CHAPTER 10

Code Optimization

Ideally, compilers should produce target code that is as good as can be written by hand. The reality is that this goal is achieved only in limited cases, and with difficulty. However, the code produced by straightforward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called *optimizations*, although the term "optimization" is a misnomer because there is rarely a guarantee that the resulting code is the best possible. Compilers that apply code-improving transformations are called *optimizing compilers*.

The emphasis in this chapter is on machine-independent optimizations, program transformations that improve the target code without taking into consideration any properties of the target machine. Machine-dependent optimizations, such as register allocation and utilization of special machine-instruction sequences (machine idioms) were discussed in Chapter 9.

The most payoff for the least effort is obtained if we can identify the frequently executed parts of a program and then make these parts as efficient as possible. There is a popular saying that most programs spend ninety per cent of their execution time in ten per cent of the code. While the actual percentages may vary, it is often the case that a small fraction of a program accounts for most of the running time. Profiling the run-time execution of a program on representative input data accurately identifies the heavily traveled regions of a program. Unfortunately, a compiler does not have the benefit of sample input data, so it must make its best guess as to where the program hot spots are.

In practice, the program's inner loops are good candidates for improvement. In a language that emphasizes control constructs like while and for statements, the loops may be evident from the syntax of the program; in general, a process called control-flow analysis identifies loops in the flow graph of a program.

This chapter is a cornucopia of useful optimizing transformations and techniques for implementing them. The best technique for deciding what transformations are worthwhile to put into a compiler is to collect statistics about the source programs and evaluate the benefit of a given set of optimizations on a

representative sample of real source programs. Chapter 12 describes transformations that have proven useful in optimizing compilers for several different languages.

One of the themes of this chapter is data-flow analysis, a process of collecting information about the way variables are used in a program. The information collected at various points in a program can be related using simple set equations. We present several algorithms for collecting information using data-flow analysis and for effectively using this information in optimization. We also consider the impact of language constructs such as procedures and pointers on optimization.

The last four sections of this chapter deal with more advanced material. They cover some graph-theoretic ideas relevant to control-flow analysis and apply these ideas to data-flow analysis. The chapter concludes with a discussion of general-purpose tools for data-flow analysis and techniques for debugging optimized code. The emphasis throughout this chapter is on optimizing techniques that apply to languages in general. Some compilers that use these ideas are reviewed in Chapter 12.

10.1 INTRODUCTION

To create an efficient target language program, a programmer needs more than an optimizing compiler. In this section, we review the options available to a programmer and a compiler for creating efficient target programs. We mention the types of code-improving transformations that a programmer and a compiler writer can be expected to use to improve the performance of a program. We also consider the representation of programs on which transformations will be applied.

Criteria for Code-Improving Transformations

Simply stated, the best program transformations are those that yield the most benefit for the least effort. The transformations provided by an optimizing compiler should have several properties.

First, a transformation must preserve the meaning of programs. That is, an "optimization" must not change the output produced by a program for a given input, or cause an error, such as a division by zero, that was not present in the original version of the source program. The influence of this criterion pervades this chapter; at all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

Second, a transformation must, on the average, speed up programs by a measurable amount. Sometimes we are interested in reducing the space taken by the compiled code, although the size of code has less importance than it once had. Of course, not every transformation succeeds in improving every program, and occasionally an "optimization" may slow down a program slightly, as long as on the average it improves things.

Third, a transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code-improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. Certain local or "peephole" transformations of the kind discussed in Section 9.9 are simple enough and beneficial enough to be included in any compiler.

Some transformations can only be applied after detailed, often time-consuming, analysis of the source program, so there is little point in applying them to programs that will be run only a few times. For example, a fast, nonoptimizing, compiler is likely to be more helpful during debugging or for "student jobs" that will be run successfully a few times and thrown away. Only when the program in question takes up a significant fraction of the machine's cycles does improved code quality justify the time spent running an optimizing compiler on the program.

Getting Better Performance

Dramatic improvements in the running time of a program — such as cutting the running time from a few hours to a few seconds — are usually obtained by improving the program at all levels, from the source level to the target level, as suggested by Fig. 10.1. At each level, the available options fall between the two extremes of finding a better algorithm and of implementing a given algorithm so that fewer operations are performed.

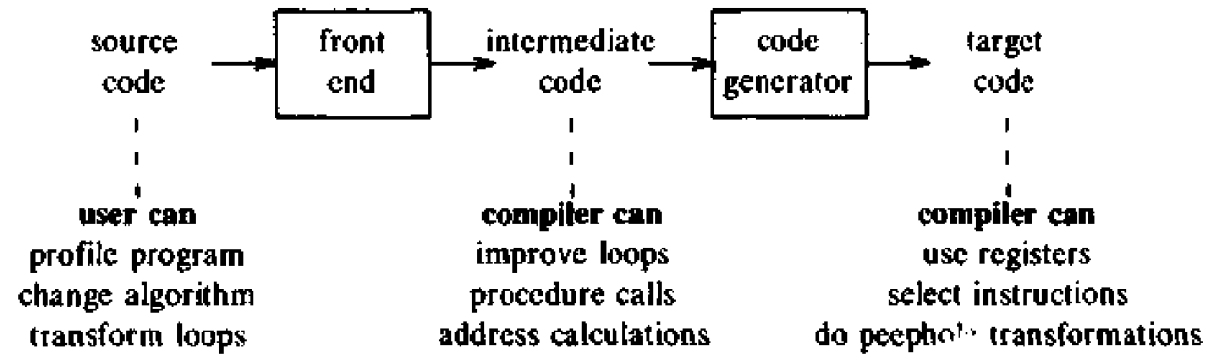


Fig. 10.1. Places for potential improvements by the user and the compiler.

Algorithmic transformations occasionally produce spectacular improvements in running time. For example, Bentley [1982] relates that the running time of a program for sorting N elements dropped from $2.02N^2$ microseconds to $12N\log_2 N$ microseconds when a carefully coded "insertion sort" was replaced by "quicksort."[1] For $N = 100$ the replacement speeds up the program by a

[1] See Aho, Hopcroft, and Ullman [1983] for a discussion of these sorting algorithms and their speeds.

factor of 2.5. For $N = 100{,}000$ the improvement is far more dramatic: the replacement speeds up the program by a factor of more than a thousand.

Unfortunately, no compiler can find the best algorithm for a given program. Sometimes, however, a compiler can replace a sequence of operations by an algebraically equivalent sequence, and thereby reduce the running time of a program significantly. Such savings are more common when algebraic transformations are applied to programs in very-high level languages, e.g., query languages for databases (see Ullman [1982]).

In this section and the next, a sorting program called quicksort will be used to illustrate the effect of various code-improving transformations. The C program in Fig. 10.2 is derived from Sedgewick [1978], where hand-optimization of such a program is discussed. We shall not discuss the algorithmic aspects of this program here — in fact, `a[0]` must contain the smallest and `a[max]` the largest element to be sorted for the program to work.

```
void quicksort(m,n)
int m,n;
{
    int i,j;
    int v,x;
    if ( n <= m ) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while(1) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if (i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Fig. 10.2. C code for quicksort.

It may not be possible to perform certain code-improving transformations at the level of the source language. For example, in a language such as Pascal or Fortran, a programmer can only refer to array elements in the usual way, e.g., as `b[i,j]`. At the level of the intermediate language, however, new opportunities for code improvement may be exposed. Three-address code, for example, provides many opportunities for improving address calculations, especially in loops. Consider the three-address code for determining the value of `a[i]`, assuming that each array element takes four bytes:

```
t1 := 4*i;   t2 := a[t1]
```

Naive intermediate code will recalculate `4*i` every time `a[i]` appears in the source program, and the programmer has no control over the redundant address calculations, because they are implicit in the implementation of the language, rather than being explicit in the code written by the user. In these situations, it behooves the compiler to clean them up. In a language like C, however, this transformation can be done at the source level by the programmer, since references to array elements can be systematically rewritten using pointers to make them more efficient. This rewriting is similar to transformations that optimizing compilers for Fortran traditionally apply.

At the level of the target machine, it is the compiler's responsibility to make good use of the machine's resources. For example, keeping the most heavily used variables in registers can cut running time significantly, often by as much as a half. Again, C allows a programmer to advise the compiler that certain variables be held in registers, but most languages do not. Similarly, the compiler can speed up programs significantly by choosing instructions that take advantage of the addressing modes of the machine to do in one instruction what naively we might expect to require two or three, as we discussed in Chapter 9.

Even if it is possible for the programmer to improve the code, it may be more convenient to have the compiler make some of the improvements. If a compiler can be relied upon to generate efficient code, then the user can concentrate on writing clear code.

An Organization for an Optimizing Compiler

As we have mentioned, there are often several levels at which a program can be improved. Since the techniques needed to analyze and transform a program do not change significantly with the level, this chapter concentrates on the transformation of intermediate code using the organization shown in Fig. 10.3. The code-improvement phase consists of control-flow and data-flow analysis followed by the application of transformations. The code generator, discussed in Chapter 9, produces the target program from the transformed intermediate code.

For convenience of presentation, we assume that the intermediate code consists of three-address statements. Intermediate code, of the sort produced by the techniques in Chapter 8, for a portion of the program in Fig. 10.2 is shown in Fig. 10.4. With other intermediate representations, the temporary variables $t_1, t_2, \ldots, t_{15}$ in Fig. 10.4 need not appear explicitly, as discussed in Chapter 8.

The organization in Fig. 10.3 has the following advantages:

1. The operations needed to implement high-level constructs are made explicit in the intermediate code, so it is possible to optimize them. For example, the address calculations for `a[i]` are explicit in Fig. 10.4, so the recomputation of expressions like `4*i` can be eliminated as discussed in the next section.

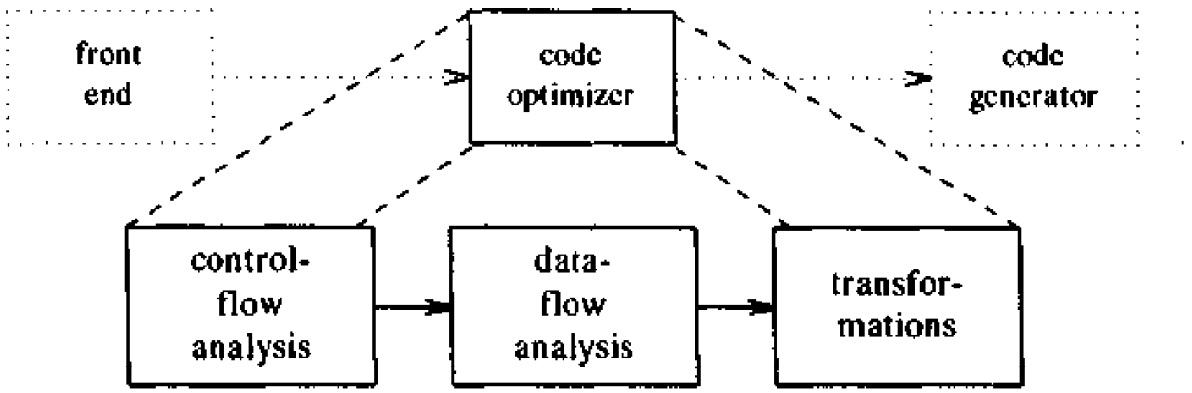


Fig. 10.3. Organization of the code optimizer.

```
(1)   i := m-1
(2)   j := n
(3)   t1 := 4*n
(4)   v := a[t1]
(5)   i := i+1
(6)   t2 := 4*i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)   j := j-1
(10)  t4 := 4*j
(11)  t5 := a[t4]
(12)  if t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)  x := a[t6]
(16)  t7 := 4*i
(17)  t8 := 4*j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4*j
(21)  a[t10] := x
(22)  goto (5)
(23)  t11 := 4*i
(24)  x := a[t11]
(25)  t12 := 4*i
(26)  t13 := 4*n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4*n
(30)  a[t15] := x
```

Fig. 10.4. Three-address code for fragment in Fig. 10.2.

2. The intermediate code can be (relatively) independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for a different machine. The intermediate code in Fig. 10.4 assumes that each element of the array a takes four bytes. Some intermediate codes, e.g., P-code for Pascal, leave it to the code generator to fill in the size of array elements, so the intermediate code is independent of the size of a machine word. We could have done the same in our intermediate code if we replaced 4 by a symbolic constant.

In the code optimizer, programs are represented by flow graphs, in which edges indicate the flow of control and nodes represent basic blocks, as discussed in Section 9.4. Unless otherwise specified, a program means a single procedure. In Section 10.8, we discuss interprocedural optimization.

Example 10.1. Figure 10.5 contains the flow graph for the program in Fig. 10.4. B_1 is the initial node. All conditional and unconditional jumps to statements in Fig. 10.4 have been replaced in Fig. 10.5 by jumps to the block of which the statements are leaders.

In Fig. 10.5, there are three loops. B_2 and B_3 are loops by themselves. Blocks B_2, B_3, B_4, and B_5 together form a loop, with entry B_2. □

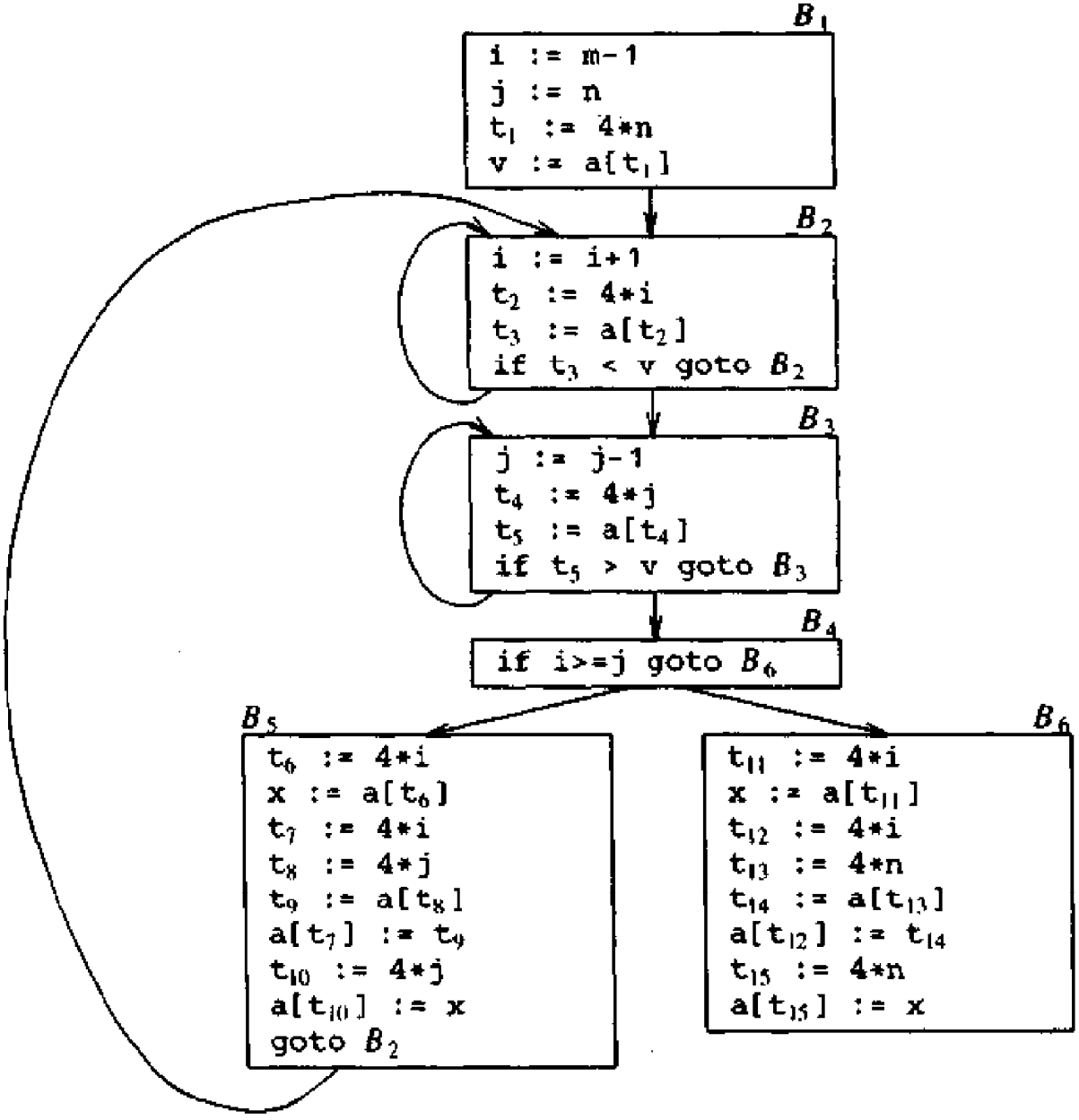


Fig. 10.5. Flow graph.

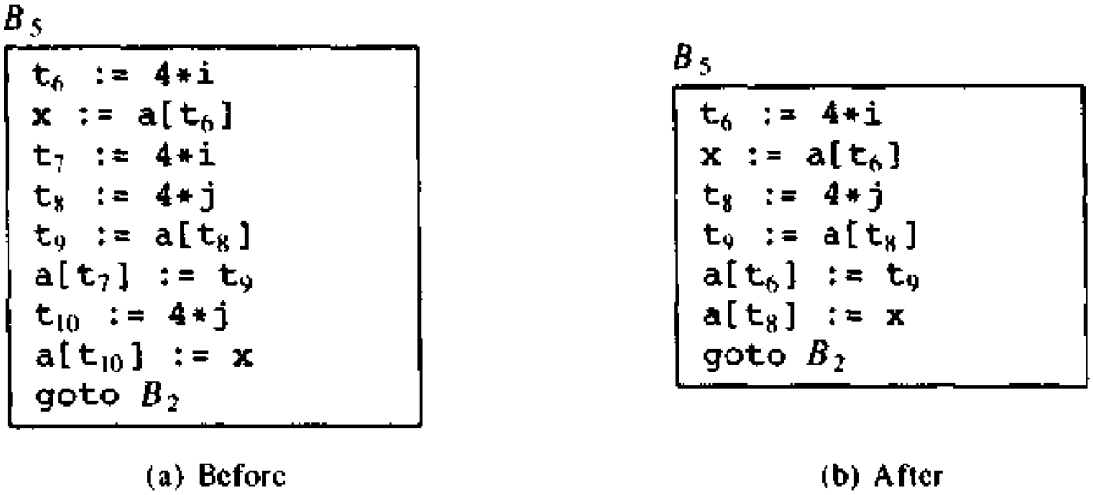
10.2 THE PRINCIPAL SOURCES OF OPTIMIZATION

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`.



(a) Before (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.

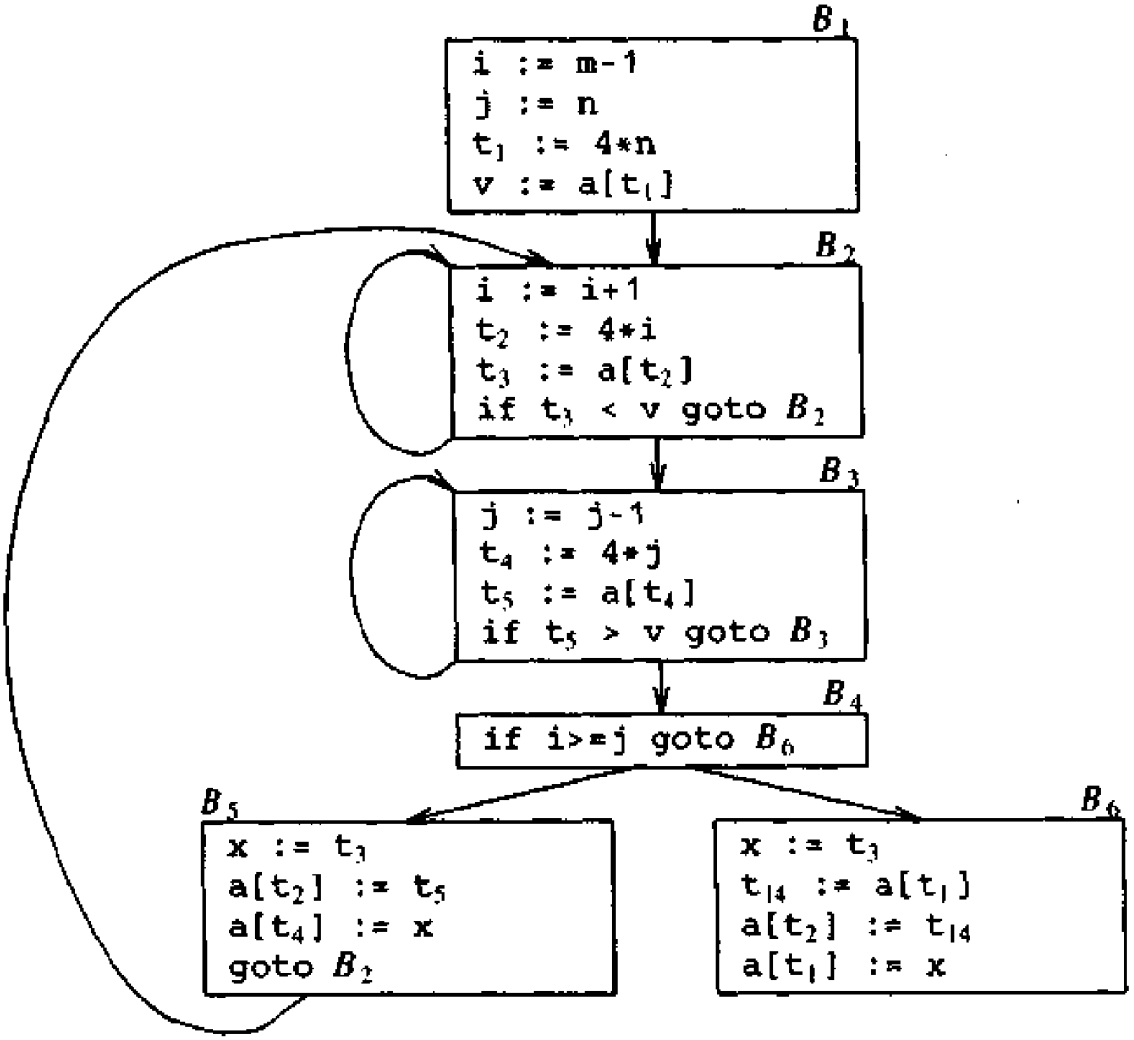


Fig. 10.7. B_5 and B_6 after common subexpression elimination.

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

```
t8 := 4*j;  t9 := a[t8];  a[t8] := x
```

in B_5 can be replaced by

```
t9 := a[t4];  a[t4] := 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
a[t4] := t3                    (10.1)
goto B2
```

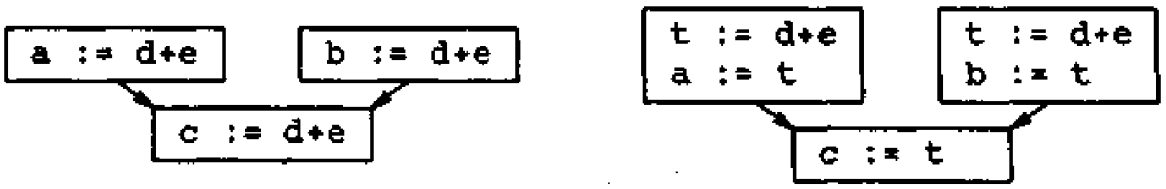


Fig. 10.8. Copies introduced during common subexpression elimination.

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

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

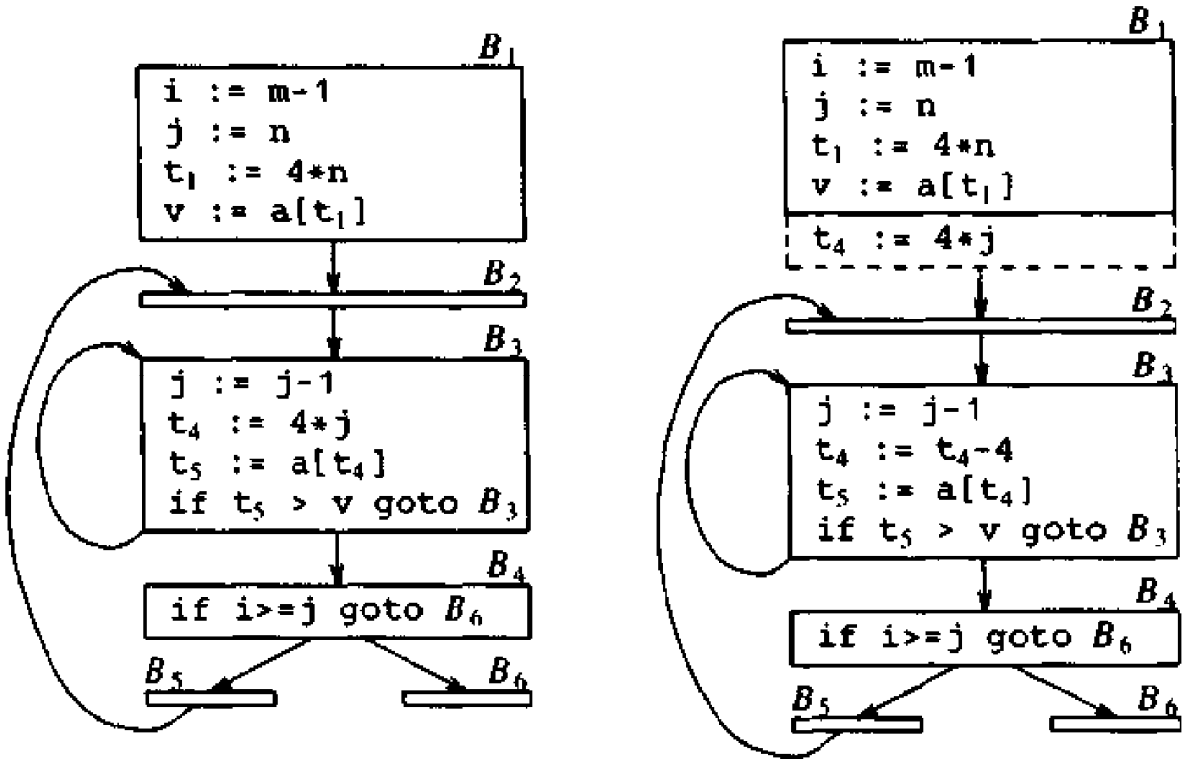


Fig. 10.9. Strength reduction applied to `4*j` in block B_3.

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. □

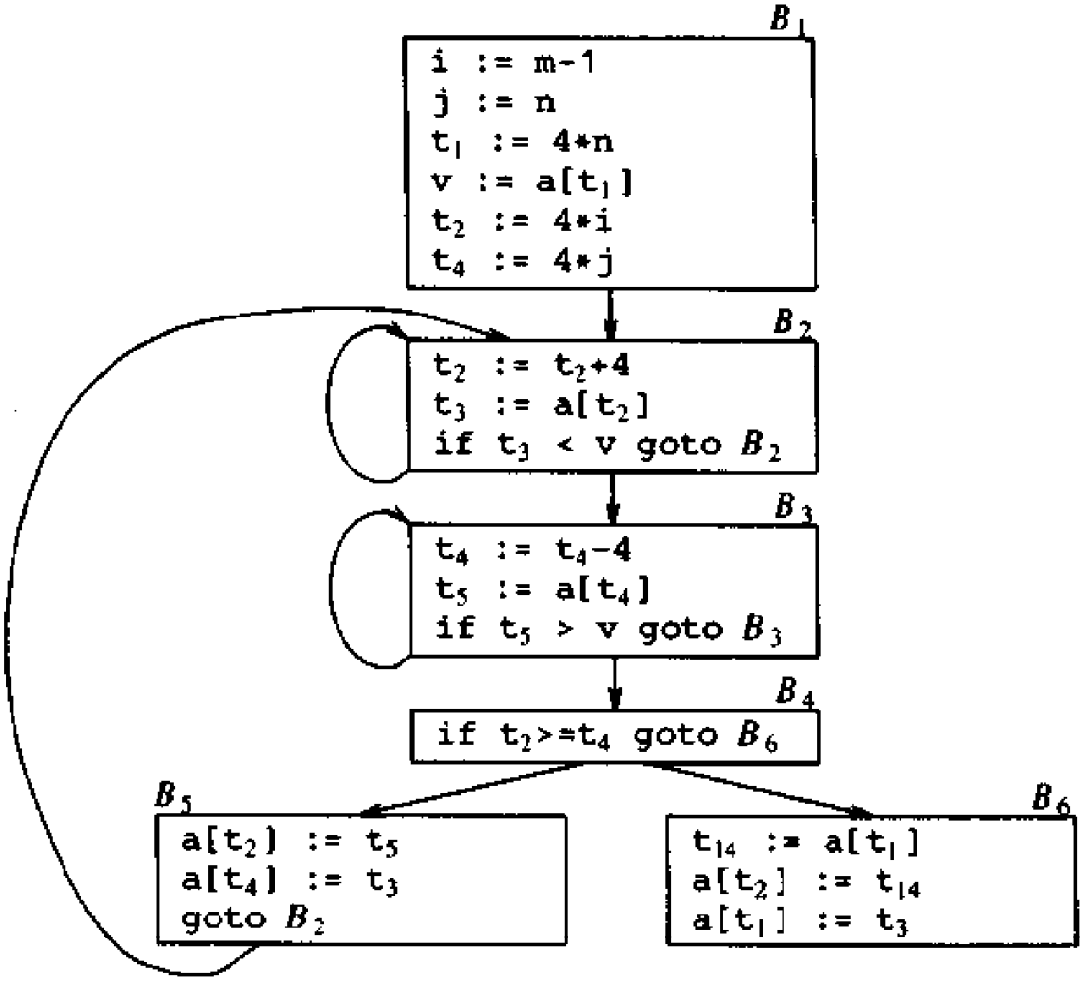


Fig. 10.10. Flow graph after induction-variable elimination.

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

applied at s, and also attached to n is the list of variables for which it is the last definition within the block. We also note those nodes, if any, whose values are live on exit from the block; these are the output nodes.

Common subexpressions can be detected by noticing, as a new node m is about to be added, whether there is an existing node n with the same children, in the same order, and with the same operator. If so, n computes the same value as m and may be used in its place.

Example 10.5. A dag for the block (10.3)

```
a := b + c
b := a - d
c := b + c                                    (10.3)
d := a - d
```

is shown in Fig. 10.11. When we construct the node for the third statement `c:=b+c`, we know that the use of `b` in `b+c` refers to the node of Fig. 10.11 labeled -, because that is the most recent definition of `b`. Thus, we do not confuse the values computed at statements one and three.

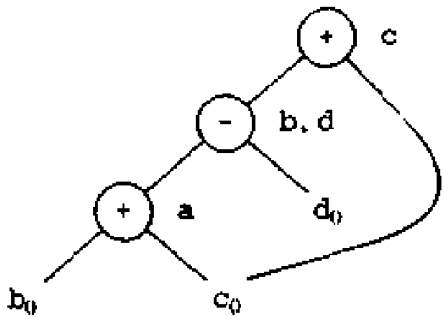


Fig. 10.11. Dag for basic block (10.3).

However, the node corresponding to the fourth statement `d:=a-d` has the operator - and the nodes labeled `a` and d_0 as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add `d` to the list of definitions for the node labeled -. □

It might appear that, since there are only three nodes in the dag of Fig. 10.11, block (10.3) can be replaced by a block with only three statements. In fact, if either `b` or `d` is not live on exit from the block, then we do not need to compute that variable, and can use the other to receive the value represented by the node labeled - in Fig. 10.11. For example, if `b` is not live on exit, we could use:

```
a := b + c
d := a - d
c := d + c
```

However, if both b and d are live on exit, then a fourth statement must be used to copy the value from one to the other.[2]

Note that when we look for common subexpressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed. Thus, the dag method will miss the fact that the expression computed by the first and fourth statements in the sequence

```
a := b + c
b := b - d
c := c + d                    (10.4)
e := b + c
```

is the same, namely, b+c. However, algebraic identities applied to the dag, as discussed next, may expose the equivalence. The dag for this sequence is shown in Fig. 10.12.

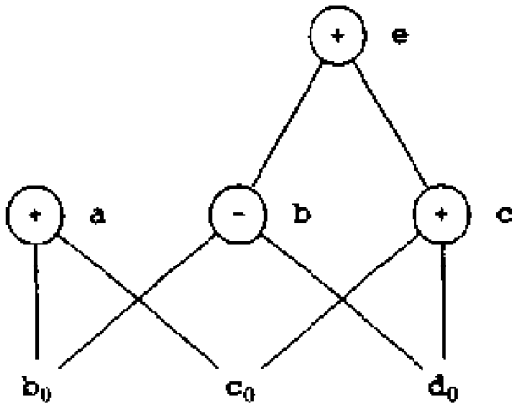


Fig. 10.12. Dag for basic block (10.4).

The operation on dags that corresponds to dead-code elimination is quite straightforward to implement. We delete from a dag any root (node with no ancestors) that has no live variables. Repeated application of this transformation will remove all nodes from the dag that correspond to dead code.

The Use of Algebraic Identities

Algebraic identities represent another important class of optimizations on basic blocks. In Section 9.9, we introduced some simple algebraic transformations that one might try during optimization. For example, we may apply arithmetic identities, such as

[2] In general, we have to be careful when reconstructing code from dags to choose the names of variables corresponding to nodes carefully. If a variable x is defined twice, or if it is assigned once and the initial value x_0 is also used, then we must make sure that we do not change the value of x until we have made all uses of the node whose value x previously held.