Local Optimization

- The simplest form of optimization
- Optimize one basic block. (No need to analyze the whole procedure body).

Some statements can be deleted

```
x := x + 0
x := x * 1
```

Eliminating these instructions are great optimizations as we are removing an entire instruction.

Some statements can be simplified

```
x := x * 0  =>  x := 0
```

Maybe the instruction on the right is faster than the instruction on the right. More importantly, assigning to a constant allows more cascading optimizations, as we will see later.

Another example:

```
y := y ** 2  => y := y * y
```

Typically, an architecture will not have the exponentiation instruction and so this operator may have to be implemented by an exponentiation loop in software. For the special cases (e.g., exponent = 2), we can simply replace by multiplication.

Another example:

```
x := x * 8  => x := x << 3
x := x * 15  => t := x << 4; x := t - x
```

Replace multiplication by a power-of-two to a shift-left operation. Can also do this for non powers-of-two. On some machines, left-shift is faster than multiply, but not on all! Modern machines have their own "rewriting logic" in hardware that can deal with these special cases really fast. In other words, some of these compiler local optimizations may become less relevant on modern hardware.

All these transformations are examples of *algebraic simplifications*.

Operations on constants can be computed at compile time

- If there is a statement `x := y op z`
- And `y` and `z` are constants.
- Then `y op z` can be computed at compile time.

Examples

- `x:= 2 + 2` can be changed to `x := 4`
- `if 2 < 0 jump L` can be deleted.
- `if 2 > 0 jump L` can be replaced by `jump L`

This class of optimizations is called *constant folding*. One of the most consequential and most common optimizations performed by compilers.

Another important optimization: eliminate unreachable basic blocks (dead code):

- Code that is unreachable from the initial block
  - e.g., basic blocks that are not the target of any jump or "fall through" from a conditional
- Removing unreachable code makes the program smaller
  - And sometimes also faster
    - Due to memory cache effects

        - Increased spatial locality

Called dead-code elimination.

Why would unreachable basic blocks occur?

- e.g., `if (DEBUG) then { .... }`
    - Need constant propagation followed with dead-code elimination to remove this whole code.
- Libraries: a library may supply a 100 methods but we are using only a 3 of those methods, the other 97 are dead-code.
- Usually other optimizations may result in more dead code

Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment.

- Rewrite intermediate code in *single assignment* form

  ```
  x := z + y
  a := x
  z := 2 * x
  ```

  Change to

  ```
  b := z + y
  a := b
  x := 2 * b
  ```

More complicated in general, due to loops

If

- Basic block is in single assignment form
- A definition `x :=` is the first use of `x` in a block

Then

- When two assignments have the same rhs, they compute the same value.

Example

```
x := y + z
....
....
w := y + z
```

We can be sure that the values of `x`, `y`, and `z` do not change in the code. Hence, we can replace the assignment to `w` as follows:

```
x := y + z
....
....
w := x
```

This optimization is called *common-subexpression elimination*. This is also another very common and consequential compiler optimization.

If we see `w := x` appears in a block, replace subsequent uses of `w` with uses of `x`

- Assumes single assignment form

Example:

```
b := z + y
a := b
x := 2 * a
```

This can be replaced with

```
b := z + y
a := b
x := 2 * b
```

This is called *copy propagation*. This is useful for enabling other optimizations

- Constant folding. e.g., if the propagated value is a constant
- Dead code elimination. e.g., the copy statement can be eliminated as it is inconsequential.

Example:

```
a := 5
x := 2 * a
y := x + 6
t := x * y
```

gets transformed to

```
a := 5
x := 10
y := 16
t := x >> 4
```

We could have also assigned 160 to `t`. That would be even better.

If `w := rhs` appears in a basic block and `w` does not appear anywhere else in the program, THEN the statement `w := rhs` is dead and can be eliminated.

- Dead: does not contribute to the program's result

In our copy-propagation example, one of the assignments is dead and can be eliminated.

- Each local optimization does little by itself
- Typical optimizations interact
  - Performing one optimization enables another
- Optimizing compilers can be thought of as a *big bag of tricks*.
- Optimizing compilers repeat optimizations until no improvement is possible
  - The optimizer can also be stopped at any point to limit compilation time
  - Certain properties on these transformations (tricks) ensure that we converge (and not oscillate) in this fixed point procedure. e.g., we always *improve* for some notion of improvement. This means that we are uni-directional and can never oscillate.

Example

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e + f
```

Final form

```
a := x * x
f := a + a
g := 6 * f
```

Possible to simplify further, but the compiler may get stuck in "local minima". e.g., should it have changed a + a to 2 * a, it would have had a better optimization opportunity (replacing g = 12*f and eliminating computation of f).

Peephole optimization

- Optimizations can be directly applied to assembly code. Typically for optimizations that got missed at IR stage (perhaps due to local minima problem).
  . At the assembly-level, we have greater visibility into instruction costs and instruction opcodes. E.g., hardware opcodes may be higher-level than IR opcodes.
- Peephole optimization is effective for improving assembly code
  - The "peephole" is a short sequence of (usually contiguous) instructions
  - The optimizer replaces the sequence with another equivalent one (but faster)

Peephole optimizations are usually written as replacement rules

```
i1; i2; i3; ..; in --> j1; j2; ..; jm
```

Example:

```
move $a $b; move $b $a --> move $a $b
```

Works if the second instruction is not the target of a jump (i.e., both instructions belong to a basic block).

Another example:

```
addiu $a $a i; addiu $a $a j --> addiu $a $a (i+j)
```

Many (but not all) of the basic block optimizations can be cast as peephole optimizations

- Example: `addiu $a $b 0` --> `move $a $b`
- Example: `move $a $a` -->
- These two together eliminate `addiu $a $a 0`.

As for local optimizations, peephole optimizations must be applied repeatedly for maximum effect

- Need to ensure that the replacement rules cannot cause oscillations. e.g., each replacement rule can only "improve" the code.

Many simple optimizations can still be applied on assembly language.

"Program optimiztion" is grossly misnamed

- Code "optimizers" have no intention or even pretense of working towards the optimal program.
- Code "improvers" is a better term.