Local Optimizations

Lecture 11
Compiler Front-end and Back-end (Revisited)

1. Lexical analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator
5. Code Optimizer
6. Code Generator
7. Peephole Optimization

Analyses

Syntheses

Source Program

Target Program

1, 2, 3, 4, 5: Front-End
6, 7: Back-End
Organization of a Code Optimizer

- Front-end
- Code optimizer
  - Control-flow analysis
  - Data-flow analysis
  - Transformations
- Code generator
Optimization

• Most complexity in modern compilers is in the optimizer
  – Also by far the largest phase

• First, we need to discuss intermediate languages
Why Intermediate Languages?

• When should we 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
  - On an intermediate language
    • Pro: Machine independent
    • Pro: Exposes optimization opportunities
Intermediate Languages

- Intermediate language = high-level assembly
  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
  - Uses opcodes but some are higher level
    - Most opcodes correspond directly to assembly opcodes
Three-Address Intermediate Code

• Each instruction is of the form
  \[ x := y \text{ op } z \]
  \[ x := \text{ op } y \]
  - \( y \) and \( z \) are registers or constants
  - Common form of intermediate code

• The expression \( x + y \ast z \) is translated
  \[ t_1 := y \ast z \]
  \[ t_2 := x + t_1 \]
  - Each subexpression has a “name”
Generating Intermediate Code

• Similar to assembly code generation

• But use any number of IL registers to hold intermediate results
An Intermediate Language

\[
P \rightarrow SP | \epsilon \\
S \rightarrow id := id \text{ op } id \\
| id := \text{ op } id \\
| id := id \\
| \text{ if } id \text{ rel } id \text{ goto } L \\
| L: \\
| \text{ jump } L
\]

- 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 into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - A basic block is a single-entry, single-exit, straight-line code segment
Basic Block Example

- Consider the basic block
  1. L:
  2. \( t := 2 \times x \)
  3. \( w := t + x \)
  4. if \( w > 0 \) goto L’

- (3) executes only after (2)
  - 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 pass 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., execution can fall-through from block A to block B
Example of Control-Flow Graphs

- 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 resource utilization
  - Execution time (most often)
  - Code size
  - Memory usage
  - Disk access

• 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), 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 compilation time.
  - Some optimizations have low benefit.
  - Many fancy optimizations are all three.

• Goal: Maximum benefit for minimum 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**
  
  \[
  x := x \times 0 \Rightarrow x := 0 \\
  y := y^{**} 2 \Rightarrow y := y \times y \\
  x := x \times 8 \Rightarrow x := x \ll 3 \\
  x := x \times 15 \Rightarrow \dagger := x \ll 4; x := \dagger - x
  \]

  (on some machines \ll is faster than \times; but not on all!)

---

Prof. Aiken
Constant Folding

- Operations on constants can be computed at compile time
  - 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
Flow of Control Optimizations

• Eliminate unreachable basic blocks:
  - Code that is unreachable from the initial block
    • E.g., basic blocks that are not the target of any jump or “fall through” from a conditional

• 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)
Flow of Control Optimizations (Cont.)

- Why would unreachable basic blocks occur?
  - Debug mode
    ```
    #define DEBUG 0
    If (DEBUG) then ...
    ```
  - Libraries
  - Result of other optimizations
Single Assignment Form

• Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment

• Rewrite intermediate code in single assignment form

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

(b is a fresh register)
Common Subexpression Elimination

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

• Then
  - When two assignments have the same rhs, they compute the same value

• Example:
  $$x := y + z \quad \Rightarrow \quad w := y + z$$
  $$\Rightarrow \quad x := y + z \quad \Rightarrow \quad w := x$$

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

• If \( w := x \) appears in a block, replace subsequent uses of \( w \) with uses of \( x \)
  - Assumes single assignment form

• Example:
  \[
  b := z + y \quad \Rightarrow \quad b := z + y \\
  a := b \quad \Rightarrow \quad a := b \\
  x := 2 \times a \quad \Rightarrow \quad x := 2 \times b
  \]

• Only useful for enabling other optimizations
  - Constant folding
  - Dead code elimination
Copy Propagation and Constant Folding

- Example:

\[
a := 5
\]

\[
x := 2 \times a
\]

\[
y := x + 6
\]

\[
t := x \times y
\]

\[
\Rightarrow
\]

\[
a := 5
\]

\[
x := 10
\]

\[
y := 16
\]

\[
t := x \times 4
\]

\[
or
\]

\[
t := 160
\]
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
- \( \text{Dead} = \) does not contribute to the program’s result

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

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

turn to single assignment form  copy propagation  dead code elimination
Applying Local Optimizations

• Each local optimization does little by itself

• Typically optimizations interact
  – Performing one optimization enables another

• Optimizing compilers repeat optimizations until no improvement is possible
  – The optimizer can also be stopped at any point to limit compilation time
An Example

• Initial code:

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

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

- **Algebraic optimization:**

```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

Prof. Aiken
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 \ll 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:**

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

• Copy propagation:

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

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

- This is the final form
Loop Optimization

1. Code Motion

2. Reduction in Strength

3. Induction Variables elimination
Code Motion

• 'n := 2 + m' can be moved out of the loop
Reduction in Strength

- $i$ is increased by 1
- $t_1$ is increased by 8
- '*' can be replaced by '+'

L:
\[
\begin{align*}
    & t_1 := i \times 8 \\
    & t_2 := A[t_1] \\
    & dp := dp + t_2 \\
    & i := i + 1 \\
    & \text{if } i < n \text{ goto } L
\end{align*}
\]

\[
\begin{align*}
    & dp := 0 \\
    & i := 1 \\
    & n := 2 + m \\
    & t_1 := 0
\end{align*}
\]

\[
\begin{align*}
    & dp := 0 \\
    & i := 1 \\
    & n := 2 + m \\
    & t_1 := 0
\end{align*}
\]

\[
\begin{align*}
    & dp := 0 \\
    & i := 1 \\
    & n := 2 + m \\
    & t_1 := 0
\end{align*}
\]
Induction Variables Elimination

- i and t1 here are regarded as 'induction variables'
- i can be removed and t1 is used instead

\[ dp := 0 \]
\[ i := 1 \]
\[ n := 2 + m \]
\[ t1 := 0 \]

\[ L: \]
\[ t1 := t1 + 8 \]
\[ t2 := A[t1] \]
\[ dp := dp + t2 \]
\[ i := i + 1 \]
\[ if \ i < n \ goto \ L \]

\[ dp := 0 \]
\[ n := 2 + m \]
\[ t3 := 8 * n \]
\[ t1 := 0 \]

\[ L: \]
\[ t1 := t1 + 8 \]
\[ t2 := A[t1] \]
\[ dp := dp + t2 \]
\[ if \ t1 < t3 \ goto \ L \]

- 'i := i + 1' is then a dead code and can be removed
Peephole Optimizations on Assembly Code

• These optimizations work on intermediate code
  - Target independent
  - But they can be applied on assembly language also

• Peephole optimization is effective 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:
  \[ \text{move } $a \hspace{1em} \text{move } $b $a \rightarrow \text{move } $a $b} \]
  - Works if \text{move } $b $a is not the target of a jump

• Another example
  \[ \text{addiu } $a $a i, \text{addiu } $a $a j \rightarrow \text{addiu } $a $a i+j \]
Peephole Optimizations (Cont.)

3AC

\[
\begin{align*}
  x & := y + 1; \\
  z & := x + 2;
\end{align*}
\]

**Machine Code**

```
move  $a y
addiu $a $a 1
move  x $a
move  $a x  /* This move statement is not needed */
addiu $a $a 2
move  $a z
```
Use of specialized instructions

\[
\begin{align*}
\text{move} & \quad \text{\$a} \quad \text{\$a} \\
\text{addiu} & \quad \text{\$a} \quad \text{\$a} \quad 1 \\
\text{move} & \quad \text{\$a} \quad \text{\$a}
\end{align*}
\Rightarrow \quad \text{increment \$a}
\]
Peephole Optimizations (Cont.)

Some machine codes can be deleted

\[
\text{multu } a \ a \ 1 \\
\text{addiu } a \ a \ 0
\]

Using shift to left instead of multiplication by powers of 2

Using shift to right instead of division into powers of 2
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
Question?

Which of the following are valid local optimizations for the given basic block? Assume that only $g$ and $x$ are referenced outside of this basic block.

- Copy propagation: Line 4 becomes $d := a \times b$.
- Common subexpression elimination: Line 5 becomes $e := d$.
- Dead code elimination: Line 3 is removed.
- After many rounds of valid optimizations, the entire block can be reduced to $g := 5$. 

```
1  a := 1
2  b := 3
3  c := a + x
4  d := a \times 3
5  e := b \times 3
6  f := a + b
7  g := e - f
```