10 CODE OPTIMIZATION

10.1 INTRODUCTION TO CODE OPTIMIZATION

The translation of a source program to an object program is basically one to many mappings; that is, there are many object programs for the same source program, which implement the same computations. Some of these object-programs may be better than other object programs when it comes to storage requirements and execution speeds. Code optimization refers to techniques a compiler can employ in order to produce an improved object code for a given source program.

How beneficial the optimization is depends upon the situation. For a program that is only expected to be run a few times, and which will then be discarded, no optimization is necessary. Whereas if a program is expected to run indefinitely, or if it is expected to run many times, then optimization is useful, because the effort spent on improving the program's execution time will be paid back, even if execution time is only reduced by a small percentage.

What follows are some optimization techniques that are useful when designing optimizing compilers.

10.2 WHAT IS CODE OPTIMIZATION?

Code optimization refers to the techniques used by the compiler to improve the execution efficiency of the generated object code. It involves a complex

analysis of the intermediate code and the performance of various transformations; but every optimizing transformation must also preserve the semantics of the program. That is, a compiler should not attempt any optimization that would lead to a change in the program's semantics.

Optimization can be machine-independent or machine-dependent. Machine-independent optimizations can be performed independently of the target machine for which the compiler is generating code; that is, the optimizations are not tied to the target machine's specific platform or language. Examples of machine-independent optimizations are: elimination of loop invariant computation, induction variable elimination, and elimination of common subexpressions.

On the other hand, machine-dependent optimization requires knowledge of the target machine. An attempt to generate object code that will utilize the target machine's registers more efficiently is an example of machine-dependent code optimization. Actually, code optimization is a misnomer; even after performing various optimizing transformations, there is no guarantee that the generated object code will be optimal. Hence, we are actually performing code improvement. When attempting any optimizing transformation, the following criteria should be applied:

1. The optimization should capture most of the potential improvements without an unreasonable amount of effort.
2. The optimization should be such that the meaning of the source program is preserved.
3. The optimization should, on average, reduce the time and space expended by the object code.

10.3 LOOP OPTIMIZATION

Loop optimization is the most valuable machine-independent optimization because a program's inner loops are good candidates for improvement. The important loop optimizations are elimination of loop invariant computations and elimination of induction variables. A loop invariant computation is one that computes the same value every time a loop is executed. Therefore, moving such a computation outside the loop leads to a reduction in the execution time. Induction variables are those variables used in a loop; their values are in lock-step, and hence, it may be possible to eliminate all except one.

10.3.1 Eliminating Loop Invariant Computations

To eliminate loop invariant computations, we first identify the invariant computations and then move them outside loop if the move does not lead to a change in the program's meaning. Identification of loop invariant computation requires the detection of loops in the program. Whether a loop exists in the program or not depends on the program's control flow, therefore, requiring a control flow analysis. For loop detection, a graphical representation, called a "program flow graph," shows how the control is flowing in the program and how the control is being used. To obtain such a graph, we must partition the intermediate code into basic blocks. This requires identifying leader statements, which are defined as follows:

1. The first statement is a leader statement.
2. The target of a conditional or unconditional goto is a leader.
3. A statement that immediately follows a conditional goto is a leader.

A basic block is a sequence of three-address statements that can be entered only at the beginning, and control ends after the execution of the last statement, without a halt or any possibility of branching, except at the end.

10.3.2 Algorithm to Partition Three-Address Code Into Basic Blocks

To partition three-address code into basic blocks, we must identify the leader statements in the three-address code and then include all the statements, starting from a leader, and up to, but not including, the next leader. The basic blocks into which the three-address code is partitioned constitute the nodes or vertices of the program flow graph. The edges in the flow graph are decided as follows. If $B1$ and $B2$ are the two blocks, then add an edge from $B1$ to $B2$ in the program flow graph, if the block $B2$ follows $B1$ in an execution sequence. The block $B2$ follows $B1$ in an execution sequence if and only if:

1. The first statement of block $B2$ immediately follows the last statement of block $B1$ in the three-address code, and the last statement of block $B1$ is not an unconditional goto statement.
2. The last statement of block $B1$ is either a conditional or unconditional goto statement, and the first statement of block $B2$ is the target of the last statement of block $B1$.

For example, consider the following program fragment:

```
Fact(x)
{
    int f = 1;
    for(i = 2; i<=x; i++)
    f = f*i;
    return(f);
}
```

The three-address-code representation for the program fragment above is:

(1) $f = 1$;
(2) $i = 2$
(3) if $i <= x$ goto(8)
(4) $f = f * i$
(5) $t1 = i + 1$
(6) $i = t1$
(7) goto(3)
(8) goto calling program

The leader statements are:

- Statement number 1, because it is the first statement.
- Statement number 3, because it is the target of a goto.
- Statement number 4, because it immediately follows a conditional goto statement.
- Statement number 8, because it is a target of a conditional goto statement.

Therefore, the basic blocks into which the above code can be partitioned are as follows, and the program flow graph is shown in Figure 10.1.

- **Block *B1*:** $f = 1$; $i = 2$
- **Block *B2*:** if $i <= x$ goto(8)
- **Block *B3*:** $f = f * i$; $t1 = i + 1$; $i = t1$; goto(3)
- **Block *B4*:** goto calling program

Even though x is not used outside the loop, the statement $x = 2$ in the block $B2$ cannot be moved to the preheader, because the use of x in $B4$ is also reached by the definition $x = 1$ in $B1$. Therefore, if we move $x = 2$ to the preheader, then the value that will get assigned to a in $B4$ will always be a 1, which is not the case in the original program.

10.4 ELIMINATING INDUCTION VARIABLES

We define basic induction variables of a loop as those names whose only assignments within the loop are of the form $I = I \pm C$, where C is a constant or a name whose value does not change within the loop. A basic induction variable may or may not form an arithmetic progression at the loop header.

For example, consider the flow graph shown in Figure 10.10. In the loop formed by $B2$, I is a basic induction variable.

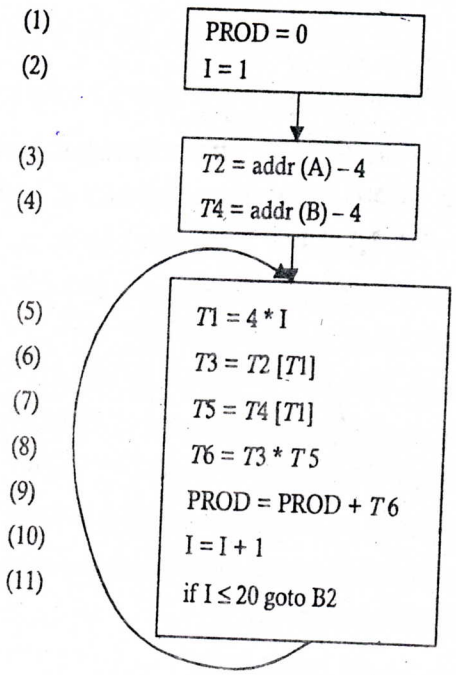

FIGURE 10.10 Flow graph where I is a basic induction variable.

We then define an induction variable of loop L as either a basic induction variable or a name J for which there is a basic induction variable I, such that each time J is assigned in L, J's value is some linear function or value of I. That is, the value of J in L should be $C_1I + C_2$, where C_1 and C_2 could be

functions of both constants and loop invariant names. For example, in loop L, I is a basic induction variable; and $T1$ is also an induction variable, because the only assignment of $T1$ in the loop assigns a value to $T1$ that is a linear function of I, computed as $4 * I$.

Algorithm for Detecting and Eliminating Induction Variables

An algorithm exists that will detect and eliminate induction variables. Its method is as follows:

1. Find all of the basic induction variables by scanning the statements of loop L.
2. Find any additional induction variables, and for each such additional induction variable A, find the family of some basic induction B to which A belongs. (If the value of A at the point of assignment is expressed as $C_1B + C_2$, then A is said to belong to the family of basic induction variable B). Specifically, we search for names A with single assignments to A within loop L, and which have one of the following forms:

$$A = B * C$$
$$A = C * B$$
$$A = B/C$$
$$A = B \pm C$$
$$A = C \pm B$$

where C is a loop constant, and B is an induction variable, basic or otherwise. If B is basic, then A is in the family of B. If B is not basic, let B be in the family of D, then the additional requirements to be satisfied are:

(*a*) There must be no assignment to D between the lone point of assignment to B in L and the assignment to A.

(*b*) There must be no definition of B outside of L reaches A.

3. Consider each basic induction variable B in turn. For every induction variable A in the family of B:

(*a*) Create a new name, temp.

(*b*) Replace the assignment to A in the loop with A = temp.

(*c*) Set temp to $C_1B + C_2$ at the end of the preheader by adding the statements:

```
temp = C1 * B
temp = temp + C2 /* omit if C2 = 0*/
```

(*d*) Immediately after each assignment $B = B + D$, where D is a loop invariant, append:

$$\text{temp} = \text{temp} + C_1 * D$$

If D is a loop invariant name, and if $C_1 \neq 1$, create a new loop invariant name for $C_1 * D$, and add the statements:

$$\text{temp1} = C_1 * D$$

$$\text{temp} = \text{temp} + \text{temp1}$$

(*e*) For each basic induction variable B whose only uses are to compute other induction variables in its family and in conditional branches, take some A in B's family, preferably one whose function expresses its value simply, and replace each test of the form B reloop X goto Y by:

$$\text{temp2} = C_1 * X$$

$$\text{temp2} = \text{temp2} + C_2 \text{ /* omit if } C_2 = 0 \text{ */}$$

$$\text{if temp reloop temp2 goto } Y$$

Delete all assignments to B from the loop, as they will now be useless.

(*f*) If there is no assignment to temp between the introduced statement A = temp (step 1) and the only use of A, then replace all uses of A by temp and delete the statement A = temp.

In the flow graph shown in Figure 10.10, we see that I is a basic induction variable, and $T1$ is the additional induction variable in the family of I, because the value of $T1$ at the point of assignment in the loop is expressed as $T1 = 4 * I$. Therefore, according to step 3b, we replace $T1 = 4 * I$ by $T1$ = temp. And according to step 3c, we add temp = $4 * I$ to the preheader. We then append the statement temp = temp + 4 after statement (10), as shown in Figure 10.10 as per step 3d. And according to step 3e, we replace the statement if $I \leq 20$ goto $B2$ by:

$$\text{temp1} = 80$$

$$\text{if (temp} \leq \text{temp1) goto } B2\text{, and delete } I = I + 1$$

The results of these modifications are shown in Figure 10.11.

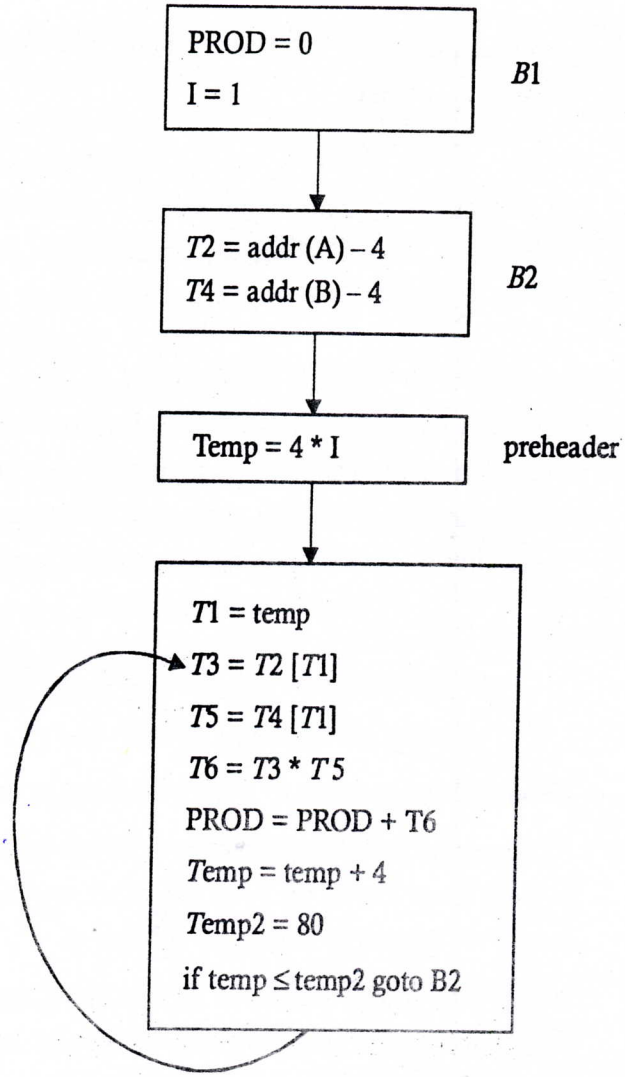

FIGURE 10.11 Modified flow graph.

By step 3f, replace *T*1 by temp. And by copy propagation, temp = 4 * *I*, in the preheader, can be replaced by temp = 4, and the statement *I* = 1 can be eliminated. In *B*1, the statement if temp ≤ temp2 goto *B*2 can be replaced by if temp ≤ 80 goto *B*2, and we can eliminate temp2 = 80, as shown in Figure 10.12.

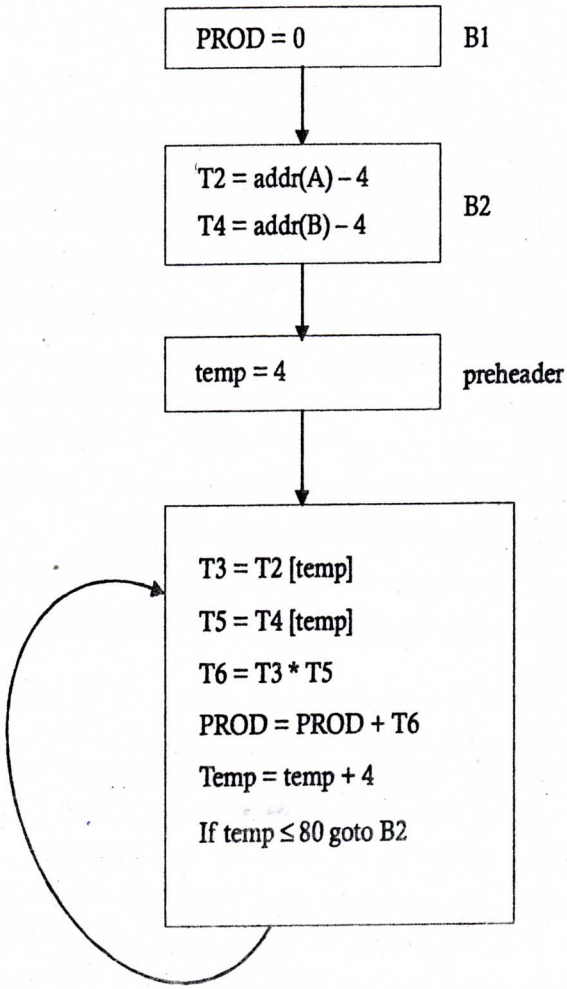

FIGURE 10.12 Flow graph preheader modifications.

10.5 ELIMINATING LOCAL COMMON SUBEXPRESSIONS

The first step in eliminating local common subexpressions is to detect the common subexpression in a basic block. The common subexpressions in a basic block can be automatically detected if we construct a directed acyclic graph (DAG).

DAG Construction

For constructing a basic block DAG, we make use of the function node(id), which returns the most recently created node associated with id. For every three-address statement $x = y\ op\ z$, $x = op\ y$, or $x = y$ in the block we:

do

{

1. If node(y) is undefined, create a leaf labeled y, and let node(y) be this node. If node(z) is undefined, create a leaf labeled z, and let that leaf be node(z). If the statement is of the form $x = op\ y$ or $x = y$, then if node(y) is undefined, create a leaf labeled y, and let node(y) be this node.
2. If a node exists that is labeled op whose left child is node(y) and whose right child is node(z) (to catch the common subexpressions), then return this node. Otherwise, create such a node, and return it. If the statement is of the form $x = op\ y$, then check if a node exists that is labeled op whose only child is node(y). Return this node. Otherwise, create such a node and return. Let the returned node be n.
3. Append x to the list of identifiers for the node n returned in step 2. Delete x from the list of attached identifiers for node(x), and set node(x) to be node n.

}

Therefore, we first go for a DAG representation of the basic block. And if the interior nodes in the DAG have more than one label, then those nodes of the DAG represent the common subexpressions in the basic block. After detecting these common subexpressions, we eliminate them from the basic block. The following example shows the elimination of local common subexpressions, and the DAG is shown in Figure 10.13.

(1) $S1 := 4 * I$

(2) $S2$: addr(A) – 4

(3) $S3 : S2\ [S1]$

(4) $S4 : 4 * I$

(5) $S5 :=$ addr(B) – 4

(6) $S6 := S5\ [S4]$

(7) $S7 := S3 * S6$

(8) $S8$: PROD + $S7$

(9) PROD : = $S8$

(10) $S9 := I + 1$

(11) $I = S9$

(12) if $I \leq 20$ goto (1).

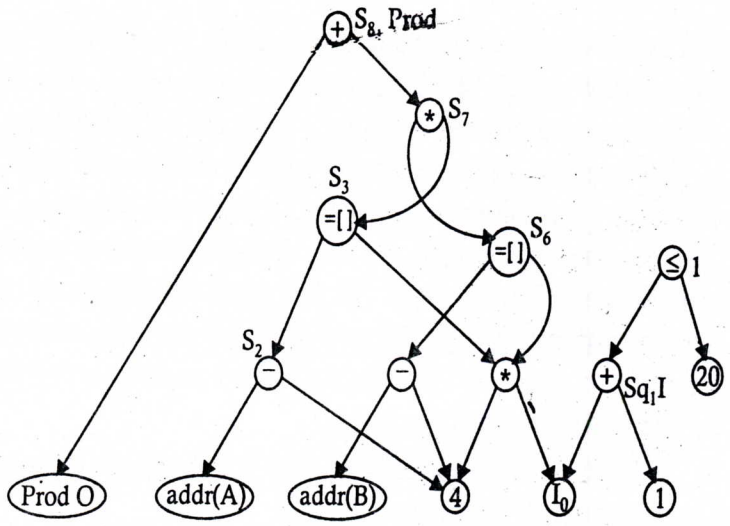

FIGURE 10.13 DAG representation of a basic block.

In Figure 10.13, PROD 0 indicates the initial value of PROD, and *I*0 indicates the initial value of *I*. We see that the same value is assigned to *S*8 and PROD. Similarly, the value assigned to *S*9 is the same as *I*. And the value computed for *S*1 and *S*4 are the same; hence, we can eliminate these common subexpressions by selecting one of the attached identifiers (one that is needed outside the block). We assume that none of the temporaries is needed outside the block. The rewritten block will be:

(1) $S1 := 4 * I$
(2) $S2 := \text{addr}(A) - 4$
(3) $S3 := S2\ [S1]$
(4) $S5 := \text{addr}(B) - 4.$
(5) $S6 := S5\ [S1]$
(6) $S7 := S3 * S6$
(7) $\text{PROD} := \text{PROD} + S7$
(8) $I := I + 1$
(9) if $I \leq 20$ goto (1)

10.6 ELIMINATING GLOBAL COMMON SUBEXPRESSIONS

Global common subexpressions are expressions that compute the same value but in different basic blocks. To detect such expressions, we need to compute available expressions.

```
        INnew(bi) = Φ
        for each predecessor p of bi
        INnew(bi) = INnew(bi) ∩ OUT(p)
        if INnew(bi) ≠ IN(bi) then
            {
            flag = true
            IN(bi) = INnew(bi)
            OUT(bi) = IN(bi) – KILL(bi) ∪ GEN(bi)
             }
        }
    }
```

After computing IN(b) and OUT(b), eliminating the global common subexpressions is done as follows. For every statement s of the form $x = y\ op\ z$ such that $y\ op\ z$ is available at the beginning of the block containing s, and neither y nor z is defined prior to the statement $x = y\ op\ z$ in that block, do:

1. Find all definitions reaching up to the s statement block that have $y\ op\ z$ on the right.
2. Create a new temp.
3. Replace each statement $U = y\ op\ z$ found in step 1 by:

   temp = $y\ op\ z$

   U = temp
4. Replace the statement $x = y\ op\ z$ in block by x = temp.

10.7 LOOP UNROLLING

Loop unrolling involves replicating the body of the loop to reduce the required number of tests if the number of iterations are constant. For example consider the following loop:

```
I = 1
while (I <= 100)
{
    x[I] = 0;
    I++;
}
```

In this case, the test $I <= 100$ will be performed 100 times. But if the body of the loop is replicated, then the number of times this test will need to be performed will be 50. After replication of the body, the loop will be:

```
I = 1
while(I<= 100)
{
    x[I] = 0;
    I++;
    X[I] = 0;
    I++;
}
```

It is possible to choose any divisor for the number of times the loop is executed, and the body will be replicated that many times. Unrolling once—that is, replicating the body to form two copies of the body—saves 50% of the maximum possible executions.

10.8 LOOP JAMMING

Loop jamming is a technique that merges the bodies of two loops if the two loops have the same number of iterations and they use the same indices. This eliminates the test of one loop. For example, consider the following loop:

```
{
for (I = 0; I < 10; I++)
    for (J = 0; J < 10; J++)
        X[I,J] = 0;
    for (J = 0; I < 10; I++)
        X[I,I] = 1;
}
```

Here, the bodies of the loops on I can be concatenated. The result of loop jamming will be:

```
{
for (I = 0; I < 10; I++)
    {
        for (J = 0; J < 10; J++)
```

```
            X[I,J] = 0;
    X[I,I] = 1;
    }
}
```

The following conditions are sufficient for making loop jamming legal:

1. No quantity is computed by the second loop at the iteration I if it is computed by the first loop at iteration $J \geq I$.
2. If a value is computed by the first loop at iteration $J \geq I$, then this value should not be used by second loop at iteration I.