PEEP-HOLE OPTIMIZATION

Most production compilers produce good code through careful instruction selection and register allocation, a few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program.

A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.

Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

Characteristic of peephole optimization:

- Each improvement may spawn opportunities for additional improvements.
- Repeated passes over the target code are necessary to get the maximum benefit.

Examples of program transformations those are characteristic of peephole optimizations:

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Eliminating Redundant Loads and Stores

If we see the instruction sequence in a target program,

```
LD a , R0
ST R0 , a
```

We can delete store instructions because whenever it is executed. First instruction will ensure that the value of has already been loaded into register R0.

Eliminating Unreachable Code

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabelled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions.

For example, for debugging purposes, a large program may have within it certain code fragments that are executed only if a variable debug is equal to 1.

In C, the source code might look like:

```
#define debug 0
....

if ( debug ) {
Print debugging information

}
```

In the intermediate representation, this code may look like

```
if debug == 1 goto L1
goto L2
L 1 : print debugging information
L2 :
```

Flow-of-Control Optimizations

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the sequence

```
goto L1
L1 : goto L2
```

by the sequence

```
goto L2
L1 : goto L2
```

If there are now no jumps to L1, then it may be possible to eliminate the statement L1 : goto L2 provided it is preceded by an unconditional jump.

Similarly, the sequence

```
if a < b goto L1
L1  : goto L2
```

can be replaced by the sequence

```
if a < b goto L2
L1 : goto L2
```

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence goto L1

```
L1 : if a < b goto L2
L3 :
```

may be replaced by the sequence

```
L3 :
if a < b goto L2
goto L3
```

Algebraic Simplification and Reduction in Strength

These algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as

```
x = x + 0
      or
x= x * 1
```

in the peephole.

Similarly, reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

Use of Machine Idioms

• The target machine may have hardware instructions to implement certain specific operations efficiently.

• Detecting situations that permit the use of these instructions can reduce execution time significantly.

For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.

The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like x = x + 1.

$x = x + 1 \rightarrow x{+}{+}$

$x = x - 1 \rightarrow x{-}{-}$