Optimizations

• Improve program performance
• **Must maintain original semantics**
  – The observable behavior of the optimized program should be equivalent to that of the original program
• Typically cannot obtain “optimal program”
• Can optimize various aspects of program execution
  – Memory
  – Time
  – Code size
  – Power
  – ...
Why Optimize?

- Programmer may introduce inefficiencies
- Compiler may introduce inefficiencies
  - Make earlier compiler stages easier to deal with
Reminder: Data-flow Analysis

\[
\begin{align*}
x & := 0 \\
y & := 1
\end{align*}
\]

while \((y < n)\)

\[
\begin{align*}
x & := x + 2 \\
y & := y + 1
\end{align*}
\]

print \(x\)

print \(y\)
Reminder: Data-flow Analysis

\[
x := 0 \\
y := 1
\]

\[
\text{while} \ (y < n) \\
x := x + 2 \\
y := y + 1
\]

\[
\text{print} \ x \\
\text{print} \ y
\]

\[
\{x \mapsto T, \ y \mapsto T, \ n \mapsto T\}
\]
Reminder: Data-flow Analysis

\[
x := 0 \\
y := 1
\]

\[
\text{while (y < n)}
\]
\[
x := x + 2 \\
y := y + 1
\]

\[
\text{print } x \\
\text{print } y
\]

\[
\{x \mapsto T, y \mapsto T, n \mapsto T\}
\]

\[
\{x \mapsto E, y \mapsto O, n \mapsto T\}
\]
Reminder: Data-flow Analysis

\[
\begin{align*}
x &:= 0 \\
y &:= 1
\end{align*}
\]

\[
\text{while } (y < n)
\]

\[
\begin{align*}
x &:= x + 2 \\
y &:= y + 1
\end{align*}
\]

\[
\text{print } x \\
\text{print } y
\]

\[
\{x \mapsto T, y \mapsto T, n \mapsto T\}
\]

\[
\{x \mapsto E, y \mapsto O, n \mapsto T\}
\]

\[
\{x \mapsto E, y \mapsto O, n \mapsto T\}
\]
Reminder: Data-flow Analysis

\[
\begin{align*}
x &:= 0 \\
y &:= 1
\end{align*}
\]

while \((y < n)\)

\[
\begin{align*}
x &:= x + 2 \\
y &:= y + 1
\end{align*}
\]

\[
\begin{align*}
\text{print } x \\
\text{print } y
\end{align*}
\]
Reminder: Data-flow Analysis

\[
\begin{align*}
\text{x := 0} \\
\text{y := 1}
\end{align*}
\]

while \((y < n)\)

\[
\begin{align*}
\text{x := x + 2} \\
\text{y := y + 1}
\end{align*}
\]

\[
\begin{align*}
\text{print x} \\
\text{print y}
\end{align*}
\]
Reminder: Data-flow Analysis

\[
\begin{align*}
&x := 0 \\
y := 1 \\
\text{while } (y < n) \\
x := x + 2 \\
y := y + 1 \\
\text{print } x \\
\text{print } y \\
\end{align*}
\]
Reminder: Data-flow Analysis

\[
x := 0 \\
y := 1
\]

\[
\text{while } (y < n) \\
x := x + 2 \\
y := y + 1
\]

\[
\text{print } x \\
\text{print } y
\]
Reminder: Data-flow Analysis

\[
\begin{align*}
&x := 0 \\
y := 1
\end{align*}
\]

\[
\begin{align*}
&\text{while (} y < n \text{)}
\end{align*}
\]

\[
\begin{align*}
x &:= x + 2 \\
y &:= y + 1
\end{align*}
\]

\[
\begin{align*}
&\text{print } x \\
&\text{print } y
\end{align*}
\]

\[
\begin{align*}
&x \mapsto T, \ y \mapsto T, \ n \mapsto T \\
&x \mapsto E, \ y \mapsto O, \ n \mapsto T \\
&x \mapsto E, \ y \mapsto O \sqcup E, \ n \mapsto T \\
&x \mapsto E, \ y \mapsto O, \ n \mapsto T \\
&x \mapsto E, \ y \mapsto E, \ n \mapsto T \\
&x \mapsto E, \ y \mapsto O, \ n \mapsto T
\end{align*}
\]

\[
O \sqcup E = T
\]
Reminder: Data-flow Analysis

```
x := 0
y := 1

while (y < n)
  x := x + 2
  y := y + 1

print x
print y
```
Reminder: Data-flow Analysis

\[
\begin{align*}
\text{x := 0} \\
\text{y := 1} \\
\text{while (y < n)} \\
\text{x := x + 2} \\
\text{y := y + 1} \\
\text{print x} \\
\text{print y}
\end{align*}
\]

\[
\begin{array}{r}
\{x \mapsto T, y \mapsto T, n \mapsto T\} \\
\{x \mapsto E, y \mapsto O, n \mapsto T\} \\
\{x \mapsto E, y \mapsto \top \cup E, n \mapsto T\} \quad \text{O \cup E = T} \\
\{x \mapsto E, y \mapsto T, n \mapsto T\} \\
\{x \mapsto E, y \mapsto T, n \mapsto T\} \\
\{x \mapsto E, y \mapsto E, n \mapsto T\} \\
\{x \mapsto E, y \mapsto O, n \mapsto T\}
\end{array}
\]
Reminder: Data-flow Analysis

\[
x := 0
y := 1
\]

while (y < n)

\[
x := x + 2
y := y + 1
\]

print x
print y

\[
\begin{align*}
x &\mapsto T, y \mapsto T, n \mapsto T \\
x &\mapsto E, y \mapsto O, n \mapsto T \\
O \sqcup E &= T \\
x &\mapsto E, y \mapsto O \sqcup E, n \mapsto T \\
x &\mapsto E, y \mapsto T, n \mapsto T \\
x &\mapsto E, y \mapsto T, n \mapsto T \\
x &\mapsto E, y \mapsto T, n \mapsto T \\
x &\mapsto E, y \mapsto O, n \mapsto T \\
\end{align*}
\]
Reminder: Data-flow Analysis

\[
\begin{align*}
x & := 0 \\
y & := 1
\end{align*}
\]

while \( (y < n) \)

\[
\begin{align*}
x & := x + 2 \\
y & := y + 1
\end{align*}
\]

\[
\begin{align*}
\text{print } x \\
\text{print } y
\end{align*}
\]

\[
\begin{align*}
\{x \mapsto \top, y \mapsto \top, n \mapsto \top\} & \quad \{x \mapsto E, y \mapsto O, n \mapsto \top\} \\
\{x \mapsto E, y \mapsto O \sqcup E, n \mapsto \top\} & \quad O \sqcup E = \top \\
\{x \mapsto E, y \mapsto \top, n \mapsto \top\} & \quad \{x \mapsto E, y \mapsto \top, n \mapsto \top\} \\
\{x \mapsto E, y \mapsto \top, n \mapsto \top\} & \quad \{x \mapsto E, y \mapsto \top, n \mapsto \top\} \\
\{x \mapsto E, y \mapsto \top, n \mapsto \top\} & \quad \{x \mapsto E, y \mapsto \top, n \mapsto \top\}
\end{align*}
\]
Reminder: Data-flow Analysis

\[\begin{align*}
  &x := 0 \\
  &y := 1 \\
&\text{while } (y < \text{n}) \\
&  \quad x := x + 2 \\
&  \quad y := y + 1 \\
&  \text{print } x \\
&  \text{print } y
\end{align*}\]

\[
\begin{align*}
{x} &\mapsto \top, \quad {y} \mapsto \top, \quad {\text{n}} \mapsto \top \\
{x} &\mapsto \text{E}, \quad {y} \mapsto \text{O}, \quad {\text{n}} \mapsto \top \\
{x} &\mapsto \text{E}, \quad {y} \mapsto \top \sqcup \text{E}, \quad {\text{n}} \mapsto \top \\
\end{align*}
\]

\[\text{O} \sqcup \text{E} = \top\]
Constant Propagation

• Idea: if on every run, a variable can only take one value, it can be replaced with a constant.

• DFA over the following lattice:

```plaintext
    T
   /   \
  1     2
 /       /
1 2 3    ...
 \
  

1. z = 3
2. x = 1
3. while (x > 0)
4. if (x == 1)
5. y = 7
6. y = z + 4
7. x = 3
8. print y
```
Constant Propagation

- Idea: if on every run, a variable can only take one value, it can be replaced with a constant.
- DFA over the following lattice:

\[
\{x \mapsto \top, y \mapsto \top, z \mapsto \top\}
\]
Basic Optimizations

• Common-subexpression Elimination
  – Eliminate repeated computation inside a basic block (DAG representation)
  – Next time we will see how to do it beyond the boundaries of a single basic block

• Copy-propagation
  – Following an assignment $x = y$, try to use “$y$” whenever “$x$” is used (if possible)
  – Potentially makes “$x$” into a dead variable, saving the assignment $x=y$
Common Subexpression Elimination

• Avoid recomputations
  ‣ ...but be careful:

\[
\begin{align*}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= a - d
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= b
\end{align*}
\]
Static Single-Assignment Form (SSA)

- Every assignment writes to a distinct variable
- Every variable is only assigned once

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```
SSA

if (f)
    x = 42;
else
    x = 73;
y = x * a;
SSA

• Branches?

if (f)
  x = 42;
else
  x = 73;
y = x * a;

if (f)
  x1 = 42;
else
  x2 = 73;
x3 = \phi(x1,x2);
y = x3 * a;

• \(\phi\) (phi) function combines different definitions
• \(\phi\) returns the value of \(x1\) if control passes through the true branch and the value of \(x2\) if it passed through the false branch
SSA

• Branches?

if (f)
  x = 42;
else
  x = 73;
y = x * a;

• \(\phi\) (phi) function combines different definitions

• \(\phi\) returns the value of \(x_1\) if control passes through the true branch and the value of \(x_2\) if it passed through the false branch

if (f)
  x1 = 42;
else
  x2 = 73;
x3 = \(\phi(x_1, x_2)\);
y = x3 * a;
SSA — why should we care?

- Makes it easy to apply many optimizations
  - constant propagation, dead code elimination...

Before:

\[
\begin{align*}
x &= 42 \\
x &= 73 \\
y &= x
\end{align*}
\]

After:

\[
\begin{align*}
x1 &= 42 \\
x2 &= 73 \\
y &= x2
\end{align*}
\]
DAG Representation of Basic Blocks

\[
\begin{align*}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= a - d
\end{align*}
\]

\[
\begin{align*}
a &= b_0 + c_0 \\
b &= a - d_0 \\
c &= b + c_0 \\
d &= a - d_0
\end{align*}
\]
DAG Representation of Basic Blocks

\[
\begin{align*}
a &= b + c \\
b &= b - d \\
c &= c + d \\
e &= b + c \\
\end{align*}
\]

\[
\begin{align*}
a &= b_0 + c_0 \\
b &= b_0 - d_0 \\
c &= c_0 + d_0 \\
e &= b + c \\
\end{align*}
\]
Aliasing is a problem

- We don’t know whether $i = j$
- Same thing happens with pointers
- Have to handle it conservatively
  - Don’t know that it’s safe $\Rightarrow$ must assume that it’s conflicting
- A major obstacle to effective optimization
Peephole Optimizations

- Optimizing long code sequences is hard
- A simple and efficient (but sub-optimal) alternative: peephole optimizations
  - Examine a small window ("peep hole") over the code
  - Identify local optimization opportunities
  - Rewrite code "in the window"
- Example
  - Local algebraic simplifications
Peephole Optimizations

- Algebraic simplification

\[ a = b + c \]
\[ b = d - d \]
\[ c = c \times b \]
\[ e = a + c \]
Peephole Optimizations

- Algebraic simplification

\[
\begin{align*}
a &= b + c \\
b &= d - d \\
c &= c \times b \\
e &= a + c
\end{align*}
\]
Peephole Optimizations

- Algebraic simplification

\[
\begin{align*}
a &= b + c \\
b &= d - d \\
c &= c \times b \\
e &= a + c
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
b &= 0 \\
c &= c \times b \\
e &= a + c
\end{align*}
\]
Peephole Optimizations

- Algebraic simplification

\[
\begin{align*}
a &= b + c \\
b &= d - d \\
c &= c \times b \\
e &= a + c
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
b &= 0 \\
c &= c \times b \\
e &= a + c
\end{align*}
\]
Peephole Optimizations

- Algebraic simplification

\[
\begin{align*}
a &= b + c \\
b &= d - d \\
c &= c \times b \\
e &= a + c
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
b &= 0 \\
c &= c \times b \\
e &= a + c
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
b &= 0 \\
c &= c \times 0 \\
e &= a + c
\end{align*}
\]
Peephole Optimizations

- Algebraic simplification

1. \( a = b + c \)
   \( b = d - d \)
   \( c = c \times b \)
   \( e = a + c \)

2. \( a = b + c \)
   \( b = 0 \)
   \( c = c \times b \)
   \( e = a + c \)

3. \( a = b + c \)
   \( b = 0 \)
   \( c = c \times 0 \)
   \( e = a + c \)
Peephole Optimizations

- Algebraic simplification

\[
\begin{align*}
\text{a} &= \text{b} + \text{c} \\
\text{b} &= \text{d} - \text{d} \\
\text{c} &= \text{c} \times \text{b} \\
\text{e} &= \text{a} + \text{c}
\end{align*}
\]
Peephole Optimizations

- Algebraic simplification

\[
\begin{align*}
a &= b + c \\
b &= d - d \\
c &= c \times b \\
e &= a + c
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
b &= 0 \\
c &= c \times b \\
e &= a + c
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
b &= 0 \\
c &= c \times 0 \\
e &= a + c
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
b &= 0 \\
c &= 0 \\
e &= a
\end{align*}
\]
Peephole Optimizations

- **Algebraic simplification**
  
  
  - $a = b + c$
  - $b = d - d$
  - $c = c \times b$
  - $e = a + c$

  
  
  - $a = b + c$
  - $b = 0$
  - $c = c \times b$
  - $e = a + c$

  
  
  - $a = b + c$
  - $b = 0$
  - $c = c \times 0$
  - $e = a + c$

  
  
  - $a = b + c$
  - $b = 0$
  - $c = 0$
  - $e = a$

- **Flow of control**

  
  
  1: if $d == 1$ goto 3
  2: goto 4
  3: print($x$)
  4: $e = a + c$
Peephole Optimizations

- **Algebraic simplification**
  
  1. $a = b + c$
  2. $b = d - d$
  3. $c = c \times b$
  4. $e = a + c$

- **Flow of control**
  
  1. if $d == 1$ goto 3
  2. goto 4
  3. print($x$)
  4. $e = a + c$
Peephole Optimizations

- **Algebraic simplification**
  
  1. $a = b + c$
  2. $b = d - d$
  3. $c = c * b$
  4. $e = a + c$

- **Flow of control**
  
  1: if $d == 1$ goto 3
  2: goto 4
  3: print(x)
  4: $e = a + c$

  1: if $d != 1$ goto 3
  2: print(x)
  3: $e = a + c$
Eliminating Redundant Code

1:  a := x
2:  x := a
3:  a := someFunction(a)
4:  x := someOtherFunction(a, x)
5:  if a > x goto (2)

- Peephole optimizations are only performed within basic block boundaries
1: \( x := 2; \)
2: \( y := 4; \)
3: \( x := 1; \)
4: if \( y > x \)
5: then \( z := y \)
6: else \( z := y \times y \); 
7: \( x := z \)
Live Variables

1: \( x := 2; \)
2: \( y := 4; \)
3: \( x := 1; \)
4: if \( y > x \)
5: then \( z := y \)
6: else \( z := y \times y; \)
7: \( x := z \)
1: x := 2;
2: y := 4;
3: x := 1;
4: if y > x
5: then z := y
6: else z := y * y;
7: x := z

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a</td>
<td>{ x }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>skip</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>cond</td>
<td>∅</td>
<td>FV(cond)</td>
</tr>
</tbody>
</table>

Live Variables
Live Variables — Solution

1: x := 2;
2: y := 4;
3: x := 1;
4: if y > x
5: then z := y
6: else z := y * y;
7: x := z

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a</td>
<td>{ x }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>skip</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>cond</td>
<td>Ø</td>
<td>FV(cond)</td>
</tr>
</tbody>
</table>

in(7) = Ø
Live Variables — Solution

1: \( x := 2; \)
2: \( y := 4; \)
3: \( x := 1; \)
4: if \( y > x \)
5: \( \text{then } z := y \)
6: \( \text{else } z := y \ast y; \)
7: \( x := z \)

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := a )</td>
<td>{ x }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>skip</td>
<td>( \varnothing )</td>
<td>( \varnothing )</td>
</tr>
<tr>
<td>cond</td>
<td>( \varnothing )</td>
<td>FV(cond)</td>
</tr>
</tbody>
</table>
Live Variables — Solution

1: \( x := 2; \)
2: \( y := 4; \)
3: \( x := 1; \)
4: if \( y > x \)
5: \( \text{then } z := y \)
6: \( \text{else } z := y \ast y; \)
7: \( x := z \)

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := a )</td>
<td>{ x }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>( \text{skip} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{cond} )</td>
<td>( \emptyset )</td>
<td>( \text{FV(cond)} )</td>
</tr>
</tbody>
</table>
Live Variables — Solution

1: \( x := 2; \)
2: \( y := 4; \)
3: \( x := 1; \)
4: \( \text{if } y > x \)
5: \( \text{then } z := y \)
6: \( \text{else } z := y \times y; \)
7: \( x := z \)

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := a )</td>
<td>{ x }</td>
<td>{ \text{FV}(a) }</td>
</tr>
<tr>
<td>skip</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( cond )</td>
<td>( \emptyset )</td>
<td>\text{FV}(\text{cond})</td>
</tr>
</tbody>
</table>
Live Variables — Solution

1: \(x := 2\)
2: \(y := 4\)
3: \(x := 1\)
4: if \(y > x\)
5: then \(z := y\)
6: else \(z := y \times y\)
7: \(x := z\)

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x := a)</td>
<td>{ (x) }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>skip</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(cond)</td>
<td>(\emptyset)</td>
<td>FV((cond))</td>
</tr>
</tbody>
</table>
1: x := 2;
2: y := 4;
3: x := 1;
4: if y > x
5: then z := y
6: else z := y * y;
7: x := z
Live Variables — Solution

1: \( x := 2 \)
2: \( y := 4 \)
3: \( x := 1 \)
4: if \( y > x \)
5: then \( z := y \)
6: else \( z := y * y \)
7: \( x := z \)

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := a )</td>
<td>( { x } )</td>
<td>( { \text{FV}(a) } )</td>
</tr>
<tr>
<td>skip</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>cond</td>
<td>( \emptyset )</td>
<td>( \text{FV}(\text{cond}) )</td>
</tr>
</tbody>
</table>
Live Variables — Solution

1: x := 2;
2: y := 4;
3: x := 1;
4: if y > x
5: then z := y
6: else z := y * y;
7: x := z

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a</td>
<td>{ x }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>skip</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>cond</td>
<td>∅</td>
<td>FV(cond)</td>
</tr>
</tbody>
</table>
Live Variables — Solution

1: \( x := 2 \);
2: \( y := 4 \);
3: \( x := 1 \);
4: if \( y > x \) then \( z := y \)
5: \( \) else \( z := y \cdot y \);
6: \( x := z \)

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := a )</td>
<td>{ x }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>skip</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \textit{cond} )</td>
<td>( \emptyset )</td>
<td>( \text{FV(\textit{cond})} )</td>
</tr>
</tbody>
</table>
1: \( x := 2; \)
2: \( y := 4; \)
3: \( x := 1; \)
4: \( \text{if } y > x \)
5: \( \quad \text{then } z := y \)
6: \( \quad \text{else } z := y \times y; \)
7: \( x := z \)

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := a )</td>
<td>{ x }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>skip</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{cond} )</td>
<td>( \emptyset )</td>
<td>( \text{FV(cond)} )</td>
</tr>
</tbody>
</table>
Eliminating Redundant Code

• Any assignment after which the assigned variable is dead, is redundant

1: \( x := 2 \)  
2: \( y := 4 \)  
3: \( x := 1 \)  
4: if \( y \leq x \) goto 7  
5: \( z := y \)  
6: goto 8  
7: \( z := y \ast y \)  
8: \( x := z \)  

1: \( y := 4; \)  
2: \( x := 1; \)  
3: if \( y \leq x \) goto 6  
4: \( z := y \)  
5: goto 7  
6: \( z := y \ast y \)  
7: \( x := z \)  

(in(1) = \emptyset)
Code Motion

• Identify repeated computation inside a loop that depends only on values that are not modified inside the loop

• Move computation outside of the loop

```java
while (x - 3 < y) {
  // ... instructions that do not change x
}
```
Code Motion

- Identify repeated computation inside a loop that depends only on values that are not modified inside the loop
- Move computation outside of the loop

```c
while (x - 3 < y) {
    // ... instructions that do not change x
}
```

```c
t_1 = x - 3;
while (t_1 < y) {
    // ... instructions that do not change x or t_1
}
```
Induction Variables & Strength Reduction

(1) \( i = 0; \)
(2) \( t_1 = i \times 4 \)
(3) \( t_2 = a[t_1] \)
(4) if \( (t_2 > 100) \) goto (19)
(5) ...
...
(17) \( i = i + 1 \)
(18) goto (2)
(19) ...

- Identify loop counters and relationship with other variables
Induction Variables & Strength Reduction

- Identify loop counters and relationship with other variables

(1) i = 0;
(2) \( t_1 = i \times 4 \)
(3) \( t_2 = a[t_1] \)
(4) if \( t_2 > 100 \) goto (19)
(5) ...
...
(17) i = i + 1
(18) goto (2)
(19) ...

(1) i = 0;
(2) \( t_1 = 0 \)
(3) \( t_2 = a[t_1] \)
(4) if \( t_2 > 100 \) goto (20)
(5) ...
...
(17) i = i + 1
(18) \( t_1 = t_1 + 4 \)
(19) goto (3)
(20) ...
Program Transformations

• Code motion and strength reduction can improve performance, provided that they maintain program semantics.

➢ When does an optimization apply to a given program?

➢ How can this be done automatically?
Program Transformations

- How can a loop be identified?
  - For loop invariant code motion

- Easy to detect in high-level language

- Hard to detect in low-level language and CFG
  - Java bytecode
  - Control Flow Graph
Loop detection

• Loop in CFG has:
  ‣ Loop header
  ‣ Back edge
  ‣ Set of nodes
Dominators

• Use dominators to identify loops
• Node d dominates node n
  – all path from the entry node to n go through d
• Every node dominates itself
Dominators
Dominators

• 1 dominates 1, 2, 3, 4
Dominators

- 1 dominates 1, 2, 3, 4
- 2 dominates 2
Dominators

- 1 dominates 1, 2, 3, 4
- 2 dominates 2
- 3 dominates 3
Dominators

- 1 dominates 1, 2, 3, 4
- 2 dominates 2
- 3 dominates 3
- 4 dominates 4
Loop detection

- **Header**
  - dominates loop nodes
- **Back edge**
  - Target dominates source
- **Loop identification**
  - Back edge identification
Post dominators

- Node d post-dominates node n
  - all path from n to the exit node go through d

- d post-dominates n does not imply n dominates d
Dominator
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
- 4 post-dominates 1, 2, 3, 4
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
- 4 post-dominates 1, 2, 3, 4

- 1 dominates 4
Dominators

• 1 post-dominates 1
• 2 post-dominates 2
• 3 post-dominates 3
• 4 post-dominates 1, 2, 3, 4
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
- 4 post-dominates 1, 2, 3, 4

- 1 does not dominate 4
Computing Dominators

- Using DFA (previous lecture)
- If 1 dominates 2, 3, 4 and 5
  - 1 dominates 6
- If 1 dominates 6
  - 1 dominates 2, 3, 4 and 5
Computing Dominators

- Dom(u)
  - Set of nodes that dominate u
- Dom(u_0) = \{u_0\}
- Dom(u) = \bigcap \{Dom(v) \mid v \in \text{pred}(u)\} \cup \{u\}
Computing Dominators

- $\text{Dom}(u_0) = \{u_0\}$
- $\text{Dom}(u) = \cap \{\text{Dom}(v) \mid v \in \text{pred}(u)\} \cup \{u\}$
Identify Loops

• Back edge
  – Edge $t \leadsto h$ s.t. $h$ dominates $t$

• Loop of a back edge $t \leadsto h$
  – $h$ is the loop header
  – Loop nodes
    • all nodes that can reach $t$ without going through $h$
Identify Loops: Algorithm

- Compute dominator relation
- Identify back edges
- Compute the loop nodes for each back edge
Identify Loops: Algorithm

for each node h in dominator tree
  for each node n for which there exists a back edge n→h
    define the loop with header := h
    body := of all nodes reachable from n
      by a depth first search backwards
      from n that stops at h
Loop Preheader

- Needed by several optimizations
  - Loop invariant code motion

- Additional code is placed there.
Loop Optimization

• Step 1: identify loops (header, pre-header)

• Apply loop optimizations:
  ‣ Loop invariant code motion
  ‣ Strength reduction of induction variable
  ‣ Induction variable elimination
Code Motion — Loop Invariant

```c
for (i=0; i < 10; ++i)
a[i] = 10*i + x*x;
```

- An expression is **loop invariant** if it does not change throughout the execution of the loop
  - Can be hoisted — computed only once
Code Motion — Loop Invariant

- An expression is **loop invariant** if it does not change throughout the execution of the loop
  - Can be hoisted — computed only once

```c
for (i=0; i < 10; ++i)
a[i] = 10*i + x*x;
```

```c
t = x*x;
for (i=0; i < 10; ++i)
a[i] = 10*i + t;
```
Loop Invariant Computation

- $a := b \land c$ is loop invariant if $b$ and $c$:
  - Are constant, or
  - Have only definitions outside the loop, or
  - Have only one definition (each) that are themselves loop invariant

- Reaching definition analysis can be used (DFA)
Loop Invariant Computation

INV := ∅

repeat
  for each definition ℓ in loop such that I ∉ INV
    if each operand in ℓ:
      is constant, or
      has no definitions inside the loop, or
      has exactly one definition d ∈ INV
    then INV := INV ∪ {ℓ}

until no changes in INV
Code Motion — Loop Invariant

• Assume $a := b \land c$ is loop invariant

  ▸ When can we hoist it out of the loop?
  ▸ Where to?
Valid Code Motion

Move \( d: a := b \quad \diamond \quad c \) to pre-header is valid when:

1. \( d \) dominates all loop exits where \( a \) is live
2. \( d \) is the only definition of \( a \) in the loop
3. All uses of \( a \) in loop can only be reached from \( d \)
Valid code motion

1. d dominates all loop exits where a is live

\[
\begin{align*}
a &= 0; \\
\text{for } (i=0; i<10; ++i) &\quad \text{if } (f(i)) \ a = x*x; \ \text{else break}; \\
\text{b} &= a;
\end{align*}
\]
Valid code motion

1. \textit{d} dominates all loop exits where \textit{a} is live

\begin{verbatim}
a = 0;
for (i=0; i<10; ++i)
  if (f(i)) a = x*x; else break;
b = a;
\end{verbatim}
Valid code motion

1. d dominates all loop exits where a is live

```c
a = 0;
for (i=0; i<10; ++i)
  if (f(i))  a = x*x; else break;

b = a;
```

Use dominator relation to check whether each loop exit is dominated by d
Valid code motion

2. \textit{d} is the only definition of a in the loop

\begin{verbatim}
for (i=0; i<10; ++i)
    if (f(i)) a = x*x;
else a = 0;
\end{verbatim}
Valid code motion

2. d is the only definition of a in the loop

```c
for (i=0; i<10; ++i)
    if (f(i)) a = x*x;
else a = 0;
```

Scan loop body for any other definitions of a
Valid code motion

3. All uses of a in loop can only be reached from d

```c
a = 0;
for (i=0; i<10; ++i)
    if (f(i)) a = x*x;
else buf[i] = a;
```
Valid code motion

3. All uses of a in loop can only be reached from d

```c
a = 0;
for (i=0; i<10; ++i)
    if (f(i)) a = x*x;
else buf[i] = a;
```
Valid code motion

3. All uses of a in loop can only be reached from d

```c
a = 0;
for (i=0; i<10; ++i)
    if (f(i)) a = x*x;
else buf[i] = a;
```

Apply reaching definitions analysis and check each use of a for any definitions of a other than d
Valid Code Motion

\[ d: a := b \triangleleft c \]

1. \( d \) dominates all loop exits where \( a \) is live
2. \( d \) is the only definition of \( a \) in the loop
3. All uses of \( a \) in loop can only be reached from \( d \)

```c
for (i=0; i < 10; ++i)
    a[i] = 10*i + x*x;
```
Valid Code Motion

d: a := b ◇ c

1. d dominates all loop exits where a is live
2. d is the only definition of a in the loop
3. All uses of a in loop can only be reached from d

```plaintext
for (i=0; i <10; ++i)
    t_1 := 10*i
    t_2 := x*x
    a[i] := t_1 + t_2
```
Valid Code Motion

\[ d : a := b \land c \]

1. \( d \) dominates all loop exits where \( a \) is live
2. \( d \) is the only definition of \( a \) in the loop
3. All uses of \( a \) in loop can only be reached from \( d \)

\begin{align*}
\text{for } (i=0; i < 10; ++i) \\
t_1 &:= 10*i \\
t_2 &:= x*x \\
a[i] &:= t_1 + t_2
\end{align*}

\begin{align*}
\text{for } (i=0; i < 10; ++i) \\
t_1 &:= 10*i \\
t_2 &:= x*x \\
a[i] &:= t_1 + t_2
\end{align*}
Strength Reduction

• If an expression $e$ is not loop invariant, code motion may still be applied:
  ‣ Compute $e$ once in loop pre-header
  ‣ Whenever any variable occurring in $e$ is modified, update computed value

• When is this beneficial?
  ‣ If the cost of **updating** $e$ is smaller (“weaker”) than **recomputing** $e$ anew
Strength Reduction

• Differencing rules:

\[
t_1 := c \times i \quad t_1 := c \times i
\]

\[
\vdots
\]

\[
i := i + 1 \quad i := i + 1
\]

\[
t_1 = c \times i \quad t_1 = t_1 + c
\]

(c is loop invariant)
Strength Reduction

- Differencing rules:

\[
\begin{align*}
t_1 &:= c \times i \\
i &:= i + 1
\end{align*}
\]

(c is loop invariant)
Strength Reduction

- Differencing rules:

\[
\begin{align*}
    t_1 & := c \times i \\
    \vdots & \\
    i & := i + 1 \\
    t_1 & := t_1 + c
\end{align*}
\]

(c is loop invariant)

\[
\begin{align*}
    t_1 & := c \times i \\
    \vdots & \\
    i & := i + 1 \\
    t_1 & := t_1 + c
\end{align*}
\]

\[
\begin{align*}
    t_1 & := i \times j \\
    \vdots & \\
    j & := j + 1 \\
    t_1 & := i \times j
\end{align*}
\]

\(\text{‘+’ < ‘*’}\)
Strength Reduction

- Differencing rules:

\[ t_1 := c \cdot i \]
\[ t_1 := c \cdot i \]
\[ i := i + 1 \]
\[ t_1 = c \cdot i \]
\[ t_1 = t_1 + c \]

(c is loop invariant)

\[ t_1 := i \cdot j \]
\[ t_1 := i \cdot j \]
\[ i := i + 1 \]
\[ t_1 = i \cdot j \]
\[ t_1 = t_1 + j \]

\[ t_1 := i \cdot j \]
\[ t_1 := i \cdot j \]
\[ j := j + 1 \]
\[ t_1 = i \cdot j \]
\[ t_1 = t_1 + i \]

\[ '+' \prec '*' \]
Summary

• Optimizations
  ✓ Not real “optimal” code
  ✓ But code that is “better” in some respect
  ✓ Often focuses on runtime

• Techniques
  ✓ Local: expressions in basic block, peephole
  ✓ Global: (function-level) Code motion, strength reduction — based on running a DFA first
Activation Records (functions)