. Srikant

Control-Flow Analysis
Depth of a Flow Graph and Convergence of DFA

- Given a depth-first spanning tree of a CFG, the largest number of retreating edges on any cycle-free path is the depth of the CFG
- The number of passes needed for convergence of the solution to a forward DFA problem is \(1 + \text{depth of CFG}\)
- One more pass is needed to determine no change, and hence the bound is actually \(2 + \text{depth of CFG}\)
- This bound can be actually met if we traverse the CFG using the depth-first numbering of the nodes
- For a backward DFA, the same bound holds, but we must consider the reverse of the depth-first numbering of nodes
- Any other order will still produce the correct solution, but the number of passes may be more
Depth of a CFG - Example 1

Flow Graph

Dominator Tree

Adapted from the “Dragon Book”, A-W, 1986

Nodes of the CFG show the DF-numbering

Depth of the CFG = 2 (10-7-3)
Depth of a CFG - Example 2

Depth of the CFG = 3 (10-7-4-3)

Flow Graph

Adapted from the “Dragon Book”, A-W, 1986
Algorithms for Machine-Independent Optimizations

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Outline of the Lecture

- Global common sub-expression elimination
- Copy propagation
- Simple constant propagation
- Loop invariant code motion
Elimination of Global Common Sub-expressions

- Needs available expression information
- For every $s : x := y + z$, such that $y + z$ is available at the beginning of $s$’ block, and neither $y$ nor $z$ is defined prior to $s$ in that block, do the following
  1. Search backwards from $s$’ block in the flow graph, and find first block in which $y + z$ is evaluated. We need not go through any block that evaluates $y + z$.
  2. Create a new variable $u$ and replace each statement $w := y + z$ found in the above step by the code segment \{ $u := y + z$; $w := u$ \}, and replace $s$ by $x := u$
  3. Repeat 1 and 2 above for every predecessor block of $s$’ block
- Repeated application of GCSE may be needed to catch “deep” CSE
GCSE Conceptual Example

Demonstrating the need for repeated application of GCSE
Copy Propagation

- Eliminate copy statements of the form $s: x := y$, by substituting $y$ for $x$ in all uses of $x$ reached by this copy.

Conditions to be checked

1. u-d chain of use $u$ of $x$ must consist of $s$ only. Then, $s$ is the only definition of $x$ reaching $u$.

2. On every path from $s$ to $u$, including paths that go through $u$ several times (but do not go through $s$ a second time), there are no assignments to $y$. This ensures that the copy is valid.

The second condition above is checked by using information obtained by a new data-flow analysis problem.

- $c_{\text{gen}}[B]$ is the set of all copy statements, $s: x := y$ in $B$, such that there are no subsequent assignments to either $x$ or $y$ within $B$, after $s$.

- $c_{\text{kill}}[B]$ is the set of all copy statements, $s: x := y$, $s$ not in $B$, such that either $x$ or $y$ is assigned a value in $B$.

Let $U$ be the universal set of all copy statements in the program.
Copy Propagation - The Data-flow Equations

- \( c_{\text{in}}[B] \) is the set of all copy statements, \( x := y \) reaching the beginning of \( B \) along every path such that there are no assignments to either \( x \) or \( y \) following the last occurrence of \( x := y \) on the path.

- \( c_{\text{out}}[B] \) is the set of all copy statements, \( x := y \) reaching the end of \( B \) along every path such that there are no assignments to either \( x \) or \( y \) following the last occurrence of \( x := y \) on the path.

\[
c_{\text{in}}[B] = \bigcap_{P \text{ is a predecessor of } B} c_{\text{out}}[P], \text{ } B \text{ not initial}
\]

\[
c_{\text{out}}[B] = c_{\text{gen}}[B] \cup (c_{\text{in}}[B] - c_{\text{kill}}[B])
\]

\[
c_{\text{in}}[B_1] = \phi, \text{ where } B_1 \text{ is the initial block}
\]

\[
c_{\text{out}}[B] = U - c_{\text{kill}}[B], \text{ for all } B \neq B_1 \text{ (initialization only)}
\]

Y.N. Srikant | Machine-Independent Optimizations
Algorithm for Copy Propagation

For each copy, $s : x := y$, do the following

1. Using the $du - chain$, determine those uses of $x$ that are reached by $s$.

2. For each use $u$ of $x$ found in (1) above, check that
   - (i) u-d chain of $u$ consists of $s$ only
     - This implies that $s$ is the only definition of $x$ that reaches this block.
   - (ii) $s$ is in $c_{in}[B]$, where $B$ is the block to which $u$ belongs.
     - This ensures that no definitions of $x$ or $y$ appear on this path from $s$ to $B$.
   - (iii) no definitions $x$ or $y$ occur within $B$ prior to $u$ found in (1) above.

3. If $s$ meets the conditions above, then remove $s$ and replace all uses of $x$ found in (1) above by $y$. 

Y.N. Srikant

Machine-Independent Optimizations
Copy Propagation Example 1

C_in[B2] = \{s1, s2\}
C_out[B2] = \{s2, s4\}

B1
s1: x:=y
s2: p:=q

C_in[B1] = \Phi
C_out[B1] = \{s1, s2\}

B2
s3: c:=a+b
s4: x:=z

B3
s5: x:=z

C_in[B3] = \{s1, s2\}
C_out[B3] = \{s2, s5\}

B4
s6: k:=x+6

C_in[B4] = \{s2, s5\}
C_out[B4] = \{s2, s5\}

B5
s7: m:=x+9
s8: n:=p

C_in[B5] = \{s2\}
C_out[B5] = \{s2, s8\}

x in s6 can be replaced by z in s5

Adapted from
“The Dragon Book”
A-W 1986

x in s7 cannot be replaced by z in s4 or s5 (two different copies of z)

p in s8 can be replaced by q in s2 (s2 reaches B5 thro' both the paths)
Copy Propagation on Running Example 1.1

B1: i = 0

B2: t1 = i > 1
   if !t1 goto B9
   true
   j = 0
   t2 = i - 1
   false

B3: stop
   B9

B4: t3 = j < t2
   if !t3 goto B8
   true
   t21 = t2
   i = t21
   goto B2
   false

B5: t4 = 4 * j
   t5 = a[t4]
   t6 = j + 1
   t7 = 4 * t6
   t8 = a[t7]
   t9 = t5 > t8
   if !t9 goto B7
   true

B6: t10 = t4
   t11 = a[t10]
   temp = t11
   t12 = t4
   t13 = a + t12
   t14 = t6
   t15 = 4 * t14
   t16 = a[t15]
   *t13 = t16
   t17 = t6
   t18 = 4 * t17
   t19 = a + t18
   *t19 = temp

B7: t20 = t6
   j = t20
   goto B4

Y.N. Srikant  Machine-Independent Optimizations
Copy Propagation on Running Example 1.2

Y. N. Srikant

Machine-Independent Optimizations
GCSE and Copy Propagation on Running Example 1.2

```
B1:  i = 0

B2:
  t1 = i>1
  if !t1 goto B9

B3:
  j = 0
  t2 = i-1

B4:
  t3 = j<t2
  if !t3 goto B8

B5:
  t4 = 4*j
  t5 = a[t4]
  t6 = j+1
  t7 = 4 * t6
  t8 = a[t7]
  t9 = t5 > t8
  if !t9 goto B7

B6:
  t11 = a[t4]
  temp = t11
  t13 = a + t4
  t16 = a[t7]
  *t13 = t16
  t19 = a + t7
  *t19 = temp

B7:
  j = t6
  goto B4
```

Y.N. Srikant  Machine-Independent Optimizations
Simple Constant Propagation

\[
\begin{align*}
\text{Stmtpile} &= \{ S \mid S \text{ is a statement in the program} \} \\
\text{while} \text{ Stmtpile is not empty} \{ \\
S &= \text{remove}(\text{Stmtpile}); \\
\text{if} \ S \text{ is of the form } x = c \text{ for some constant } c \\
&\quad \text{for all statements } T \text{ in the } du-chain \text{ of } x \text{ do} \\
&\quad \text{if usage of } x \text{ in } T \text{ is reachable only by } S \\
&\quad \quad \{ \text{ substitute } c \text{ for } x \text{ in } T; \text{ simplify } T \} \\
&\quad \text{Stmtpile} = \text{Stmtpile} \cup \{ T \}
\end{align*}
\]

Note: If all usages of \( x \) are replaced by \( c \), then \( x = c \) becomes dead code and a separate dead code elimination pass will remove it.
Simple Constant Propagation Example

\[ d1: \ x = 7 \]

\[ u1 \text{ is reached by only } d1. \text{ Hence } x \text{ in } u1 \text{ can be replaced by value of } x \text{ in } d1 \]

\[ u1 \text{ and } u2 \text{ are usages of def } d1 \]

\[ = x + 6 \ (u1) \]

\[ d2: \ x = 9 \]

\[ = x + 3 \ (u2) \]

\[ u2 \text{ is reached by both } d1 \text{ and } d2. \text{ Hence } x \text{ in } u2 \text{ cannot be replaced by either value of } x \]