UNIT V

CODE OPTIMIZATION

The front end translates a source program into an intermediate representation from which the back end generates the target code.

Benefits of using a machine independent form are:

1) A compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2) Machine independent code optimizer can be applied to the intermediate representation.

5.1 CODE OPTIMIZATION

Codes produced after compilation can be made to run faster or take less space or both. This improvement is achieved by program transformations that are traditionally called optimizations.

Optimizing Compilers

Compilers that apply code improving transformations are called optimizing compilers.

Machine Independent Optimizations

It improves the larger code without taking into consideration any properties of the target machines.

Machine Dependent Optimization

It improves the target code by taking the target machines into consideration.

Example:

- Register allocation.
- Utilization of special machine instruction sequences and (machine idoms).

  We can optimize the code with less effort if we can identify the frequently executed parts of a program and then make these parts as efficient as possible. Most programs spend ninety percent of their execution time in ten percent of the code. Profiling the

run time execution of a program accurately identifies the heavily traveled regions of a program.

- The program inner loops are good candidates for improvement.

Criteria for Code Improving Transformations

The best program transformations are those that yield the most benefit for the least effort. The transformation provided by an optimizing compiler should have the following properties.

1) The transformation must preserve the meaning of programs. "That is an optimization" must not change the output produced by a program for a given input or cause an error that was not present in the original version of the source program.
2) A transformation must on the average speed up programs by a measurable amount.
3) A transformation must be worth the effort. Certain local or "peophole" transformation are simple enough and beneficial enough to be included in any compiler.

Getting Better Performance

Dramatic improvements in the running time of a program are usually obtained by improving the program at all levels from the source level to the target level.

An Organization for an Optimizing Compiler

There are often several levels at which a program can be improved. The code improvement phase consists of control flow and data flow analysis followed by the application of transformations.

Organization of the Code Optimizer

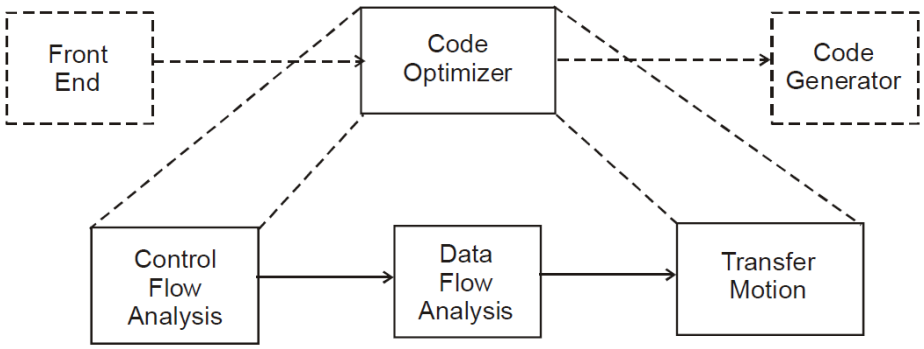


Figure 5.1

Advantages of the preceeding organization

1) The operations needed to implement high level constructs, as the address calculation for a[i], are made explicit in the intermediate code, so it is possible to optimize them.

For example the recomputation of 4 * *i* can be eliminated.

2) The intermediate code is independent of the target machine so the optimizer does not have to change much of the code generator is replaced by one for a different machine.

NOTE:

- In the code optimizes programs are represented by flow graphs in which edges indicates the flow of control and nodes represents basic blocks.

5.2 THE 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.

Function Preserving Transformations:

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Example: Common subexpression elimination, copy propagation, dead-code elimiantion, etc.

Local common subexpressions could be removed as the dag for the basic block was constructed.

Local Common Subexpression Elimination

Before	**After**
B_2	**B_2**
$t_1 := 4 * i$	$t_1 := 4 * i$
$t_2 := a[t\ ]$	$t_2 := a[t\ ]$
$t_3 := 4 * i$	$t_4 := b[t\ ]$
$t_4 := b[t3]$	

Common Subexpressions

An ‘occurrence’ of an expression E is called a common subexpression if E was previously computed and the values of variable in E have not changed since the previous computation.

We can avoid recomputing the expression if we can use the previously computed value. This change would result in the reconstruction of the intermediate code.

Flow graph before and after eliminating the local common subexpressions in B_2.

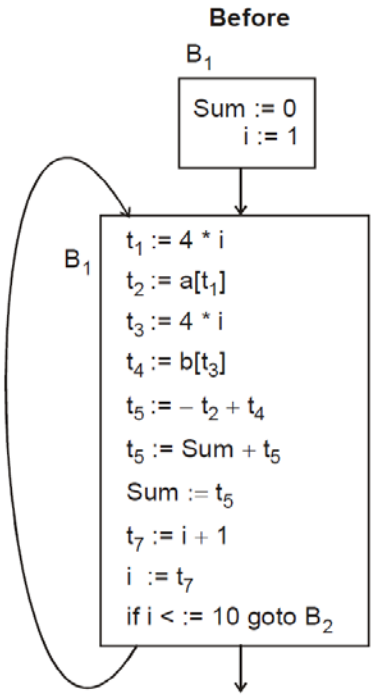


Figure 5.2

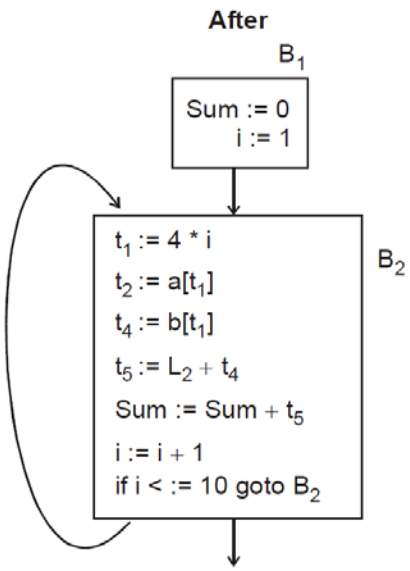


Figure 5.3

NOTE: This example is also applicable for copy propagation as sum is used in the place of t_6, and dead code elimination as 'sum = to' statement is removed.

Loop Optimizations

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. Three techniques are important for loop optimizataion.

1) Code Motion:

- which moves code outside a loop, if its value remains unchanged.

2) Induction Variable Elimination:

If the values of '*x*' and '*y*' remain in lock-step, (i.e) everytime the value of 'y' increase by 'i', the value of 'x' increases by '2' because of the statement x := y * 1. Then such identifiers as 'x' and 'y' 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.

3) Reduction in Strength:

- which replaces an expensive operation by a cheaper one.

Example: multiplication by an addition.

(1) Code Motion

This transformation takes an expression that yields the same result independent of the number of times a loop is executed and places the expression before the loop.

Example:

Evaluation of limit–2 is a loop invariant computation in the following while statement.

while (i <= limit – 2)

Code motion will result in the equivalence of t = limit–2 ;

while (i < = t).

(2) Induction Variable Elimination

- It is already discussed in detail.

Copy Propagation:

The idea behind copy propagation transformation is to use '*y*' for '*x*' wherever possible after the copy statement x := y.

For example copy propagation applied to the following block B

B_2

$x := t_3$
$a[t_4] := x$
goto B_2

yields the resultant block, as follows:

B_2

$x := t_3$
$a[t_4] := t_3$
goto B_2

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. The dead code may appear as the result of previous transformations.

For Example: In the following statement

if (debug) print ... .

If the value of debug is always false due to a preceding statement.

debug := false. , then the print statement is dead because it cannot be reached.

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*a nd transforms.

B_2

$x := t_3$

$a[t_4] := x$

goto B_2

Into

B_2

$a[t_4] := t_3$

goto B_2

This code is a further improvement of Block B_2.

Reduction in strength (detailed explanation)

It replaces expensive operation by a cheaper one such as multiplication by an addition.

For Example: The block B in the following flow graph.

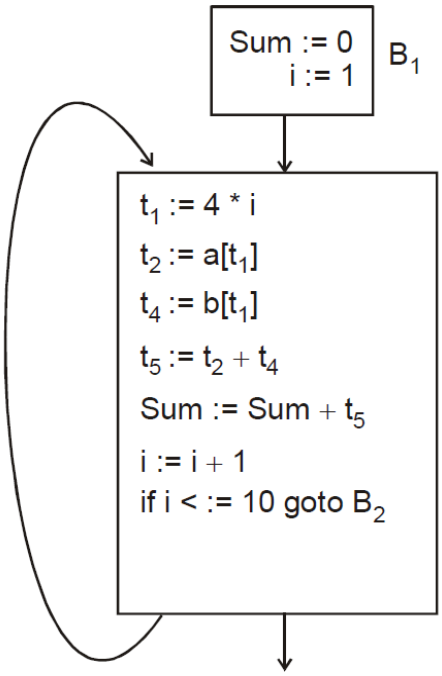


Figure 5.4

after reduction in strength replaces 't_1:= 4 * i" by "t_1:= t+ 4" as both the computations results in the same final answer.

Example: If i = 1

$t_1 = 4 * i = 4.$

Similarly $t_1 := 0 + 4 = 4$ | If i = 2

|t : = 4 * 2 = 8

Similarly |t= 4 + 4 = 8.

Thus the resultant flow graph after reduction in strength appears as follows:

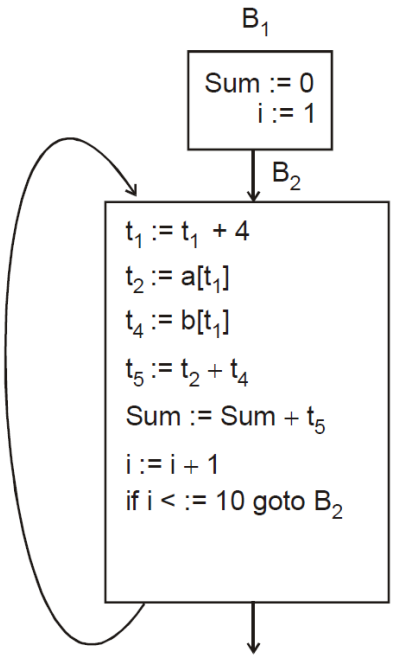


Figure 5.5

5.3 PEEPHOLE OPTIMIZATION

The quality of the larger code can be improved by applying “optimizing” transformation to the target program.

A simple but efficient techniques for “locally improving” the largest code is peophole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions. (called the peophole) and replacing these instructions by a shorter or faster sequence hidden ever possible.

Use:

Peophole optimization can not only be used to improve the quality of the target code, but, it can also be used to improve the intermediate representation.

NOTE:

- Repeated passes over the target code are necessary to get the maximum benefit of this optimization technique.

Characteristics of Peephole Optimizations:

- Redundant instruction elimination
- Flow of control optimizations
- Algebraic simplifications
- Use of machine idioms.

Redundant Load and Stores:

If we see the instruction sequence:

(1) MOV R0, a

(2) MOV a, R0

We can delete instruction (2) because the value of 'a' will already be present in R0.

Unreachable Code:

Another opportunity for peophole optimization is the removal of unreachable instructions. In 'C' a piece of code

```
# define debug 0
if (debug)      {
                print debugging information
                }
```

In the intermediate representation the if statement may be translated as:

if debug = 1 goto L_1

goto L_2

L: print debugging information.

L_2:

One obvious peophole optimization is to eliminate jumps over jumps. Thus the preceding statement may be replaced by:

if debug $\neq$ 1 goto L_2.

print debugging information.

L_2:

since debug is set to '0' is in the "# define debug 0" statement, constant "0" should replace debug as follows:

if $0 \neq 1$ go to L2

print debugging information.

L_2:

Zero is always not equal to one so, all the statements to print the debugging information is unreachable, so they can be eliminated one at a time.

Flow of Control Optimizations:

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peophole optimization.

We can replace the jump sequence

goto L_1

L_1 : goto L_2

by

goto L_2

L_1 : goto L_2

If there are no jumps to L1 , it may be possible to eliminate the statement.

L_1 : goto L_2

Similarly the sequence:

if $x > y$ goto L_1.

L_1 : goto L_2

can be replaced by:

if $x > y$ go to L_2

L_1 : goto L_2

The statement labeled L_1 can now be eliminated if there are no jumps to L_1.

The sequence

goto L_1

L_1 : if $x > y$ goto L_2

L_3 :

may be replaced by:

if $x > y$ goto L_2

goto L_3

L_3.

Algebraic Simplification:

There are number of algebraic simplification that can be attempted through peephole optimization. For (example) statements such as:

x := x + 0 (or)

x := x * 1.

can be easily eliminated through peophole optimization:

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

For example than calling an exponential routein to compute $x2$, $x * x$ computation can be cheaper.

Use of Machine Idioms:

The target machine may have hardware instructions to implement certain specific operations efficiently. Directing situations that permit the use of their instructions can reduce execution time significantly.

Example: Some machine have auto increment and auto decrement addressing modes. The use of these modes greatly improves the quality of code.

Approaches to compiler development:

There are several general approaches that a compiler writer can adopt to implement a compiler. The simplest is to retarget or an existing compiler.

If there is no suitable existing compiler the compiler writer might adopt the organization of a known compiler for a similar language and implement in corresponding components. The use of compiler building tools can be a significant help in this regard.

Boot Strapping

Using the facilities offered by a language to compile itself is the essence of bootstrapping.

Example: 'C' compilers are written in C.

For bootstrapping purpose a compiler is characterized by 3 languages.

- The source language 'S' that it compiles.
- The target language 'T' that it generates.
- The implementation language that it is written in.

The 3 language is represented using the following diagram called a 'T' diagram.

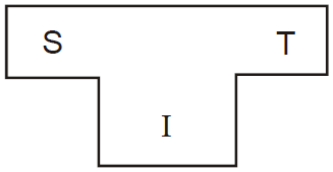


Figure 5.6 The 'T' diagram is abbreviated as SIT

The 3 languages may all be quite different.

Cross Compiler

It is a compiler which may run on one machine and produce target code for another machine.

- Suppose we write a cross-compiler for a new language 'L' in implementation language 'S' to generate code for machine N that is we create L_SN.
- If an existing compiler for 'S' runs on machine M and generates code for M it is characterized by S_MM.
- If LSN is run through S_MM we get a computer L_MN. This process is illustrated in the following Figure 5.10.

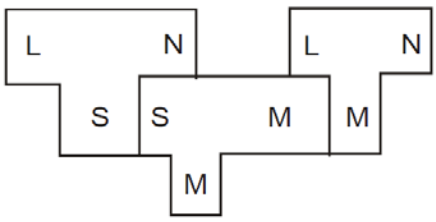


Figure 5.7

This can be represented by the following equation:

$$L_SN + S_MM = L_MN$$

Using boostrapping techniques, an optimizing compiler can optimize itself.

- Suppose all development is done an machine M.

  If we have S_SM a good optimizing computer for a language S written in 'S', and we want S_MM a good optimizing compiler for S written in M.

- First the optimizing compiler S_SM is translated by the quick and dirty compiler. SXX, to produce a poor implementation of the optimizing compiler, but one that does produce good code (SXM).

- The good optimizing compiler SMM is obtained by recompiling SSM through SXM.

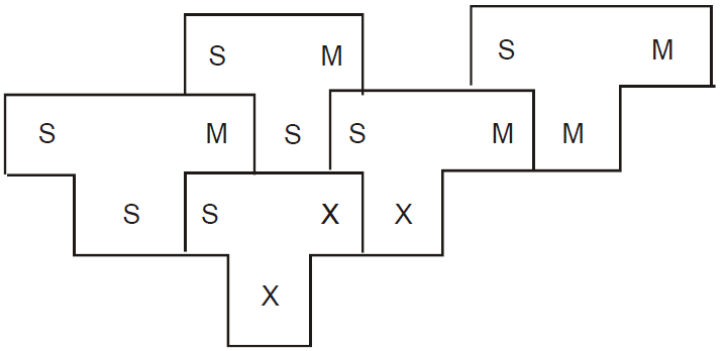


Figure 5.8

5.4 THE DAG REPRESENTATION OF BASIC BLOCKS

Directed a cyclic graphs (dags) are useful data structures for implementing transformation on basic blocks.

- A Dag gives a picture of Row the value computed by each statement in a basic block is used in subsequent statements of the block.

  A Dag for a basic block is a directed a cyclic graph with the following labels on nodes.

  1) Leaves are labeled by unique identifiers, variable names or constants.

  2) Interior nodes are labeled by an operator symbol.

  3) Nodes are also optionally given a sequence of identifiers for labels.

- The interior noes represent computed values and the identifier labeling a node are deemed to have that value.

Dag Construction

To construct a dag for a basic block, we process each statement of the block in turn.

- When we see a statement of the form x := y + z we took for the node labeled '+' and whose left child is y+ and right child is 'z'.

  If so label that node with an additional label 'x'. Otherwise create a new node labeled

  '+' and whose left child is 'y' and right child is 'z'.

- For the assignment statements of the form x := y we do not create a new node. Rather we append label 'x' to the list of names on the node folding the current value of 'y'.

  We how give the algorithm to comupte a dag from a block.

Algorithm: Constructing a dag.

Input: A basic block.

Output: A dag for the basic block contains the following informations.

1. A label for each node.
   - For leaves the label is an identifier.
   - For interior nodes the label is an operator symbol.
2. For each node a list of attached identifiers.

Method:

The dag construction process is to do the following steps, (1) through, (3) for each statement of its block in turn.

(1) If the current 3 address statement is x := yopz.

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).

(2) If the 3 address statement is x := yopz determine if there is a node labeled 'op' whose left child is node (y) and whose right child is node (z).

If *not create such a node.

- If the 3 address statement is of the form. x := y, determine whether there is a node labeled op whose lone child is node (y). If not create such a node.
- If the 3 address statement is of the form x := y. Find the node which holds, the value 'y'.

(3) Delete 'x' from the list of attached identifiers for node (x). Append 'x' to the list of attached identifiers for the node 'n' found in (2), and set node *c* = 1 to '*n*'.

The Dag for the statement *x* := *y* + *z* is:

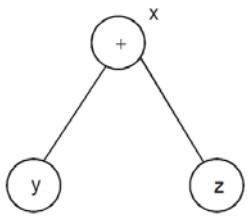


Figure 5.9

The Dag for the statements.

$t_1 := 4 * i$

$t_2 := a[t]$

is

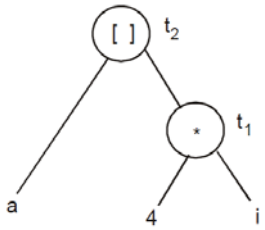


Figure 5.10

Applications of Dag's

There are several pieces of useful information that we can obtain as we are executing the preceding algorithm.

- First we can automatically detect, common subexpressions.
- Second we can determine which identifiers have their values used in the block.

  $\Rightarrow$ they are exactly those for which a leaf is created in step (1).
- Third we can determine which statements compute values that could he used outside the block $\Rightarrow$ they are exactly those statements S.

  whose node '*n*' constructed or found in step 2 still far node (*x*) = *n*, at the end of the dag construction where '*x*' is the identifier assigned by statement S.

Example: ***Consider the statements:***

(1) $x := a + b$

(2) $y := p + q$

(3) $x := y + d$

In this case the value of '*x*' in statement number (1) is not live on exit but the value of '*x*' in statement (3) is live, on exist.

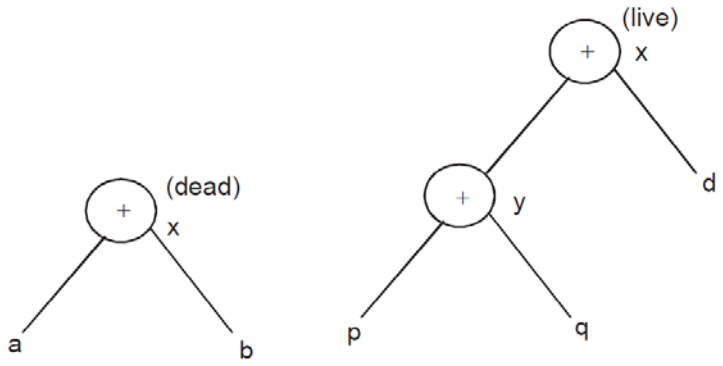


Figure 5.11

- Another important use to which the dag may be put is to reconstruct a simplified list of quadruples taking advantages of common subexpressions.

NOTE: We may generally evaluate the interior nodes of the dag in any order that is a topological sort of the dag. In a topological sort, a node is not evaluated until all of its children that are interior nodes have been evaluated.

Arrays Pointers and Procedure calls:

Arrays

Consider the basic block:

x := a[i]

a[j] := y

z := a[i]

If the dag is constructed a[i] would become a common subexpression and the optimized block would turn out to be:

x := a[i]

z := x a[j] = y

For each array it is convenient to have a list of all nodes currently not killed. A node must be killed when a new element is assigned to an element of the array.

Pointers

If we have an assignment statement such as **p* := ω , and if *p* could only point to *r* or *s* then only node (*r*) and node (*s*) must be killed.

Procedure Call

A procedure call in a basic block kills all nodes, since any variable may be changed as a side effect of the called procedure.

5.5 OPTIMIZATION OF BASIC BLOCKS

Code improving transformations for basic blocks includes:

- common subexpression elimination
- dead code elimination
- reduction in strength.

Many of the structure preserving transformation can be implemented by constructing a dag for a basic block.

Common subexpressions can be detected by noticing, a new node ‘m’ which about to be added in a dag, with the same children of an existing node ‘n’ in the same order. If so ‘n’computes the same value as ‘m’ and may be used in its place.

For Example:

The dag for the following block:

a := b + c

b := a – d

c := b + c

d := a – d is shown as follows:

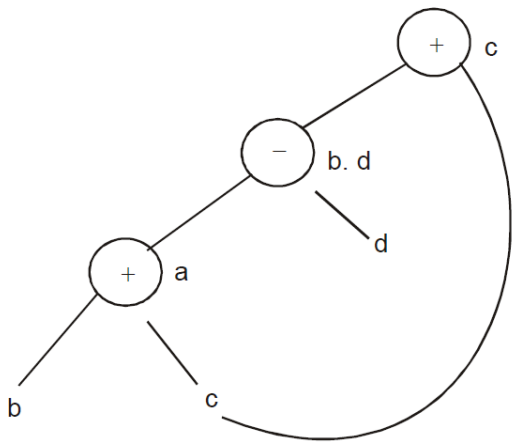


Figure 5.12

If *b* is not live on exit we can eliminate the common subexpressions and transform the block as follows:

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

Corresponding dag is:

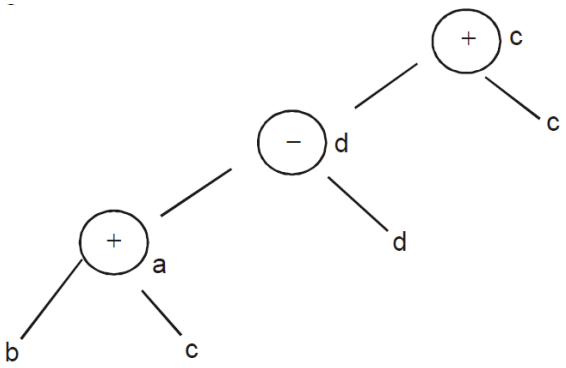


Figure 5.13

The operation on dag that corresponds to dead code elimination is to delete from a dag any root (node with no ancestors) that has no live variable. Repeated application of the transformation will remove all nodes from the dag that correspond to dead code.

The Use of Algebraic Identities:

We may apply authentic identities such as:

```
x + 0 = 0 + x = x
x – 0 = x
x * 1 = 1 × x
x/1 = x
```

Another class of algebraic optimization includes reduction in strength that is replacing a more expensive operator by a cheaper one as in

x * * 2 = x * x

2.0 * x = x + x

x / 2 = x * 0.5

Another class of related optimizations is constant folding.

Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2 * 3 would be replaced by 6.

General Algebraic Transformations are:

- commutativity and
- associativity.

For example if '*' is commutative

x * y = y * x. Therefore before creating a node '*' with left child * and right child y check for any existing node 'x' with its left child 'y' and right child 'x'.

Associative laws may also be applied to expose common subexpressions.

For example if the source code has the assignments:

a := b + c

e := c + d + b.

The following intermediate code might be generated:

a := b + c

t := c + d

e: = t + b.

If 't' is not needed outside the block we can change this sequence to:

a := b + c

e := a + d.

5.6 GLOBAL DATA FLOW ANALYSIS

- Data-flow analysis refers to a body of techniques that derive information about the flow of data along program execution paths.

5.6.1 BASIC BLOCKS AND FLOW GRAPHS

A graph representation of 3 address statements is called a flow graph.

Nodes in the flow graph represent computations and the edges represents the flow of control.

Use

We can find the inner loops where a program is expected to spend most of its time using flow graphs.

Basic Blocks

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

The following sequence of three address, statements forms a basic block:

$t := a * a$

$t := a * b$

$t := 2 * t$

$t := t + t$.

- A name in a basic block is said to be live at a given point if the 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 this basic block.

Algorithm

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 determines the set of leaders, the first statement of basic blocks. The rules for the selection of leaders 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 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 source code shown below, which is used to compute the sum of two arrays

   begin

   sum := 0

   i := 1

```
do begin
    sum := sum + a[i] + b[i];
        i = i + 1;
    end
    While i < - 10
end
```

A list of 3 address statements performing the preceding computation is:

(1) sum := 0

(2) i = 1

(3) t1 := 4 * *i*

(4) t2 := *a*[*t*1]

(5) t3 := 4 * *i*

(6) t4 := *b*[*t*3]

(7) t5 := t2 + t4

(8) t6 := sum + t5

(9) sum := t6

(10) t7 := i + 1

(11) i = t7

(12) i < = 10 goto 3

NOTE: The size of each element in the array is 4 bytes.

∴ 1st element is in position a[4] 2nd element is in position a[8] and so on.

- statement (1) is a leader by rule (1)
- statement (3) is a leader by rule (2) (since statement (12) jumps to it)

∴ The block 1 and block 2 appears as follows:

Block B1

Sum:=0
i:=1

Block B2

```
t1 := 4 * i
t2 := a[t1]
t3 := 4 * i
t4 := b[t3]
t5 := t2 + t4
t6 := sum + t5
sum := t6
t7 := i + 1
i = t7
if i < = 10 goto B2
```

Transformations on Basic Blocks

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. Many of these transformation are useful for improving the quality of code that will be ultimately generated from a basic block.

There are two important classes of local transformations that can be applied to basic blocks they are structure preserving transformations and the algebraic transformation.

Structure Preserving Transformations:

The primary structure preserving transformations on basic blocks are:

(1) Common subexpression elimination

(2) Dead code elimination

(3) Renaming of temporary variables

(4) Intercharge of two independent adjacent statements.

(1) Common Subexpression Elimination

Consider the basic block

$$a := b + c$$

$d := b + c.$

As both the statements performs the same computation in the block, it can be transformed into an equivalent block as shown next

$a := b + c$

$d := a.$

(2) Dead Code Ellimination

Suppose the statement x := y + z appears in a basic block, and if 'x' is not subsequently used, 'x' is dead. Then this statements may be safely removed without changing the value of the basic block.

(3) Renaming Temporary Variables

Suppose we have a statement t := b + c where 't' is a temporary. If we change this statement to $u := b + c$, where 'u' is a new temporary variable, then change all uses of this instance of 't' to u is then the value of the basic block is not changed.

NOTE: 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 basic block a normal form block

(4) Interchange of Statements

Suppose we have a block with the two adjacent statements

$t_1 := b + c$

$t_2 := x + y$

Then we can inter-change the two statements without affecting the value of the block.

Algebraic Transformations:

Countless number of algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

For example statement such as:

x := x + 0. or

x := x * 1.

can be eliminated from a block.

Without changing the set of expressions it computes.

The exponentiation operator in the statement:

x := y ↑ 2

Usually requires a function call to implement. Using an algebraic transformation the statement can be replaced by the cheaper but equivalent statement:

x := y * y

Flow Graphs:

We can add the flow of control information to the set of basic blocks, 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. B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence

1) There is a jump from the task statement of B_2 to the first statement of B_2.
2) B_2 immediately follows B1 in the order of the program.

We say that B_1 is a predecessor of B_2 and B_2 is a successor of B_1. Thus the flow graph to add the elements of two arrays is represented as follows:

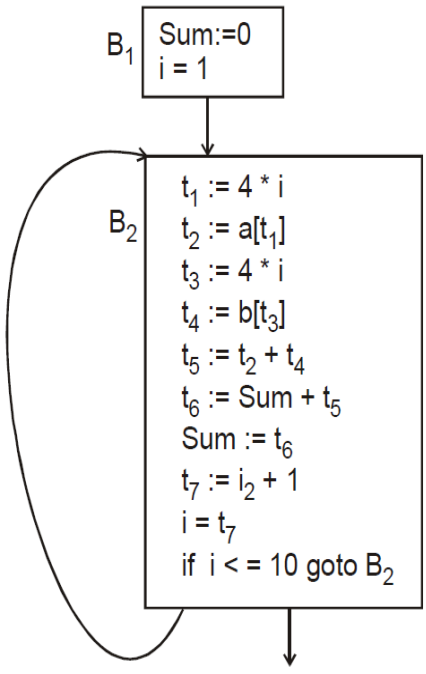


Figure 5.14

Loops

In the preceding flow graph there is one loop consisting of block B_2.

Loop is a Collection of nodes in a flow graph such that

(1) All nodes in the collection are strongly connected that is from any node in the loop to any other, there is a path of length one or more, wholly within the loop and

(2) The collection of nodes has a unique entry that is a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

A loop that contains no other loops is called an inner loop.

Representation of Basic Blocks

Basic block can be represented by a variety of data structures.

- Each basic block can be represented by a record consisting of a count of the number of quadruples in the block.

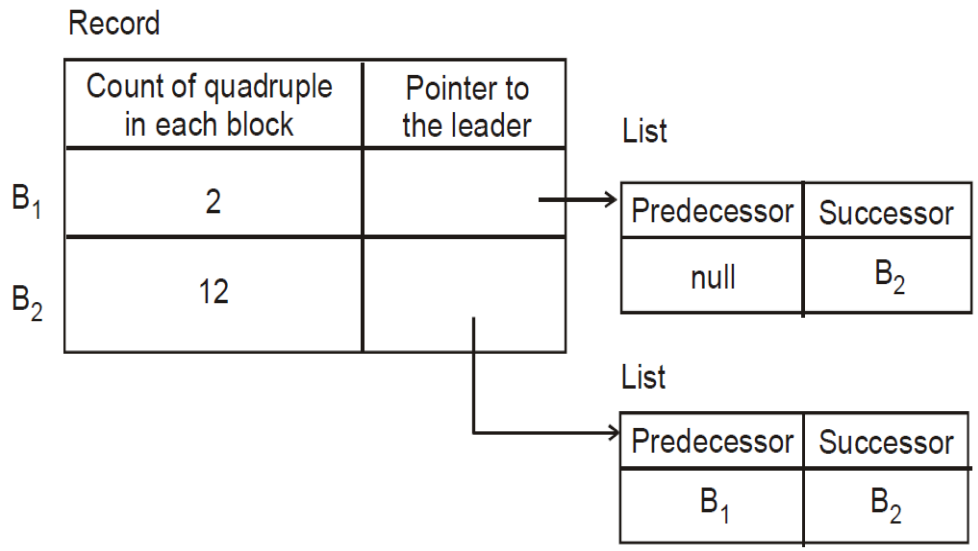


Figure 5.15

- An alternative is to make a linked list of the quadruples in each block.

5.6.2 DATA-FLOW ABSTRACTION

Each execution of an intermediate-code statement transforms an input state to a new output state. The input state is associated with the program point before the statement and the output state is associated with the program point after the statement.

Let us see what the flow graph tell us about the possible execution paths.

- Within one basic block, the program point after a statement is the same as the program point before the next.
- If there is an edge from block B_1 to block B_2 then the program point after the last statement of B_1 may be followed immediately by the program point before the first statement of B_2.

Thus, we may define an execution path from point P1 to point Pn to be a sequence of points P1, P2, ..., Pn such that for each $i = 1, 2, ..., n - 1$, either.

1. P_i is the point immediately proceeding a statement and P_{i+1} is the point immediately following that same statement, or
2. P_i is the end of some block and P_{i+1} is the beginning of a successor block.

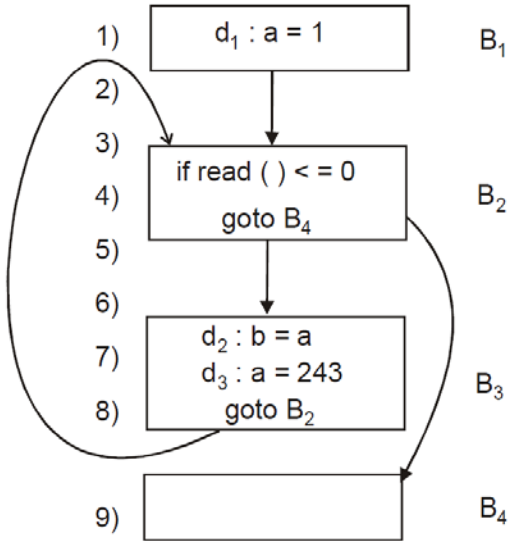


Figure 5.16 Example program for data-flow abstraction

5.6.3 THE DATA-FLOW ANALYSIS SCHEMA

We denote the data-flow values before and after each statement S by IN [s] and OUT [s], respectively. The data-flow problem is to find a solution to a set of constraints on the IN [s]'s and OUT [s]'s, for all statements S. There are two set of constraints; those based on the semanting of the statements and those based on the flow of control.

Transfer Functions

The data-flow values before and after a statement are constrained by the semantics of the statement.

It has two flavours:

I. Forward along execution paths.

II. Backwards up the execution paths.

I. Forward along execution paths

The transfer function of a statement S, which we shall usually denote f_s, takes the data-flow the before the statement and produces a new data-flow value after the statement.

$$OUT[s] = fs\ (IN\ [s])$$

II. Backward-Flow Problem

The transfer function f_s for statement S converts a data-flow value after the statement to a new data-flow value before the statement

$$IN[s] = fs\ [OUT\ [s])$$

Control-Flow Constraints

The second set of constraints on data-flow values is derived from the flow of control within a basic block, control flow is simple. If a block B consists of statement S_1 , S_2 , ..., S_n in that order, then the control-flow value out of S_1 is the same as the control-flow value into S_{i+1}. That is:

$$IN\ [S_{i+1}] = OUT\ [S_i], \text{ for all } i = 1, 2, ..., n - 1.$$

5.6.4 DATA-FLOW SCHEMES ON BASIC BLOCKS

We denote the data-flow values immediately before and immediately after each basic block By IN[B] and OUT[B], respectively.

I. Form and Flow Problem

$$IN\ [B] = \cup\ p \text{ a predecessor of B } OUT\ [P]$$

II. Backwards-Flow Problem

$$IN\ [B] = f_B\ (OUT\ [B])$$

$$OUT\ [B] = \cup_S \text{ a successor of B } IN[s].$$

5.6.5 REACHING DEFINITIONS

- It is used for data-flow schemes.
- By knowing where in a program each variable x may have been defined when control reaches each point P, we can determine many things about x.
- A definition d reaches a point P if there is a path from the point immediately following d to p, such that d is not “killed” along that path. We kill a definition of a variable x, if

there is any other definition of x reaches point P, then d the value of x used at P was last defined.

5.6.6 TRANSFER EQUATIONS FOR REACHING DEFINITION

Consider a definition

$$d : V = V + W$$

definition d

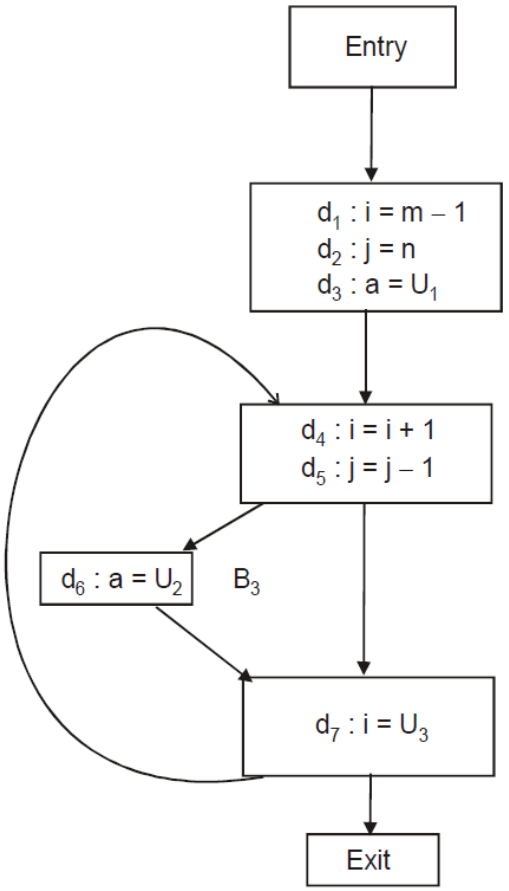
Variable V

binary operator +

transfer function,

$$f_d(x) = gen_d \cup (x - kill_d)$$

where $gen_d = \{d\}$, the set of definitions generated by the statement, and $kill_d$ is the set of all other definition of V in the program.



B_1 $\quad gen_1 = \{d_1, d_2, d_3\}$

$\quad kill\ B_1 = \{d_4, d_5, d_6, d_7\}$

$\quad gen\ B_2 = \{d_4, d_5\}$

B_2 $\quad kill\ B_2 = \{d_1, d_2, d_7\}$

$\quad gen\ B_3 = \{d_6\}$

B_3 $\quad kill\ B_3 = \{d_3]$

$\quad gen\ B_4 = \{d_7\}$

B_4 $\quad kill\ B_4 = \{d_1, d_4\}$

Figure 5.17

$f_2(f_1(x)) = gen_2 \cup (gen_1 \cup (x - kill_1)$

$= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2))$

Transfer function for block B may be written as:

$f_B(x) = gen_B \cup (x - kill_B)$

where $kill_B = kill_1 \cup kill_2 \cup ... \cup kill_n$,

and

$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup ... \cup (gen_1 - kill_2 - kill_3 - ... - kill_n)$

5.6.7 CONTROL-FLOW EQUATIONS

We consider the set of constraints derived from the control flow between basic blocks. If one path can be reaches:

$$OUT [P] \subseteq IN [B]$$

$$IN [B] = \bigcup_{p} \text{a predecessor of B } OUT [P].$$

5.6.8 LIVE-VARIABLE ANALYSIS

In live-variable analysis we wish to know for variable *x* and point *p* whether the value of *x* at *p* could be used along some path in the flow graph starting at *p*.

If so, we say *x* is live at *p*; otherwise, *x* is dead at *p*.

Define:

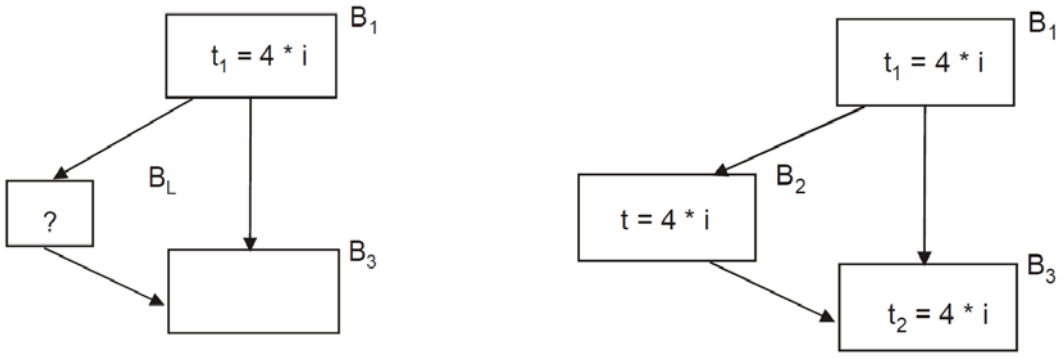
1. def_B as the set of variables defined in B prior to any use of that variable in B and
2. Use_B as the set of variables whose values may be used in B prior to any definitions of the variable.

5.6.9 AVAILABLE EXPRESSIONS

An expression x + y is available at a point p if every path from the entrynode to p evaluate x + y, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y.

In available expression (x + y),

(a) block kill expression, (b) block generation expression.



(a) 4*i expression in block ***(b) 4*i recomputed after i***

Figure 5.18 Subexpressions across blocks

5.7 EFFICIENT

Algorithm: Iterative solution to general data - flow - frameworks.

INPUT: A data-flow framework with the following components:

1. A data-flow graph, with specially labeled ENTRY and EXIT nodes.
2. A direction of the data-flow D.
3. A set of values V.
4. A meet operator A.
5. A set of function F, where f_B in f is the transfer function or block B, and
6. A constant value V_{ENTRY} or V_{EXIT} in V, representing the boundary condition or forward and backward frameworks, respectively.

OUTPUT: Values in V for IN [E] and OUT [B] for each block B in the data-flow graph.

METHOD:

Properties of Iterative Algorithm.

- If algorithm of data flow, the result is a solution to the data-flow equations.
- If the framework is monotone, then the solution found is the maximum fixed point (MFP) of the data-flow equations. A maximum fixed point is a solution with the property that in any other solution, the values of IN[B] and OUT[B] are $\leq$ the corresponding values of the MFP.

(a) Iterative Algorithm for a Forward Data-Flow Algorithm

1. 1) OUT [ENTR] = V_{ENTRY};
2. for (ach basic block B other than ENTRY) OUT [B] □ T'
3. while (changes to any OUT occur)
4. for (each basic block B other than ENTRY) { OUT [p];
5. IN [B] = ∧p a predecessor of B

   OUT [B] = f_B(IN [B]);

   }

(b) Iterative Algorithm for a Backward Data-Flow Problem

6. IN [EXIT] = V_{EXIT};
7. for (each basic block B other than EXIT) IN [B] = T;
8. while (changes to any IN occur)
9. for (each basic block B other then EXIT) { IN[s];
10. OUT [B] =∧s a successor of B
11. IN [B] = fB (OUT [B]);

    }