CHAPTER NINE
9. CODE OPTIMIZATION

9.1. Introduction

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

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 *optimization*.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- ✓ The output code must not, in any way, change the meaning of the program.
- ✓ Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- ✓ Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- ✓ At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- ✓ After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- ✓ While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types: machine independent and machine dependent.

- ❖ Machine - independent Optimization: In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. It improves the code without taking into consideration any properties of the target machine. For example:

```
do
{
   item = 10;
   value = value + item;
}while(value<100);
```

This code involves repeated assignment of the identifier *item*, which if we put this way:

```
item = 10;
do
{
    value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

- Machine - dependent Optimization: Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy. It considers register allocation and utilization of special machine-instruction sequences.

Criteria for Code-Improving Transformations

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.

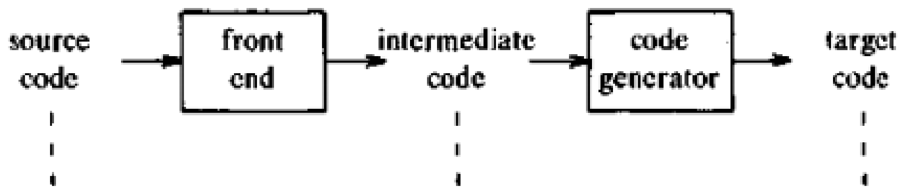
First, a transformation must preserve the meaning of programs. That is, an "optimization" must not change the output produced by a program for a given input, or cause an error, such as a division by zero, that was not present in original version of the source program.

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.

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 in the following figure. 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.



user can	**compiler can**	**compiler can**
profile program	improve loops	use registers
change algorithm	procedure calls	select instructions
transform loops	address calculations	do peephole transformations

Figure 9.1: Places for potential improvements by the user and the compiler

9.2. 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 level. Local transformations are usually performed first.

❖ A Running Example: Quicksort

In the following, we shall use a fragment of a sorting program called *quicksort* to illustrate several important code-improving transformations. The following is the c code for quick sort.

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[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; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Figure 9.2: C code for quick sort

In this example we assume that integers occupy four bytes. The assignment `x = a[i]` is translated into the two three-address statements t_6 = 4*i, x = a[t_6] as shown in steps (14) and (15) of the following figure. Similarly, `a[j] = x` become `t10 = 4*j, a[t10] = x` in steps (20) and (21). The three address code for the above code segment is the following.

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

Figure 9.3: Three address code for quick sort

The flow graph for the above three address code is the following.

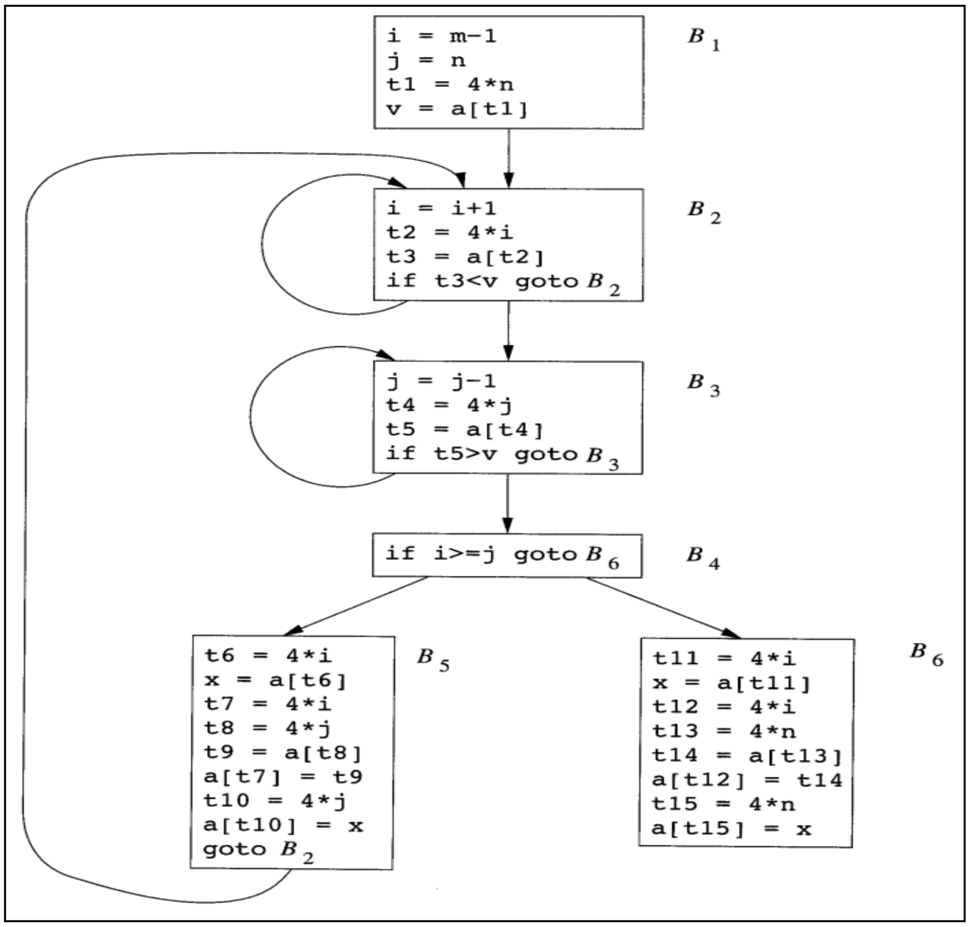
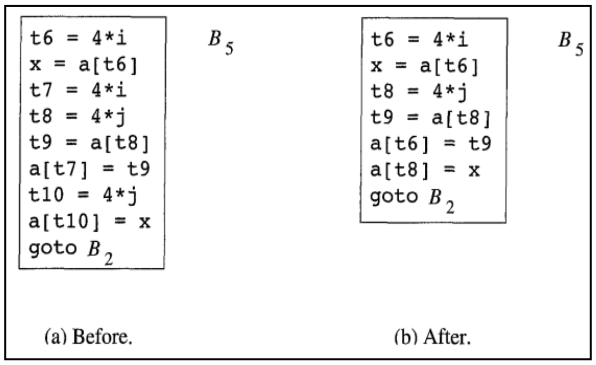


Figure 9.4: Flow graph for quick sort fragment

❖ Function-Preserving Transformation

There are a number of ways in which a compiler can improve a program without changing the function it computes. Common-sub expression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such function-preserving (or *semantics-preserving)* transformations.

Frequently, a program will include several calculations of the same value, such as an offset in an array. 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 *B5* shown in the following figure (a) recalculates `4 * i` and `4* j`, although none of these calculations were requested explicitly by the programmer.



```
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
```

B5

(a) Before.

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

B5

(b) After.

Figure 9.5: Local common-sub expression elimination

❖ **Common Subexpressions**

An occurrence of an expression *E* is called a *common subexpression* if *E* was previously computed and the values of the variables in *E* have not changed since the previous computation. We can avoid re-computing the expression if we can use the previously computed value.

Example-1: The assignments to $\mathbf{t_7}$ and $\mathbf{t_{10}}$ in Fig. 9.5(a) compute the common subexpressions `4*i` and 4 * j, respectively. These steps have been eliminated in Fig. 9.4(b), which uses $\mathbf{t_6}$ instead of $\mathbf{t_7}$ and $\mathbf{t_8}$ instead of $\mathbf{t_{10}}$.

Example-2: The following figure shows the result of eliminating both global and local common subexpressions from blocks B_5 and B_6 in the flow graph of Fig. 9.4. We first discuss the transformation of B_5 and then mention some subtleties involving arrays.

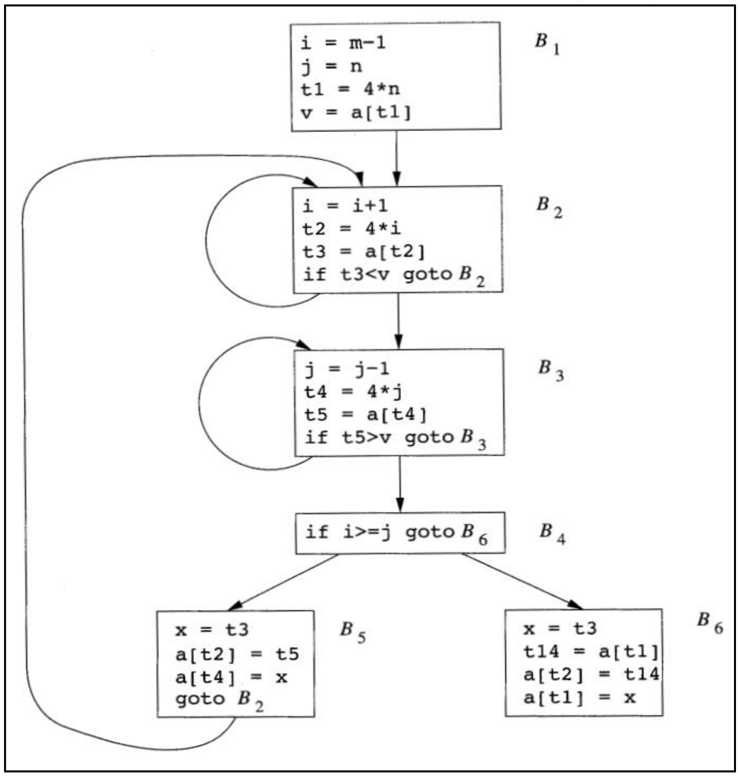


Figure *9.6:* B_5 and B_6 after common-subexpression elimination

After local common subexpressions are eliminated, B_5 still evaluates 4*i and 4 * *j* , as shown in Fig. 9.5(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. *9.6*, observe that as control passes from the evaluation of *4*j* in B_3 to B_5, there is no change to *j* and no change to t_4, 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

$$t_9 = a[t_4]$$
$$a[t_6] = t_9$$

in B_5 therefore can be replaced by

$$a[t_6] = t_5$$

Analogously, the value assigned to x in block B_5 of Fig. 9.5(b) is seen to be the same as the value assigned to t_3 in block B_2. Block B_5 in Fig. 9.6 is the result of eliminating common subexpressions corresponding to the values of the source level expressions a[i] and a[j] from B5 in Fig. 9.5(b). A similar series of transformations has been done to B_6 in Fig. 9.6.

The expression a[t_l] in blocks B_1 and B_6 of Fig. 9.6 is not considered a common subexpression, although t_l 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_l] 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_l] as a common subexpression.

❖ Copy Propagation

Block B5 in Fig. 9.6 can be further improved by eliminating x, using two new transformations. One concerns assignments of the form `u = v` called copy statements, or copies for short.

Had we gone into more detail in Example-2 above, copies would have arisen much sooner, because the normal algorithm for eliminating common subexpressions introduces them, as do several other algorithms.

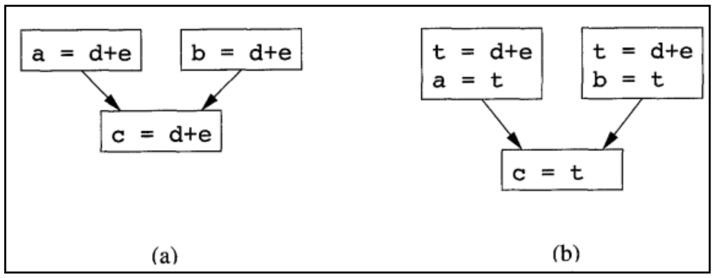
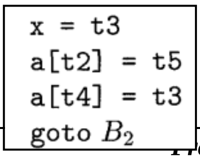


Figure 9.7: Copies introduced during common subexpression elimination

Example-3: In order to eliminate the common subexpression from the statement `c = d+e` in Fig. 9.7(a), we must use a new variable t to hold the value of `d+e`. The value of variable t, instead of that of the expression `d+e`, is assigned to c in Fig. 9.7(b). 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 v for u, wherever possible after the copy statement u = v. For example, the assignment x = t_3 in block B_5 of Fig. 9.6 is a copy. Copy propagation applied to B_5 yields the code in Fig. 9.8. This change may not appear to be an improvement, but it gives us the opportunity to eliminate the assignment to x.

Figure 9.8: Basic block *B5* after copy propagation



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

❖ 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.

Example-4: Suppose `debug` is set to TRUE or FALSE at various points in the program, and used in statements like

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

It may be possible for the compiler 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 must be the last assignment to debug prior to any tests of the value of debug, 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 the print operation 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 the code in Fig 9.8 into

$$\begin{aligned}&a[t_2] = t_5\\&a[t_4] = t_3\\&\text{goto } B2\end{aligned}$$

This code is a further improvement of block *B5* in Fig. 9.6.

❖ Code Motion

Loops are a very important place for optimizations, 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.

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 evaluates the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop, that is, one basic block to which all jumps from outside the loop go.

Example-5: 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 code

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

Now, the computation of *limit- 2* is performed once, before we enter the loop. Previously, there would be *n+1* calculations of *limit-2* if we iterated the body of the loop *n* times.

9.3. Optimization of Basic Blocks

We only see here one method for optimizing basic blocks: use of algebraic identities. See the other method, induction variable elimination, from the slide.

Algebraic identities represent another important class of optimizations on basic blocks. For example, we may apply arithmetic identities, such as

$x + 0 = 0 + x$ $x - 0 = x$

$x * 1 = 1 * x$ $x / 1 = x$

to eliminate computations from a basic block.

Another class of algebraic optimizations includes local reduction in strength that is, replacing a more expensive operator by a cheaper one as in:

Expensive		*Cheaper*
X^2	=	$x * x$
$2 * x$	=	$x + x$
$X / 2$	=	$x * 0.5$

A third class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their value. Thus the expression *2 ** 3.14 would be replaced by 6.28. Many constant expressions arise in practice because of the frequent use of symbolic constants in programs.

9.4. Peephole Optimization

While 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. 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.

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.

The peephole is a small, sliding window on a 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
- ✓ Use of machine idioms

❖ Eliminating Redundant Loads and Stores

If we see the instruction sequence

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

in a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of a has already been loaded into register **R_0.** Note that if the store instruction had a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction. Put another way, the two instructions have to be in the same basic block for this transformation to be safe.

❖ Eliminating 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 code fragments that are executed only if a variable debug is equal to 1. In the intermediate representation, this code may look like

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

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of debug, the code sequence above can be replaced by

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

If debug is set to 0 at the beginning of the program, constant propagation would transform this sequence into:

```
  if (0 != 1) goto L2
  print debugging information
L2:
```

Now the argument of the first statement always evaluates to true, so the statement can be replaced by goto L2. Then all statements that print debugging information are unreachable and can be eliminated one at a time.

❖ **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

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

While the number of instructions in the two sequences is the same, we sometimes skip the unconditional jump in the second sequence, but never in the first. Thus, the second sequence is superior to the first in execution time.

❖ **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.