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

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

- Unreachable code

```c
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.,

```c
L1: IF X > 2 GOTO L4
    X := X + 1
    GOTO L1
L4: ...
```
More Peephole Optimizations

- **Algebraic Simplification**

\[
\begin{align*}
  x + 0 &= 0 + x = x \\
  x - 0 &= x \\
  x * 1 &= 1 * x = x \\
  x/1 &= x
\end{align*}
\]

- **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.

\[
\text{umul} \ %l1, 8, \ %l2 \ \Rightarrow \ \text{sll} \ %l1, 3, \ %l2
\]

- **Use of machine idioms**

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

\[
\begin{align*}
  \text{set} \ 372, \ %l1 \\
  \text{add} \ %l2,%l1,%l3 \ \Rightarrow \ \text{add} \ %l2,372,%l3
\end{align*}
\]
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**

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

- **Algebraic Identities**

  E.g., use associativity and commutativity of +

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

- **Iterate!** Optimizations enable further optimizations.

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

Source: $i := j + 1$
$\text{a}[i] := \text{b}[i] + j + 1$

Naive IR: After CSE:

$t1 := \text{addr } j$
$t2 := \ast t1$
$t3 := \text{const } 1$
$t4 := t2 + t3$
$t5 := \text{addr } i$
$\ast t5 := t4$

$t6 := \text{addr } b$
$t7 := \text{addr } i$
$t8 := \ast t7$
$t9 := \text{const } 4$
$t10 := t8 \ast t9$
$t11 := t6 + t10$
$t12 := \ast t11$
$t13 := \text{addr } j$
$t14 := \ast t13$
$t15 := \text{const } 1$
$t16 := t14 + t15$
$t17 := t12 + t16$
$t18 := \text{addr } a$
$t19 := \text{addr } i$
$t20 := \ast t19$
$t21 := \text{const } 4$
$t22 := t20 \ast t21$
$t23 := t18 + t22$
$\ast t23 := t17$

$t1 : = \text{addr } j$
$t2 : = \ast t1$
$t3 : = \text{const } 1$
$t4 : = t2 + t3 ; j + 1$
$t5 : = \text{addr } i$
$\ast t5 : = t4$

$t6 : = \text{addr } b$
$t7 : = \text{addr } i$
$t8 : = \ast t7$
$t9 : = \text{const } 4$
$t10 : = t4 \ast t9$
$t11 : = t6 + t10 ; \&(\text{b}[i])$
$t12 : = \ast t11$
$t13 : = \text{addr } j$
$t14 : = \ast t13$
$t15 : = \text{const } 1$
$t16 : = t14 + t15 ; j + 1$
$t17 : = t12 + t4$
$t18 : = \text{addr } a$
$t19 : = \text{addr } i$
$t20 : = \ast t19$
$t21 : = \text{const } 4$
$t22 : = t20 \ast t21$
$t23 : = t18 + t10 ; \&(\text{a}[i])$
$\ast t23 : = t17$
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.

```pseudocode
a = 0;
for (i = 0; i < 20; i++)
    a = a + b[i] * c[i];
return a;
```
IR for Dot Product

B1  t1 := const 0
    t2 := addr a
    *t2 := t1
    t3 := addr i
    *t3 := t1

B2  L2:
    t5 := addr i
    t6 := *t5
    t7 := const 20
    if t6 >= t7 goto L4

B3  t8 := addr a
    t9 := *t8
    t10 := addr b
    t11 := addr i
    t12 := *t11
    t13 := const 4
    t14 := t12 * t13
    t15 := t10 + t14 ; &(b[i])
    t16 := *t15
    t17 := addr c
    t18 := t17 + t14 ; &(c[i])
    t19 := *t18
    t20 := t16 * t19

B4  L4:

    t21 := t9 + t20
    *t8 := t21
    t22 := const 1
    t23 := t12 + t22
    *t11 := t23
    goto L2

    t24 := addr a
    t25 := *t24
    return t25
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.

\[
\begin{align*}
t1 & := \text{const 0} \\
t2 & := \text{addr a} \\
*\text{t2} & := t1 \\
t3 & := \text{addr i} \\
*\text{t3} & := t1
\end{align*}
\]

\[
\begin{align*}
t1 & := \text{const 0} \\
t9 & := t1 \ ; \ a \\
t6 & := t1 \ ; \ i \times 4 \\
t13 & := \text{const 4} \\
t7 & := \text{const 80} \\
t8 & := \text{addr a} \\
t10 & := \text{addr b} \\
t17 & := \text{addr c}
\end{align*}
\]
L2:
\[
t5 := \text{addr } i \\
t6 := *t5 \\
t7 := \text{const } 20 \\
\text{if } t6 \geq t7 \text{ goto L4}
\]
\[
t8 := \text{addr } a \\
t9 := *t8 \\
t10 := \text{addr } b \\
t11 := \text{addr } i \\
t12 := *t11 \\
t13 := \text{const } 4 \\
t14 := t12 * t13 \\
t15 := t10 + t14 \\
t16 := *t15 \\
t17 := \text{addr } c \\
t18 := t17 + t14 \\
t19 := *t18 \\
t20 := t16 * t19 \\
t21 := t9 + t20 \\
* t8 := t21 \\
t22 := \text{const } 1 \\
t23 := t12 + t22 \\
* t11 := t23 \\
goto L2 \\
\]

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

L2:
\[
t5 := \text{addr } i \\
t6 := *t5 \\
t7 := \text{const } 20 \\
\text{if } t6 \geq t7 \text{ goto L4}
\]
\[
t8 := \text{addr } a \\
t9 := *t8 \\
t10 := \text{addr } b \\
t11 := \text{addr } i \\
t12 := *t11 \\
t13 := \text{const } 4 \\
t14 := t12 * t13 \\
t15 := t10 + t14 \\
t16 := *t15 \\
t17 := \text{addr } c \\
t18 := t17 + t14 \\
t19 := *t18 \\
t20 := t16 * t19 \\
t21 := t9 + t20 \\
* t8 := t21 \\
t22 := \text{const } 1 \\
t23 := t12 + t22 \\
* t11 := t23 \\
goto L2 \\
\]

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