Code-Improving Transformations ("Optimization")

Why Optimize

• To encourage good programming
  – Reduce the penalty for writing clean, well-organized code (and so reduce bugs)

• To simplify code generation
  – Reduce the penalty for generating simple code (and so reduce bugs)

• To make programs fast and small
  – Least important of the three, but important in some domains

Classes of Optimization

• Local
  – Within a single basic block (straight-line code)
  – Example: Common-subexpression elimination

• "Global"
  – Across blocks, within a procedure (method)
    • So we get to badly chosen names in one term, "global optimization"
  – Example: Lifting code from a loop

• Inter-procedural
  – Across procedures
  – Examples: Procedure in-lining and specialization

Some Local Optimizations

• Copy propagation
  • Substituting in a known value

• Constant folding
  • A kind of partial evaluation

• Dead code elimination
  • Often exposed by other optimizations

• Algebraic identities
  • Recognizing special cases, e.g., x**2 = x*x
Algebraic Identity

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

Copy Propagation

\[ a := x \times x \]
\[ b := 3 \]
\[ c := x \]
\[ d := c \times c \]
\[ e := b \ll 1 \]
\[ f := a + d \]
\[ g := e \times f \]

Constant Folding

\[ a := x \times x \]
\[ b := 3 \]
\[ c := x \]
\[ d := x \times x \]
\[ e := 3 \ll 1 \]
\[ f := a + d \]
\[ g := e \times f \]

Common Sub-Expression Elimination

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

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

Dead Code Elimination

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

Observations

- Transformations cascade
  - One transformation enables another, etc.
  - Some (like copy propagation) are useful primarily to expose others
- Continue applying until no more changes
  - Repetition also needed if they are applied as "peephole" optimizations on object code

Some “Global” Transformations

- Copy propagation, constant propagation, dead code elimination, ...
  - Versions of the local optimizations, but using data flow analysis to test applicability
- Code motion
  - Hoist invariant code from a loop
    - Most important
  - Loop unrolling
    - Exposes other optimizations
Induction Variables

```c
for (i=0; i < 10; ++i) {
    t = 2*i;
    A[i] = B[t];
    t' = t' + 2;
}
```

Do you see the multiplication by 4 that we are going to avoid by recognizing induction variables? (Semi-trick question.)

Inter-procedural: Inlining

- Inlining: Copy called procedure into caller (treat call like a macro)
  - When is this safe?
- Why?
  - Eliminate call overhead
  - Expose optimization opportunities (specialization)

Notes on Optimization

- Lots and lots and lots ...
  - 20+ years of compiler research on optimization, including many for special situations (parallelizing, caching improvement, ...)
- Wider scope => ++profit, ++cost
  - Doing more with more; but when is it worthwhile?
- Optimizations interact
  - Cascading (good), interfering (bad); no one fully understands how to order them (yet)