Code Optimizations

- Code optimizations are code transformations
- Optimization is a misnomer
  - can sometimes make things worse
- Can be applied at various level of code: high, intermediate and low
- Can be applied to various regions of programs
  - local - basic block
  - global - control flow graph of method
  - inter-procedure - across methods

Optimization Phase

- Next to last phase of compiler
- Goal is to improve the quality of the generated code
  - Time
  - Memory
  - Power
- Typically performed on some (or several levels of) intermediate language
Why Intermediate Languages?

- When to perform optimizations
  - On AST
    - Pro: Machine independent
    - Con: Too high level
  - On assembly language
    - Pro: Exposes optimization opportunities
    - Con: Machine dependent
    - Con: Must re-implement optimizations when retargeting
  - On an intermediate language
    - Pro: Machine independent
    - Pro: Exposes optimization opportunities

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 \( x + y \ast z \) is translated as
    \[
    t_1 := y \ast z \\
    t_2 := x + t_1
    \]
    - Each subexpression has a "home"
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:
  
  \[
  \text{igen}(e_1 + e_2, t) =
  \begin{align*}
  &\text{igen}(e_1, t_1) \\
  &\text{igen}(e_2, t_2) \\
  &t := t + t_1
  \end{align*}
  \]

  \(t_1\) is a fresh register
  \(t_2\) is a fresh register

- Unlimited number of registers
  \(\implies\) simple code generation

An Intermediate Language

- \(P \Rightarrow S \mid e\)
- \(S \Rightarrow \text{id} := \text{id} \ op \ \text{id} \mid \text{id} := \text{op} \ \text{id} \mid \text{id} := \text{id} \mid \text{push} \ \text{id} \mid \text{id} := \text{pop} \mid \text{id} \ \text{if} \ \text{rel} \ \text{id} \ \text{goto} \ L \mid L \mid \text{jump} \ L\)

- \(S\) statements
- \(P\) commands
- \(e\) expressions
- \(id\) identifiers
- \(op\) operators
- \(L\) labels

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

- \(+\) addition
- \(-\) subtraction
- \(*\) multiplication

- \(\mathrm{id} := \mathrm{id} \ op \ \mathrm{id}\)
- \(\mathrm{id} := \mathrm{op} \ \mathrm{id}\)
- \(\mathrm{id} := \mathrm{id}\)
- \(\mathrm{push} \ \mathrm{id}\)
- \(\mathrm{id} := \mathrm{pop}\)
- \(\mathrm{id} \ \mathrm{if} \ \mathrm{rel} \ \mathrm{id} \ \mathrm{goto} \ L\)
- \(L\)
- \(\mathrm{jump} \ L\)
Structure intermediate program to analyze

- Generate intermediate code
  - want to analyze code to apply optimizations
  - Control flow important
  - Data flow important
- Use control flow graph - node are basic blocks, edges show flow of control

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
  - All instructions will be executed if first one is

Basic Block Example

- Consider the basic block
  - L:
    - t := 2 * x
    - w := t + x
    - if w > 0 goto L’
- No way for (3) to be executed without (2) having been executed right before
  - We know 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 jump L_B
  - E.g., the execution can fall-through from block A to block B

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, 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 Java there are three granularities of optimizations
- Local optimizations
  - Apply to a basic block in isolation
- Global optimizations
  - Apply to a control-flow graph (method body) in isolation
  - 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
  \[ x := x * 0 \Rightarrow x := 0 \]
  \[ y := y ** 2 \Rightarrow y := y * y \]
  \[ x := x * 8 \Rightarrow x := x << 3 \]
  \[ x := x * 15 \Rightarrow t := x << 4; x := t - x \]

  (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
  \[ x := y \text{ op } z \]
  - 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

Control Flow 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)
Static Single Assignment Form (SSA)

- Some optimizations are simplified if each name occurs only once on the left-hand side of an assignment.
- Intermediate code can be rewritten to be in single assignment form.

\[
\begin{align*}
  x := z + y & \quad x_1 := z + y \\
  a := x & \quad a := x_1 \\
  x := 2 * x & \quad x_2 := 2 * x_1
\end{align*}
\]

- 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.
  - If any assignment have the same rhs, they compute the same value.
- Example:

\[
\begin{align*}
  x := y + z & \quad x := y + z \\
  \ldots & \quad \ldots \\
  w := y + z & \quad w := x \\
  \text{(the values of } x, y, \text{ and } z \text{ do not change in the \ldots code)}
\end{align*}
\]

Copy Propagation

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

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

- This does not make the program smaller or faster but might enable other optimizations.
  - Constant folding.
  - Dead code elimination: - see \( a := b \) above.
Copy Propagation and Constant Folding

- Example:
  - \( a := 5 \)
  - \( x := 2 \times a \Rightarrow x := 10 \)
  - \( y := x + 6 \Rightarrow y := 16 \)
  - \( t := x \times y \Rightarrow t := x \ll 4 \)

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
  - Dead = does not contribute to the program’s result

- Example: (\( a \) is not used anywhere else)
  - \( x := z + y \)
  - \( a := x \Rightarrow a := x \Rightarrow x := 2 \times x \)
  - \( x := 2 \times a \Rightarrow x := 2 \times z \times x \)

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

An Example

- Algebraic optimization:
  
  \[
  \begin{align*}
  a & := x ^ { \ast 2 } \\
  b & := 3 \\
  c & := x \\
  d & := c ^ { \ast } c \\
  e & := b ^ { \ast } 2 \\
  f & := a + d \\
  g & := e ^ { \ast } f
  \end{align*}
  \]
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:
  
  ```
  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:
  
  \[
  \begin{align*}
  a &:= x \times x \\
  b &= 3 \\
  c &:= x \\
  d &:= x \times x \\
  e &= 6 \\
  f &:= a + d \\
  g &:= e \times f
  \end{align*}
  \]

- 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:
  
  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: Final:

<table>
<thead>
<tr>
<th>Initial code</th>
<th>Final:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a := x \times 2)</td>
<td>(a := x \times x)</td>
</tr>
<tr>
<td>(b := 3)</td>
<td></td>
</tr>
<tr>
<td>(c := x)</td>
<td></td>
</tr>
<tr>
<td>(d := c \times c)</td>
<td></td>
</tr>
<tr>
<td>(e := b \times 2)</td>
<td></td>
</tr>
<tr>
<td>(f := a + d)</td>
<td>(f := a + a)</td>
</tr>
<tr>
<td>(g := e \times f)</td>
<td>(g := 6 \times f)</td>
</tr>
</tbody>
</table>

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 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
- Move to global optimizations

Local Optimization

Recall the simple basic-block optimizations
  - Constant propagation
  - Dead code elimination

\[
\begin{align*}
X &:= 3 \\
Y &:= Z \ast W \\
Q &:= X + Y \\
\end{align*}
\]

\[
\begin{align*}
X &:= 3 \\
Y &:= Z \ast W \\
Q &:= X + Y \\
\end{align*}
\]
Global Optimization

These optimizations can be extended to an entire control-flow graph.

\[
\begin{align*}
X &:= 3 \\
B &> 0 \\
Y &:= Z + W \\
Y &:= 0 \\
A &:= 2 \times X
\end{align*}
\]
Correctness

- How do we know it is OK to globally propagate constants?
- There are situations where it is incorrect:

\[ X := 3 \]
\[ B > 0 \]
\[ Y := Z + W \]
\[ X := 4 \]
\[ Y := 0 \]
\[ A := 2 * X \]

Correctness (Cont.)

To replace a use of \( x \) by a constant \( k \) we must know that:

On every path leading to the use of \( x \), the last assignment to \( x \) is \( x := k \)

Example 1 Revisited

\[ X := 3 \]
\[ B > 0 \]
\[ Y := Z + W \]
\[ X := 4 \]
\[ Y := 0 \]
\[ A := 2 * X \]
Example 2 Revisited

Discussion
- The correctness condition is not trivial to check
  - "All paths" includes paths around loops and through branches of conditionals
  - Checking the condition requires global analysis
    - An analysis of the entire control-flow graph