10.2 THE PRINCIPAL SOURCES OF OPTIMIZATION

In this section, we introduce some of the most useful code-improving transformations. Techniques for implementing these transformations are presented in subsequent sections. A transformation of a program is called *local* if it can be performed by looking only at the statements in a basic block; otherwise, it is called *global*. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes. Common subexpression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such function-preserving transformations. Section 9.8 on the dag representation of basic blocks showed how local common subexpressions could be removed as the dag for the basic block was constructed. The other transformations come up primarily when global optimizations are performed, and we shall discuss each in turn.

Frequently, a program will include several calculations of the same value, such as an offset in an array. As mentioned in Section 10.1, some of these duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language. For example, block B_5 shown in Fig. 10.6(a) recalculates 4*i and 4*j.

B_5

```
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
```

(a) Before

B_5

```
t6 := 4*i
x := a[t6]
t8 := 4*j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```

(b) After

Fig. 10.6. Local common subexpression elimination.

Common Subexpressions

An occurrence of an expression E is called a *common subexpression* if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value. For example, the assignments to t_7 and

t_{10} have the common subexpressions `4*i` and `4*j`, respectively, on the right side in Fig. 10.6(a). They have been eliminated in Fig. 10.6(b), by using t_6 instead of t_7 and t_8 instead of t_{10}. This change is what would result if we reconstructed the intermediate code from the dag for the basic block.

Example 10.2. Figure 10.7 shows the result of eliminating both global and local common subexpressions from blocks B_5 and B_6 in the flow graph of Fig. 10.5. We first discuss the transformation of B_5 and then mention some subtleties involving arrays.

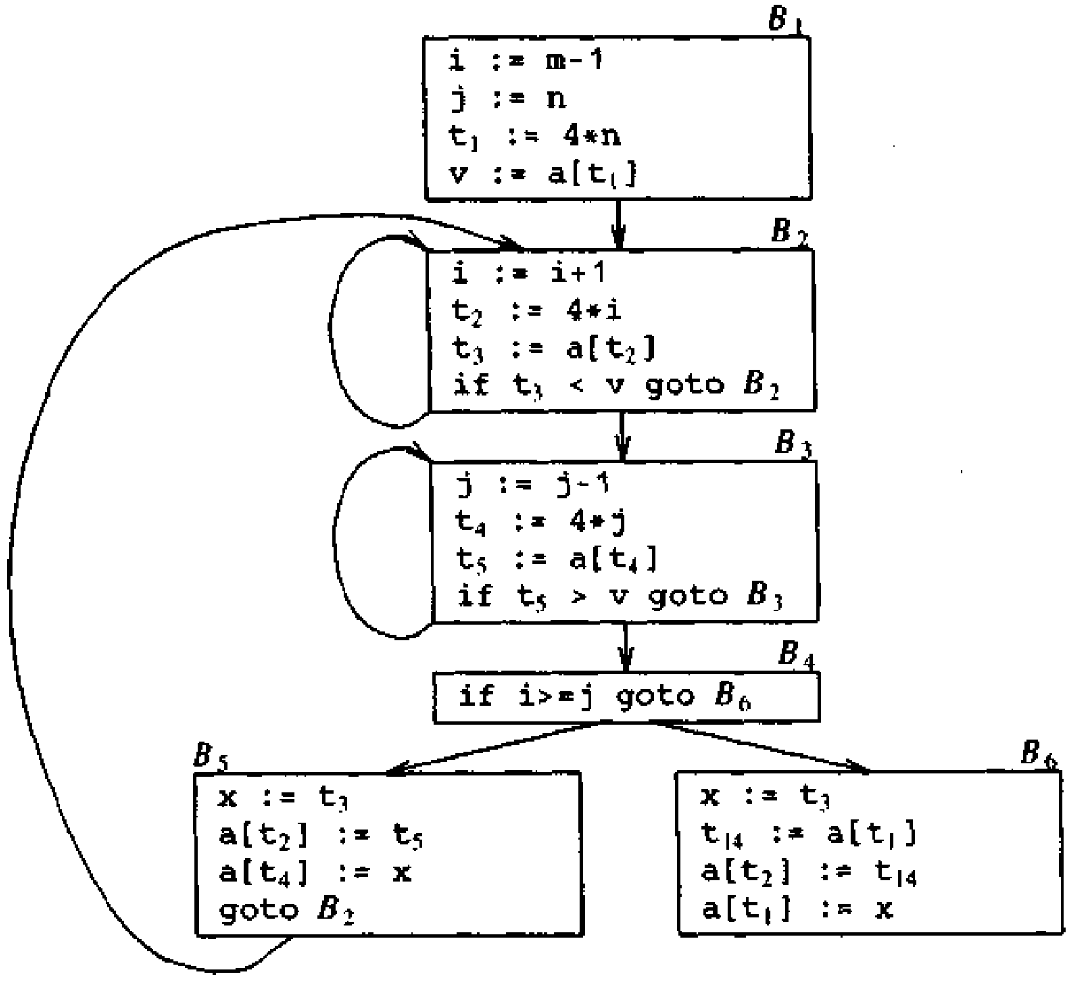


Fig. 10.7. B_5 and B_6 after common subexpression elimination.

After local common subexpressions are eliminated, B_5 still evaluates `4*i` and `4*j`, as shown in Fig. 10.6(b). Both are common subexpressions; in particular, the three statements

t_8 := 4*j; t_9 := a[t_8]; a[t_8] := x

in B_5 can be replaced by

t_9 := a[t_4]; a[t_4] := x

using t_4 computed in block B_3. In Fig. 10.7, observe that as control passes from the evaluation of 4*j in B_3 to B_5, there is no change in j, so t_4 can be used if 4*j is needed.

Another common subexpression comes to light in B_5 after t_4 replaces t_8. The new expression a[t_4] corresponds to the value of a[j] at the source level. Not only does j retain its value as control leaves B_3 and then enters B_5, but a[j], a value computed into a temporary t_5, does too because there are no assignments to elements of the array a in the interim. The statements

```
t9 := a[t4];   a[t6] := t9
```

in B_5 can therefore be replaced by

```
a[t6] := t5
```

Analogously, the value assigned to x in block B_5 of Fig. 10.6(b) is seen to be the same as the value assigned to t_3 in block B_2. Block B_5 in Fig. 10.7 is the result of eliminating common subexpressions corresponding to the values of the source level expressions a[i] and a[j] from B_5 in Fig. 10.6(b). A similar series of transformations has been done to B_6 in Fig. 10.7.

The expression a[t_1] in blocks B_1 and B_6 of Fig. 10.7 is not considered a common subexpression, although t_1 can be used in both places. After control leaves B_1 and before it reaches B_6, it can go through B_5, where there are assignments to a. Hence, a[t_1] may not have the same value on reaching B_6 as it did on leaving B_1, and it is not safe to treat a[t_1] as a common subexpression. □

Copy Propagation

Block B_5 in Fig. 10.7 can be further improved by eliminating x using two new transformations. One concerns assignments of the form f:=g called *copy statements*, or *copies* for short. Had we gone into more detail in Example 10.2, copies would have arisen much sooner, because the algorithm for eliminating common subexpressions introduces them, as do several other algorithms. For example, when the common subexpression in c:=d+e is eliminated in Fig. 10.8, the algorithm uses a new variable t to hold the value of d+e. Since control may reach c:=d+e either after the assignment to a or after the assignment to b, it would be incorrect to replace c:=d+e by either c:=a or by c:=b.

The idea behind the copy-propagation transformation is to use *g* for f, wherever possible after the copy statement f:=g. For example, the assignment x:=t_3 in block B_5 of Fig. 10.7 is a copy. Copy propagation applied to B_5 yields:

```
x := t3
a[t2] := t5                    (10.1)
a[t4] := t3
goto B2
```

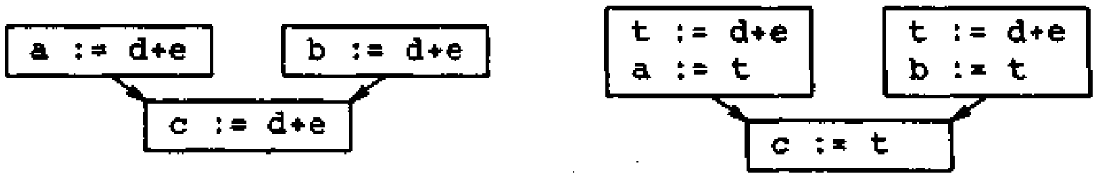


Fig. 10.8. Copies introduced during common subexpression elimination.

This may not appear to be an improvement, but as we shall see, it gives us the opportunity to eliminate the assignment to x.

Dead-Code Elimination

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. For example, in Section 9.9 we discussed the use of debug that is set to true or false at various points in the program, and used in statements like

```
if (debug) print ...                                     (10.2)
```

By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement, the value of debug is false. Usually, it is because there is one particular statement

```
debug := false
```

that we can deduce to be the last assignment to debug prior to the test (10.2), no matter what sequence of branches the program actually takes. If copy propagation replaces debug by false, then the print statement is dead because it cannot be reached. We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as *constant folding*.

One advantage of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to x and transforms (10.1) into:

```
a[$t_2$] := $t_5$
a[$t_4$] := $t_3$
goto $B_2$
```

This code is a further improvement of block B_5 in Fig. 10.7.

Loop Optimizations

We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop. Three techniques are important for loop optimization: *code motion*, which moves code outside a loop; *induction-variable elimination*, which we apply to eliminate `i` and `j` from the inner loops B_2 and B_3 of Fig. 10.7; and, *reduction in strength*, which replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a *loop-invariant computation*) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of `limit-2` is a loop-invariant computation in the following while-statement:

```
while ( i <= limit-2 ) /* statement does not change limit */
```

Code motion will result in the equivalent of

```
t = limit-2;
while ( i <= t ) /* statement does not change limit or t */
```

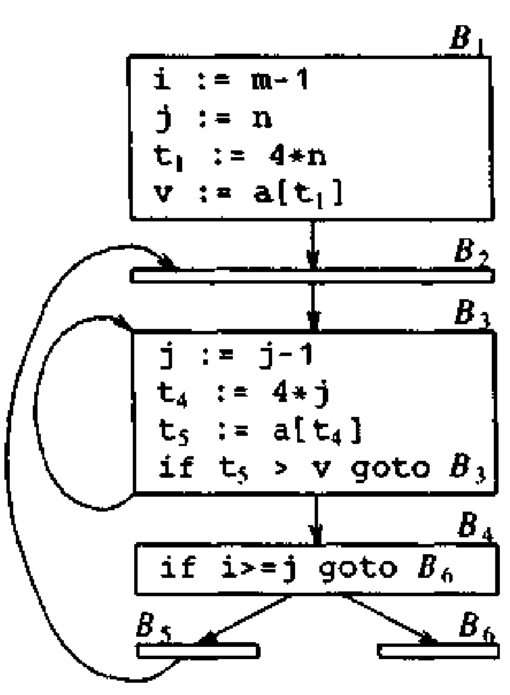
Induction Variables and Reduction in Strength

While code motion is not applicable to the quicksort example we have been considering, the other two transformations are. Loops are usually processed inside out. For example, consider the loop around B_3. Only the portion of the flow graph relevant to the transformations on B_3 is shown in Fig. 10.9.

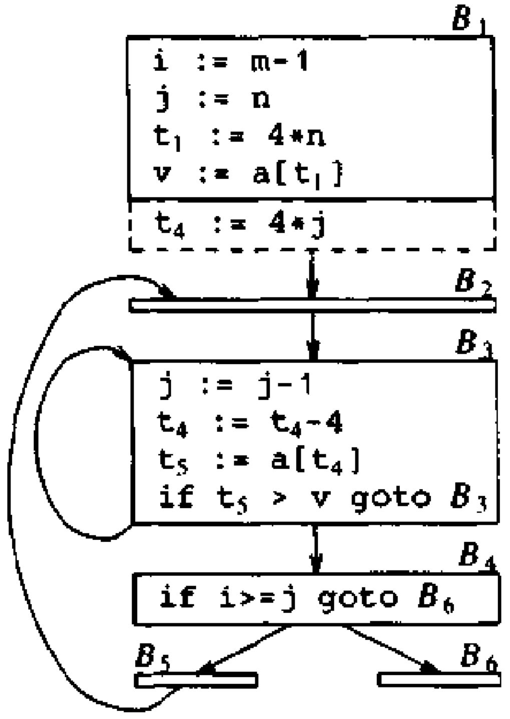
Note that the values of `j` and t_4 remain in lock-step; every time the value of `j` decreases by 1, that of t_4 decreases by 4 because `4*j` is assigned to t_4. Such identifiers are called *induction variables*.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B_3 in Fig. 10.9(a), we cannot get rid of either `j` or t_4 completely; t_4 is used in B_3 and `j` in B_4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually, `j` will be eliminated when the outer loop of B_2-B_5 is considered.

Example 10.3. As the relationship t_4 = `4*j` surely holds after such an assignment to t_4 in Fig. 10.9(a), and t_4 is not changed elsewhere in the inner loop around B_3, it follows that just after the statement `j:=j-1` the



(a) Before



(b) After

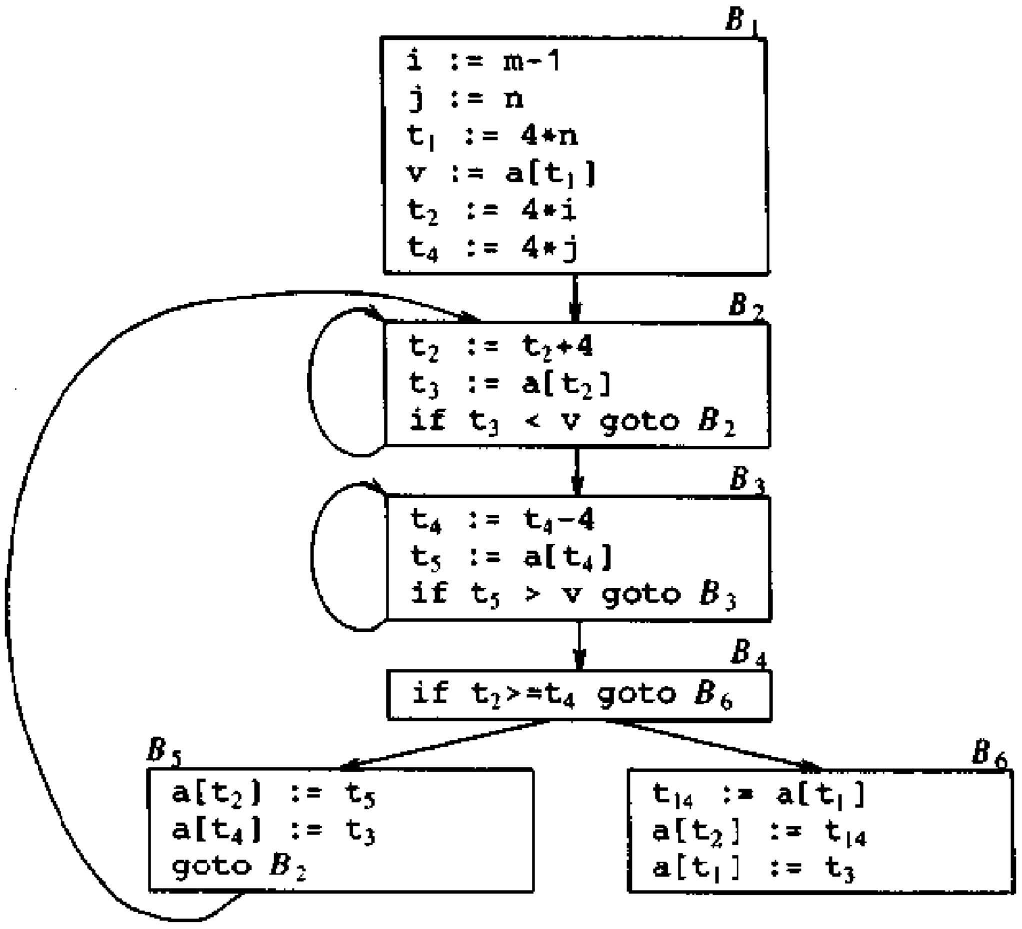
Fig. 10.9. Strength reduction applied to `4*j` in block B_3.

relationship t_4 = `4*j-4` must hold. We may therefore replace the assignment t_4`:=4*j` by t_4`:=`t_4`-4`. The only problem is that t_4 does not have a value when we enter block B_3 for the first time. Since we must maintain the relationship t_4 = `4*j` on entry to the block B_3, we place an initialization of t_4 at the end of the block where `j` itself is initialized, shown by the dashed addition to block B_1 in Fig. 10.9(b).

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines. □

Section 10.7 discusses how induction variables can be detected and what transformations can be applied. We conclude this section with one more example of induction-variable elimination that treats `i` and `j` in the context of the outer loop containing B_2, B_3, B_4, and B_5.

Example 10.4. After reduction in strength is applied to the inner loops around B_2 and B_3, the only use of `i` and `j` is to determine the outcome of the test in block B_4. We know that the values of `i` and t_2 satisfy the relationship t_2 = `4*i`, while those of `j` and t_4 satisfy the relationship t_4 = `4*j`, so the test t_2`>=`t_4 is equivalent to `i>=j`. Once this replacement is made, `i` in block B_2 and `j` in block B_3 become dead variables and the assignments to them in these blocks become dead code that can be eliminated, resulting in the flow graph shown in Fig. 10.10. □



```
B1:
i := m-1
j := n
t1 := 4*n
v := a[t1]
t2 := 4*i
t4 := 4*j

B2:
t2 := t2+4
t3 := a[t2]
if t3 < v goto B2

B3:
t4 := t4-4
t5 := a[t4]
if t5 > v goto B3

B4:
if t2>=t4 goto B6

B5:
a[t2] := t5
a[t4] := t3
goto B2

B6:
t14 := a[t1]
a[t2] := t14
a[t1] := t3
```

Fig. 10.10. Flow graph after induction-variable elimination.

The code-improving transformations have been effective. In Fig. 10.10, the number of instructions in blocks B_2 and B_3 has been reduced from 4 to 3 from the original flow graph in Fig. 10.5, in B_5 it has been reduced from 9 to 3, and in B_6 from 8 to 3. True, B_1 has grown from four instructions to six, but B_1 is executed only once in the fragment, so the total running time is barely affected by the size of B_1.

10.3 OPTIMIZATION OF BASIC BLOCKS

In Chapter 9, we saw a number of code-improving transformations for basic blocks. These included structure-preserving transformations, such as common subexpression elimination and dead-code elimination, and algebraic transformations such as reduction in strength.

Many of the structure-preserving transformations can be implemented by constructing a dag for a basic block. Recall that there is a node in the dag for each of the initial values of the variables appearing in the basic block, and there is a node n associated with each statement s within the block. The children of n are those nodes corresponding to statements that are the last definitions prior to s of the operands used by s. Node n is labeled by the operator