# OPTIMIZATION

## 8.1 PRINCIPLE SOURCES OF OPTIMIZATION

A transformation of a program is called local if it can be performed by looking only at the statements in a bas9ic 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 sub expression elimination, copy propagation, deadcode elimination, and constant folding are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed. 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 recalculates 4*i and 4*j.

Common Sub expressions An occurrence of an expression E is called a common sub expression if E was previously computed, and the values of 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. For example, the assignments to t7 and t10 have the common sub expressions 4*I and 4*j, respectively, on the right side in Fig. They have been eliminated in Fig by using t6 instead of t7 and t8 instead of t10. This change is what would result if we reconstructed the intermediate code from the dag for the basic block.

Example: the above Fig shows the result of eliminating both global and local common sub expressions from blocks B5 and B6 in the flow graph of Fig. We first discuss the transformation of B5 and then mention some subtleties involving arrays.

After local common sub expressions are eliminated B5 still evaluates 4*i and 4*j, as

Shown in the earlier fig. Both are common sub expressions; in particular, the three statements t8:= 4*j; t9:= a[t[8]; a[t8]:=x in B5 can be replaced by t9:= a[t4]; a[t4:= x using t4 computed in block B3. In Fig. observe that as control passes from the evaluation of 4*j in B3 to B5, there is no change in j, so t4 can be used if 4*j is needed.

Another common sub expression comes to light in B5 after t4 replaces t8. The new expression a[t4] corresponds to the value of a[j] at the source level. Not only does j retain its value as control leaves b3 and then enters B5, but a[j], a value computed into a temporary t5, does too because there are no assignments to elements of the array a in the interim. The statement t9:= a[t4]; a[t6]:= t9 in B5 can therefore be replaced by

a[t6]:= t5 The expression in blocks B1 and B6 is not considered a common sub expression although t1 can be used in both places. After control leaves B1 and before it reaches B6,it can go through B5,where there are assignments to a. Hence, a[t1] may not have the same value on reaching B6 as it did in leaving B1, and it is not safe to treat a[t1] as a common sub expression.

Copy Propagation

Block B5 in Fig. 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 sub expressions introduces them, as do several other algorithms. For example, when the common sub expression in c:=d+e is eliminated in Fig., 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:=t3 in block B5 of Fig. is a copy. Copy propagation applied to B5 yields:

x:=t3

a[t2]:=t5

a[t4]:=t3

goto B2 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.

## 8.2 DEAD-CODE ELIMINATIONS

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, 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. 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 no matter what sequence of branches the program actually takes. If copy propagation replaces debug by false, then theprint statement is dead because it cannot be reached. We can eliminate both the test and printing from the o9bject 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 1.1 into

a [t2 ] := t5

a [t4] := t3

goto B2

## 8.3 PEEPHOLE OPTIMIZATION

A statement-by-statement code-generations strategy often produce 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.

A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving 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.

The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this. We shall give the following examples of program transformations that are characteristic of peephole optimizations:

• Redundant-instructions elimination
• Flow-of-control optimizations
• Algebraic simplifications
• Use of machine idioms

**REDUNTANT LOADS AND STORES**

If we see the instructions sequence

(1) (1) MOV R0,a

(2) (2) MOV a,R0

-we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

**UNREACHABLE CODE**

Another opportunity for peephole optimizations 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 representations the if-statement may be translated as:

```
If debug =1 goto L2
Goto L2
L1: print debugging information
```

L2: .....(a)

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

If debug ≠1 goto L2

Print debugging information

L2: .....(b)

As the argument of the statement of (b) evaluates to a constant true it can be replaced by

If debug ≠0 goto L2

Print debugging information

L2: .....(c)

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

## 8.4 FLOW-OF-CONTROL OPTIMIZATIONS

The 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 L2

....

L1 : gotoL2

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

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: .....(1)

may be replaced by

if a<b goto L2

goto L3

.....

L3: .....(2)

While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

## 8.5 REGISTER ALLOCATION

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

1. During register allocation, we select the set of variables that will reside in registers at a point in the program.

2. During a subsequent register assignment phase, we pick the specific register that a variable will reside in. Finding an optimal assignment of registers to variables is difficult, even with single register values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.

Certain machines require register pairs (an even and next odd numbered register) for some operands and results. For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs. The multiplication instruction is of the form M x, y where x, is the multiplicand, is the even register of an even/odd register pair.

The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The division instruction is of the form D x, y where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor. After division, the even register holds the remainder and the odd register the quotient. Now consider the two three address code sequences (a) and (b) in which the only difference is

the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c). Ri stands for register i. L, ST and A stand for load, store and add respectively. The optimal choice for the register into which 'a' is to be loaded depends on what will ultimately happen to e.

t := a + b t := a + b

t := t * c t := t + c

t := t / d t := t / d

(a) (b)

Two three address code sequences

L R1, a L R0, a

A R1, b A R0, b

M R0, c A R0, c

D R0, d SRDA R0,

ST R1, t D R0, d

ST R1, t

(a) (b)

## 8.6 CHOICE OF OF EVALUATION ORDER

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem. Initially, we shall avoid the

problem by generating code for the three -address statements in the order in which they have been produced by the intermediate code generator.

## 8.7 APPROCHES TO CODE GENERATION

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that code generator must face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal Reference Counting Garbage Collection The difficulty in garbage collection is not the actual process of collecting the garbage--it is the problem of finding the garbage in the first place. An object is considered to be garbage when no references to that object exist. But how can we tell when no references to an object exist? A simple expedient is to keep track in each object of the total number of references to that object. That is, we add a special field to each object called a reference count . The idea is that the reference count field is not accessible to the Java program. Instead, the reference count field is updated by the Java virtual machine itself.

Consider the statement

Object p = new Integer (57);

which creates a new instance of the Integer class. Only a single variable, p, refers to the object. Thus, its reference count should be one.

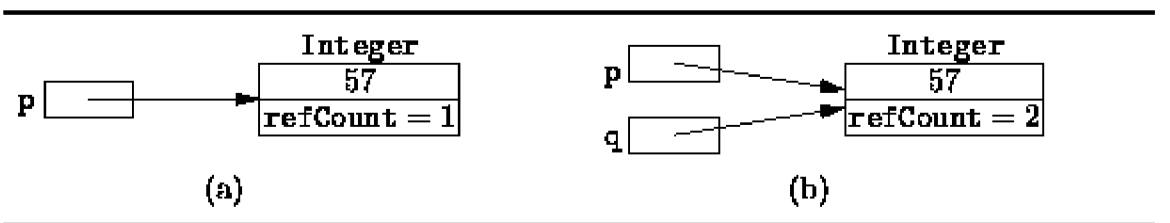


Figure: Objects with reference counters.

Now consider the following sequence of statements:

Object p = new Integer (57);

Object q = p;

This sequence creates a single Integer instance. Both p and q refer to the same object. Therefore, its reference count should be two.

In general, every time one reference variable is assigned to another, it may be necessary to update several reference counts. Suppose p and q are both reference variables. The assignment

p = q;

would be implemented by the Java virtual machine as follows:

```
if (p != q)
{
   if (p != null)
         --p.refCount;
   p = q;
   if (p != null)
         ++p.refCount;
}
```

For example suppose p and q are initialized as follows:

Object p = new Integer (57);

Object q = new Integer (99);

As shown in Figure ⌟(a), two Integer objects are created, each with a reference count of one. Now, suppose we assign q to p using the code sequence given above. Figure ⌟(b) shows that after the assignment, both p and q refer to the same object--its reference count is two. And the reference count on Integer(57) has gone to zero which indicates that it is garbage.

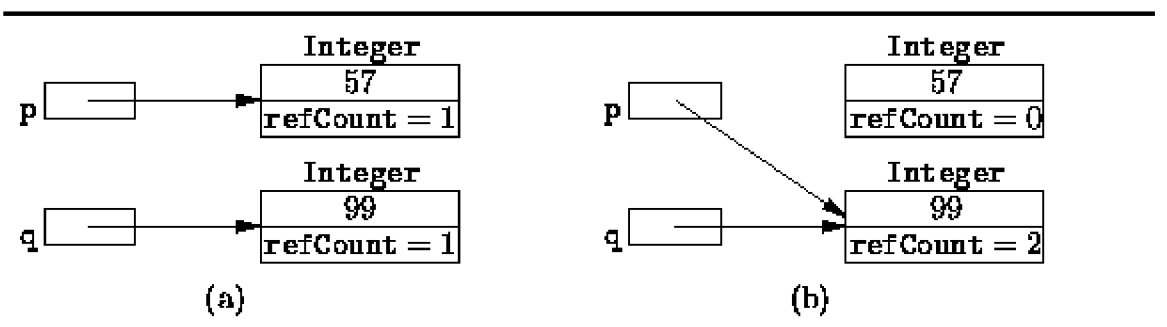


Figure: Reference counts before and after the assignment p = q.

The costs of using reference counts are twofold: First, every object requires the special reference count field. Typically, this means an extra word of storage must be allocated in each object. Second, every time one reference is assigned to another, the reference counts

must be adjusted as above. This increases significantly the time taken by assignment statements.

The advantage of using reference counts is that garbage is easily identified. When it becomes necessary to reclaim the storage from unused objects, the garbage collector needs only to examine the reference count fields of all the objects that have been created by the program. If the reference count is zero, the object is garbage.

It is not necessary to wait until there is insufficient memory before initiating the garbage collection process. We can reclaim memory used by an object immediately when its reference goes to zero. Consider what happens if we implement the Java assignment p = q in the Java virtual machine as follows:

```
if (p != q)
{
   if (p != null)
         if (--p.refCount == 0)
            heap.release (p);
   p = q;
   if (p != null)
         ++p.refCount;
}
```

Notice that the release method is invoked immediately when the reference count of an object goes to zero, i.e., when it becomes garbage. In this way, garbage may be collected incrementally as it is created.

**TEXT BOOKS:**


1. Compilers, Principles Techniques and Tools- Alfred V Aho, Monical S Lam, Ravi Sethi, Jeffrey D. Ullman,2nd ed, Pearson,2007.
2. Principles of compiler design, V. Raghavan, 2nd ed, TMH, 2011.
3. Principles of compiler design, 2nd ed, Nandini Prasad, Elsevier