CSE450
Translation of Programming Languages

Lecture 24: Optimization: Introduction, Basic Blocks, Local Optimizations
Structure of a Compiler

Front End

Back End

Source Language

Lexical Analyzer

Syntax Analyzer

Semantic Analyzer

Int. Code Generator

Intermediate Code

Code Optimizer

Target Code Generator

Target Language

Today!

Project 1 - ✔

Project 2 - ✔

Project 4 - ✔

Project 3 - ✔

Project 6 - ✔
Why are Intermediate Codes used?

An intermediate code is the layer between the front-end and back-end of the compiler.

It is a simplified assembly language that is easy to create, and easy to convert to other languages.

What else needs to be done?
Why are Intermediate Codes used?

An intermediate code is the layer between the front-end and back-end of the compiler.

It is a simplified assembly language that is easy to create, and easy to convert to other languages.

What if we add intermediate code?
Why are Intermediate Codes used?

An intermediate code is the layer between the front-end and back-end of the compiler.

It is a simplified assembly language that is easy to create, and easy to convert to other languages.

Can this make it easier to write our compiler?
Why are Intermediate Codes used?

An intermediate code is the layer between the front-end and back-end of the compiler.

It is a simplified assembly language that is easy to create, and easy to convert to other languages.
Optimization Overview

- Optimization seeks to improve a program's utilization of one or more resources:
  - Execution time
  - Memory footprint
  - Code size
  - Network messages
  - Battery power
  - etc.
- Must not alter the program's computation results
Classes of Optimizations

- High level languages typically allow 3 types of optimizations:
  - **Local** Optimizations - Basic Block level
  - **Global** Optimizations - Control Flow Graph level (method body)
  - **Inter-procedural** Optimizations - Across method boundaries, and sometimes across translation unit boundaries

- **Most** compilers do **local** optimizations
- **Many** do **global** optimizations
- **Few** do **inter-procedural** optimizations
Cost of Optimizations

- In practice, a conscious decision is made to not implement all possible optimizations
  - Some are difficult to implement
  - Some have costly compilation times
  - Some are hard to prove correct
- Goal: Maximize improvement, minimize cost
Local Optimizations

- Simplest form of optimizations, no need to analyze the whole procedure, just a basic block

- A basic block is a segment of code with no jumps or jump targets (labels) inside
  - Jump targets must be the first instruction
  - Jumps must be at the end of the block
  - Basic blocks can be single expressions
Identifying Basic Blocks

- **Input**: Intermediate Code
- **Output**: A list of basic blocks, with each intermediate code instruction assigned to exactly one block
- **Method**: Determine leader instructions
  - First instruction of the intermediate code is a leader instruction
  - Any instruction that is a target of a jump (labeled)
  - Any instruction that follows a conditional jump
  - For each leader, basic block is itself plus all instructions that follow until the next leader instruction
Next Use Information

- Knowing when a variable will be used next is important for generating good code.
- Several optimizations need next use information.
- Definition:
  - Suppose statement $i$ assigns to variable $x$
  - If subsequent statement $j$ has $x$ as an operand and there are no intervening assignments to $x$, statement $j$ uses $x$. Variable $x$ is live at statement $i$. 
Static Single Assignment Form

- Static single assignment (SSA) form is an intermediate representation where each variable is assigned exactly once.
- Current intermediate code can be transformed into SSA form by splitting variables into versions of the original, identified by a subscript.
- SSA form enables and simplifies the application of various optimizations.
  - For example: some versions of a variable may be eliminated (optimized away), while others need to be stored for future use.
Calculating Next Use for Basic Blocks

- **Input**: Basic Block - assume that the symbol table shows the last version of all non-temporary variables as live after the block
- **Output**: Each statement $i: x = y \text{ op } z$ has liveness and next use information attached for $x$, $y$, and $z$
- **Method**: Start at the last statement, scan backwards
  - Attach to statement $i$ the current info found for $x$, $y$, and $z$
  - Mark $x$ as not live, no next use
  - Mark $y$ and $z$ as live and next uses of $y$ and $z$ to statement $i$
- In SSA form, versions are independent
Algebraic Simplification

- Some statements can simply be removed (algebraic identities)
  \[ x = x + 0 \]
  \[ x = x \times 1 \]

- Some can be simplified (reduction in strength)
  \[ x = x \times 0 \] \[ \Rightarrow \] \[ x = 0 \]
  \[ x = x \wedge 2 \] \[ \Rightarrow \] \[ x = x \times x \]
  \[ x = x \times 8 \] \[ \Rightarrow \] \[ x = x \ll 3 \]
  \[ x = x \times 15 \] \[ \Rightarrow \] \[ t = x \ll 4; \ x = t - x \]

- Shift operations are much faster than multiplication on most machines
Constant Folding

- Many 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 (or set to 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 removed.
Copy propagation

- Replacing occurrences of a variable that is being directly assigned with the value of the assignment

\[
\begin{align*}
y &= x \\
z &= 6 \times y
\end{align*} \quad \rightarrow \quad
\begin{align*}
y &= x \\
z &= 6 \times x
\end{align*}

- Copy propagation is a cleanup optimization

- Some optimizations require copy propagation to be run, in order to achieve their intended efficiency increase
Common Subexpression Elimination

- Identify nodes with the same source operands and the same operator.
- The result of the first operation can be used in place of the second.

\[
\begin{align*}
    a &= b + c \\
    b &= a - d \\
    c &= b + c \\
    d &= a - d
\end{align*}
\]

\[
\begin{align*}
    a &= b + c \\
    d &= a - d \\
    c &= d + c
\end{align*}
\]

- Note that if both of the results are live after the current block, must still store the result in both.
Dead Code Elimination

- Remove any instructions that have no live variables attached

<table>
<thead>
<tr>
<th>a = b + c</th>
<th>a and b live</th>
<th>a = b + c</th>
<th>a = b + c</th>
</tr>
</thead>
<tbody>
<tr>
<td>b = b - d</td>
<td></td>
<td>b = b - d</td>
<td>b = b - d</td>
</tr>
<tr>
<td>c = c + d</td>
<td>c and e are not</td>
<td>c = c + d</td>
<td></td>
</tr>
<tr>
<td>e = b + c</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Applying Local Optimizations

- Local optimizations do little on their own
- Typically optimizations interact: performing one enables others to be used
- Optimizing compilers will usually repeat optimizations until no further improvement is possible
  - The optimizer can also be stopped at any time to limit compilation time
Example

- Initial un-optimized code:

```plaintext
a = x ^ 2
b = 3
c = x
d = c * c
e = b * 2
f = a + d
g = e * f
```

...
Example

- **Algebraic Simplification:**
  
  ```plaintext
  a = x ^ 2
  b = 3
  c = x
  d = c * c
  e = b * 2
  f = a + d
  g = e * f
  ```
Example

- Algebraic Simplification:

  
  \[ a = x^2 \]
  \[ b = 3 \]
  \[ c = x \]
  \[ d = c \times c \]
  \[ e = b \times 2 \]
  \[ f = a + d \]
  \[ g = e \times f \]

  ...
Example

- Copy Propagation:

```plaintext
a = x * x
b = 3
c = x
d = c * c
e = b << 1
f = a + d
g = e * f
...```

...
Example

- Copy Propagation:
  
  ```
  ...  
  a = x * x
  b = 3
  c = x
  d = c * c
  e = b << 1
  f = a + d
  g = e * f
  ...  
  ```
Example

- Copy Propagation:

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

- Copy Propagation:

  ```
  a = x * x
  b = 3
  c = x
  d = c * c
  e = 3 << 1
  f = a + d
  g = e * f
  ```
Example

- Copy Propagation:

  
  ```
  a = x * x
  b = 3
  c = x
  d = x * x
  e = 3 << 1
  f = a + d
  g = e * f
  ```

  ...
Example

- Constant Folding:
  
  ```
  a = x * x
  b = 3
  c = x
  d = x * x
  e = 3 << 1
  f = a + d
  g = e * f
  ```
Example

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

- Constant Folding:

```
  a = x * x
  b = 3
  c = x
  d = x * x
  e = 6
  f = a + d
  g = e * f
```

...
Example

- Common Sub-expression Elimination:
  
  ... 
  a = x * x
  b = 3
  c = x
  d = x * x
  e = 6
  f = a + d
  g = e * f
  ...
Example

- Common Sub-expression Elimination:

  \[
  \begin{align*}
  a &= x \times x \\
  b &= 3 \\
  c &= x \\
  d &= x \times x \\
  e &= 6 \\
  f &= a + d \\
  g &= e \times f \\
  \end{align*}
  \]
Example

- Common Sub-expression Elimination:

  \[
  \cdots \\
  a = x \ast x \\
  b = 3 \\
  c = x \\
  d = a \\
  e = 6 \\
  f = a + d \\
  g = e \ast f \\
  \cdots
  \]
Example

- Copy Propagation (round 2):
  ...
  \[ a = x \times x \]
  \[ b = 3 \]
  \[ c = x \]
  \[ d = a \]
  \[ e = 6 \]
  \[ f = a + d \]
  \[ g = e \times f \]
  ...

Example

- Copy Propagation (round 2):

  
  $a = x \times x$
  $b = 3$
  $c = x$
  $d = a$
  $e = 6$
  $f = a + d$
  $g = e \times f$

  
  ...
Example

- Copy Propagation (round 2):
  
  
  \[ a = x \times x \]
  \[ b = 3 \]
  \[ c = x \]
  \[ d = a \]
  \[ e = 6 \]
  \[ f = a + a \]
  \[ g = e \times f \]
  
  \[ \ldots \]
Example

- Copy Propagation (round 2):

  \[\ldots\]

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

  \[\ldots\]
Example

- Copy Propagation (round 2):
  
  \[
  \begin{align*}
  a &= x \times x \\
  b &= 3 \\
  c &= x \\
  d &= a \\
  e &= 6 \\
  f &= a + a \\
  g &= 6 \times f \\
  \ldots
  \end{align*}
  \]
Example

- Dead Code Elimination:

  ```
  ... 
  a = x * x
  b = 3
  c = x
  d = a
  e = 6
  f = a + a
  g = 6 * f
  ... 
  ```
Example

- Dead Code Elimination:

```
... 
  a = x * x 
  b = 3 
  c = x 
  d = a 
  e = 6 
  f = a + a 
  g = 6 * f 
... 
```
Example

- Dead Code Elimination:
  ...
  a = x * x

  f = a + a
  g = 6 * f
  ...

Example

- The Final Program:

  ```
  a = x * x
  f = a + a
  g = 6 * f
  ```
What makes a good Intermediate Code/Representation?

- Can easily translate an AST into the intermediate code (IC)
- Can easily convert the IC into any target assembly language

What does this entail?

- Narrow interface - few commands, yet complete
- Easy to perform optimizations
- Flexible commands - easily retargeted to multiple assembly languages
Intermediate Code Issues

- Intermediate code translations are straightforward, but inefficient!
  - Lots of temporaries
  - Lots of instructions
  - Repeated operations
- Can we do this more intelligently?
  - Should we worry about it?