Compilers

Local Optimization
Local Optimization

- The simplest form of optimization
- Optimize one basic block
  - No need to analyze the whole procedure body
• Some statements can be deleted
  \[ x := x + 0 \]
  \[ x := x \times 1 \]

• Some statements can be simplified
  \[ x := x \times 0 \implies x := 0 \]
  \[ y := y \times^{2} \implies y := y \times y \]
  \[ x := x \times 8 \implies x := x \ll 3 \]
  \[ x := x \times 15 \implies t := x \ll 4; x := t - x \]

(on some machines \( \ll \) is faster than \( \times \); but not on all!)
Operations on constants can be computed at compile time

- If there is a statement \( x := y \text{ op } z \)
- And \( y \) and \( z \) are constants
- Then \( y \text{ op } z \) can be computed at compile time

Example: \( x := 2 + 2 \Rightarrow x := 4 \)

Example: if \( 2 < 0 \) jump L can be deleted
• Constant folding can be dangerous.
Local Optimization

- **Eliminate unreachable basic blocks:**
  - Code that is unreachable from the initial block
    - E.g., basic blocks that are not the target of any jump or “fall through” from a conditional

- **Removing unreachable code makes the program smaller**
  - And sometimes also faster
    - Due to memory cache effects
    - Increased spatial locality
• Why would unreachable basic blocks occur?
• Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment

• Rewrite intermediate code in *single assignment* form

\[
\begin{align*}
  x &:= z + y \\
  a &:= x \quad \Rightarrow \quad a := b \\
  x &:= 2 \times x \\
  x &:= 2 \times b
\end{align*}
\]

(b is a fresh register)

– More complicated in general, due to loops
Local Optimization

- If
  - Basic block is in single assignment form
  - A definition $x :=$ is the first use of $x$ in a block
- Then
  - When two assignments have the same rhs, they compute the same value
- Example:
  - $x := y + z$
  - $x := y + z$
  - $w := y + z$  $\Rightarrow$  $w := x$
  - (the values of $x$, $y$, and $z$ do not change in the ... code)
• If \( w := x \) appears in a block, replace subsequent uses of \( w \) with uses of \( x \)
  – Assumes single assignment form

• Example:
  \[
  b := z + y \\
  a := b \\
  x := 2 * a
  \]
  \[
  b := z + y \\
  a := b \\
  x := 2 * b
  \]

• Only useful for enabling other optimizations
  – Constant folding
  – Dead code elimination
• Example:

\[
a := 5 \\
x := 2 \times a \\
y := x + 6 \\
t := x \times y
\]

\[
\Rightarrow \\
a := 5 \\
x := 10 \\
y := 16 \\
t := x \ll 4
\]
Local Optimization

If 
\[ w := \text{rhs} \] appears in a basic block 
\[ w \] does not appear anywhere else in the program

Then

the statement \( w := \text{rhs} \) is dead and can be eliminated
– \text{Dead} = \text{does not contribute to the program’s result}

Example: \( (a \text{ is not used anywhere else}) \)
\[
\begin{align*}
x &:= z + y \\
b &:= z + y \\
b &:= z + y \\
a &:= x & \Rightarrow & a &:= b & \Rightarrow & x &:= 2 \times b \\
x &:= 2 \times a & \Rightarrow & x &:= 2 \times b
\end{align*}
\]
• Each local optimization does little by itself

• Typically optimizations interact
  – Performing one optimization enables another

• Optimizing compilers repeat optimizations until no improvement is possible
  – The optimizer can also be stopped at any point to limit compilation time
• Initial code:

\[
\begin{align*}
  a & := x ** 2 \\
  b & := 3 \\
  c & := x \\
  d & := c * c \\
  e & := b * 2 \\
  f & := a + d \\
  g & := e * f
\end{align*}
\]
• Algebraic optimization:
  
  \[
  \begin{align*}
  a & := x \times 2 \\
  b & := 3 \\
  c & := x \\
  d & := c \times c \\
  e & := b \times 2 \\
  f & := a + d \\
  g & := e \times f
  \end{align*}
  \]
• **Algebraic optimization:**

\[
\begin{align*}
a &:= x \times x \\
b &:= 3 \\
c &:= x \\
d &:= c \times c \\
e &:= b \ll 1 \\
f &:= a + d \\
g &:= e \times f
\end{align*}
\]
• Copy propagation:
  
a := x \times x
b := 3
c := x
d := c \times c
e := b \ll 1
f := a + d
g := e \times f
Local Optimization

- Copy propagation:
  
  
  \[
  \begin{align*}
  a & := x \times x \\
  b & := 3 \\
  c & := x \\
  d & := x \times x \\
  e & := 3 \ll 1 \\
  f & := a + d \\
  g & := e \times f
  \end{align*}
  \]
• **Constant folding:**

\[
\begin{align*}
    a &:= x \times x \\
    b &:= 3 \\
    c &:= x \\
    d &:= x \times x \\
    e &:= 3 \ll 1 \\
    f &:= a + d \\
    g &:= e \times f
\end{align*}
\]
• **Constant folding:**

\[
\begin{align*}
  a &:= x \times x \\
  b &:= 3 \\
  c &:= x \\
  d &:= x \times x \\
  e &:= 6 \\
  f &:= a + d \\
  g &:= e \times f
\end{align*}
\]
Local Optimization

- **Common subexpression elimination:**

  ```
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 6
  f := a + d
  g := e * f
  ```
• Common subexpression elimination:

\[
\begin{align*}
a &:= x \times x \\
b &:= 3 \\
c &:= x \\
d &:= a \\
e &:= 6 \\
f &:= a + d \\
g &:= e \times f
\end{align*}
\]
• Copy propagation:

\[
\begin{align*}
a & := x \times x \\
b & := 3 \\
c & := x \\
d & := a \\
e & := 6 \\
f & := a + d \\
g & := e \times f
\end{align*}
\]
• Copy propagation:
  
  a := x * x
  b := 3
  c := x
  d := a
  e := 6
  f := a + a
  g := 6 * f
Local Optimization

- **Dead code elimination:**

  \[
  \begin{align*}
  a & := x \times x \\
  b & := 3 \\
  c & := x \\
  d & := a \\
  e & := 6 \\
  f & := a + a \\
  g & := 6 \times f 
  \end{align*}
  \]
Local Optimization

- Dead code elimination:
  
  \[
  a := x \times x
  \]
  
  \[
  f := a + a \\
  g := 6 \times f
  \]
  
- This is the final form
Which of the following are valid local optimizations for the given basic block? Assume that only \( g \) and \( x \) are referenced outside of this basic block.

1. Copy propagation: Line 4 becomes \( d := a \times b \).
2. Common subexpression elimination:
   Line 5 becomes \( e := d \).
3. Dead code elimination: Line 3 is removed.
4. After many rounds of valid optimizations, the entire block can be reduced to \( g := 5 \).