That is, when reordering code, uses of an array a may not cross each other, and no statement may cross a procedure call or an assignment through a pointer.

9.9 PEEPHOLE OPTIMIZATION

A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying "optimizing" transformations to the target program. The term "optimizing" is somewhat misleading because there is no guarantee that the resulting code is optimal under any mathematical measure. Nevertheless, many simple transformations can significantly improve the running time or space requirement of the target program, so it is important to know what kinds of transformations are useful in practice.

A simple but effective technique for locally improving the target code is *peephole optimization*, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the *peephole*) and replacing these instructions by a shorter or faster sequence, whenever possible. Although we discuss peephole optimization as a technique for improving the quality of the target code, the technique can also be applied directly after intermediate code generation to improve the intermediate representation.

The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements. In general, repeated passes over the target code are necessary to get the maximum benefit. In this section, we shall give the following examples of program transformations that are characteristic of peephole optimizations:

- redundant-instruction elimination
- flow-of-control optimizations
- algebraic simplifications
- use of machine idioms

Redundant Loads and Stores

If we see the instruction sequence

```
(1) MOV   R0, a
(2) MOV   a, R0
```
(9.7)

we can delete instruction (2) because whenever (2) is executed, (1) will ensure that the value of a is already in register R0. Note that if (2) had a label[9] we

[9] One advantage of generating assembly code is that labels will be present, facilitating peephole optimizations such as this. If machine code is generated, and peephole optimization is desired, we can use a bit to mark the instructions that would have labels.

could not be sure that (1) was always executed immediately before (2) and so we could not remove (2). Put another way, (1) and (2) have to be in the same basic block for this transformation to be safe.

While target code such as (9.7) would not be generated if the algorithm suggested in Section 9.6 were used, it might be if a more naive algorithm like the one mentioned at the beginning of Section 9.1 were used.

Unreachable Code

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled 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 segments that are executed only if a variable `debug` is 1. In C, the source code might look like:

```
#define debug 0
...
if ( debug ) {
    print debugging information
}
```

In the intermediate representation the if-statement may be translated as:

```
    if debug = 1 goto L1
    goto L2                                    (9.8)
L1: print debugging information
L2:
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of `debug`, (9.8) can be replaced by:

```
    if debug ≠ 1 goto L2
    print debugging information                (9.9)
L2:
```

Now, since `debug` is set to 0 at the beginning of the program,[10] constant propagation should replace (9.9) by

```
    if 0 ≠ 1 goto L2
    print debugging information                (9.10)
L2:
```

As the argument of the first statement of (9.10) evaluates to a constant **true**, it can be replaced by `goto L2`. Then all the statements that print debugging aids are manifestly unreachable and can be eliminated one at a time.

[10] To tell that `debug` has the value 0 we need to do a global "reaching definitions" data-flow analysis, as discussed in Chapter 10.

Flow-of-Control Optimizations

The intermediate code generation algorithms in Chapter 8 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 jump sequence

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

by the sequence

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

If there are now no jumps to `L1`,[11] 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

```
        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          (9.11)
L3:
```

may be replaced by

```
        if a < b goto L2
        goto L3
        ...
L3:                               (9.12)
```

While the number of instructions in (9.11) and (9.12) is the same, we

[11] If this peephole optimization is attempted, we can count the number of jumps to each label in the symbol-table entry for that label; a search of the code is not necessary.

sometimes skip the unconditional jump in (9.12), but never in (9.11). Thus, (9.12) is superior to (9.11) in execution time.

Algebraic Simplification

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. However, only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

```
x := x + 0
```

or

```
x := x * 1
```

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength

Reduction in strength replaces 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. For example, x^2 is invariably cheaper to implement as `x*x` than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented (approximated) as multiplication by a constant, which may be cheaper.

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 these 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 `i:=i+1`.

9.10 GENERATING CODE FROM DAGS

In this section, we show how to generate code for a basic block from its dag representation. The advantage of doing so is that from a dag we can more easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples. Central to our discussion is the case where the dag is a tree. For this case we can generate code that we can prove is optimal under such criteria as