Lecture 10: Data Flow Analysis II
Review: Compiler Middle End (Optimizer)

- Front end works from the syntax of the source code
- Rest of the compiler works from intermediate representation (IR)
  - Analyzes IR to learn about the code
  - Transforms IR to improve final code
  - Examples
    1. Parallelization
    2. Redundancy elimination
    3. Data flow analysis (DFA) framework
    4. Single static assignment (SSA)
Review: Compiler Middle End (Optimizer)

- Front end works from the syntax of the source code
- Rest of the compiler works from intermediate representation (IR)
  - Analyzes IR to learn about the code
  - Transforms IR to improve final code
  - Examples
    1. Parallelization
    2. Redundancy elimination
    3. Data flow analysis (DFA) framework
    4. Single static assignment (SSA)
An expression $x+y$ is redundant if and only if, along every path from the procedure’s entry, it has been evaluated, and its constituent subexpressions ($x$ & $y$) have not been re-defined.

If the compiler can prove that an expression is redundant

- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to the problem

- Proving that $x+y$ is redundant
- Rewriting the code to eliminate the redundant evaluation

One single-pass, local technique to accomplish both is called value numbering (VN)
Review

**Redundancy Elimination**

- Local Value Numbering (LVN)
- Superlocal Value Numbering (SVN)
- Dominator Value Numbering (DVN)
- Global Common Subexpression Elimination (GCSE)
Review: Local Value Numbering

Local Value Numbering (LVN)
Find redundant operation within a block

LVN finds 2 redundant ops
LVN misses 8 redundant ops!
Review: Local Value Numbering

• The Algorithm

For each operation $o$ in the block
1. Get value numbers for the operands from a hash lookup
2. Hash <$\text{operator}, \text{VN}(o_1), \text{VN}(o_2)$> to get a value number for $o$
3. If $o$ already had a value number, replace $o$ with a reference
4. If $o_1$ & $o_2$ are constant, evaluate it & use a “load immediate”

Looks at operand’s value number, not its name.
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundant operation within a block

Superlocal Value Numbering (SVN)
Find redundant operation within a EBB
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
- Find redundant operation within a block

Superlocal Value Numbering (SVN)
- Find redundant operation within a EBB

An Extended Basic Block (EBB)
- Set of blocks $B_1, B_2, \ldots, B_n$
- $B_1$ is either the entry node to the procedure, or it has $> 1$ pred.
- All other $B_i$ have 1 pred. & that pred. is in the EBB

Three EBBs in this CFG
1. $\{ A, B, C, D, E \}$
2. $\{ F \}$
3. $\{ G \}$
Review: Superlocal Value Numbering

The SVN Algorithm

1. Identify EBBs
2. In depth-first order over an EBB, starting with the head of the EBB, $p_0$
   (I) Apply LVN to $p_i$
   (II) Invoke SVN on each of $p_i$’s EBB successors
     • When going from $p_i$ to its EBB successor $p_j$, extend the symbol table with a new scope for $p_j$, apply LVN to $p_j$, & process $p_j$’s EBB successors
     • When going from $p_j$ to its EBB predecessor $p_i$, discard the scope for $p_j$

Use a scoped table & the right name space
Local Value Numbering (LVN)
Find redundant operation within a block

Superlocal Value Numbering (SVN)
Find redundant operation within a EBB

An Extended Basic Block (EBB)
Set of blocks $B_1, B_2, \ldots, B_n$
- $B_1$ is either the entry node to the procedure, or it has $> 1$ pred.
- All other $B_i$ have 1 pred. & that pred. is in the EBB

Three EBBs in this CFG
1. $\{ A, B, C, D, E \}$
2. $\{ F \}$
3. $\{ G \}$
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

SVN finds 5 redundant ops.
SVN misses 5 redundant ops!
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree
An expression \( x+y \) is **redundant** if and only if, **along every path from** the procedure’s entry, it has been evaluated, and its constituent subexpressions (\( x \) & \( y \)) have **not** been re-defined.

**We need to understand the properties of those paths**
- Specifically, what blocks occur on any path from entry to \( p \)
- If we can find those blocks, we can use their hash tables

**For example:**
- \( A \) is on every path to any other node; and \( C \) is on every path to \( D, E, \) and \( F. \)
- Only \( A \) is on every path to \( G. \)
Review: Dominance in a Control Flow Graph

Definition

*In a flow graph, x dominates y if and only if every path from the entry node of the control-flow graph to y includes x*

- By definition, x dominates x itself
- We associate a DOM set with each node
- DOM(x) contains the set of nodes that dominate x
- |DOM(x)| ≥ 1

<table>
<thead>
<tr>
<th>Block</th>
<th>DOM</th>
<th>IDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>A,B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A,C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>A,C,D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>A,C,E</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>A,C,F</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>A,G</td>
<td>A</td>
</tr>
</tbody>
</table>
Immediate dominator

- For any node $x$, there must be a $y$ in $\text{DOM}(x)$ closest to $x$
- We call this $y$ the immediate dominator of $x$
  - $x$ cannot be its own immediate dominator, unless $x$ is $n_0$
- As a matter of notation, we write this as $\text{IDOM}(x)$

<table>
<thead>
<tr>
<th>Block</th>
<th>DOM</th>
<th>IDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>A,B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A,C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>A,C,D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>A,C,E</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>A,C,F</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>A,G</td>
<td>A</td>
</tr>
</tbody>
</table>
Computing Dominators

To use dominance information, we need to compute DOM sets

- A node \( n \) dominates node \( m \) iff \( n \) is on every path from \( n_0 \) to \( m \)
  - Every node dominates itself, by definition
  - \( n \)'s immediate dominator is the node in \( \text{DOM}(n) \) that is closest to \( n \) in the graph

\[ \text{IDOM}(n) \neq n, \text{ unless } n \text{ is } n_0, \text{ by convention.} \]
Computing Dominators

Computing DOM

• We formulate the computation of DOM sets as a data-flow analysis (DFA) problem
• Simultaneous equations over sets associated with the nodes in the CFG

Equations relate the set value at a node n to those of its predecessors and successors in the CFG

Initially, \( \text{DOM}(n) = N, \forall n \neq n_0 \).

1. \( \text{DOM}(n_0) = \{ n_0 \} \)
2. \( \text{DOM}(n) = \{ n \} \cup (\cap_{p \in \text{preds}(n)} \text{DOM}(p)) \)
Computing Dominators

Initially, $\text{DOM}(n) = N, \forall n \neq n_0$.

1. $\text{DOM}(n_0) = \{ n_0 \}$
2. $\text{DOM}(n) = \{ n \} \cup (\cap_{p \text{ preds}(n)} \text{DOM}(p))$

Example:

$\text{DOM}(B_7) = \{ B_7 \} \cup \{ \text{DOM}(B_6) \cap \text{DOM}(B_8) \}$
Computing Dominators

A “fixed-point” data flow analysis (DFA) solution:

- Build a control-flow graph to identify the blocks & their predecessors
- Initialize $\text{DOM}(n)$ appropriately, for all $n$ in the CFG
  Set $\text{DOM}(n)$ to $N$, for all $n \neq n_0$
  Set $\text{DOM}(n_0)$ to $\{ n_0 \}$
- Repeatedly apply the following equation until the sets stop changing

$$\text{DOM}(n) = \{ n \} \cup (\bigcap_{p \in \text{preds}(n)} \text{DOM}(p))$$

<table>
<thead>
<tr>
<th>DOM($n_0$)</th>
<th>${ n_0 }$</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $x \leftarrow n_1$ to $n_n$</td>
<td></td>
</tr>
<tr>
<td>$\text{DOM}(x)$</td>
<td>${ \text{all nodes in graph, } N }$</td>
</tr>
<tr>
<td>change</td>
<td>$\leftarrow$ true</td>
</tr>
<tr>
<td>while (change)</td>
<td></td>
</tr>
<tr>
<td>change</td>
<td>$\leftarrow$ false</td>
</tr>
<tr>
<td>for $x \leftarrow n_0$ to $n_n$</td>
<td></td>
</tr>
<tr>
<td>TEMP</td>
<td>$\leftarrow { x } \cup \bigcap_{y \in \text{preds}(x)} \text{DOM}(y)$</td>
</tr>
<tr>
<td>if $\text{DOM}(x) \neq \text{TEMP}$ then</td>
<td></td>
</tr>
<tr>
<td>change</td>
<td>$\leftarrow$ true</td>
</tr>
<tr>
<td>$\text{DOM}(x)$</td>
<td>$\leftarrow \text{TEMP}$</td>
</tr>
</tbody>
</table>
Computing Dominators

Initially,
\[ \text{DOM}(n) = N, \ \forall \ n \neq n_0. \]
\[ \text{DOM}(n_0) = \{ n_0 \} \]

Data Flow Equation:
\[ \text{DOM}(n) = \{ n \} \cup (\cap_p \text{preds}(n) \ \text{DOM}(p)) \]

<table>
<thead>
<tr>
<th></th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
<th>B8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0}</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>{0}</td>
<td>{0,1}</td>
<td>{0,1,2}</td>
<td>{0,1,2,3}</td>
<td>{0,1,2,3,4}</td>
<td>{0,1,5}</td>
<td>{0,1,5,6}</td>
<td>{0,1,5,7}</td>
<td>{0,1,5,8}</td>
</tr>
<tr>
<td>2</td>
<td>{0}</td>
<td>{0,1}</td>
<td>{0,1,2}</td>
<td>{0,1,3}</td>
<td>{0,1,3,4}</td>
<td>{0,1,5}</td>
<td>{0,1,5,6}</td>
<td>{0,1,5,7}</td>
<td>{0,1,5,8}</td>
</tr>
<tr>
<td>3</td>
<td>{0}</td>
<td>{0,1}</td>
<td>{0,1,2}</td>
<td>{0,1,3}</td>
<td>{0,1,3,4}</td>
<td>{0,1,5}</td>
<td>{0,1,5,6}</td>
<td>{0,1,5,7}</td>
<td>{0,1,5,8}</td>
</tr>
</tbody>
</table>
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

1. Find blocks $DOM(p)$ occur on any path from entry to $p$.
2. Use hash table of $DOM(p)$ to remove redundancy
Two principles
• Each name is defined by exactly one operation
• Each operand refers to exactly one definition

To reconcile these principles with real code
• Insert $\phi$-functions at merge points to reconcile name space
• Add subscripts to variable names for uniqueness
Static Single Assignment (SSA)

Original Form

SSA Form

\[
\begin{align*}
A & : m \leftarrow a + b \\
 & n \leftarrow a + b \\
B & : p \leftarrow c + d \\
 & r \leftarrow c + d \\
C & : q \leftarrow a + b \\
 & s \leftarrow a + b \\
 & t \leftarrow c + d \\
 & u \leftarrow e + f \\
D & : e \leftarrow b + 18 \\
E & : e \leftarrow a + 17 \\
 & s \leftarrow a + b \\
 & t \leftarrow c + d \\
 & u \leftarrow e + f \\
F & : y \leftarrow a + b \\
 & z \leftarrow c + d \\
G & : v \leftarrow a + b \\
 & w \leftarrow c + d \\
 & x \leftarrow e + f \\
\end{align*}
\]
Use Dominator for Superlocal Value Numbering

SVN did not help with blocks F or G

• Multiple predecessors
• Must decide what facts hold in F and in G
• Can look at F’s or G’s DOM sets
• Or use table from IDOM(x) to start x
  - Use C for F and A for G
  - Imposes a DOM-tree application order

Dominator Tree

```
A
B          C          G
D          E          F
```

```
m_0 ← a + b
n_0 ← a + b
```

```
q_0 ← a + b
r_1 ← c + d
```

```
p_0 ← c + d
r_0 ← c + d
e_0 ← b + 18
s_0 ← a + b
u_0 ← e + f
```

```
e_1 ← a + 17
t_0 ← c + d
u_1 ← e + f
```

```
r_2 ← \varphi(r_0,r_1)
y_0 ← a + b
z_0 ← c + d
e_3 ← \varphi(e_0,e_1)
u_2 ← \varphi(u_0,u_1)
v_0 ← a + b
w_0 ← c + d
x_0 ← e + f
```
Use Dominator for Superlocal Value Numbering

SVN did not help with blocks F or G

- Multiple predecessors
- Must decide what facts hold in F and in G
- Can look at F’s or G’s DOM sets
- Or use table from IDOM(x) to start x
  - Use C for F and A for G
  - Imposes a DOM-tree application order

Dominator Tree
Use Dominator for Superlocal Value Numbering

SVN did not help with blocks F or G
- Multiple predecessors
- Must decide what facts hold in F and in G
- Can look at F’s or G’s DOM sets
- Or use table from IDOM(x) to start x
  - Use C for F and A for G
  - Imposes a DOM-tree application order

Dominator Tree

![Dominator Tree Diagram]
Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

DVN finds 8 redundant ops.
DVN misses 2 redundant ops!

Another shortcoming of DVN
No loop-carried redundancies
Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

Global Common Subexpression Elimination (GCSE)
Find redundancy in all basic blocks
Global Redundancy Elimination

**Availability definition:**

An expression is *available* on entry to block $b$ if and only if, along every path from $n_0$ to $b$, it has been evaluated and none of its constituent subexpressions has been re-defined.
Data Flow Analysis for Expression Availability

Annotate each block $b$ with a set $\text{AVAIL}(b)$

$x+y \in \text{AVAIL}(b)$ iff $x+y$ is available at the entry to block $b$

$$\text{AVAIL}(b) = \cap_{x \in \text{predecessors}(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) - \text{EXPRKILL}(x)))$$

$\text{DEEXPR}(x)$: the set of expressions defined in block $x$ and not killed before the end of block $x$  
**(downward exposed expressions)**

$\text{EXPRKILL}(x)$: the set of expressions killed in block $x$  
**(killed expressions)**

Initial Conditions:

$\text{AVAIL}(n_0) = \emptyset$

$\text{AVAIL}(x) = \{ \text{all expressions} \}$

- $a = b + c$
- $f = a$
- $u = f + e$
- $l = b + u$

$\text{EXPRKILL}: \{ f + e \}$

$\text{DEExpr}: \{ b + c \}$

$\text{AVAIL}: \{ b + c, b + u \}$
Computing Availability

To compute AVAIL sets:

1. Build a control-flow graph (CFG)
2. For each block $b$ in the CFG, compute $\text{DEEXPR}$ and $\text{EXPRKILL}$
3. Solve the equations for AVAIL
   I. Use a data-flow solver (e.g., round-robin iterative solver)
   II. The AVAIL equations are well-behaved (unique fixed point, fast convergence)

$$
\text{AVAIL}(b) = \cap_{x \in \text{predecessors}(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) - \text{EXPRKILL}(x)))
$$
Example

\[
\begin{align*}
A & : m \leftarrow a + b \\
& \quad n \leftarrow a + b \\
B & : p \leftarrow c + d \\
& \quad r \leftarrow c + d \\
C & : q \leftarrow a + b \\
& \quad r \leftarrow c + d \\
D & : e \leftarrow b + 18 \\
& \quad s \leftarrow a + b \\
& \quad u \leftarrow e + f \\
E & : e \leftarrow a + 17 \\
& \quad t \leftarrow c + d \\
& \quad u \leftarrow e + f \\
F & : v \leftarrow a + b \\
& \quad w \leftarrow c + d \\
& \quad x \leftarrow e + f \\
G & : y \leftarrow a + b \\
& \quad z \leftarrow c + d \\
\end{align*}
\]

**AVAIL(A)**

- $\varnothing$

**AVAIL(B)**

- $\{a+b\} \cup (\varnothing \cap \text{all})$
- $\{a+b\}$

**AVAIL(C)**

- $\{a+b\}$

**AVAIL(D)**

- $\{a+b, c+d\} \cup (\{a+b, c+d\} \cap \text{all})$
- $\{a+b, c+d\}$

**AVAIL(E)**

- $\{a+b, c+d\}$

**AVAIL(F)**

- $\{b+18, a+b, e+f\} \cup (\{a+b, c+d\} \cap \{\text{all} - e+f\})$
- $\{a+b, c+d, e+f\}$

**AVAIL(G)**

- $\{c+d\} \cup (\{a+b\} \cap \text{all})$
- $\{a+b, c+d, e+f\} \cup (\{a+b, c+d, e+f\} \cap \text{all})$
- $\{a+b, c+d\}$

<table>
<thead>
<tr>
<th>DEExpr</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b+c+d</td>
<td>a+b</td>
<td>a+b+c+d</td>
<td>b+18, a+b, e+f</td>
<td>a+b, c+d, e+f</td>
<td>a+b, c+d, e+f</td>
<td>a+b, c+d</td>
<td>a+b, c+d</td>
</tr>
<tr>
<td>ExprKill</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>e+f</td>
<td>e+f</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>
Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

Global Common Subexpression Elimination (GCSE)
Find redundancy in all basic blocks

AVAIL(F) = \{a+b, c+d, e+f\}
AVAIL(G) = \{a+b, c+d\}
Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

Global Common Subexpression Elimination (GCSE)
Find redundancy in all basic blocks

GCSE finds 10 redundant ops.
GCSE misses 0 redundant ops!
Summary

Redundancy Elimination

- **Local Value Numbering (LVN)**
  - Finds redundancy, constants, & algebraic identities in a block
- **Superlocal Value Numbering (SVN)**
  - Extends local value numbering to EBBs
  - Use SSA-like name space to simplify bookkeeping
- **Dominator Value Numbering (DVN)**
  - Extends scope to “almost” global (no back edges)
  - Use dominance information to handle join points in CFG
- **Global Common Subexpression Elimination (GCSE)**
  - Calculate available expression at every basic block
  - Use data-flow analysis to calculate available expressions
# Comparison of Redundancy Elimination Techniques

<table>
<thead>
<tr>
<th>Name</th>
<th>Scope</th>
<th>Operates On</th>
<th>Basis of Identity</th>
<th>Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVN</td>
<td>Local</td>
<td>blocks</td>
<td>value</td>
<td>no</td>
</tr>
<tr>
<td>SVN</td>
<td>Superlocal</td>
<td>EBBs</td>
<td>value</td>
<td>no</td>
</tr>
<tr>
<td>DVN</td>
<td>Regional</td>
<td>dom. Tree</td>
<td>value</td>
<td>no</td>
</tr>
<tr>
<td>GCSE</td>
<td>Global</td>
<td>CFG</td>
<td>lexical</td>
<td>yes</td>
</tr>
</tbody>
</table>
Review: Compiler Middle End (Optimizer)

- Front end works from the syntax of the source code
- Rest of the compiler works from intermediate representation (IR)
  - Analyzes IR to learn about the code
  - Transforms IR to improve final code
  - Examples
    1. Parallelization
    2. Redundancy elimination
    3. Data flow analysis (DFA) framework
    4. Single static assignment (SSA)
Review: Compiler Middle End (Optimizer)

- Front end works from the syntax of the source code
- Rest of the compiler works from intermediate representation (IR)
  - Analyzes IR to learn about the code
  - Transforms IR to improve final code
  - Examples
    1. Parallelization
    2. Redundancy elimination
    3. Data flow analysis (DFA) framework
    4. Single static assignment (SSA)
Live Variables

A variable $v$ is live at point $p$ if and only if there is a path from $p$ to a use of $v$ along which $v$ is not redefined.

Uses:
1. Global register allocation
2. Improve SSA construction: reduce # of $\phi$-functions
3. Detect references to uninitialized variables & defined but not used variables
4. Drive transformations: useless-store elimination
Equations for Live Variables

- **LIVEOUT**(n) contains the name of every variable that is live on exit from n
- **UEVAR**(n) contains the upward-exposed variables in n, i.e. those that are used in n before any redefinition in n
- **VARKILL**(n) contains all the variables that are defined in n
- Data Flow Equation (n_f is the exit node of the CFG)

\[
\text{LIVEOUT}(n_f) = \phi
\]
\[
\text{LIVEOUT}(n) = \bigcup_{m \in \text{succ}(n)} \text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)})
\]
Three Steps in Live Variable Analysis

1. Build a CFG
2. Gather the initial information for each block
3. Use an iterative fixed-point algorithm to propagate information around the CFG

\[
\text{LIVEOUT}(n_f) = \phi \\
\text{LIVEOUT}(n) = \bigcup_{m \in \text{succ}(n)} \text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \text{VARKILL}(m))
\]
// assume block b has k operations
// of form "x ← y op z"
for each block b
    Init(b)

Init(b)
    UEVar(b) ← Ø
    VarKill(b) ← Ø
    for i ← 1 to k
        if y ∉ VarKill(b)
            then add y to UEVar(b)
        if z ∉ VarKill(b)
            then add z to UEVar(b)
        add x to VarKill(b)
Terms

- **Postorder**: visits as many of a nodes’ children as possible before visiting the node
- **Reverse Postorder (RPO)**: visits as many of a nodes’ predecessors as possible before visiting the node
- Forward problem: RPO on CFG.
- Backward problem: Postorder on CFG or RPO on reverse CFG.
Comparison with AVAIL Analysis

- Common
  - Three steps
  - Fixed-point algorithm
- Differences
  - AVAIL: domain is a set of expressions
    LIVEOUT: domain is a set of variables
  - AVAIL: forward problem
    LIVEOUT: backward problem
  - AVAIL: intersection of all paths (all path problem)
    LIVEOUT: union of all paths (any path problem)
Reaching Definitions

A definition $d$ of some variable $v$ reaches operation $i$ iff $i$ uses the defined value of $v$ before $v$ is redefined.

- **REACHES**($n$): the set of definitions that reach the start of node $n$
- **DEDEF**($n$): the set of downward-exposed definitions in $n$
  
  i.e., their defined variables are not redefined before leaving $n$
- **DEFKILL**($n$): all definitions killed by a definition in $n$.

**Initial Condition:**

$\text{REACHES}(n_0) = \emptyset$

**Data Flow Equation:**

$\text{REACHES}(n) = \bigcup_{m \in \text{proc}(n)} \text{DEDEF}(m) \cup (\text{REACHES}(m) \cap \text{DEFKILL}(m))$
Example: Def-Use Chains

\[
\begin{align*}
\text{a} & \leftarrow 5 \\
\text{b} & \leftarrow 3 \\
\text{c} & \leftarrow \text{b} + 2 \\
\text{d} & \leftarrow \text{a} - 2 \\
\text{e} & \leftarrow \text{a} + \text{b} \\
\text{e} & \leftarrow \text{e} + \text{c} \\
\text{f} & \leftarrow 2 + \text{e} \\
\text{e} & \leftarrow 13
\end{align*}
\]

\text{d} is dead
It has no use

Write \text{f}
More data flow analysis in next class
Global Register Coloring
Global Register Allocation

The Big Picture

• At each point in the code
• Determine which values will reside in registers
• Select a register for each such value
• The goal is an allocation that “minimizes” running time

Most modern, global allocators use a graph-coloring paradigm

• Build a “conflict graph” or “interference graph”
• Find a $k$-coloring for the graph, where $k$ is the number of available registers, or change the code to a nearby problem that it can $k$-color
Building the Interference Graph

What is an “interference” ? (or conflict)

• Two values *interfere* if there exists an operation where both are simultaneously live
• If x and y interfere, they cannot occupy the same register
• Interference graph construction relies on LIVE information

The interference graph, $G_I = (N_I, E_I)$

• Nodes in $G_I$ represent values, or live ranges
• Edges in $G_I$ represent individual interferences
  For $x, y \in N_I$, $\langle x, y \rangle \in E_I$ iff $x$ and $y$ interfere

A $k$-coloring of $G_I$ can be mapped into an allocation to $k$ registers
Live Range

Live ranges are simpler in a single block

A value is live from its definition to its last use.

- A live range is just the interval in the block from first definition to last use
- In a single block, live ranges form an interval graph

Simple Example:

<table>
<thead>
<tr>
<th>#</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a ← ...</td>
</tr>
<tr>
<td>1</td>
<td>b ← ...</td>
</tr>
<tr>
<td>2</td>
<td>c ← ... a</td>
</tr>
<tr>
<td>3</td>
<td>d ← ... b</td>
</tr>
<tr>
<td>4</td>
<td>e ← ... a</td>
</tr>
<tr>
<td>5</td>
<td>f ← ... e</td>
</tr>
</tbody>
</table>

a’s live range is [0,4]
b’s live range is [1,3]
e’s live range is [4,5]

Live ranges may, of course, overlap
- a & b are simultaneously live, so they cannot occupy the same PR
- a & e can occupy the same PR, as could b & e
Live Range

In the multi-block case, live ranges are more complex than in the local case.

• Consider x, y & z in the code to the right
  ‣ x has 2 different live ranges
Live Range

In the multi-block case, live ranges are more complex than in the local case.

- Consider x, y & z in the code to the right
  - x has 2 different live ranges
  - y has 2 different live ranges
Live Range

In the multi-block case, live ranges are more complex than in the local case.

- Consider x, y & z in the code to the right
  - x has 2 different live ranges
  - y has 2 different live ranges
  - z has 1 live range
Finding Live Ranges

We can use SSA to find live ranges in a simple way

• Build static single assignment form (SSA form)
• Consider each SSA name a set
• At each phi-function, union together the sets of the phi-function arguments
• Each remaining set is a live range
• Rename into live ranges

SSA form will be covered in next class.
Live Range

Example in (Pruned) SSA Form
• Each name is defined in exactly one operation
• Each use refers to one name
• Live ranges are
  1. \((x_0, x_2, x_3)\) and \((x_1)\)
  2. \((y_0)\) and \((y_1, y_2, y_3)\)
  3. \((z_0, z_1, z_2)\)
    As predicted several slides ago
Rename

```
B0 | x0 ← ...
   | z0 ← ...

B1 | z1 ← φ(z0,z2)
   | y0 ← ...

x1 ← y0
y1 ← x1
x2 ← ...

B3 | ... ← y0
   | y2 ← z1

B4 | x3 ← φ(x2,x0)
   | y3 ← φ(y1,y2)
   | ... ← ...
   | z2 ← ...

B5 | ... ← x3 + y3 + z2

B0 | x0 ← ...
   | z0 ← ...

B1 | y0 ← ...

B2 | x1 ← y0
   | y1 ← x1
   | x0 ← ...

B3 | ... ← y0
   | y1 ← z0

B4 | ...
   | z0 ← ...

B5 | ... ← x0 + y1 + z0
```
Chaitin’s Allocation Algorithm

Assume we have \( k \) physical registers: \( k \)-coloring problem

**Observation:**
Any vertex \( n \) that fewer than \( k \) neighbors in the interference graph can always be colored.

**Algorithm:**

1. Pick any vertex \( n \) such that \( \deg(n) < k \) and push it into the stack \( S \).
   Remove \( n \) and all its incident edges from the interference graph.
   If there does not exist such a vertex with degree \( < k \), prune nodes from the graph until we can find a vertex with degree \( < k \). The pruning criteria can be customized.
2. Repeat step 1 until no vertex is left in the graph.
3. Successively pop vertices off the stack \( S \) and color one node at one time.
Example

3 Registers

Stack

1 is the only node with degree < 3
Example

3 Registers

Stack

Now, 2 & 3 have degree < 3
Example

3 Registers

Stack

Now all nodes have degree < 3
Example

3 Registers

Stack

4
2
1

3
5

3
5
Example

3 Registers

Stack

Colors:
1: 🔴
2: 🔴
3: 🔵
Example

3 Registers

Stack

Colors:
1: 🟠
2: 🔴
3: 🔵
Example

3 Registers

Stack

Colors:
1: [Yellow]
2: [Light Pink]
3: [Light Blue]
Example

3 Registers

Stack

Colors:
1: 
2: 
3: 

4 — 5

3
Example

3 Registers

Stack

Colors:
1: 
2: 
3: )
Example

3 Registers

Stack

Colors:
1:
2:
3:
Chaitin-Briggs Algorithm

An Improvement over Chaitin’s algorithm

Observation:
A node that has more than k-1 neighbors is not necessarily un-colorable.

Brigg’s idea:
• Keep the pruned nodes as coloring candidates and still push them into stack
• When you pop it off, a color might be available for it. If so, color it.

Maximum degree is a loose upper bound on colorability
Example

No node has degree < 2
• Chaitin would spill a node
• Briggs picks the same node & stacks it
Example

2 Registers

Stack

Pick a node, say 1
Example

2 Registers

Stack

Pick a node, say 1
Example

2 Registers

Now, both 2 & 3 have degree < 2
Pick one, say 3
Example

Both 2 & 4 have degree < 2.
Take them in order 2, then 4.
Example

2 Registers

Stack
Example

2 Registers

Stack

Now, rebuild the graph
Example

2 Registers

Stack

Colors:
1: 
2: 

1
2
3
4
Example

2 Registers

Stack

Colors:
1:  
2:  

Colors:
1:  
2:  
Example

2 Registers

Stack

Colors:
1:
2:
Example

2 Registers

Stack

Colors:
1: 🔄
2: 🛡
Reading

• Textbook
  Engineering a Compiler
  - Chapter 5.3.4
  - Chapter 5.5
  - Chapter 8.6.1
  - Chapter 9