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!

• Redundant instructions

  `fmovd %f0, %f2`
  `fmovd %f0, %f2 ; ok to remove if in same B.B.`

• 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 Peephole Optimizations

• Algebraic Simplification

  `x + 0 = 0 + x = x`
  `x - 0 = x`
  `x * 1 = 1 * 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.

  `umul %l1, 8, %l2 ⇒ sll %l1, 3, %l2`

• Use of machine idioms

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

  `set 372, %l1
  add %l2, %l1, %l3 ⇒ add %l2, 372, %l3`
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 \( a[i] \).

- Copy Propagation

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

- Algebraic Identities

E.g., use associativity and commutativity of +

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

- Iterate! Optimizations enable further optimizations.

- Primary technique: build directed acyclic graph (DAG) for basic block.

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.

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

IR for Dot Product

<table>
<thead>
<tr>
<th>Basic Block</th>
<th>After CSE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>t1 := const 0</td>
</tr>
<tr>
<td></td>
<td>t2 := addr a</td>
</tr>
<tr>
<td></td>
<td>*t2 := t1</td>
</tr>
<tr>
<td></td>
<td>t3 := addr i</td>
</tr>
<tr>
<td></td>
<td>*t3 := t1</td>
</tr>
<tr>
<td>B2</td>
<td>t5 := addr i</td>
</tr>
<tr>
<td></td>
<td>t6 := *t5</td>
</tr>
<tr>
<td></td>
<td>t7 := const 20</td>
</tr>
<tr>
<td></td>
<td>if t6 &gt;= t7 goto L4</td>
</tr>
<tr>
<td>B3</td>
<td>t8 := addr a</td>
</tr>
<tr>
<td></td>
<td>t9 := *t8</td>
</tr>
<tr>
<td></td>
<td>t10 := addr b</td>
</tr>
<tr>
<td></td>
<td>t11 := addr i</td>
</tr>
<tr>
<td></td>
<td>*t11 := t23</td>
</tr>
</tbody>
</table>

PSU CS322 W'05 Lecture 11 © Andrew Tolmach 1992-2005
Example: effects of global optimization

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

```plaintext
t1 := const 0
t2 := addr a
*t2 := t1

t3 := addr i
*t3 := t1

t9 := t1 ; a
t6 := t1 ; i * 4

t13 := const 4
t7 := const 80
t8 := addr a
t10 := addr b
t17 := addr c

L2:

t5 := addr i
t6 := *t5

if t6 >= t7 goto L4

if t6 >= t7 goto L4

if t6 >= t7 goto L4

L4:

t24 := addr a

t25 := *t24

return t25
```

```plaintext
L2:

t8 := addr a
t9 := *t8
t10 := addr b
t11 := addr i
t12 := *t11
t13 := const 4
t14 := t12 * t13
t15 := t10 + t14

t16 := *t15
t17 := *t15
t18 := t17 + t14
t19 := *t18
t20 := t16 * t19
t21 := t9 + t20

*t8 := t21
t22 := const 1
t23 := t12 + t22

t15 := t10 + t6

t16 := *t15

t17 := addr c
t18 := t17 + t14
t19 := *t18
t20 := t16 * t19
t21 := t9 + t20

*t8 := t21
t22 := const 1
t23 := t12 + t22

t6 := t6 + t13

t1 := const 0
```