1. Code Optimization Strategies

- We can try to improve the performance of the target program by performing code-improving transformations within basic blocks. This approach is called local optimization.
- A more thorough, more global job of code optimization can be done by looking at transformations across the basic blocks of a procedure, a task sometimes called intra-procedural optimization.
- We can also look at inter-procedural optimization where we try to improve the performance of a program as a whole.
- The general strategy for code optimization is to look for program transformations that give the most bang for the buck: they should be easy to implement, they should not take too much compilation time, and they should have high payoff. As with many tasks in compilation, code optimization is a study in tradeoffs.

2. Peephole Optimization

- One strategy for generating good code is to first use a naive code generation algorithm and then apply local improvements to the code by examining a sliding window of instructions, called the peephole, and replacing an instruction sequence within the peephole by shorter or faster sequence of code. Here are some typical peephole transformations:
  - Eliminating redundant loads and stores
    - In the instruction sequence
      - `LD R0, a`
      - `ST a, R0`
    - the store instruction is redundant and can be eliminated.
• Eliminating unreachable code
  o In the instruction sequence
    
    L1: goto L2
    x = y + z
    L2: a = b + c
    
    the second statement is unreachable and can be eliminated.

• Eliminating unnecessary jumps
  o In the instruction sequence
    
    L1: if x < y goto L2
    ...
    L2: goto L3
    
    the jump to a jump can be replaced by
    
    L1: if x < y goto L3
    ...
    L2: goto L3

• Algebraic simplification
  o Three-address statements such as
    
    x = x + 0 or x = x * 1
    
    where x is an integer can be eliminated entirely.

• Reduction in strength
  o An expensive operation such as \( x^2 \) can be replaced by a cheaper operation such as \( x \times x \).

3. Common Subexpression Elimination

• Local common subexpression elimination
  
  o In the following BEFORE basic block, the assignments to \( t7 \) and \( t10 \) compute the subexpressions \( 4 \times i \) and \( 4 \times j \), which have been eliminated in the AFTER block by local common subexpression elimination:

  BEFORE          AFTER
  t6 = 4 \times i  t6 = 4 \times i
  x = a[t6]       x = a[t6]
  t7 = 4 \times i  t8 = 4 \times j
  t8 = 4 \times j
  t9 = a[t8]      t9 = a[t8]
  a[t7] = t9      a[t6] = t9
  t10 = 4 \times j
  a[t10] = x      a[t8] = x
  goto B2        goto B2
Global common subexpression elimination

- In the following flow graph, block B5 computes the common subexpressions 4 * i and 4 * j, which are computed in blocks B2 and B3, respectively.

- Notice that block B5 can be replaced by the following block since block B2 has computed 4*i into t2 and a[t2] into t3:
x = t3
t8 = 4 * j
t9 = a[t8]  
a[t2] = t9  
a[t8] = x  
goto B2

- This block can be replaced by following block by noticing that block B3 has computed 4*j into t4 and a[t4] into t5:

  x = t3  
t9 = a[t4]  
a[t2] = t9  
a[t4] = x  
goto B2

- We now notice that block B3 has already computed a[t4] into t5 so we can replace the second and third statements by the assignment a[t2] = t5 to obtain the following optimized block:

  x = t3  
a[t2] = t5  
a[t4] = x  
goto B2

So far we have reduced the original nine-statement block B5 into a four-statement block.

4. Copy Propagation

- A three-address statement of the form \( u = v \) is called a copy statement, or copy for short.
- We can introduce copy statements to avoid recomputing common subexpressions:

```
a = d + e
b = d + e

c = d + e

\[
t = d + e  
\]
\[
a = t
\]
\[
t = d + e  
\]
\[
b = t
\]
\[
c = t
\]```
5. Dead-Code Elimination

- Statements that compute values that never get subsequently used can be eliminated.
- Often copy propagation turns copy statements into dead code.
- Consider the reduced basic block for B5:

```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```

After copy propagation this block becomes:

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

We now observe x is never used so the first statement can be eliminated. The block now becomes

```
a[t2] = t5
a[t4] = x
goto B2
```

6. Code Motion

- Loop-invariant computations are best moved outside loops.
- Consider the while-statement:

```
while (i <= limit - 2)
```

Code motion will produce a faster equivalent loop when the limit computation is performed once before entering the loop:

```
t = limit - 2
while (i <= t)
```

7. Induction Variables

- A variable x is an induction variable if its value always changes by a constant whenever it is assigned a new value.
For example, \(i\) and \(t_2\) are induction variables in block \(B_2\) of the flow graph in Section 3 above.

- Reduction in strength and induction-variable elimination can be used to speed up loops. See ALSU, Figs. 9.8 – 9.10, pp. 592-595 for an extended example.

### 8. Practice Problems

1) ALSU, Exercise 9.1.1 (p. 596).
2) ALSU, Exercise 9.1.4 (p. 596).

### 9. Reading

- ALSU, Sections 8.5, 8.7, 9.1