problem. ...ization' looks

5.1 The DAG Representation of Basic Blocks

ajority of local optimization techniques begins with transformation of basic blocks to directed acyclic graph (DAG). Let us see how to construct DAG from basic blocks.

1. For each & every initial values of the variables appearing in the basic block, there exists a node in the DAG.
2. If 'S' is a statement with in the block then there exists a node 'N' associated with each 'S'. The children of 'N' are those nodes corresponding to statements that are the last definations,, prior to 'S', of the operands used by 'S'.
3. Label node 'N' by the operator applied at 'S' and also attached to 'N'. It is list of variables for which it is the last definition within the block.
4. DAG consists of 'output Nodes'. These few nodes which are 'output nodes' are those nodes whose variables are 'live on exit' from the block.[*'Live on exit'* means that their values can be used later in another block of the flow graph]. Global flow analysis calculates these *'live variables'*.

he following code improving transformations are performed on the code present in lock. This is performed by using DAG representation of a basic block.

1. Eliminate *'local s'* [are those instructions which compute a value that have already been computed].
2. Eliminate *'dead code'* [Instruction that compute a value which is never used are 'Dead Codes'].
3. Reorder those statements that do not depend on one another. This reordering help to reduce time a temporary value needs to be preserved in a register.
4. Algebraic laws can also be used to reorder operands of three-address instructions. This helps to simplify the computation.

7.5.2. Finding Local Common Subexpressions

Two operations are *'common'* if they produce the same result. In such a case, it is better to compute the result once and reference it the second time rather than re-evaluate it. Common subexpressions can be detected by noticing, as a new node 'M' is about to be added, whether there is an existing node 'N' with the same children, in the same order, and with the same operator. In this case, N computes the same value as M and may be used in its place.

Let following instructions belong to a block:

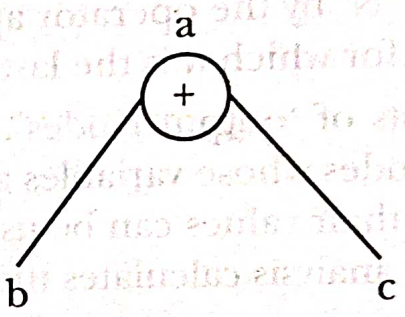
$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = a - d \quad (7.1)$$

The DAG for expressions is as shown below:

Step 1

Consider a = b + c

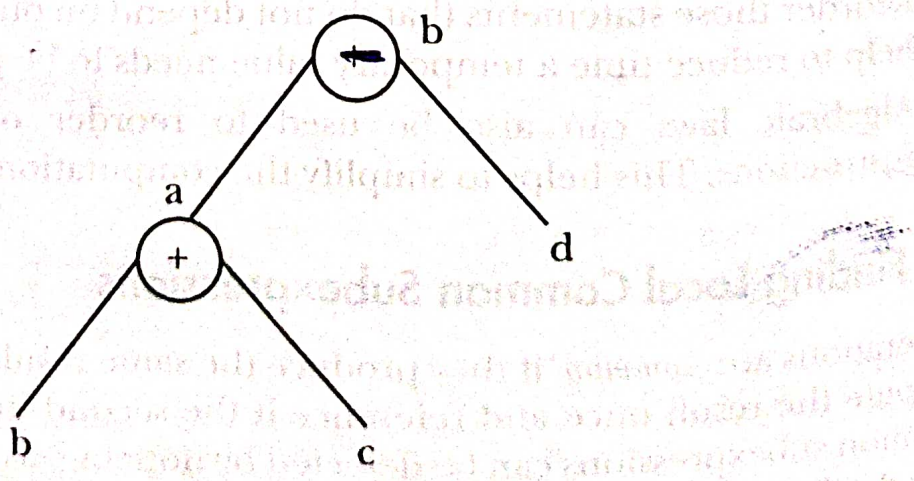
Operator



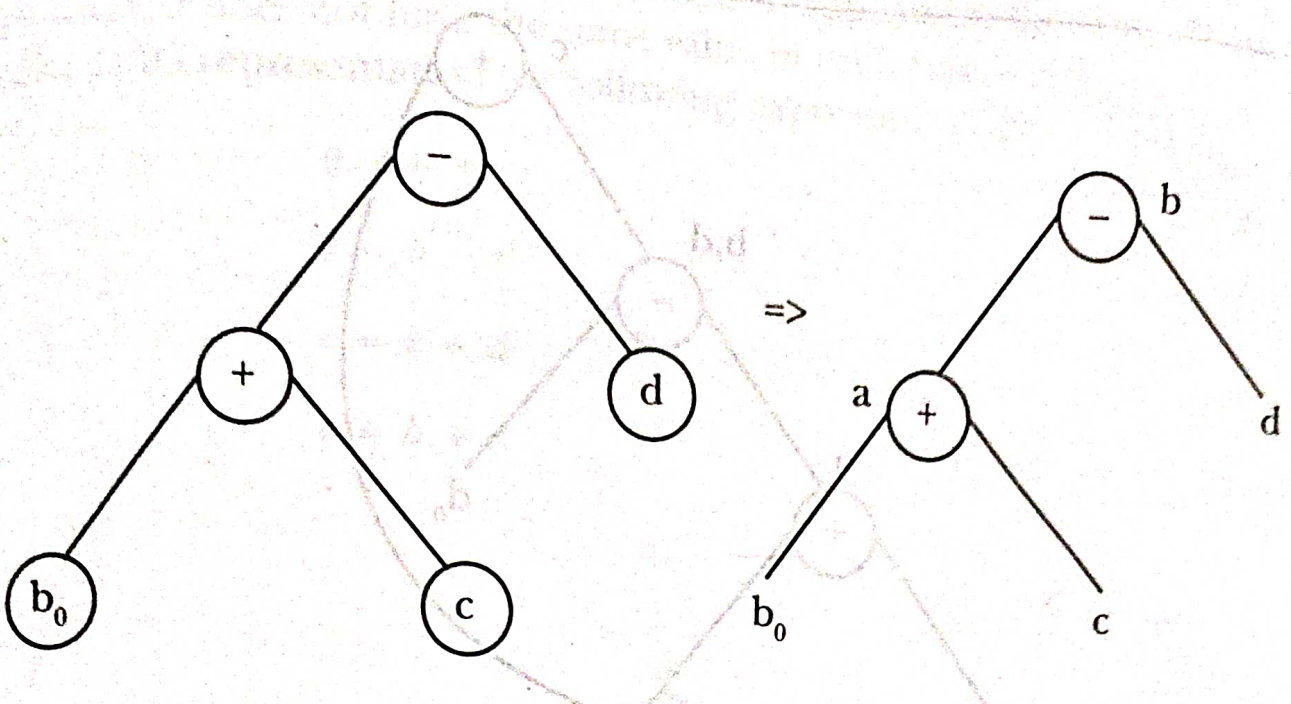
Name the node as 'a' (as a = b + c)

Step 2

Consider next instruction b = a − d

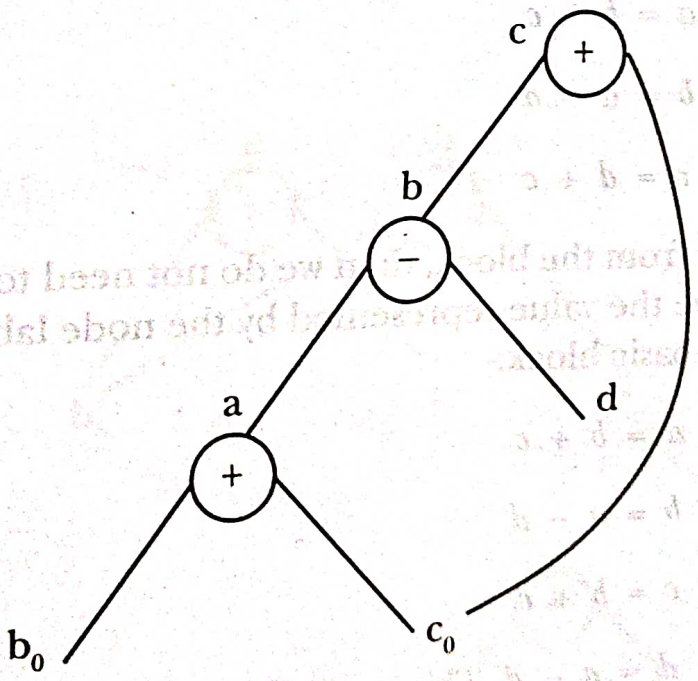


Here 'b' in step1 is named as b_0 because we have b = a − d [we may become confused between a = b + c and b = a − d, as 'b' exist in both instructions a = b + c and b = a − d]



Step 3

In the similar way write DAG for the other two instructions.



Here consider the recent 'b' [and not b_0]. Since c = b + c. 'c' in step1 is named as c_0 as shown in step3.

Step 4

d = a − d

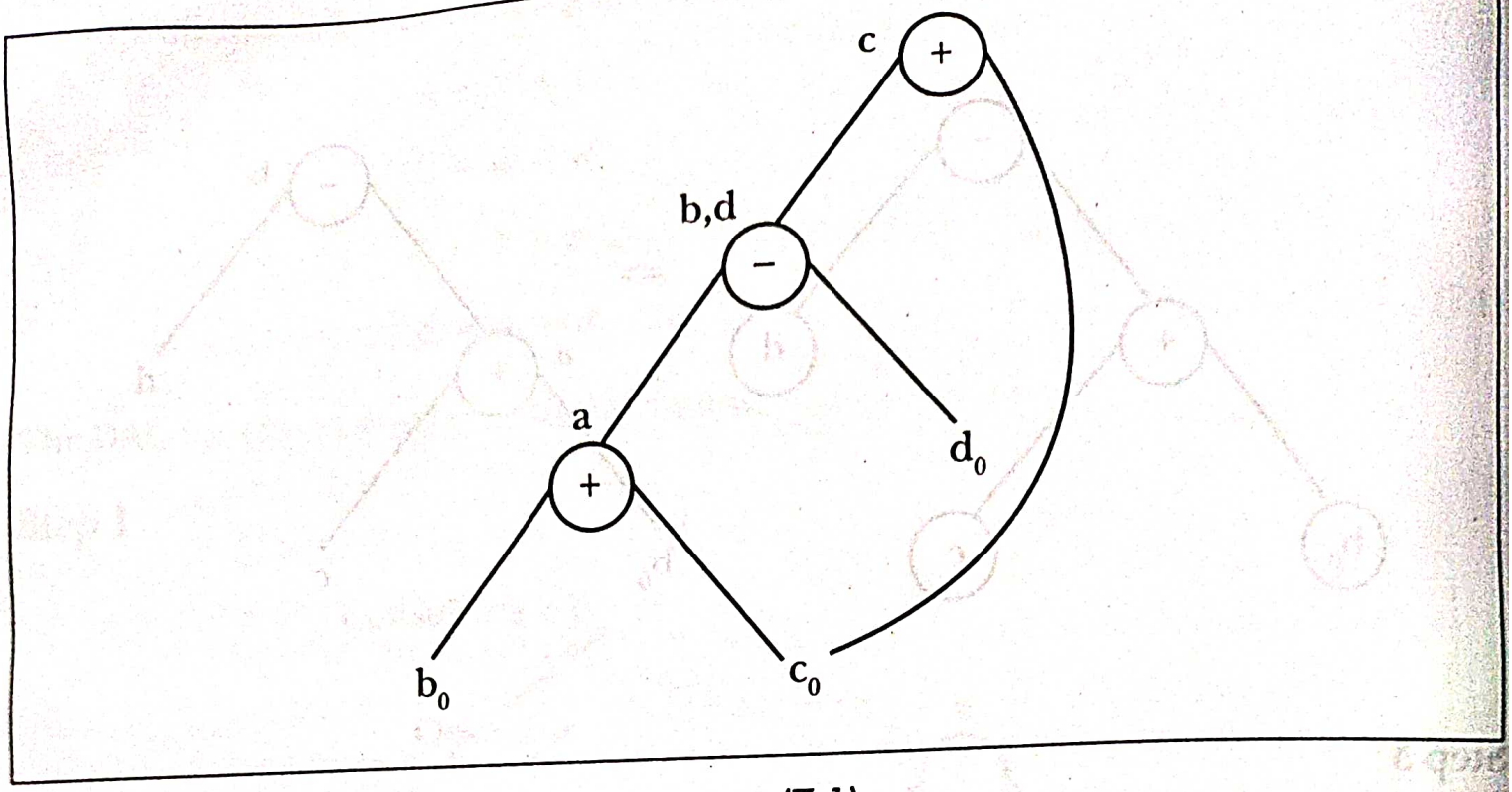


FIGURE 7.8: DAG representation for expression (7.1)

Expression (7.1) can be reduced to:-

$$a = b + c$$

$$b = a - d$$

$$c = d + c \qquad (7.2)$$

[If 'b' is not live on exit from the block, then we do not need to compute that variable and 'd' is used to receive the value represented by the node labeled (-)].
Consider the following basic block:-

$$a = b + c$$

$$b = a - d$$

$$c = b + c$$

$$d = a - d.$$ *This is expression (7.1).*

Here the second and fourth statement calculates the same value, so this block can be written as:

$$a = b + c$$

$$b = a - d$$

$$c = b + c$$

$$d = b$$

Note that, even though the same expression, b+c, appears on the right of both the fi
and third lines, it does not have the same value in each case.
Let us write DAG representation for following expression (7.2):-

$$a = b + c$$

$$b = b - d$$

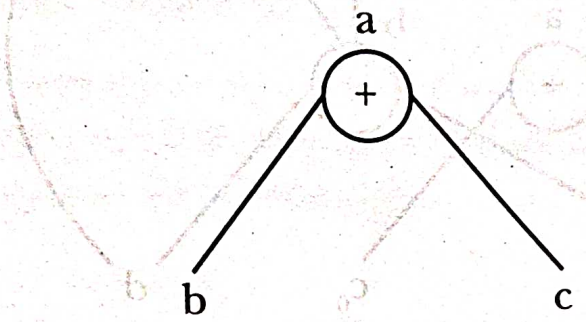
$$c = c + d$$

$$e = b + c$$

(7

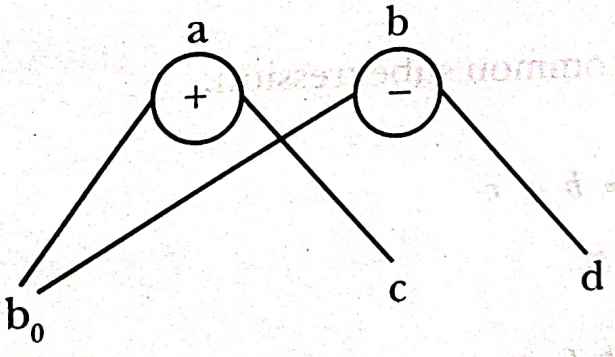
Step 1

a = b + c



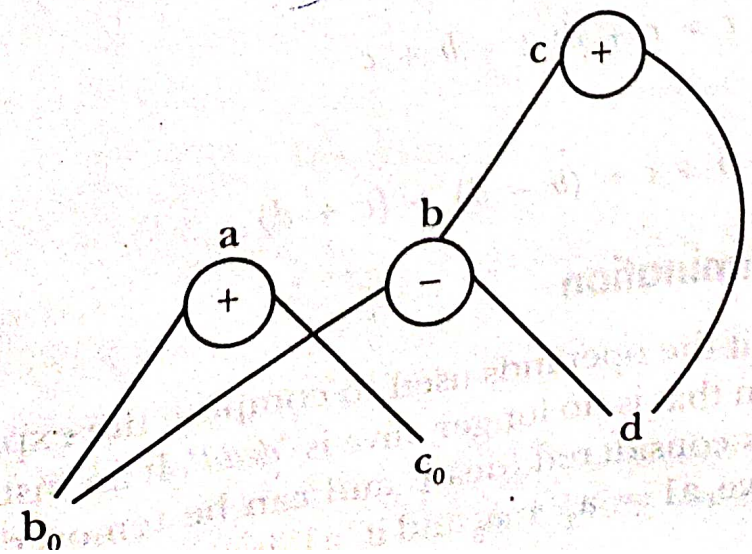
Step 2

b = b – d



Step 3

c = c + d



Step 4

e = b + c

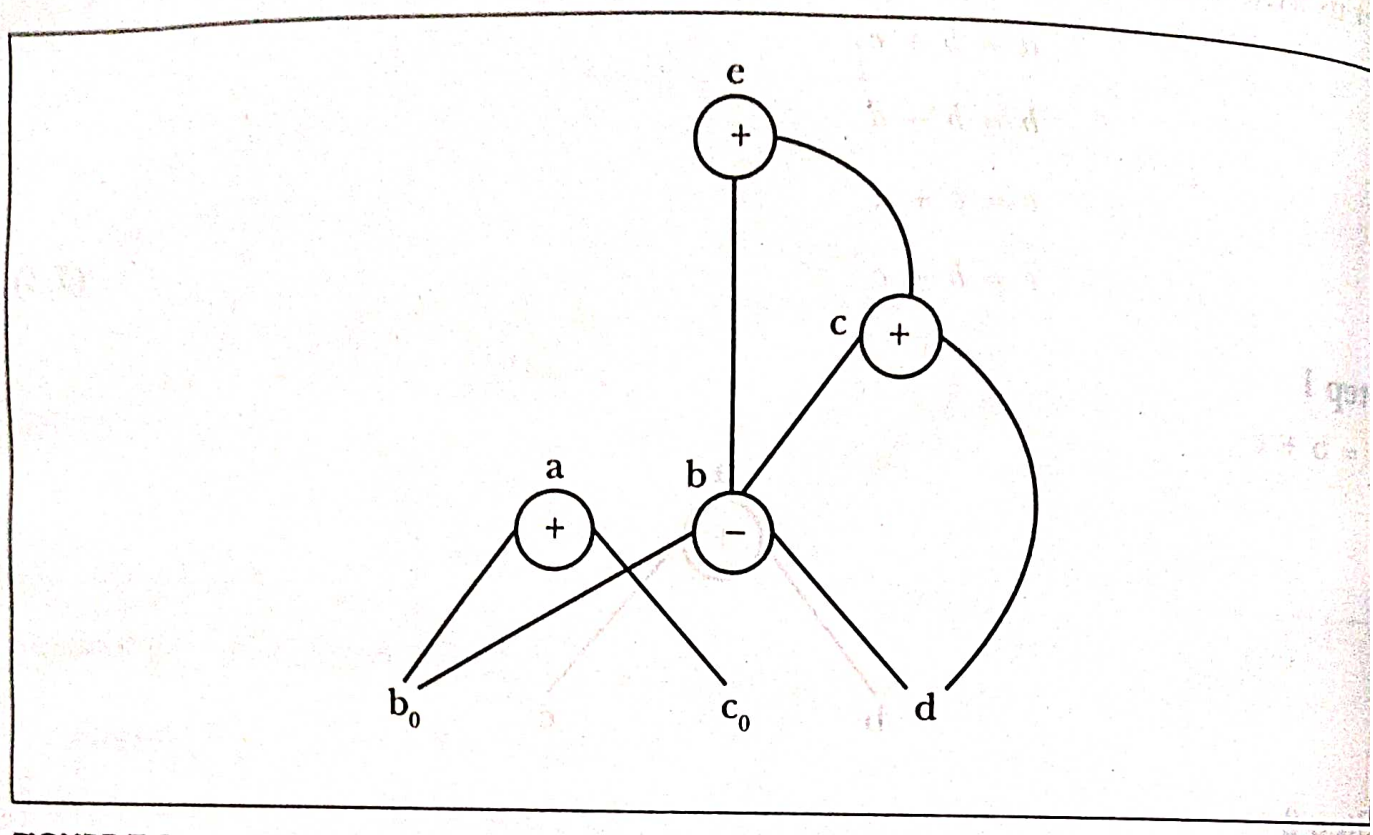


FIGURE 7.9: DAG representation for expression (7.3)

Fig.7.9 does not have any common subexression.
Here,

$$e = b + c$$

$$\Downarrow$$

$$b + c$$

Also

$$c = c + d; a = b + c$$

Therefore, we have

$$b + c = (b - d) + (c + d)$$

7.5.3 Dead Code Elimination

An expression is *'alive'* if the operands used to compute the expression have not been changed. An expression that is no longer alive is *'dead'*. If an instruction result is never used, the instruction is considered 'dead' and can be removed from the instruction stream. For eg. if we have, a1 = a_2 + a_3 and if 'a1' is never used again, we can eliminate

this instruction altogether. Little care has to be taken here. If 'a1' holds the result of some function: a1 = mcall _ over; where 'mcall_over' is some function call.

Even if 'a1' is never used again, we cannot eliminate the instruction because we cannot be sure that called function has no side effects.

The original source program can also have dead code but is more likely to have resulted from some of the optimization techniques run previously. In the fig. 7.9, 'a' and 'b' are live but 'c' and 'e' are not. Here root labeled 'e' can be immediately removed.

Suppose a = b op d and if 'a' is never used subsequently, then 'a' is dead. Hence this expression can be removed.

7.5.4 Use Of Algebraic Identities

The rules of arithmetic can come in handy when looking for redundant calculations to eliminate. The following are few examples that allow us to replace an expression on the left with a simpler equivalent on the right:

$0 + x = x$

$1 * x = x$

$0 / x = x$

b && true = b etc.

Example

Expensive	Cheaper
X - 0	X
X^3	X × X× X
X / 3	X × 0.33
X = Y× 64	X = Y << 6

Replacing a more expensive operation by a cheaper one is known is *'reduction in strength'*, which is part of algebraic optimization.

Another type of optimization is *'constant folding'*. Computing expressions with known values at compile time is 'constant folding.

"Constant folding and propagation" is replacing expressions consisting of constants with their final value at compile time, rather than doing the calculation at runtime. This is used in most modern languages.

For eg.: -Replacing 2 * 5 with their final value 10 at compile time.

7.5.5 Representation of Array References

Consider the following basic blocks:-

$$a = b[i]$$

$$b[j] = d$$

$$e = b[i] \qquad (7.4)$$

Here in expression (7.4), optimization can be carried out by replacing the third instruction e = b [i] by the simpler e = a; here 'j' could be = 'i', then the second statement may be changed to b [i]. This is not a legal change.

Therefore (7.3) can be optimized as:

$$a = b[i]$$

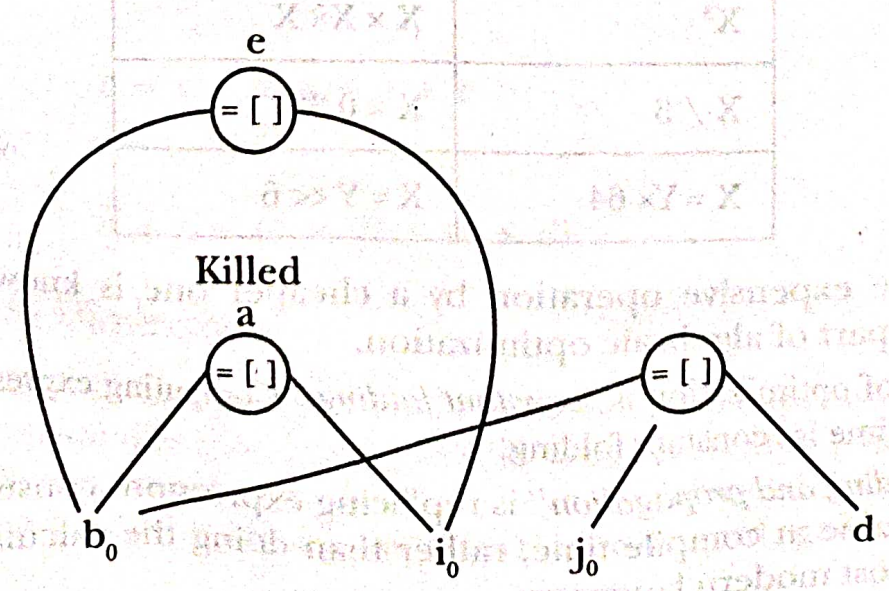
$$e = a$$

$$b[j] = d.$$

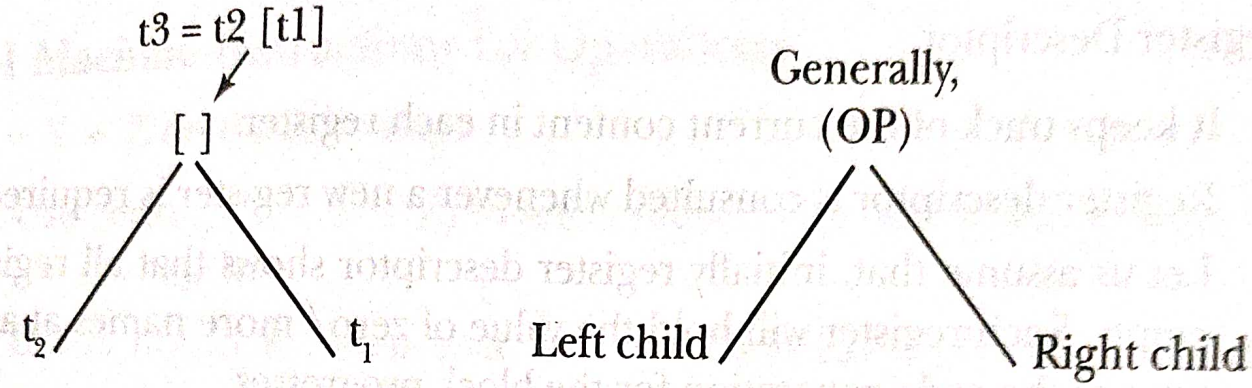
We have, a = b [i]. This can be represented by creating a node with operator [ ] and two children's representing the initial value of b_o & index 'i'. Variable 'a' becomes a label of this new node.

We also have, b [j] = d. This is represented by a new node with operator [ ] = and three children representing b_o, j and d. Here we do not have any variable labeling this node. The creation of this node 'kills' all currently constructed nodes whose value depends on b_o. The node which has been killed cannot receive any more labels i.e. it cannot become a common sub expression.

The DAG representation for expression (7.4) is:-



Note



8.2 PRINCIPAL SOURCES OF OPTIMIZATION

Optimization can either be *local optimization* or *global optimization*. Transformation applied to the same block is local optimization else it is global optimization. Usually local transformations are done first. The meaning (ie. Semantics) of the source program should not be changed while applying optimization.

The transformation can be performed at both the local level and global level. Usually local transformation is performed first.

Different ways in which a compiler can improve a program without changing the function (ie. Function-preserving transformations) are:

1. Elimination of common subexpression
2. Copy propogation
3. Dead-code elimination.
4. Constant folding

8.2.1 Elimination of Common Subexpression

Common subexpressions can either be eliminated locally or globally. Local common subexpressions can be identified in a basic block. Hence first step to eliminate local common subexpressions is to construct a basic block. Then the common sub expressions in a basics block can be deleted by constructing a directed acyclic graph (DAG). We have already studied in chapter 7 about DAG construction.

Let us consider the following basic block:

B_2

$t_2 = 4*i$
$b = a[t_2]$
$t_3 = 4*i$
$t_4 = 4*j$
$t_5 = a[t_4]$
$a[t_3] = t_6$
$t_7 = 4*j$
$a[t_7] = b$
goto B_4

FIGURE 8.1: Local common subexpression

The local common subexpression in B2 are:

$t_3 = 4*i$
$t_7 = 4*j$

Hence these local common subexpressions can be eliminated. The block obtained after eliminating local common subexpression is as shown below:

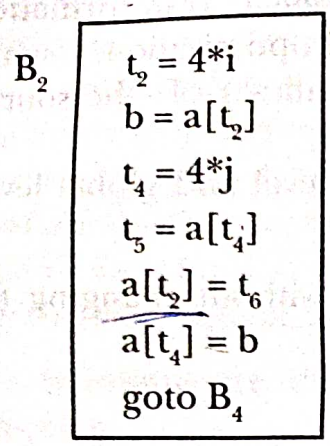


FIGURE 8.2: Elimination of local common subexpression from fig. 8.1

An expression E is said to be a "common subexpression" if E had been already computed and the values of variables in E have not changed since the previous computation. Re computing the expression E can be eliminated if preciously computed values are used as shown above.

8.2.2 Copy Propagation

Consider the assignment of the form f: = g. The statement f: = g is called as 'copy statement.Consider fig. 8.3

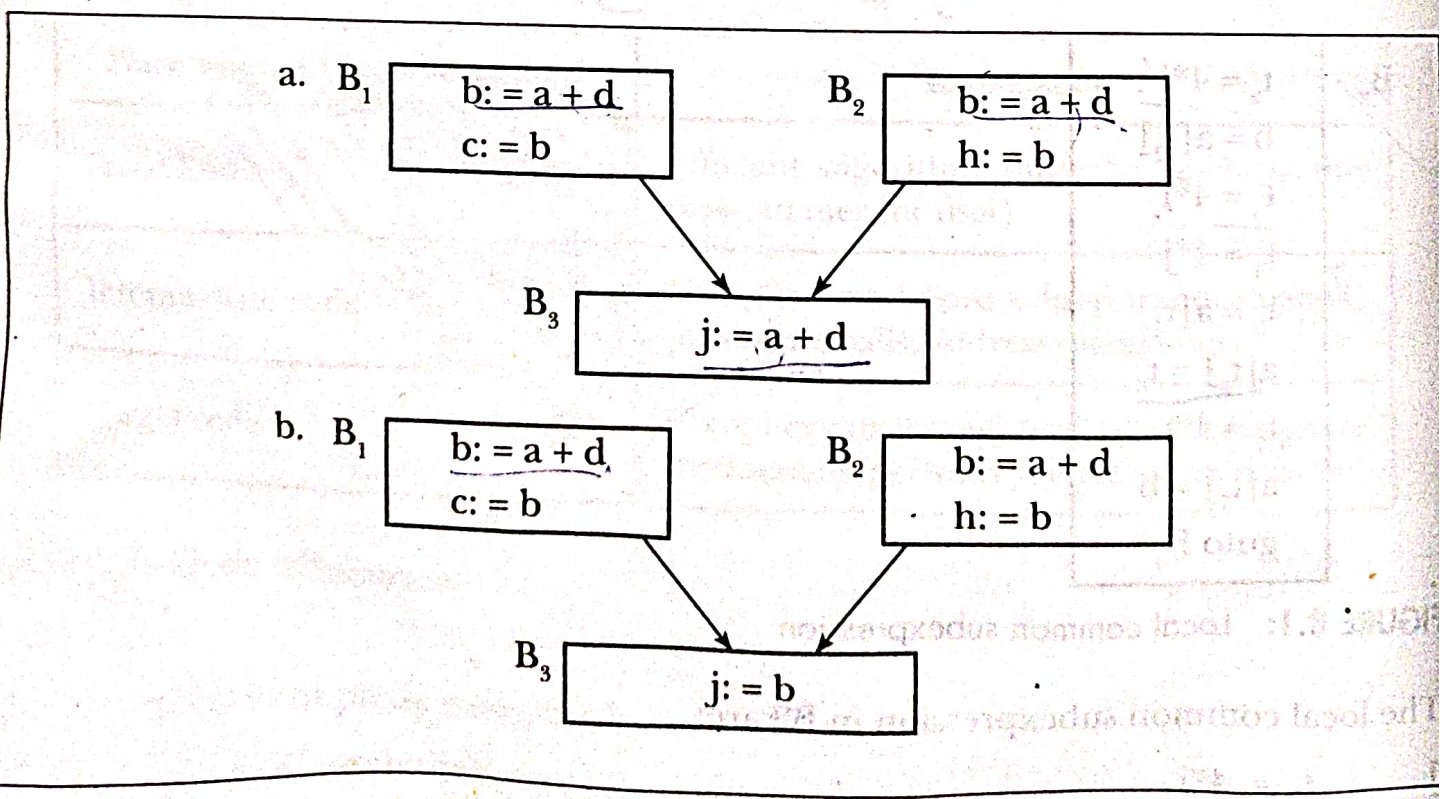


FIGURE 8.3: Copies for common subexpression elimination.

Here the common subexpression is j = a + d; when 'a + d' is eliminated fig. 8.3 uses = b. The control many reach b: = a + d either after the assignment to c (as in B1) or after the assignment to h (as in B_2).

Therefore, it is not right to replace b: = a + d by either b: = c or by b: = h. The concept behind copy-propagation transformation is to use 'g' for 'f'; wherever possible after using the copy statement f: = g.

8.2.3 Dead-Code Elimination

A variable is 'live' if the value contained into it is used subsequently. If its value is never used, the variable is said to be 'dead'.

Consider,

```
a : = t1
b [t2]: = t3
b [t4]: = t1
goto B3
```
(8.1)

Dead-code elimination from (8.1) transforms to (8.2):

```
b [t2]: = t3
b[t4]: = t1
goto B3
```
(8.2)

8.2.4 Constant Folding

In folding technique the computation of a constant is done at compile time instead of execution time and further the computed value of the constant is used.

Example

$$a = (5/2) * m$$

Here folding is done by performing the computation of (5/2) at compile time.

8.2.5 Loop Optimizations

Code optimization can be done in 'loops' of the program mainly the inner loops where Programs tend to spend the bulk of their time. If the number of instructions are less in inner loop then the running time of the program gets reduced by maximum extent LO can be done by the following 3 techniques:

Technique	Explanation
Code motion	Moves code outside a loop
Induction variable elimination	Eliminate inner loops induction variables
Strength reduction	Replaces expensive operation by a cheaper one

8.2.5.1 Code Motion

Code motion is a technique which is used to move the code outside the loop. If there exists any expression outside the loop whole result is unchanged even after executing the loop many timer, then such an expression should be placed just before the loop.

Code motion 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.

Example

```
While (i < = max - 3)
{
Sum = Sum + b [i];
}
```

This code can be optimized by removing the computation of max -3 outside the loop as shown below:

```
n = max – 3;
While (i < = n)
{
Sum = Sum + b [i];
}
```

8.2.5.2 Induction Variable Elimination

Induction variables of a loop are those names whose only assignment with in the loop are of the form L = L ± C, where 'C' is a constant or a name whose value does not change with in the loop. [It is either incremented or decremented by some constant].

B_1

```
i: = i + 1
t1: = 4 * j
t2: = a [t1]
if t2 < 20 goto B1
```

When i gets incremented by 2 then t_1 gets incremented by 4. Hence i and t_1 are induction variables. We can therefore replace the assignment t_1: = 4*j by t_1: = t_1 – 4.

8.2.5.3 Strength Reduction

Replacing an expensive operation by a cheaper one is called *reduction in strength.* For example strength of * is higher than strength of +. Hence the higher strength operators can be replaced by lower strength operators.

Example

```
for (i = 1; i< = 40; i ++)
{
---
Sum = i * 7;
---
}
```

Here we get values of 'Sum' as 7, 14, –, 100 The above code can be replaced by using strength reduction as shown below:

```
temp = 7;
for (i = 1; i < = 40; i ++)
{
---
Sum = temp;
temp = temp + 7;
-
}
```

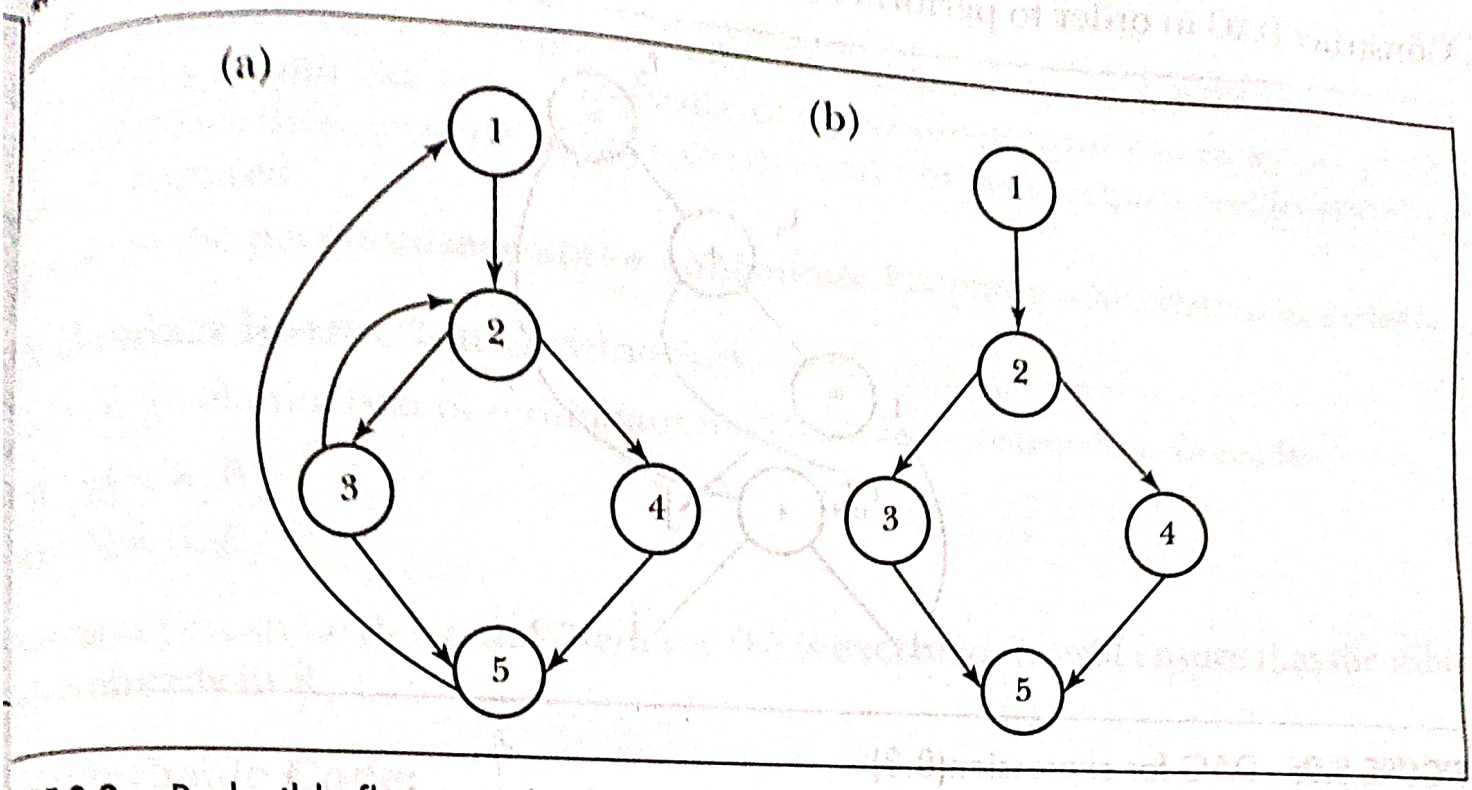
The meaning of the above code is same but the strength of operation is reduced

Example

*x**3 = x*x*x; x/4 = x*0.25.*

8.3 OPTIMIZATION OF BASIC BLOCKS

The structure – preserving transformations are: Common subexpression elimination and dead - code elimination.



URE 8.8: Reducible flow graph after removing the back edge

w graph which consists of forward edges and backward edges are reducible graphs. ese forward and backward edges which are partitioned in flow graph G are two joint groups which has following properties:

1. Forward edge form an acyclic graph such that every node is reachable from initial node of flow graph G.
2. Back edges are those edges whose head dominates their tail.

8.8 is a reducible flow graph.

5 LOCAL OPTIMIZATION

al optimization is a type of optimization which is done of some sequence of ements. It is done with in the specific basic block. DAG can be constructed to do al optimization.

ample

B_1

$$t_1 = 4 * i$$
$$t_2 = 4 * j$$
$$t_3 = t_1 + t_2$$
$$t_4 = 4 * i$$
$$t_5 = t_3 * t_4$$

(8.3)

Construct DAG in order to perform local optimization.

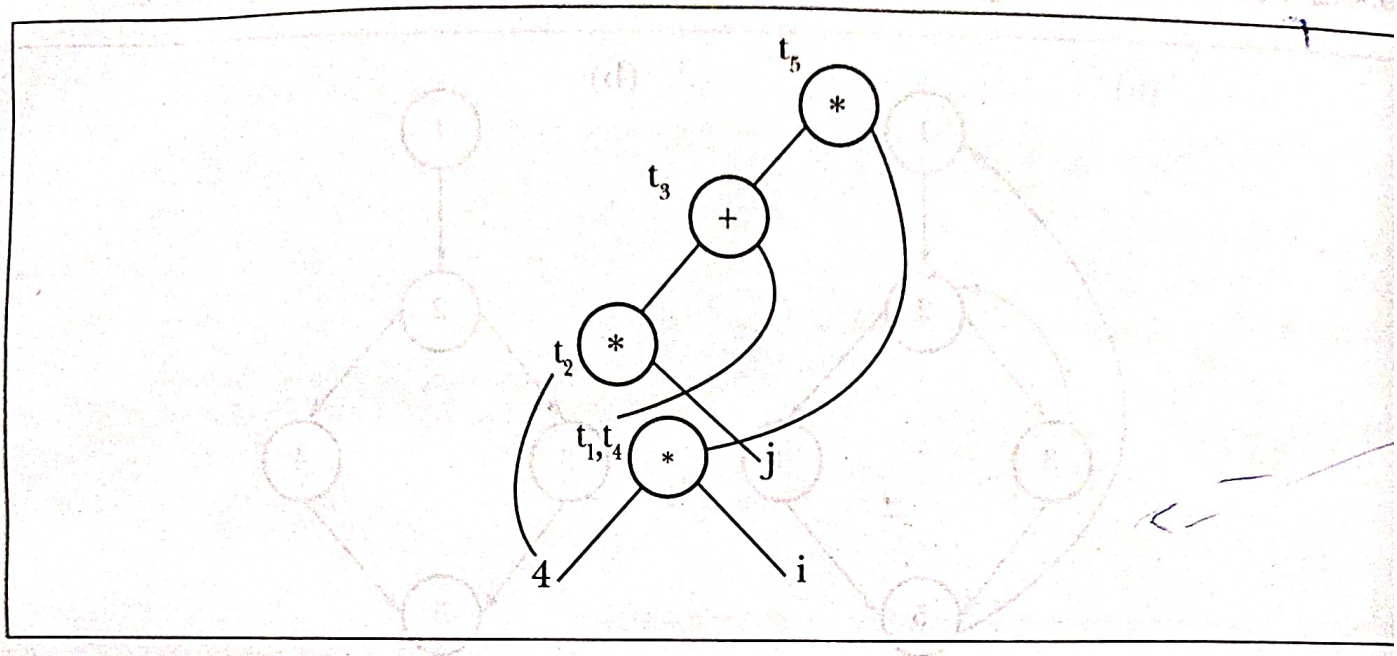


FIGURE 8.9: DAG for expression(8.3)

Local optimization is done to expression (8.3) which consists of block B_1.

Here the common subexpression is $t_4 = 4 * i$. This common subexpression is eliminated and the optimized code for the basic B_1 is:

$t_1 = 4 * i$

$t_2 = 4 * j$

$t_3 = t_1 + t_2$

$t_5 = t_3 * t_1$

The DAG for the above optimized code is:

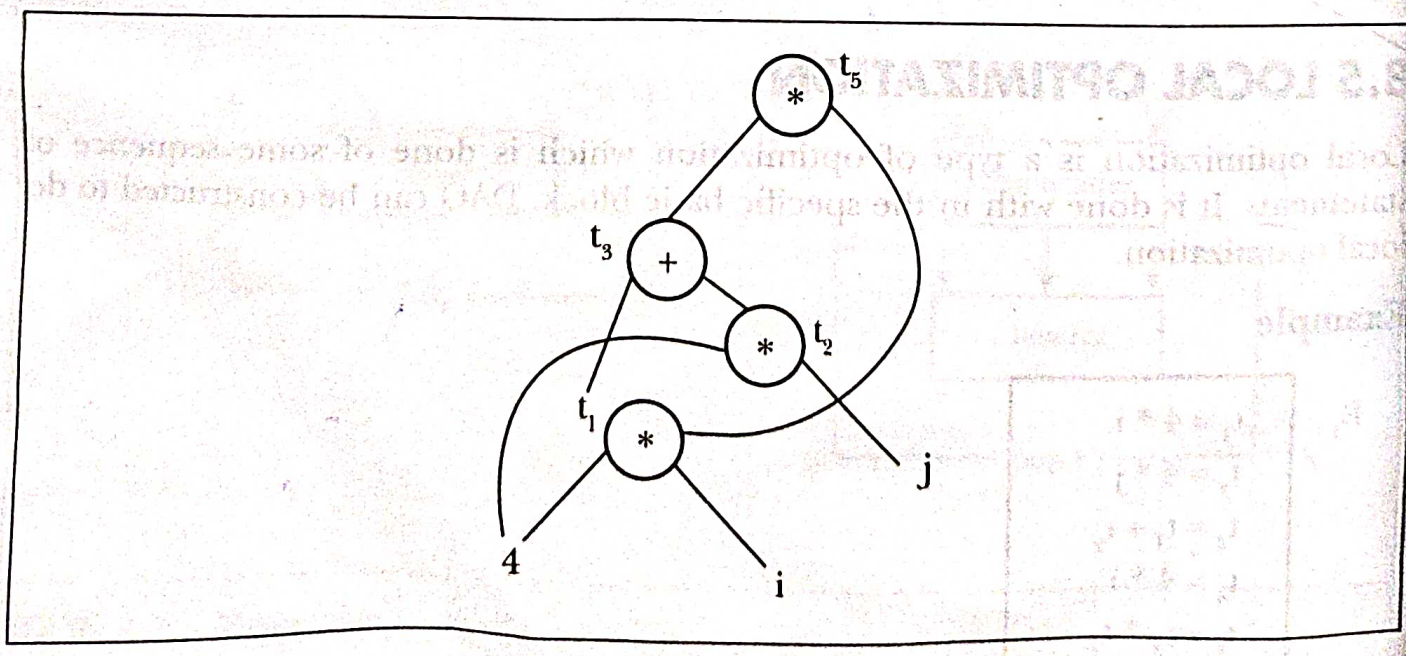


FIGURE 8.10: DAG after performing local optimization.

8.6 PEEPHOLE OPTIMIZATION

Peephole optimization is a simple but effective method used to locally improve the target code by examining a short sequence of target instructions known as 'peephole' and then replace these instructions by a shorter and / or faster sequence of instructions whenever required.

The code in the peephole may not be contiguous. Peephole optimization includes:

1. Redundant-Instruction Elimination

This includes elimination of redundant load and store instruction. Consider:

a) Mov R_1,8

b) Mov d, R_1

Instruction (b) can be deleted. Whenever (b) is executed, (a) will ensure that the value of 'd' is already in R_1.

2. Unreachable Code:

An unlabeled instruction that immediately follows an unconditional jump can be removed.

Example

```
i = j
if info = 2 goto L1
goto L2
L1: info is correct
L2:
```

⇓ Eliminate unreachable code

```
i = j
If info ≠ 2 goto L2
info is correct
L2:
```

[Here L1 immediately follows unconditional jump statement 'goto L2'].

3. Algebraic Simplifications

Algebraic identities that occur frequently and which is worth considering them can be simplified.

Example

X = X*1 is often produced by straight forward intermediate code generation algorithms. Hence they can be eliminated easily through Peephole optimization.