Local Optimizations

Lecture 21
Lecture Outline

• Local optimization

• Next time: global optimizations
Code Generation Summary

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

• Our compiler goes directly from AST to assembly language with a brief stop or two
  - If we preserved environment data from typecheck, use that;
  - cleanup other minor loose ends perhaps.
  - Simple-compile.lisp does not perform optimizations

• Most real compilers use some optimization somewhere (history of Fortran I..)
When to perform optimizations

- **On AST**
  - **Pro:** Machine independent
  - **Con:** Too high level

- **On assembly language**
  - **Pro:** Exposes more optimization opportunities
  - **Con:** Machine dependent
  - **Con:** Must reimplement optimizations when retargetting

- **On an intermediate language between AST and assembler**
  - **Pro:** Machine independent
  - **Pro:** Exposes many optimization opportunities
Intermediate Languages for Optimization

• 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 may translate to several assembly instructions
    • Perhaps some opcodes correspond directly to assembly opcodes
• Usually not stack oriented.
Texts often consider optimizing based on Three-Address Intermediate Code

- Computations are reduced to simple forms like
  \[ x := y \text{ op } z \] [3 addresses]
or maybe \[ x := \text{ op } y \]
  - \( y \) and \( z \) can be only registers or constants (not expressions!)
  - Also need control flow test/jump/call/
- New variables are generated, perhaps to be used only once (SSA = static single assignment)
- The expression \( x + y \ast z \) is translated as
  \[ t_1 := y \ast z \]
  \[ t_2 := x + t_1 \]
  - Each subexpression then has a “home” for its value
How hard to generate this kind of Intermediate Code?

• Similar technique to our assembly code generation

• Major differences
  - Use any number of IL registers to hold intermediate results
  - Not stack oriented

• Same compiler organization..
Generating Intermediate Code (Cont.)

- **Igen(e, t)** function generates code to compute the value of e in register t
- **Example:**
  \[
  \text{igen}(e_1 + e_2, t) = \\
  \text{igen}(e_1, t_1) ; (t_1 \text{ is a fresh register}) \\
  \text{igen}(e_2, t_2) ; (t_2 \text{ is a fresh register}) \\
  t := t_1 + t_2 ; (\text{instead of } “+”) \\
  \]
- **Unlimited number of registers** ⇒ simple code generation
We can define an Intermediate Language formally, too...

P → S ; P | ε
S → id := id op id
  | id := op id
  | id := id
  | push id
  | id := pop
  | if id relop id goto L
  | L:
  | jump L

• id’s are register names
• Constants can replace id’s
• Typical operators: +, -, *

• id’s are register names
• Constants can replace id’s
• Typical operators: +, -, *

---

Prof. Fateman  CS164  Lecture 21  9
Optimization Concepts

- Inside Basic Blocks
- Between/Around Basic Blocks: Control Flow Graphs
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)
  - 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. \( t := 2 \times x \)
  3. \( w := t + x \)
  4. if \( w > 0 \) goto L

• No way for (3) to be executed without (2) having been executed right before
  - We can change (3) to \( w := 3 \times x \)
  - 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 CFG ... too bad we already used this..
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

```plaintext
x := 1
i := 1

L:
x := x * x
i := i + 1
if i < 10 goto L
```
**Optimization Overview**

- Optimization seeks to improve a program’s utilization of some resource
  - Execution time (most often) [instructions, memory access]
  - Code size
  - Network messages sent,
  - Battery power used, etc.

- Optimization should not alter what the program computes
  - The answers must still be the same (* sometimes relaxed for floating point numbers... a bad idea, though)
  - Same behavior on bad input (?) e.g. array bounds?
A Classification of Optimizations

• For languages like Java 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 (function body) in isolation
  3. Inter-procedural optimizations
     • Apply across call boundaries
• Most compilers do (1), many do (2) and very few do (3)
Cost of Optimizations

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

• Why?
  – Some optimizations are hard to implement. Programs are tricky to write/debug
  – Some optimizations are costly in terms of compilation time. Even exponential time $O(2^s)$, for program of size $s$.
  – Some fancy optimizations are both hard and costly!

• Depends on goal:
  – maximum improvement with acceptable cost / debuggability
  – vs. beat competitive benchmarks
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
  \[ x := x \times 0 \quad \Rightarrow \quad x := 0 \quad ;;x \text{ not "infinity" or NaN} \]
  \[ y := y^2 \quad \Rightarrow \quad y := y \times y \]
  \[ x := x \times 8 \quad \Rightarrow \quad x := x \ll 3 \]
  \[ x := x \times 15 \quad \Rightarrow \quad t := x \ll 4; x := t - x \]
  (on some machines \( \ll \) is faster than \( \times \); but not on all!)
**Constant Folding**

- Operations on constants can be computed at compile time.
- In general, if there is a statement
  \[ x := y \text{ op } z \]
  - And \( y \) and \( z \) are constants (and \( \text{op} \) has no side effects)
  - Then \( y \text{ op } z \) can be computed at compile time [if you are computing on the same machine, at least. Eg. 32 vs 64 bit?]
- Example: \( x := 2 + 2 \Rightarrow x := 4 \)
- Example: \( \text{if } 2 < 0 \text{ jump L} \) can be deleted
- When might constant folding be dangerous?
- Why would anyone write such stupid code?
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)
Using (Static) Single Assignment Form SSA

• Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment.
• Intermediate code can be rewritten to be in single assignment form.
  \[
  \begin{align*}
  x & := z + y & b & := z + y \\
  a & := x & \Rightarrow & a & := b \\
  x & := 2 \times x & x & := 2 \times b \\
  (b & \text{ is a fresh register})
  \end{align*}
  \]
• More complicated in general, due to loops.
Common Subexpression Elimination

- Assume
  - Basic block is in single assignment form
  - A definition \( x := \) is the first use of \( x \) in a block
- All assignments with same rhs compute the same value
- Example:

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

\( w := x \)

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

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

  ⇒

  \[
  \begin{align*}
    a & := 5 \\
    x & := 10 \\
    y & := 16 \\
    t & := x << 4 \\
  \end{align*}
  \]
Copy Propagation and 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
- \( \text{Dead} = \) does not contribute to the program’s result

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

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

• Each local optimization does very little by itself
• Often the optimization seems silly “who would write code like that?” Answer: the optimizer, in a previous step! That is: typically optimizations interact so that performing one optimization enables other opts.
• Typical optimizing compilers repeatedly perform optimizations until no more improvement is produced.
• The optimizer can also be stopped at any time to limit the compilation time
An Example

• **Initial code:**
  
  
  \[
  \begin{align*}
  a & := x^2 \\
  b & := 3 \\
  c & := x \\
  d & := c \times c \\
  e & := b \times 2 \\
  f & := a + d \\
  g & := e \times f
  \end{align*}
  \]
An Example

• Algebraic optimization:
  
  
  \[
  a := x ^ 2 \\
  b := 3 \\
  c := x \\
  d := c * c \\
  e := b * 2 \\
  f := a + d \\
  g := e * f
  \]
An Example

- **Algebraic optimization:**
  
  ```plaintext
  a := x * x
  b := 3
  c := x
  d := c * c
  e := b << 1
  f := a + d
  g := e * f
  ```
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:
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 3 << 1
  f := a + d
  g := e * f
An Example

• **Constant folding:**
  
  a := x * x  
  b := 3  
  c := x  
  d := x * x  
  e := 3 << 1  
  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:*

  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := a \]
  \[ e := 6 \]
  \[ f := a + d \]
  \[ g := e \times f \]
An Example

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

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

• 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 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:
  
  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 as with other 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 time: global optimizations