Intermediate Code. Local Optimizations

Lecture 35
(Adapted from notes by R. Bodik and G. Necula)
Lecture Outline

• Intermediate code

• Local optimizations

• Next time: global optimizations
Code Generation Summary

• We have discussed
  - Runtime organization
  - Simple stack machine code generation
  - Improvements to stack machine code generation

• Our compiler goes directly from AST to assembly language
  - And does not perform optimizations

• Most real compilers use intermediate languages
Why Intermediate Languages?

- When to perform optimizations
  - On AST
    - **Pro**: Machine independent
    - **Cons**: Too high level
  - On assembly language
    - **Pro**: Exposes optimization opportunities
    - **Cons**: Machine dependent
    - **Cons**: Must reimplement optimizations when retargetting
  - On an intermediate language
    - **Pro**: Machine independent
    - **Pro**: Exposes optimization opportunities
    - **Cons**: One more language to worry about
Intermediate Languages

• Each compiler uses its own intermediate language
  - IL design is still an active area of research

• Intermediate language = high-level assembly language
  - 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
  \[ x := y \text{ op } z \]
  - \( y \) and \( z \) can be only registers or constants
  - Just like assembly
- Common form of intermediate code
- The AST expression \( x + y \times z \) is translated as
  \[ t_1 := y \times z \]
  \[ t_2 := x + t_1 \]
  - Each subexpression has a “home” in a temporary
Generating Intermediate Code

- Similar to assembly code generation
- Major difference
  - Use any number of IL registers to hold intermediate results
Generating Intermediate Code (Cont.)

- \textbf{Igen}(e, t) function generates code to compute the value of }e\textit{ in register }t\textit{.

- Example:

  \begin{align*}
  \text{igen}(e_1 + e_2, t) &= \\
  \text{igen}(e_1, t_1) &\quad (t_1 \text{ is a fresh register}) \\
  \text{igen}(e_2, t_2) &\quad (t_2 \text{ is a fresh register}) \\
  t &:= t_1 + t_2
  \end{align*}

- Unlimited number of registers
  \implies \text{simple code generation}
Intermediate Code. Notes

- Intermediate code is discussed in Ch. 8
  - Required reading

- You should be able to manipulate intermediate code
An Intermediate Language

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

- 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 in a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - Each instruction in a basic block is executed after all the preceding instructions have been executed
Basic Block Example

• Consider the basic block
  1. \( L: \)
  2. \( t := 2 * x \)
  3. \( w := t + x \)
  4. \( \text{if } w > 0 \text{ goto } L' \)

• No way for (3) to be executed without (2) having been executed right before
  - 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 flow 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., the execution can fall-through from block A to
block B

- Frequently abbreviated as \texttt{CFG}
Control-Flow Graphs. Example.

- 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
Optimization Overview

• Optimization seeks to improve a program’s utilization of some resource
  - Execution time (most often)
  - Code size
  - Network messages sent
  - Battery power used, 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 Cool 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) and very 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 terms of compilation time.
  - The fancy optimizations are both hard and costly.

• The goal: maximum improvement with minimum of 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

\[ x := x + 0 \]

\[ x := x * 1 \]

• Some statements can be simplified

\[ x := x * 0 \quad \Rightarrow \quad x := 0 \]

\[ y := y ** 2 \quad \Rightarrow \quad y := y * y \]

\[ x := x * 8 \quad \Rightarrow \quad x := x \ll 3 \]

\[ x := x * 15 \quad \Rightarrow \quad t := x \ll 4; \ x := t - x \]

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

- Operations on constants can be computed at compile time
- In general, 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 \( 2 \lt 0 \) jump L can be deleted
- When might constant folding be dangerous?
Flow of Control Optimizations

• Eliminating unreachable code:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or “fall through” from a conditional
  - Such basic blocks can be eliminated

• Why would such basic blocks occur?

• 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 assignment is to a temporary that has not appeared already in the basic block.
- Intermediate code can be rewritten to be in single assignment form:

  \[
  \begin{align*}
  &x := a + y \\
  &a := x \\
  &x := a \times x \\
  &b := x + a
  \end{align*}
  \Rightarrow
  \begin{align*}
  &x := a + y \\
  &a_1 := x \\
  &x_1 := a_1 \times x \\
  &b := x_1 + a_1
  \end{align*}
  \]

  \(x_1\) and \(a_1\) are fresh temporaries.
Common Subexpression Elimination

• Assume
  - Basic block is in *single assignment form*
• All assignments with same rhs compute the same value
• Example:
  \[
  x := y + z \\
  \ldots \\
  w := y + z \\
  \]
  \[
  x := y + z \\
  \Rightarrow \\
  w := x
  \]
  
• Why is single assignment important here?
Copy Propagation

• If \( w := x \) appears in a block, all subsequent uses of \( w \) can be replaced with uses of \( x \)

• Example:

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

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

• This does not make the program smaller or faster but might enable other optimizations
  - Constant folding
  - Dead code elimination

• Again, single assignment is important here.
Copy Propagation and Constant Folding

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

  \[
  \begin{align*}
  \Rightarrow & \quad \begin{align*}
  \text{a} & := 5 \\
  \text{x} & := 10 \\
  \text{y} & := 16 \\
  \text{t} & := \text{x} \ll 4
  \end{align*}
  \end{align*}
  \]
Dead Code Elimination

If

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

Then

the statement \[ w := \text{rhs} \] is dead and can be eliminated
- \textbf{Dead} = does not contribute to the program's result

Example: (\textit{a} is not used anywhere else)

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

• Each local optimization does very little by itself
• Typically optimizations interact
  – Performing one optimizations enables other opt.
• Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
  – The optimizer can also be stopped at any time to limit the 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:
  \[ a := x \times 2 \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := c \times c \]
  \[ e := b \times 2 \]
  \[ f := a + d \]
  \[ g := e \times f \]
An Example

• Algebraic optimization:

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

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

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

• Constant folding:
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 3 + 3
  f := a + d
  g := e * f
An Example

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

- **Common subexpression elimination:**

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

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

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

• The optimizations presented before work on intermediate code
  - They are target independent
  - But they can be applied on assembly language also

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

move $a$ $b$, move $b$ $a \rightarrow move$ $a$ $b$

- Works if move $b$ $a$ is not the target of a jump

• Another example

addiu $a$ $a$ $i$, addiu $a$ $a$ $j \rightarrow addiu$ $a$ $a$ $i+j$
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`

- Just like for local optimizations, peephole optimizations need to be applied repeatedly to get 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: global optimizations