CODE OPTIMIZATION

• Really “improvement” rather than “optimization;” results are seldom optimal.
• Remove inefficiencies in user code and (more importantly) in compiler-generated code.
• Can be applied at several levels, chiefly intermediate or assembly code.
• Can operate at several levels:
  - “Peephole” : very local IR or assembly
  - “Local” : within basic blocks
  - “Global” : entire procedures
  - “Interprocedural” : entire programs (maybe even multiple source files)
• Theoretical tools: graph algorithms, control and data flow analysis.
• Practical tools: few.
• Most of a serious modern compiler is devoted to optimization.

PEEPHOLE OPTIMIZATIONS

• Look at short sequences of statements (in IR or assembly code)
• Correct inefficiencies produced by excessively local code generation strategies.
• Repeat!
• Same effect can often be achieved by using smarter (but hence more complex) code generation in the first place.

EXAMPLE PEEPHOLE OPTIMIZATIONS

• Redundant instructions
  mov %f0, %f2
  mov %f0, %f2 ; ok to remove if in same basic block

• Unreachable code
  LOOP IF x > 2 THEN EXIT ELSE X := X + 1 END;
  L1: IF X > 2 GOTO L2
      GOTO L3
  L2: GOTO L4
      GOTO L1 ; never executed
  L3: X := X + 1
      GOTO L1
  L4: ...

• Flow-of-control fixes: remove jumps to jumps, e.g.,
  L1: IF X > 2 GOTO L4
      X := X + 1
      GOTO L1
  L4: ...
**MORE PEEP Hole OPTIMIZATIONS**

- **Algebraic Simplification**
  
  \[
  x + 0 = 0 + x = x \\
  x - 0 = x \\
  x \times 1 = 1 \times x = x \\
  x/1 = x
  \]

- **Strength Reduction**

  Target hardware may have cheaper ways to do certain operations.

  E.g., multiplication or division by a power of 2 is better done by shifting.

  \[
  \text{imull} \ $8, \%l2 \ \Rightarrow \ \text{sall} \ $3, \%l2
  \]

- **Use of machine idioms**

  Target hardware may have quirks/features that make certain sequences faster:

  \[
  \text{imull} \ $8, \%l2 \\
  \text{addl} \ \%l3, \%l2 \\
  \text{addl} \ $20, \%l2 \ \Rightarrow \ \text{leal} \ 20(\%l3, \%l2, 8)
  \]

**LOCAL (BASIC BLOCK) OPTIMIZATIONS**

- Typically applied to IR, **after** addressing is made explicit, but **before** machine dependencies appear.

- Most important: **Common Subexpression Elimination** (CSE)

  \[
  i := j + 1 \\
  a[i] := a[i] + j + 1
  \]

  Avoid duplicating the code for \(j+1\) or the addressing code for \(a[i]\). One technique: build **directed acyclic graph** (DAG) for basic block.

- **Copy Propagation**

  \[
  a := b + 1 \ \Rightarrow \ a := b + 1 \\
  c := a \quad c := c \quad \text{maybe can now omit} \\
  d := c \quad d := d
  \]

- **Algebraic Identities**

  E.g., use associativity and commutativity of +

  \[
  a := b + c \ \Rightarrow \ a := b + c \\
  b := c + d + b \ b := b + c + d \ ; \text{now do CSE}
  \]

**GLOBAL (FULL PROCEDURE) OPTIMIZATION**

Loop optimizations are most important.

- **Code motion**: “hoist” expensive calculations above the loop.

- **Use induction variables** and reduction in strength. Change only one index variable on each loop iteration, and choose one that’s cheap to change.

  Also continue to apply CSE, copy propagation, dead code elimination, etc. on global scale.

  Based on **control flow graph**.

  Example: Computing dot product (assuming \(i, a\) local; \(b, c\) global). Local CSE already performed within basic blocks.

  \[
  a = 0; \\
  \text{for} \ (i = 0; i < 20; i++) \\
  \ a = a + b[i] * c[i]; \\
  \text{return} \ a;
  \]

  Example IR...
Example: Effects of Global Optimization

- Promote locals \( a \) and \( i \) to registers.
- Induction variable: replace \( i \) with \( i \times 4 \), thus reducing strength of per-loop operation; adjust test accordingly.
- Hoist all constants out of loop.

Results on example:

B1
\[
\text{t1 := const 0}
\]
\[
\text{t2 := addr a}
\]
\[
\text{t3 := addr i}
\]
\[
\text{t4 := const 4}
\]
\[
\text{t5 := addr i}
\]
\[
\text{t6 := *t5}
\]
\[
\text{t7 := const 20}
\]
\[
\text{if t6 >= t7 goto L4}
\]

B2
\[
\text{L2:}
\]
\[
\text{t5 := addr i}
\]
\[
\text{t6 := *t5}
\]
\[
\text{t7 := const 20}
\]
\[
\text{if t6 >= t7 goto L4}
\]

B3
\[
\text{B3 t8 := addr a}
\]
\[
\text{t9 := *t8}
\]
\[
\text{t10 := addr b}
\]
\[
\text{t11 := addr i}
\]
\[
\text{t12 := *t11}
\]
\[
\text{t13 := const 4}
\]
\[
\text{t14 := t12 + t13}
\]
\[
\text{t15 := t10 + t14 ; &b[i]}
\]
\[
\text{t16 := *t15}
\]
\[
\text{t17 := addr c}
\]
\[
\text{t18 := t17 + t14 ; &c[i]}
\]
\[
\text{t19 := *t18}
\]
\[
\text{t20 := t16 + t19}
\]
\[
\text{t21 := t9 + t20}
\]
\[
\text{t22 := const 1}
\]
\[
\text{t23 := t12 + t22}
\]
\[
\text{*t11 := t23}
\]
\[
\text{goto L2}
\]

B4
\[
\text{L4:}
\]
\[
\text{t24 := addr a}
\]
\[
\text{t25 := *t24}
\]
\[
\text{return t25}
\]

Interprocedural Optimization

- Procedure inlining is most important.
- Replace a procedure call with a copy of the procedure body (including initial assignments to parameters).
- Applicable when body is not too big, or is called only once.

Benefits:
- Saves overhead of procedure entry/exit, argument passing, etc.
- Permits other optimizations to work over procedure boundaries.
- Particularly useful for languages that encourage use of small procedures (e.g. OO state get/set methods).

Cost:
- Risk of "code explosion."
- Doesn’t work when callee is not statically known (e.g. OO dynamic dispatch or FP first-class calls).