Module V

Code Optimization:

Principal sources of optimization, Machine dependent and machine independent optimizations, Local and global optimizations.

Code generation:

Issues in the design of a code generator.

The target machine, A simple code generator

Optimization

- A program transforming technique
- Tries to improve the code by making it consume smaller space and higher speed

Criteria for code improving transformations

- Transformations provided by an optimizing compiler should have several properties

1. Preserve the meaning of a program

- Optimization must not change the output produced by a program for a given input or cause an error , such as division by zero , that was not present in the original version of the source program

2. Speed up the programs by a measurable amount
- 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
3. Must be worth the effort – ie the process itself should be fast and should not delay the compiling process

Optimization

- Two types
- Machine independent
- Machine dependent

Machine-independent 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
- Eg

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

- This code involves repeated assignment of the identifier item, which can be changed as

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

- Saves the CPU cycles
- Can be run on any processor

Machine-dependent optimization

- 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

Basic Blocks

- A sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end

```
t1 := a * a
t2 := a * b
t3 := 2 * t2
t4 := t1 + t3
t5 := b * b
t6 := t4 + t5
```

Terms used

- A three-address statement x:=y + z is said to ***define*** x and to ***use*** or ***reference*** y and z
- A name in a basic block is said to be ***live*** at a given point if its value is used after that point in the program, perhaps in another basic block

- The following algorithm can be used to partition a sequence of three-address statements into basic blocks

Algorithm 9.1. Partition into basic blocks.

Input. A sequence of three-address statements.

Output. A list of basic blocks with each three-address statement in exactly one block.

Method.

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are the following.

   i) The first statement is a leader.
   ii) Any statement that is the target of a conditional or unconditional goto is a leader.
   iii) Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program. □

Consider the fragment of the source code shown in fig 9.7. It computes the dot product of two vectors a and b of length 20. A list of three address statements performing this computation on our target machine is shown in fig 9.8

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i + 1
    end
    while i <= 20
end
```

Fig. 9.7. Program to compute dot product.

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]          /* compute a[i] */
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]          /* compute b[i] */
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
```

Fig. 9.8. Three-address code computing dot product.

- Apply the algorithm 9.1 to the three-address code in fig 9.8 to determine its basic blocks
- Statement (1) is a leader by rule (i)
- Statement (3) is a leader by rule (ii)
- By rule (iii),the statement following (12) is a leader
- So , statements (1) and (2) form a basic block
- The remainder of the program beginning with statement (3) forms a second basic block

```
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.

```
(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
```

Basic blocks and flow graphs

- Flow graphs
  - A graph representation of three-address statements
  - Useful for understanding code-generation algorithms
  - Nodes represent computations
  - Edges represents the flow of control

Flow Graphs

- Flow-of-control information can be added to the set of basic blocks making up a program by constructing a directed graph called a flow graph
- The nodes of the flow graph are the basic blocks
- One node is distinguished as initial
- It is the block whose leader is the first statement
- There is a directed edge from block B1 to block B2, if B2 can immediately follow B1 in some execution sequence; ie if

1. There is a conditional or unconditional jump from the last statement of B1 to the first statement of B2 or
2. B2 immediately follows B1 in the order of the program and B1 does not end in an unconditional jump

- We say that B1 is a predecessor of B2 and B2 is a successor of B1

- The flow graph of the program of fig 9.7 is shown in Fig 9.9. B1 is the initial node
- The jump statement to (3) is replaced by an equivalent jump to the beginning of block B2

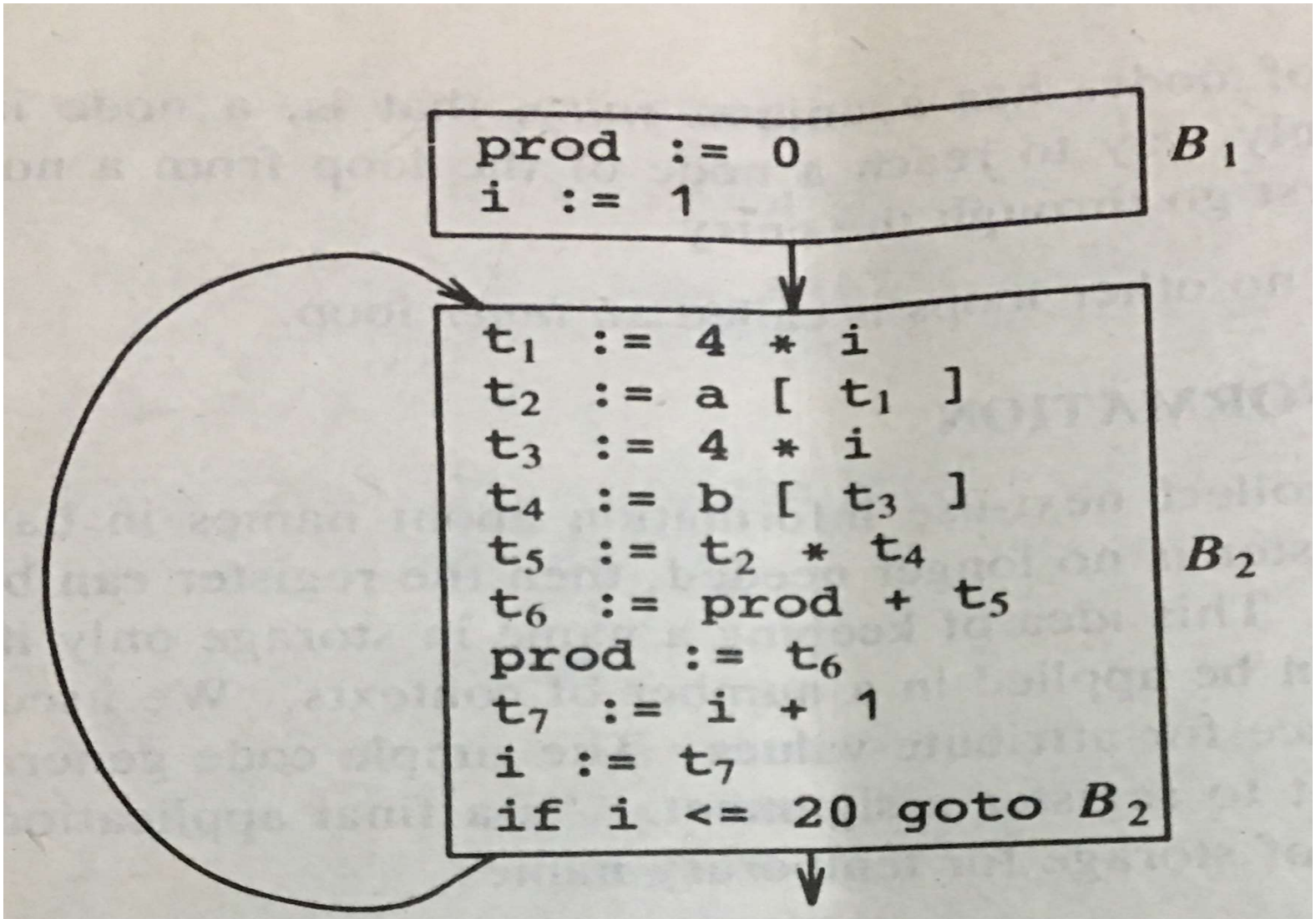


Fig. 9.9. Flow graph for program.

Transformations on Basic Blocks

- A basic block computes a set of expressions
- These expressions are the values of the names live on exit from the block
- Two basic blocks are said to be *equivalent* if they compute the same set of expressions
- A number of transformations can be applied to a basic block without changing the set of expressions computed by the block
- Most of these are useful for improving the quality of code that will be ultimately generated from a basic block

The principal sources of optimization

- Local transformations
  - If it can be performed by looking only at the statements in a basic block
- Global transformations

- Many transformations can be performed both at the local and global levels
- Local transformations are usually performed first

- Two important classes of local transformations can be applied to basic blocks
  - Structure preserving transformations
  - Algebraic transformations

Structure preserving transformations

- The primary structure preserving transformations on basic blocks are :

1. Common sub-expression elimination
2. Dead-code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent statements

1. Common subexpression elimination

- Consider the basic block

```
a := b + c
b := a –d
c := b + c                    9.2
d := a - d
```

- The second and fourth statements compute the same expression, namely, b+c-d, hence the basic block may be transformed into the equivalent block

```
a := b + c
b := a –d
c := b + c                    9.3
d := b
```

- Although the first and third statement appear to have the same expression on the right, the second statement redefines b
- Therefore the value of b in the third statement is different from the value of b in the first, and the first and the third statements do not compute the same expression

2. Dead-code elimination

- Suppose x is dead, that is, never subsequently used, at the point where the statement x:=y+z appears in a basic block
- Then this statement may be safely removed without changing the value of the basic block

3. Renaming temporary variables

- Suppose t:=b+c is a statement where t is a temporary variable
- If we change this statement to u:=b+c, where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed
- We can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary
- We call such a basic block as a ***normal-form*** block

4. Interchange of statements

- Suppose we have a block with the two adjacent statements

  t1:=b+c

  t2:=x+y

- We can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2

Algebraic Transformations

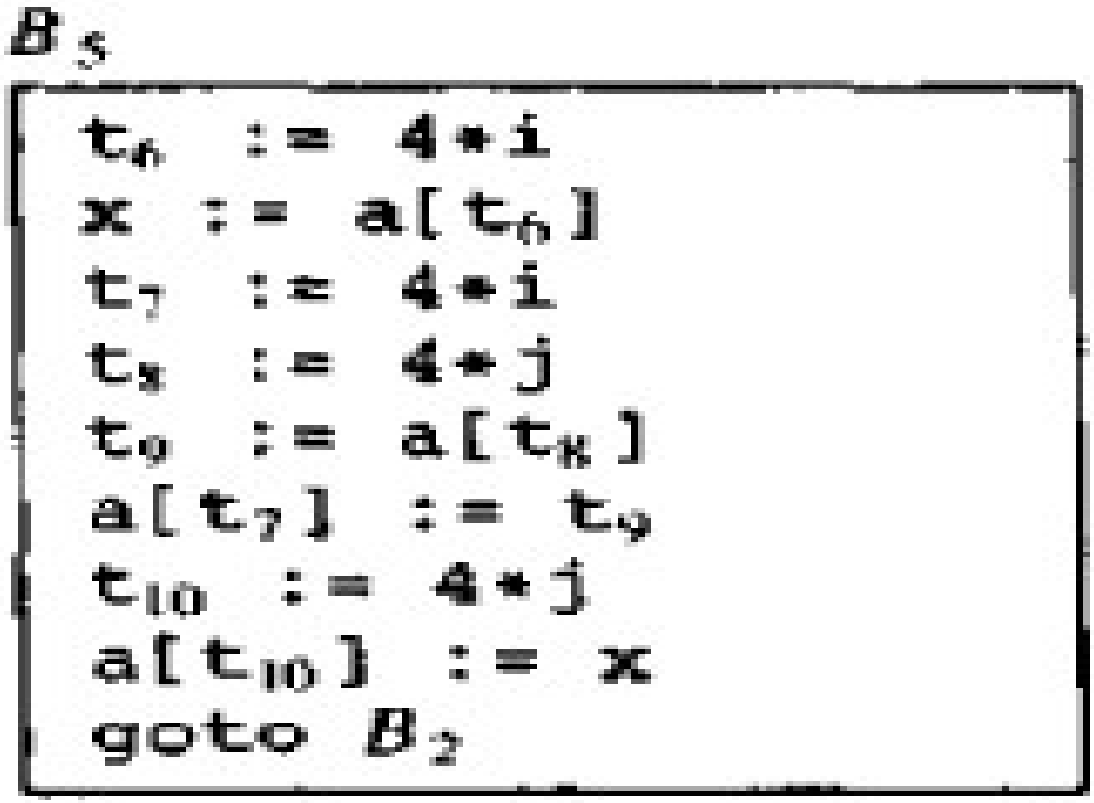
- There are many algebraic transformations that can be used to change the set of expressions computed by a basic block into an algebraically equivalent set
- The useful ones are those that simplify expressions or replace expensive operations by cheaper ones
- x = x + 0 or x = x * 1 can be eliminated from a basic block without changing the set of expressions it computes

- Expensive operations like x = y ** 2 which requires a function call can be replaced by an equivalent statement x = y * y

Function preserving transformations

- Ways in which the compiler can improve a program without changing the function it computes
  - Common sub-expression elimination
  - Copy propagation
  - Dead-code elimination and
  - Constant folding
- 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 eg:, block B5 recalculates 4*i and 4*j Fig 10.6



(a) Before

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 recomputing the expression if we use the previously computed value
- For eg, the assignments to t7 and t10 have the common subexpressions 4*i and 4*j respectively in fig 10.6 a
- They have been eliminated in fig 10.6b, by using t6 instead of t7 and t8 instead of t10

B_5

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

(b) After

Fig 10.6

- Fig 10.7 shows the result of eliminating both global and local common subexpressions from blocks B5 and B6 in the flow graph of fig 10.5

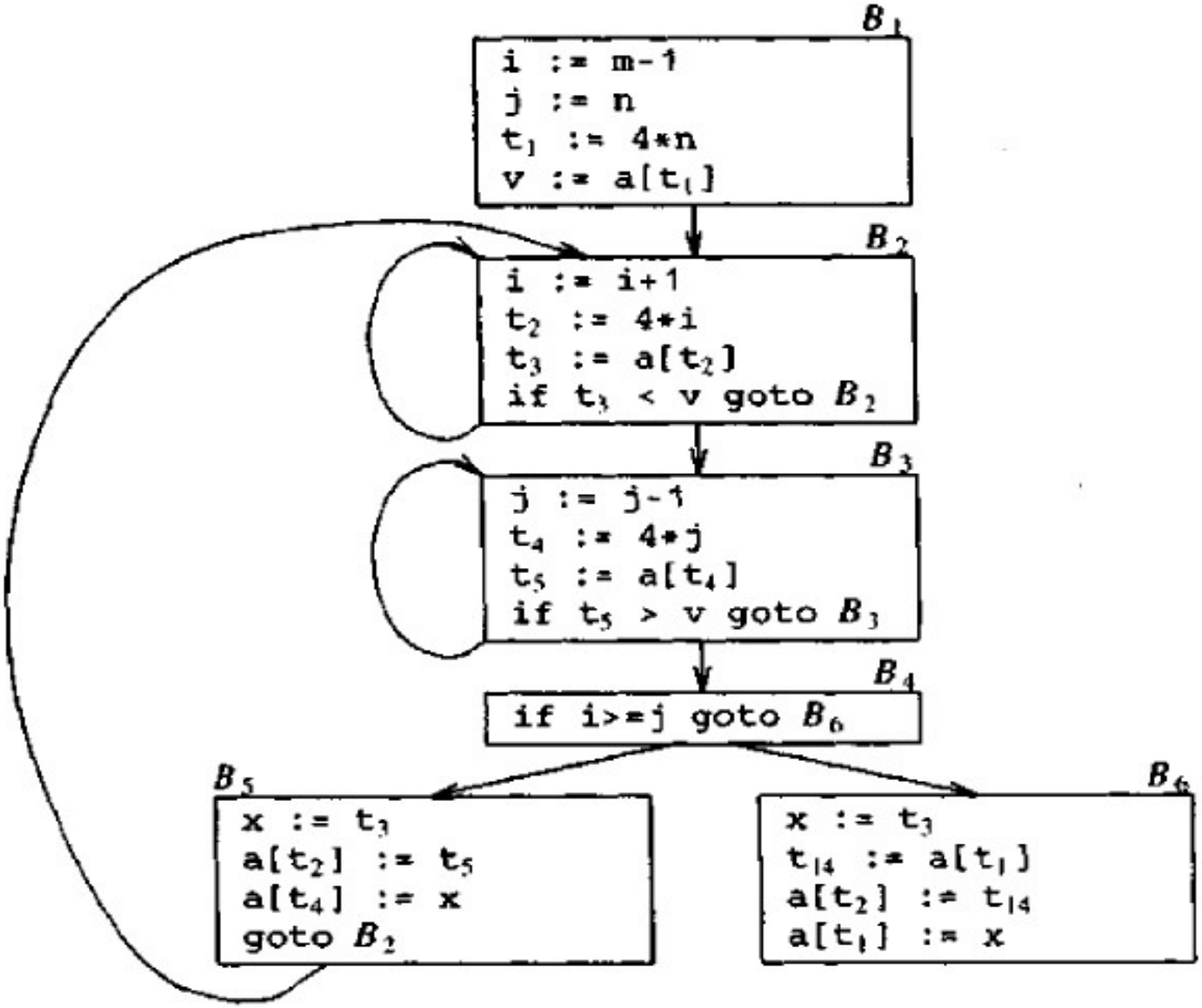


Fig. 10.7. B_5 and B_6 after common subexpression elimination.

Copy propagation

- Block B5 can be further improved by eliminating x using two new transformations
- Assignments of the form f=g called copy statements or copies for short

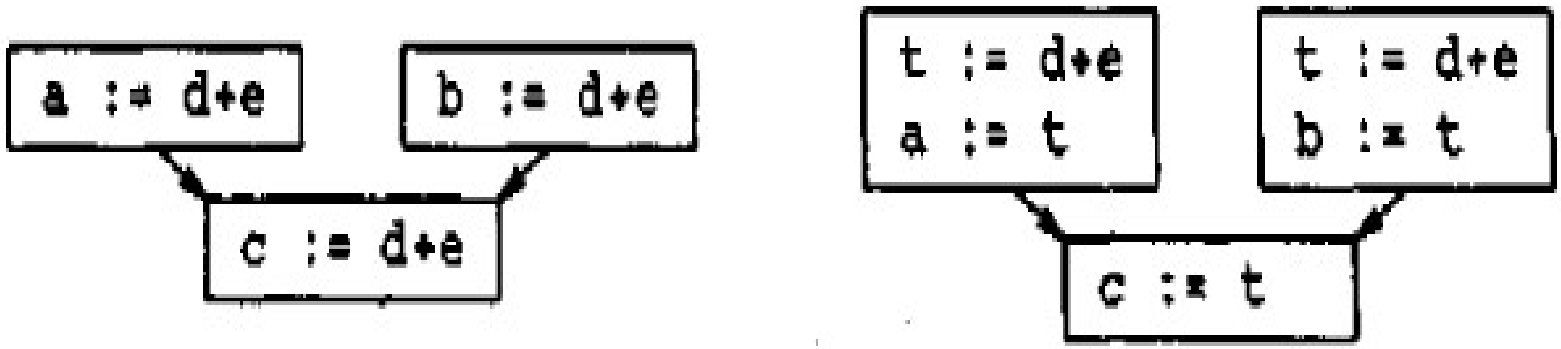


Fig. 10.8. Copies introduced during common subexpression elimination.

- The idea behind copy propagation transformation is to use g for f, wherever possible after the copy statement f=g
- For eg, the assignment x=t3 in block B5 of fig 10.7 is a copy
- Copy propagation applied to B5 yields:

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

10.1

- This may not appear as an improvement, but 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
- While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations
- For eg: If we have a test

  if (debug) print….
- Assume that there is a statement debug = false
- 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
- Generally, deducing at compile time that the value of an expression is 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 eg, the 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
```

Loop optimizations

- Important place for optimization – loops
- Especially 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
- 3 techniques
  - Code motion : moves code outside a loop
  - Induction variable elimination
  - Reduction in strength : replaces expensive operations by cheaper ones

Code motion

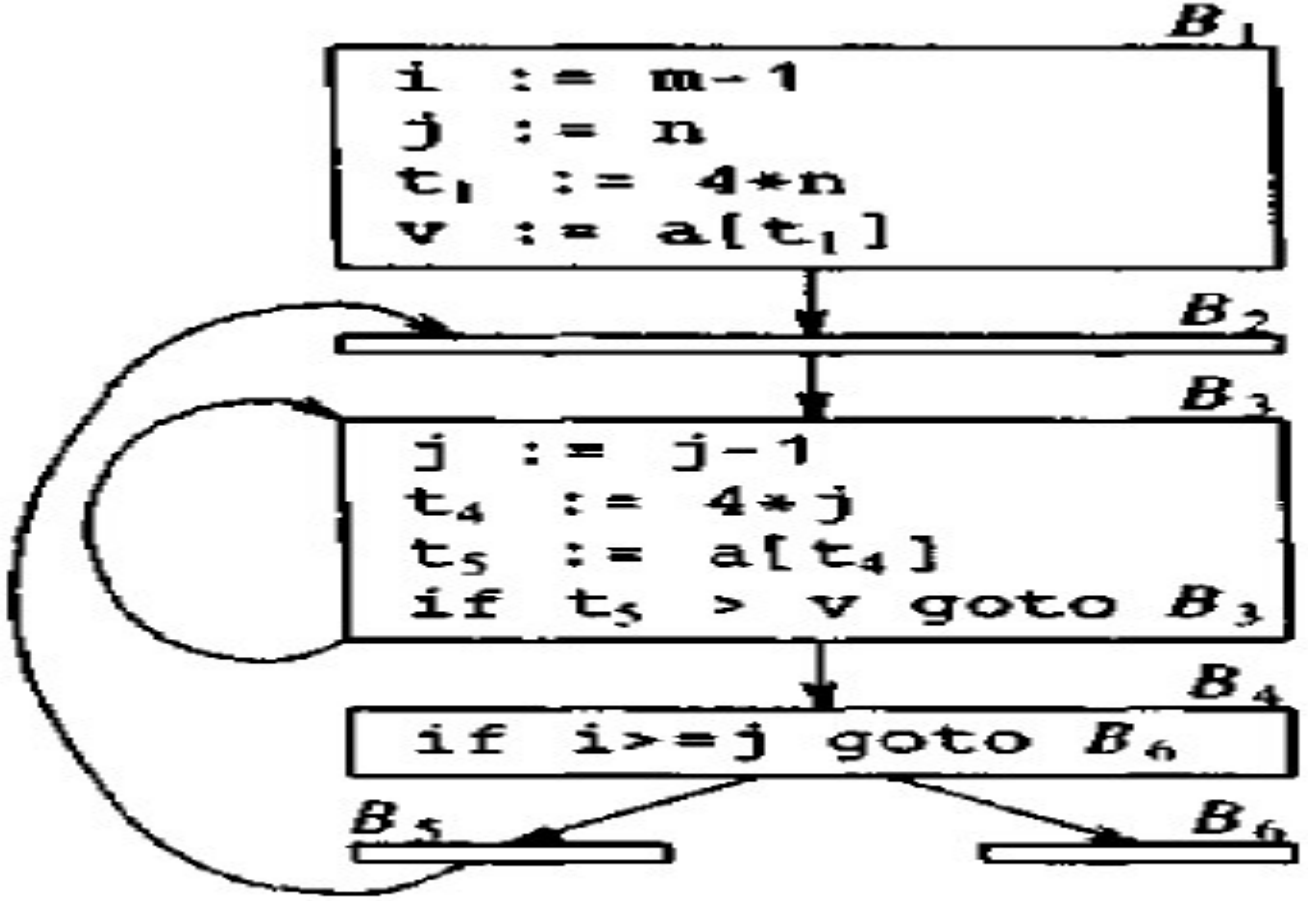
- An important modification
- Decreases the amount of code in a loop
- Takes an expression that yields the same result independent of the number of times a loop is executed (loop invariant computation) and places the expression before the loop
- For eg : consider the following while statement

  while (i <= limit-2)

- Here the evaluation of limit-2 is a loop-invariant computation
- Code motion will result in the equivalent of

```
t = limit-2
while (i<=t)
```

Induction variables and Reduction in strength

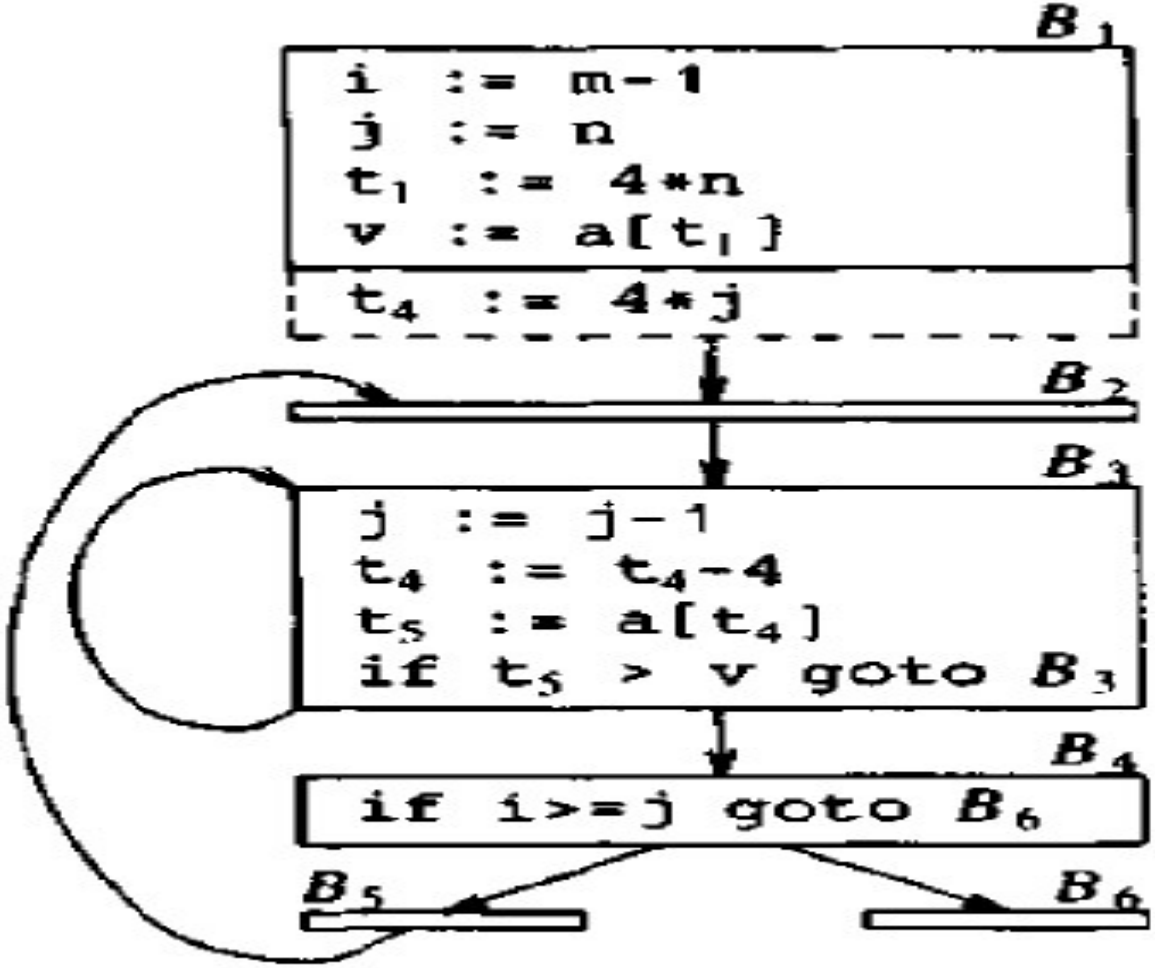


(a) Before

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

- Here 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 10.9a , we cannot get rid of either j or t4 completely;t4 is used in B3 and j in B4

- As the relationship t4=4*j surely holds after such an assignment to t4 in fig 10.9a,and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j=j-1 the relationship t4=4*j-4 must hold
- We may then replace the assignment t4=4*j by t4=t4-4
- The only problem here is that t4 does not have a value when we enter block B3 for the first time
- Since we must maintain the relationship t4=4*j on the entry to the block B3, we place an initialization of t4 at the end of the block where j itself is initialized


B1
i := m-1
j := n
t1 := 4*n
v := a[t1]
t4 := 4*j
B2
B3
j := j-1
t4 := t4-4
t5 := a[t4]
if t5 > v goto B3
B4
if i>=j goto B6
B5
B6

(b) After

- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction

- After reduction in strength is applied to the inner loops around B2 and B3, the only use of I and j is to determine the outcome of the test in block B4
- Since the values of i and t2 satisfy the relationship t2=4*i and those of j and t4 satisfy t4=4*j, test t2>=t4 is equivalent to i>=j

- Once this replacement is made, i in Block B2 and j in Block B3 becomes dead variables and the assignments to them in these blocks become dead code that can be eliminated, resulting in the flow graph in fig 10.10

- In fig 10.10, the number of instructions in blocks B2 and B3 has been reduced from 4 to 3 from the original flow graph in fig 10.5,in B5 it has been reduced from 9 to 3 and B6 from 8 to 3

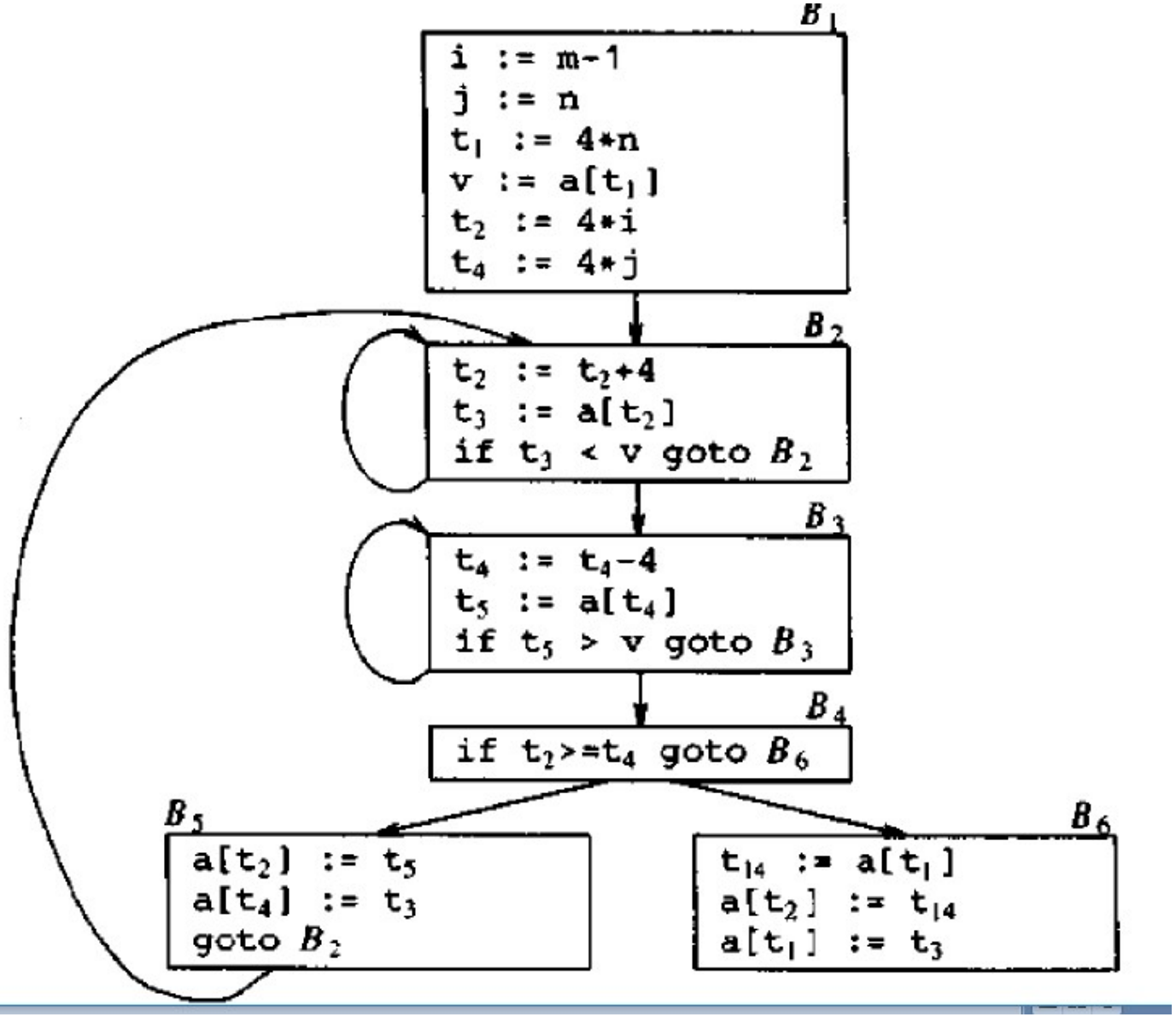


Fig. 10.10. Flow graph after induction-variable elimination.