Intermediate Code & Local Optimizations

Lecture 20
Lecture Outline

• Intermediate code

• Local optimizations

• Next time: global optimizations
Code Generation Summary

• We have discussed
  - Runtime organization
  - Simple stack machine code generation
  - Code generation for Objects

• So far, our compiler maps AST to assembly language
  - And does not perform optimizations
Optimization

- Optimization is our last compiler phase

- Most complexity in modern compilers is in the optimizer
  - Also by far the largest phase

- First, we need to discuss intermediate languages
Why Intermediate Languages?

• When should we perform optimizations?
  - On AST
    • Pro: Machine independent
    • Con: Too high level
  - On assembly language
    • Pro: Exposes optimization opportunities
    • Con: Machine dependent
    • Con: Must reimplement optimizations when retargetting
  - On an intermediate language
    • Pro: Machine independent
    • Pro: Exposes optimization opportunities
Intermediate Languages

• Intermediate language = high-level assembly

  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
  - Uses opcodes but some are higher level
    • E.g., push translates to several assembly instructions
    • Most opcodes correspond directly to assembly opcodes
Three-Address Intermediate Code

- Each instruction is of the form
  \[ x := y \text{ op } z \]
  \[ x := \text{ op } y \]
  - \( y \) and \( z \) are registers or constants
  - Common form of intermediate code
- The expression \( x + y * z \) is translated
  \[ t_1 := y * z \]
  \[ t_2 := x + t_1 \]
  - Each subexpression has a “name”
Generating Intermediate Code

• Similar to assembly code generation

• But use any number of IR registers to hold intermediate results
Generating Intermediate Code (Cont.)

• $igen(e, t)$ function generates code to compute the value of $e$ in register $t$

• Example:

  $igen(e_1 + e_2, t) =$
  
  $igen(e_1, t_1) \quad (t_1 \text{ is a fresh register})$
  
  $igen(e_2, t_2) \quad (t_2 \text{ is a fresh register})$

  $t := t_1 + t_2$

• Unlimited number of registers

  $\Rightarrow$ simple code generation
Intermediate Code Notes

• You should be able to use intermediate code
  - At the level discussed in lecture

• You are not expected to know how to generate intermediate code
  - Because we won’t discuss it
  - But really just a variation on code generation . . .
An Intermediate Language

\[ P \rightarrow S P | \varepsilon \]
\[ S \rightarrow id := id \ op \ id \]
\[ \quad | id := op \ id \]
\[ \quad | id := id \]
\[ \quad | \text{push} \ id \]
\[ \quad | id := \text{pop} \]
\[ \quad | \text{if} \ id \ \text{relop} \ id \ \text{goto} \ L \]
\[ \quad | L:\]
\[ \quad | \text{jump} \ L \]

- id’s are register names
- Constants can replace id’s
- Typical operators: +, -, *
Definition. Basic Blocks

• A **basic block** is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except in the last instruction)

• Idea:
  - Cannot jump into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - A basic block is a single-entry, single-exit, straight-line code segment
Basic Block Example

- Consider the basic block
  1. L:
  2. \( t := 2 * x \)
  3. \( w := t + x \)
  4. if \( w > 0 \) goto L

- (3) executes only after (2)
  - We can change (3) to \( w := 3 * x \)
  - Can we eliminate (2) as well?
Definition. Control-Flow Graphs

- A control-flow graph is a directed graph with
  - Basic blocks as nodes
  - An edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B
  - E.g., the last instruction in A is \texttt{jump L_B}
  - E.g., execution can fall-through from block A to block B
Example of Control-Flow Graphs

- The body of a method (or procedure) can be represented as a control-flow graph

- There is one initial node

- All “return” nodes are terminal

$x := 1$
i := 1

L: 
\[ x := x \ast x \]
\[ i := i + 1 \]
\[ \text{if } i < 10 \text{ goto } L \]
Optimization Overview

- Optimization seeks to improve a program’s resource utilization
  - Execution time (most often)
  - Code size
  - Network messages sent, etc.

- Optimization should not alter what the program computes
  - The answer must still be the same
A Classification of Optimizations

- For languages like C and C++ there are three granularities of optimizations
  1. Local optimizations
     - Apply to a basic block in isolation
  2. Global optimizations
     - Apply to a control-flow graph (method body) in isolation
  3. Inter-procedural optimizations
     - Apply across method boundaries

- Most compilers do (1), many do (2), few do (3)
Cost of Optimizations

• In practice, a conscious decision is made not to implement the fanciest optimization known

• Why?
  - Some optimizations are hard to implement
  - Some optimizations are costly in compilation time
  - Some optimizations have low benefit
  - Many fancy optimizations are all three!

• Goal: Maximum benefit for minimum cost
Local Optimizations

• The simplest form of optimizations

• No need to analyze the whole procedure body
  - Just the basic block in question

• Example: algebraic simplification
Algebraic Simplification

• Some statements can be deleted
  \[ x := x + 0 \]
  \[ x := x \times 1 \]

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

(on some machines \(\ll\) is faster than \(\times\); but not on all!)
Constant Folding

• 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
Flow of Control Optimizations

• 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

• Why would such basic blocks occur? (e.g., if-conditional always evaluates to false)

• Removing unreachable code makes the program smaller
  - And sometimes also faster
    • Due to memory cache effects (increased spatial locality)
Single Assignment Form

- 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

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

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

  (\(b\) is a fresh register)

  - More complicated in general, due to loops
Common Subexpression Elimination

• 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:

\[
\begin{align*}
  x & := y + z \\
  \ldots & \Rightarrow \ldots \\
  w & := y + z \\
  \ldots & \\
  w & := x
\end{align*}
\]

(the values of \( x, y, \) and \( z \) do not change in the ... code)
Copy Propagation

• If \( w := x \) appears in a block, replace subsequent uses of \( w \) with uses of \( x \)
  - Assumes single assignment form

• Example:
  
  \[
  \begin{align*}
  b & := z + y \\
  a & := b \\
  x & := 2 * a
  \end{align*}
  \]
  \[
  \begin{align*}
  b & := z + y \\
  a & := b \\
  x & := 2 * b
  \end{align*}
  \]

• Only useful for enabling other optimizations
  - Constant folding
  - Dead code elimination
Copy Propagation and Constant Folding

• Example:

\[
\begin{align*}
  a & := 5 \\
  x & := 2 \times a \\
  y & := x + 6 \\
  \uparrow & := x \times y \\
\end{align*}
\]

\[\Rightarrow\]

\[
\begin{align*}
  a & := 5 \\
  x & := 10 \\
  y & := 16 \\
  \uparrow & := x \ll 4 \\
\end{align*}
\]
Copy Propagation and Dead Code Elimination

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
- \( \text{Dead} = \) does not contribute to the program’s result

Example: (\( a \) is not used anywhere else)

\[
\begin{align*}
  x &:= z + y \\
  a &:= x &\Rightarrow& a := b &\Rightarrow& x := 2 \cdot b \\
  x &:= 2 \cdot a &\Rightarrow& x := 2 \cdot b
\end{align*}
\]
Applying Local Optimizations

- 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
An Example

- **Initial code:**

  ```
  a := x ** 2
  b := 3
  c := x
  d := c * c
  e := b * 2
  f := a + d
  g := e * f
  ```
An Example

• Algebraic optimization:

\[
\begin{align*}
    a & := x \; \text{**} \; 2 \\
    b & := 3 \\
    c & := x \\
    d & := c \; \ast \; c \\
    e & := b \; \ast \; 2 \\
    f & := a \; + \; d \\
    g & := e \; \ast \; f
\end{align*}
\]
An Example

- **Algebraic optimization:**

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

• Copy propagation:
  
a := x * x  
b := 3  
c := x  
d := c * c  
e := b << 1  
f := a + d  
g := e * f
An Example

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

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

- **Constant folding:**
  
  ```plaintext
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 6
  f := a + d
  g := e * f
  ```
An Example

• **Common subexpression elimination:**

  a := x * x
  b := 3
  c := x
  d := x * x
  e := 6
  f := a + d
  g := e * f
An Example

• **Common subexpression elimination:**

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

• Copy propagation:

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

• Copy propagation:
  
  a := x * x
  b := 3
  c := x
  d := a
  e := 6
  f := a + a
  g := 6 * f
An Example

• Dead code elimination:
  
  \[
  \begin{align*}
  a & := x \times x \\
  b & := 3 \\
  c & := x \\
  d & := a \\
  e & := 6 \\
  f & := a + a \\
  g & := 6 \times f
  \end{align*}
  \]
An Example

• Dead code elimination:
  \[ a := x \times x \]
  \[ f := a + a \]
  \[ g := 6 \times f \]

• This is the final form
Peephole Optimizations on Assembly Code

• These optimizations work on intermediate code
  - Target independent
  - But they can be applied on assembly language also

• **Peephole optimization** is effective for improving assembly code
  - The “peephole” is a short sequence of (usually contiguous) instructions
  - The optimizer replaces the sequence with another equivalent one (but faster)
Peephole Optimizations (Cont.)

- Write peephole optimizations as replacement rules
  \[ i_1, \ldots, i_n \rightarrow j_1, \ldots, j_m \]
  where the rhs is the improved version of the lhs

- Example:
  \[
  \text{move } $a $b, \text{ move } $b $a \rightarrow \text{move } $a $b \\
  \text{- Works if } \text{move } $b $a \text{ is not the target of a jump}
  \]

- Another example
  \[
  \text{addiu } $a $a i, \text{ addiu } $a $a j \rightarrow \text{addiu } $a $a i+j
  \]
Peephole Optimizations (Cont.)

• Many (but not all) of the basic block optimizations can be cast as peephole optimizations
  - Example: `addiu $a $b 0` → `move $a $b`
  - Example: `move $a $a` →
  - These two together eliminate `addiu $a $a 0`

• As for local optimizations, peephole optimizations must be applied repeatedly for maximum effect
Local Optimizations: Notes

- Intermediate code is helpful for many optimizations

- Many simple optimizations can still be applied on assembly language

- “Program optimization” is grossly misnamed
  - Code produced by “optimizers” is not optimal in any reasonable sense
  - “Program improvement” is a more appropriate term

- Next time: global optimizations