Local Optimization

CS440/540
Why Intermediate Languages?

- When to perform optimizations
  - On AST
    - Pro: Machine independent
    - Con: Too high level
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 reimplement optimizations when retargetting
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 reimplement optimizations when retargeting
  - On an intermediate language
    - Pro: Machine independent
    - Pro: Exposes optimization opportunities
    - Con: One more language to worry about
An Intermediate Language

\[
P \rightarrow S \ P | \epsilon \\
S \rightarrow id := id \ op \ id \\
| \ id := op \ id \\
| \ id := id \\
| \ push \ id \\
| \ id := pop \\
| \ if \ id \ relop \ id \ goto \ L \\
| \ L: \\
| \ jump \ L
\]

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

• **Basic blocks (BB)**
  - Split code into basic atomic units

• **Control-flow graphs (CFG)**
  - Connect the BBs together as a directed graph

• **Useful for representing intermediate code**
  - Use a graphical representation
  - Make control-flow explicit
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)
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. \( t := 2 \times x \)
  3. \( w := t + x \)
  4. if \( w > 0 \) goto \( L' \)
Basic Block Example

• Consider the basic block
  1. \( L: \)
  2. \( t := 2 \times x \)
  3. \( w := t + x \quad \Rightarrow \quad w := 3 \times x \)
  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 \( w := 3 \times x \)
  - 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 `jump L_B`
  - E.g., the execution can fall-through from block A to block B

- Frequently abbreviated as CFG
Control-Flow Graphs: Example

\[ x := 1 \\
 i := 1 \]

\[ L: \\
 x := x \times x \\
 i := i + 1 \\
 if \ i < 10 \ goto \ L \]
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 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 – correctness
  - The answer must still be the same
A Classification of Optimizations

- For languages like C and Cool there are three granularities of optimizations
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
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** (a.k.a. intra-procedural)
     - Apply to a control-flow graph (method body) in isolation
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** (a.k.a. intra-procedural)
     - Apply to a control-flow graph (method body) in isolation
  3. **Inter-procedural optimizations**
     - Apply across method boundaries
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** (a.k.a. intra-procedural)
     • 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
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.
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 cost
Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question
Algebraic Simplification

- Some statements can be deleted
  \[
  x := x + 0 \\
  x := x * 1
  \]
Algebraic Simplification

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

• Some statements can be simplified
  \[ x := x \times 0 \quad \Rightarrow \quad x := 0 \]
Algebraic Simplification

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

• Some statements can be simplified
  \[ x := x \times 0 \quad \Rightarrow \quad x := 0 \]
  \[ y := y \times 2 \quad \Rightarrow \quad y := y \times y \]
Algebraic Simplification

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

- Some statements can be simplified
  \[ x := x \times 0 \quad \Rightarrow \quad x := 0 \]
  \[ y := y \times 2 \quad \Rightarrow \quad y := y \times y \]
  \[ x := x \times 8 \quad \Rightarrow \quad x := x \ll 3 \]
  \[ x := x \times 15 \quad \Rightarrow \quad ? \]
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
- 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
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: \( x := 2 + 2 \Rightarrow x := 4 \)
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
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
- 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?
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 register occurs only once on the left-hand side of an assignment
Single Assignment Form

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

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

  \(b\) is a fresh register

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

- **Assume**
  - Basic block is in single assignment form
  - A definition $x :=$ is the first use of $x$ in a block
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:

  $x := y + z$               $x := y + z$
  $\cdots$                     $\Rightarrow$                 $\cdots$
  $w := y + z$               $w := x$

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

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

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

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

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

- If $w := x$ appears in a block, all subsequent uses of $w$ can be replaced with uses of $x$

- Example:
  
  $b := z + y$  \hspace{1cm} $b := z + y$
  
  $a := b$ \hspace{1cm} $a := b$
  
  $x := 2 * a$ \hspace{1cm} $x := 2 * b$

- 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:
  \[ a := 5 \]
  \[ x := 2 \times a \]
  \[ y := x + 6 \]
  \[ t := x \times y \]
Copy Propagation and Constant Folding

- Example:
  
  \[
  \begin{align*}
  a &:= 5 \\
  x &:= 2 \times a \\
  y &:= x + 6 \\
  t &:= x \times y
  \end{align*}
  \Rightarrow
  \begin{align*}
  a &:= 5 \\
  x &:= 10 \\
  y &:= 16 \\
  t &:= x \ll 4
  \end{align*}
  \]
Copy Propagation and Dead Code Elimination

If

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

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

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

Example: (a is not used anywhere else)

\[ x := z + y \]
\[ b := z + y \]
\[ a := x \quad \Rightarrow \quad a := b \]
\[ b := z + y \]
\[ x := 2 * a \quad \Rightarrow \quad x := 2 * b \]
Applying Local Optimizations

- Each local optimization does very little by itself
Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations interact
  - Performing one optimizations enables other opt.
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:**
  
  
  \[
  \begin{align*}
  a & := x \times x \\
  b & := 3 \\
  c & := x \\
  d & := c \times c \\
  e & := b \times 2 \\
  f & := a + d \\
  g & := e \times f
  \end{align*}
  \]
An Example

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

- Copy propagation:

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

- Copy propagation:
  
  \[
  a := x \times x \\
  b := 3 \\
  c := x \\
  d := x \times x \\
  e := 3 \ll 1 \\
  f := a + d \\
  g := e \times f
  \]
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:**
  
  ```
  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:
  
  \[
  \begin{align*}
  a &:= x \times x \\
  b &:= 3 \\
  c &:= x \\
  d &:= a \\
  e &:= 6 \\
  f &:= a + d \\
  g &:= e \times f
  \end{align*}
  \]
An Example

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

- Dead code elimination:
  
  \[
  a := x \times x \\
  b := 3 \\
  c := x \\
  d := a \\
  e := 6 \\
  f := a + a \\
  g := 6 \times f
  \]
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 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
Peephole Optimizations (Cont.)

- Write peephole optimizations as replacement rules

\[ i_1, ..., i_n \rightarrow j_1, ..., j_m \]

where the rhs is the improved version of the lhs

- Example:

  move $a \ $b, move $b \ $a \rightarrow \text{move } $a \ $b

  - Works if move $b \ $a is not the target of a jump
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` → Empty
  - These two together eliminate `addiu $a $a 0`
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 → Empty
  - 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