

## PRINCIPAL SOURCES OF OPTIMIZATION

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. Function preserving transformations examples:

- Common sub expression elimination
- Copy propagation
- Dead-code elimination
- Constant folding

### Common Sub expressions elimination:

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 recomputing the expression if we can use the previously computed value.

For example:

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5
```

The above code can be optimized using the common sub-expression elimination as

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

The common sub expression  $t4: = 4*i$  is eliminated as its computation is already in  $t1$  and the value of  $i$  is not been changed from definition to use.

### Copy Propagation:

Assignments of the form  $f: =g$  called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use  $g$  for  $f$ , whenever possible after the copy statement  $f: = g$ . Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate  $x$ .

Example:

```
x=Pi;
.....
A=x*r*r;
```

The optimization using copy propagation can be done as follows:  $A=Pi*r*r$ ; Here the variable  $x$  is eliminated

### **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.

Example:

```
i=0;
if(i=1)
{
a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

### **Constant folding:**

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,

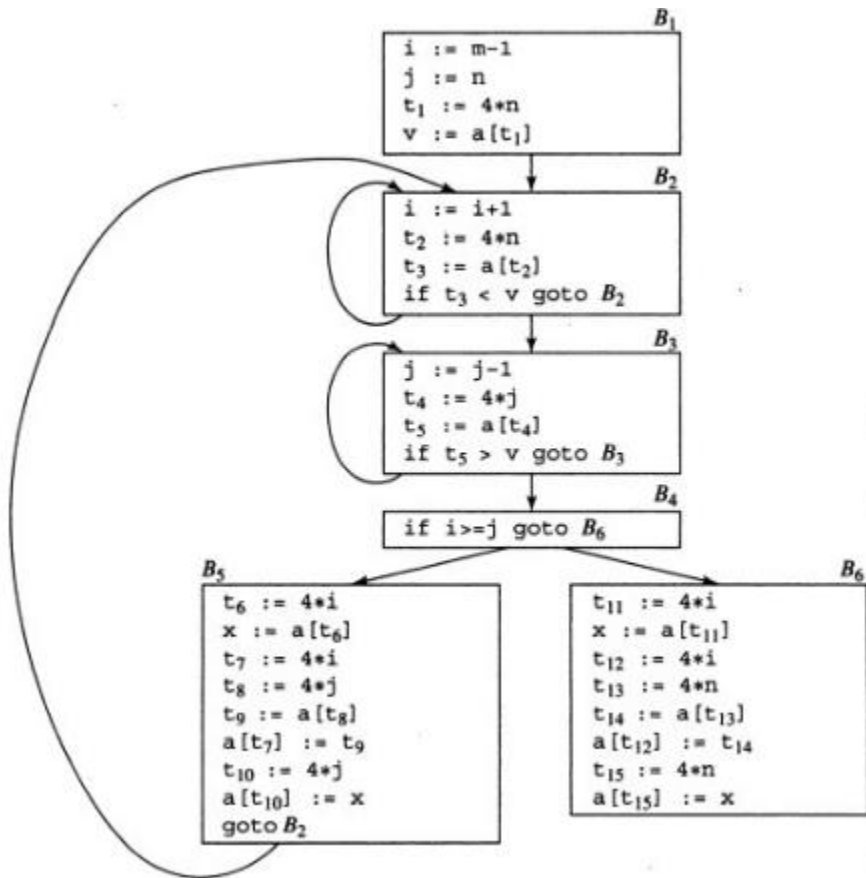
$a=3.14157/2$  can be replaced by  
 $a=1.570$  thereby eliminating a division operation.

### **Loop Optimizations:**

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, 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 replace variables from inner loop.
- Reduction in strength, which replaces and 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:**

Loops are usually processed inside out. For example, consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4\*j is assigned to t4. 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 B3 in Fig.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4. 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 B2- B5 is considered.

**Reduction in Strength:**

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example,  $x^2$  is invariably cheaper to implement as  $x*x$  than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift.