• 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 PEEPHOLE 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
c := a ; maybe can now omit
```

```
d := c
d := a
```

• **Algebraic Identities**

E.g., use associativity and commutativity of +

```
a := b + c \Rightarrow a := b + c
```

```
b := c + d + b b := b + c + d ; now do CSE
```
CSE EXAMPLE

Source:
\[
\begin{align*}
  i & := j + 1 \\
  a[i] & := b[i] + j + 1
\end{align*}
\]

Naive IR: 
After CSE:

\[
\begin{align*}
  t1 & := \text{addr } j \\
  t2 & := *t1 \\
  t3 & := \text{const } 1 \\
  t4 & := t2 + t3 \\
  t5 & := \text{addr } i \\
  *t5 & := t4 \\
  t6 & := \text{addr } b \\
  t7 & := \text{addr } i \\
  t8 & := *t7 \\
  t9 & := \text{const } 4 \\
  t10 & := t8 * t9 \\
  t11 & := t6 + t10 \\
  t12 & := *t11 \\
  t13 & := \text{addr } j \\
  t14 & := *t13 \\
  t15 & := \text{const } 1 \\
  t16 & := t14 + t15 \\
  t17 & := t12 + t16 \\
  t18 & := \text{addr } a \\
  t19 & := \text{addr } i \\
  t20 & := *t19 \\
  t21 & := \text{const } 4 \\
  t22 & := t20 * t21 \\
  t23 & := t18 + t22 \\
  *t23 & := t17
\end{align*}
\]
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.

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

Example IR...
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
    return t25
    goto L2
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:

```plaintext
B1  t1 := const 0  \hspace{1cm} t1 := const 0
    t2 := addr a  \hspace{1cm} t9 := t1 ; a
    *t2 := t1  \hspace{1cm} t6 := t1 ; i \times 4
    t3 := addr i  \hspace{1cm} t13 := const 4
    *t3 := t1  \hspace{1cm} t7 := const 80
```

\(t8 := addr a\)

\(t10 := addr b\)

\(t17 := addr c\)
B2  L2:
    t5 := addr i
    t6 := *t5
    t7 := const 20
    if t6 >= t7 goto L4  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
    t16 := *t15
    t17 := addr c
    t18 := t17 + t14
    t19 := *t18
    t20 := t16 * t19
    t21 := t9 + t20
    t22 := const 1
    t23 := t12 + t22
    goto L2
    *t8 := t21
    t24 := addr a
    t25 := *t24
    return t25
    t22 := const 1
    t23 := t12 + t22
    goto L2
    *t11 := t23
    return t25
    t6 := t6 + t13
    goto L2

B4  L4:
    t24 := addr a
    t25 := *t24
    return t25
    return t9
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).