Lecture 36: Local Optimization

[Adapted from notes by R. Bodik and G. Necula]
Introduction to Code Optimization

*Code optimization* is the usual term, but is grossly misnamed, since code produced by “optimizers” is not optimal in any reasonable sense. *Program improvement* would be more appropriate.

Topics:

- Basic blocks
- Control-flow graphs (CFGs)
- Algebraic simplification
- Constant folding
- Static single-assignment form (SSA)
- Common-subexpression elimination (CSE)
- Copy propagation
- Dead-code elimination
- Peephole optimizations
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 the beginning.
  - Cannot jump within a basic block, except at end.
  - Therefore, each instruction in a basic block is executed after all
    the preceding instructions have been executed
Basic-Block Example

• Consider the basic block

  1. L1:
  2. \( t := 2 \times x \)
  3. \( w := t + x \)
  4. if \( w > 0 \) goto L2

• 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?
Control-Flow Graphs (CFGs)

• A control-flow graph is a directed graph with basic blocks as nodes
• There is an edge from block $A$ to block $B$ if the execution can flow from the last instruction in $A$ to the first instruction in $B$:
  - The last instruction in $A$ can be a jump to the label of $B$.
  - Or execution can fall through from the end of block $A$ to the beginning of block $B$. 
Control-Flow Graphs: Example

- The body of a method (or procedure) can be represented as a CFG
- 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 should not depart from the programming language’s semantics

- So if the semantics of a particular program is deterministic, optimization must not change the answer.

- On the other hand, some program behavior is undefined (e.g., what happens when an unchecked rule in C is violated), and in those cases, optimization may cause differences in results.
A Classification of Optimizations

• For languages like C and 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 (single function body) in isolation.
  3. **Inter-procedural optimizations**: Apply across function boundaries.

• Most compilers do (1), many do (2) and very few do (3)

• Problem is expense: (2) and (3) typically require superlinear time. Can usually handle that when limited to a single function, but gets problematic for larger program.

• In practice, generally don’t implement fanciest known optimizations: some are hard to implement (esp., hard to get right), some require a lot of compilation time.

• The goal: maximum improvement with minimum cost.
Local Optimizations: Algebraic Simplification

• Some statements can be deleted
  
  \[
  \begin{align*}
  x & := x + 0 \\
  x & := x * 1
  \end{align*}
  \]

• Some statements can be simplified or converted to use faster operations:

<table>
<thead>
<tr>
<th>Original</th>
<th>Simplified</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x := x * 0)</td>
<td>(x := 0)</td>
</tr>
<tr>
<td>(y := y ** 2)</td>
<td>(y := y * y)</td>
</tr>
<tr>
<td>(x := x * 8)</td>
<td>(x := x &lt;&lt; 3)</td>
</tr>
<tr>
<td>(x := x * 15)</td>
<td>(t := x &lt;&lt; 4; x := t - x)</td>
</tr>
</tbody>
</table>

(on some machines \(<<\) is faster than \(*\); but not on all!)
Local Optimization: Constant Folding

- Operations on constants can be computed at compile time.
- Example: \( x := 2 + 2 \) becomes \( x := 4 \).
- Example: \( \text{if } 2 < 0 \text{ jump L} \) becomes a no-op.
- When might constant folding be dangerous?
Global Optimization: Unreachable code elimination

- Basic blocks that are not reachable from the entry point of the CFG may be eliminated.

- Why would such basic blocks occur?

- Removing unreachable code makes the program smaller (sometimes also faster, due to instruction-cache effects, but this is probably not a terribly large effect.)
Single Assignment Form

- Some optimizations are simplified if each assignment is to a temporary that has not appeared already in the basic block.

- Intermediate code can be rewritten to be in *(static)* single assignment *(SSA)* form:

  \[
  \begin{align*}
  x & := a + y \\
  a & := x \\
  x & := a \times x \\
  b & := x + a \\
  a_1 & := x \\
  x_1 & := a_1 \times x \\
  b & := x_1 + a_1
  \end{align*}
  \]

  where \(x_1\) and \(a_1\) are fresh temporaries.
Common SubExpression (CSE) Elimination in Basic Blocks

- A **common subexpression** is an expression that appears multiple times on a right-hand side in contexts where the operands have the same values in each case (so that the expression will yield the same value).

- Assume that the basic block on the left is in single assignment form.

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

- That is, if two assignments have the same right-hand side, we can replace the second instance of that right-hand side with the variable that was assigned the first instance.

- How did we use the assumption of single assignment here?
Copy Propagation

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

- Example:

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

- This does not make the program smaller or faster but might enable other optimizations. For example, if \( a \) is not used after this statement, we need not assign to it.

- Or consider:

  \[
  \begin{align*}
  b &:= 13 \quad b := 13 \\
  x &:= 2b \quad x := 2*13
  \end{align*}
  \]

  which immediately enables constant folding.

- Again, the optimization, as described, won’t work unless the block is in single assignment form.
Another Example of Copy Propagation and Constant Folding

\[
\begin{align*}
a & := 5 \\
x & := 2 \times a \\
y & := x + 6 \\
t & := x \times y
\end{align*}
\]
Dead Code Elimination

• If that statement \( w := \text{rhs} \) appears in a basic block and \( w \) does not appear anywhere else in the program, we say that the statement is **dead** and can be eliminated; it does not contribute to the program’s result.

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

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

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

• How have I used SSA here?
Applying Local Optimizations

• As the examples show, each local optimization does very little by itself.

• Typically, optimizations interact: performing one optimization enables others.

• So typical optimizing compilers repeatedly perform optimizations until no improvement is possible, or it is no longer cost effective.
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 II: Algebraic simplification

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

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

\begin{align*}
  &\textbf{a} := x * x \\
  &\textbf{b} := 3 \\
  &\textbf{c} := x \\
  &\textbf{d} := c * c \\
  &\textbf{e} := b + b \\
  &\textbf{f} := a + d \\
  &\textbf{g} := e * f
\end{align*}
An Example: Copy propagation

a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
f := a + d
g := e * f
An Example: Constant folding

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

\[ \begin{align*}
a & := x \ast x \\
b & := 3 \\
c & := x \\
d & := x \ast x \\
e & := 6 \\
f & := a + d \\
g & := e \ast f
\end{align*} \]
An Example: Common Subexpression Elimination

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

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

a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
An Example: Dead code elimination

\[
\begin{align*}
a & := x \times x \\
b & := 3 \\
c & := x \\
d & := a \\
e & := 6 \\
f & := a + a \\
g & := 6 \times f
\end{align*}
\]
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.
- *Peephole optimization* is a technique for improving assembly code directly
  - The "peephole" is a short subsequence of (usually contiguous) instructions, either contiguous, or linked together by the fact that they operate on certain registers that no intervening instructions modify.
  - The optimizer replaces the sequence with another equivalent, but (one hopes) better one.
  - Write peephole optimizations as replacement rules
    \[ i_1; \ldots; \text{in} \Rightarrow j_1; \ldots; j_m \]
    possibly plus additional constraints. The \( j \)'s are the improved version of the \( i \)'s.
Peephole optimization examples:

- We'll use the notation '@A' for pattern variables.
- Example:
  
  \[
  \text{movl } %@a \ %@b; \ L: \text{ movl } %@b \ %@a \Rightarrow \text{ movl } %@a \ %@b
  \]
  
  assuming \(L\) is not the target of a jump.

- Example:

  \[
  \text{addl } %@k1, \ %@a; \text{ movl } @k2(\%@a), \ %@b
  \Rightarrow \text{ movl } @k1+@k2(\%@a), \ %@b
  \]
  
  assuming \(%@a\) is “dead“.

- Example (PDP11):

  \[
  \text{mov } %@I, \ @I(@ra) \Rightarrow \text{ mov (r7), @I(@ra)}
  \]

  This is a real hack: we reuse the value \(I\) as both the immediate value and the offset from \(ra\). On the PDP11, the program counter is \(r7\).

- As for local optimizations, peephole optimizations need to be applied repeatedly to get maximum effect.
Problems:

• Serious problem: what to do with pointers? Problem is *aliasing*: two names for the same variable:
  - As a result, *t may change even if local variable t does not and we never assign to *t*.
  - Affects language design: rules about overlapping parameters in Fortran, and the `restrict` keyword in C.
  - Arrays are a special case (address calculation): is A[i] the same as A[j]? Sometimes the compiler can tell, depending on what it knows about i and j.

• What about globals variables and calls?
  - Calls are not exactly jumps, because they (almost) always return.
  - Can modify global variables used by caller