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, 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.)

- Igen(e, t) function generates code to compute the value of e in register t
- Example:
  
  \[
  \text{igen}(e_1 \cdot e_2, t) = \\
  \text{igen}(e_1, t_1) \quad (t_1 \text{ is a fresh register}) \\
  \text{igen}(e_2, t_2) \quad (t_2 \text{ is a fresh register}) \\
  t := t_1 \cdot t_2
  \]

  - Unlimited number of registers
    ⇒ 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

\[
\begin{align*}
    P & \to SP | \\
    S & \to \text{id := id op id} \\
    & \mid \text{id := id} \\
    & \mid \text{id := push id} \\
    & \mid \text{id := pop} \\
    & \mid \text{if id relop id goto \text{L}} \\
    & \mid \text{L:} \\
    & \mid \text{jump L}
\end{align*}
\]

  - 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. \text{L:}
  2. \text{t := 2 \cdot x}
  3. \text{w := t + x}
  4. \text{if w > 0 goto L'}
- No way for (3) to be executed without (2) having been executed right before
  - We can change (3) to \text{w := 3 \cdot 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 `jump L_B`
  - E.g., the execution can fall-through from block A to block B

- Frequently abbreviated as 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**
  
  \[ x := x + 0 \]
  
  \[ x := x \times 1 \]

- **Some statements can be simplified**
  
  \[ x := x \times 0 \Rightarrow x := 0 \]
  
  \[ y := y \times 2 \Rightarrow y := y \times y \]
  
  \[ x := x \times 8 \Rightarrow x := x < 3 \]
  
  \[ x := x \times 15 \Rightarrow t := x < 4; x := t - x \]

  (on some machines \(<\) 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
  
  \[ x := y \mathbin{\circ} z \]
  
  - And \(y\) and \(z\) are constants
  
  - Then \(y \mathbin{\circ} z\) can be computed at compile time.
  
  Example: \(x := 2 + 2 \Rightarrow x := 4\)

  Example: if \(2 \times 0\) 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

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

  \[
  \begin{align*}
  x_1 &:= a \times x \\
  b_1 &:= x_1 + a_1 \\
  \end{align*}
  \]

  \( (x_1 \text{ and } a_1 \text{ are fresh temporaries}) \)

### Common Subexpression Elimination

- Assume
  
  - Basic block is in single assignment form

- All assignments with same rhs compute the same value

- Example:

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

- Why is single assignment important here?

### Copy Propagation

- If \(w \mathbin{\circ} x\) appears in a block, all subsequent uses of \(w\) can be replaced with uses of \(x\)

- Example:

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

- 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:
  
  \[
  \begin{array}{ll}
  a & := 5 \\
  x & := 2 \cdot a \\
  y & := x + 6 \\
  t & := x \cdot y
  \end{array}
  \Rightarrow
  \begin{array}{ll}
  a & := 5 \\
  x & := 10 \\
  y & := 16 \\
  t & := x \cdot 4
  \end{array}
  \]

**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
- Dead \( w \) does not contribute to the program’s result

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

\[
\begin{array}{ll}
  x & := z + y \\
  a & := x \\
  b & := z + y \\
  a & := b \\
  x & := 2 \cdot a
  \end{array}
  \Rightarrow
  \begin{array}{ll}
  a & := b \\
  x & := 2 \cdot b
  \end{array}
  \]

**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:
  \[
  \begin{array}{ll}
  a & := x \cdot 2 \\
  b & := 3 \\
  c & := x \\
  d & := c \cdot c \\
  e & := b \cdot 2 \\
  f & := a + d \\
  g & := e \cdot f
  \end{array}
  \]

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

• Copy propagation:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := c \times c \]
  \[ e := b + b \]
  \[ f := a + d \]
  \[ g := e \times f \]

An Example

• Copy propagation:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := x \times x \]
  \[ e := 3 + 3 \]
  \[ f := a + d \]
  \[ g := e \times f \]

An Example

• Constant folding:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := x \times x \]
  \[ e := 3 + 3 \]
  \[ f := a + d \]
  \[ g := e \times f \]

An Example

• Constant folding:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := x \times x \]
  \[ e := 6 \]
  \[ f := a + d \]
  \[ g := e \times f \]

An Example

• Common subexpression elimination:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := x \times x \]
  \[ e := b \]
  \[ f := a + d \]
  \[ g := e \times f \]

An Example

• Common subexpression elimination:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := a \]
  \[ e := 6 \]
  \[ f := a + d \]
  \[ g := e \times f \]
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

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

  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
  \[
  i_1 \rightarrow j_1, \ldots, i_n \rightarrow j_n
  \]
  where the rhs is the improved version of the lhs
- Example:
  - move $a$ $b$, move $b$ $a$ → move $a$ $b$
  - Works if move $b$ $a$ is not the target of a jump
- Another example
  - addiu $a$ $i$, addiu $a$ $a$ $j$ → 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