RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Compiler Design**

Subject Code: **CS-603**

Semester: **6th**

UNIT- V:

Introduction to Code optimization: sources of optimization of basic blocks, loops in flow graphs, dead code elimination, loop optimization, Introduction to global data flow analysis, Code Improving transformations, Data flow analysis of structure flow graph Symbolic debugging of optimized code.

1. Code Optimization

Code optimization phase is an optional phase in the phases of a compiler, which is either before the code generation phase or after the code generation phase. This chapter focuses on the types of optimizer and the techniques available for optimizing.

The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

Optimizations are classified into two categories. They are

Machine independent optimizations:

Machine dependent optimizations:

1.1 Machine independent optimizations:

Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

1.1 Machine dependent optimizations:

Machine dependent optimizations are based on register allocation and utilization of special machine-instruction sequences.

1.1 The criteria for code improvement transformations:

Simply stated, the best program transformations are those that yield the most benefit for the least effort.

The transformation must preserve the meaning of programs. That is, the 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 source program. At all times we take the “safe” approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an “optimization” may slow down a program slightly.

The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. “Peephole” transformations of this kind are simple enough and beneficial enough to be included in any compiler.

2. Principal Sources Of Optimisation

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.

2.1 Function-Preserving Transformations

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

The transformations

- Common sub expression elimination,
- Copy propagation,
- Dead-code 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 the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

2.3 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 re-computing 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 value of i is not been changed from definition to use.

2.4 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.

For 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

2.5 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. An optimization can be done by eliminating dead code.
Example:

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

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

2.6 Constant folding:
We can eliminate both the test and printing from the object 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,
a=3.14157/2 can be replaced by
a=1.570 thereby eliminating a division operation.

2.7 Loop Optimizations:
We now give a brief introduction to a very important place for optimizations, namely loops, especially the 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.

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.

2.8 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 */
```

2.9 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. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

3. Optimization Of Basic Blocks
There are two types of basic block optimizations. They are:

Structure-Preserving Transformations
Algebraic Transformations

Structure-Preserving Transformations:
The primary Structure-Preserving Transformation on basic blocks are:

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

3.1 Common sub-expression elimination:
Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered aga in – of course providing the variable values in the expression still remain constant.
Example:
a: =b+c
b: =a-d
c: =b+c
d: =a-d
The 2nd and 4th statements compute the same expression: b+c and a-d
Basic block can be transformed to
a: = b+c
b: = a-d
c: = a
d: = b

3.2 Dead code elimination:
It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

3.3 Renaming of temporary variables:
A statement t: =b+c where t is a temporary name can be changed to u: =b+c where u is another temporary name, and change all uses of t to u.
In this we can transform a basic block to its equivalent block called normal-form block.
Interchange of two independent adjacent statements:
Two statements

t1:=b+c
t2:=x+y
can be interchanged or reordered in its computation in the basic block when value of t 1does not affect the value of t2.

3.4 Algebraic Transformations:
Algebraic identities represent another important class of optimizations on basic blocks.
This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. Reduction in strength.
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*3.14 would be replaced by 6.28.
The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.
Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments
a :=b+c
e :=c+d+b

the following intermediate code may be generated:
a :=b+c
t :=c+d
e :=t+b

4. Loops In Flow Graph
A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.
Dominators:
In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by d dom n. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.
Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:
Invariant code: A fragment of code that resides in the loop and computes the same• value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
Induction analysis: A variable is called an induction variable if its value is altered• within the loop by a loop-invariant value.
Strength reduction: There are expressions that consume more CPU cycles, time,• and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Natural Loop:
- One application of dominator information is in determining the loops of a flow graph suitable for improvement.
- The properties of loops are A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
- There must be at least one way to iterate the loop(i.e.)at least one path back to the header.

- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.

5. Dead-Code Elimination

Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code

There are some code statements who's computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.

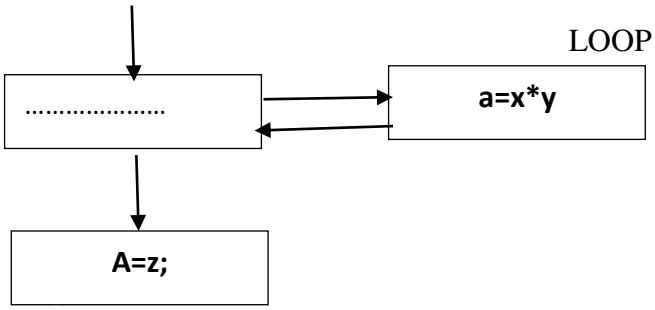

Figure 5.1: Dead–code elimination

The above control flow graph depicts a chunk of program where variable 'a' is used to assign the output of expression 'x * y'. Let us assume that the value assigned to 'a' is never used inside the loop. Immediately after the control leaves the loop, 'a' is assigned the value of variable 'z', which would be used later in the program. We conclude here that the assignment code of 'a' is never used anywhere, therefore it is eligible to be eliminated.

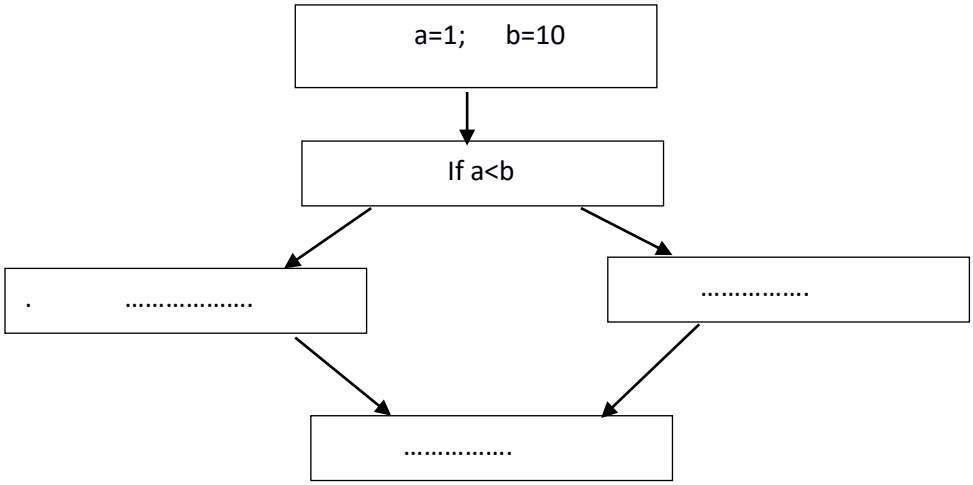

Figure 5.2: Dead–code elimination

Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

5.1 Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands. Whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,

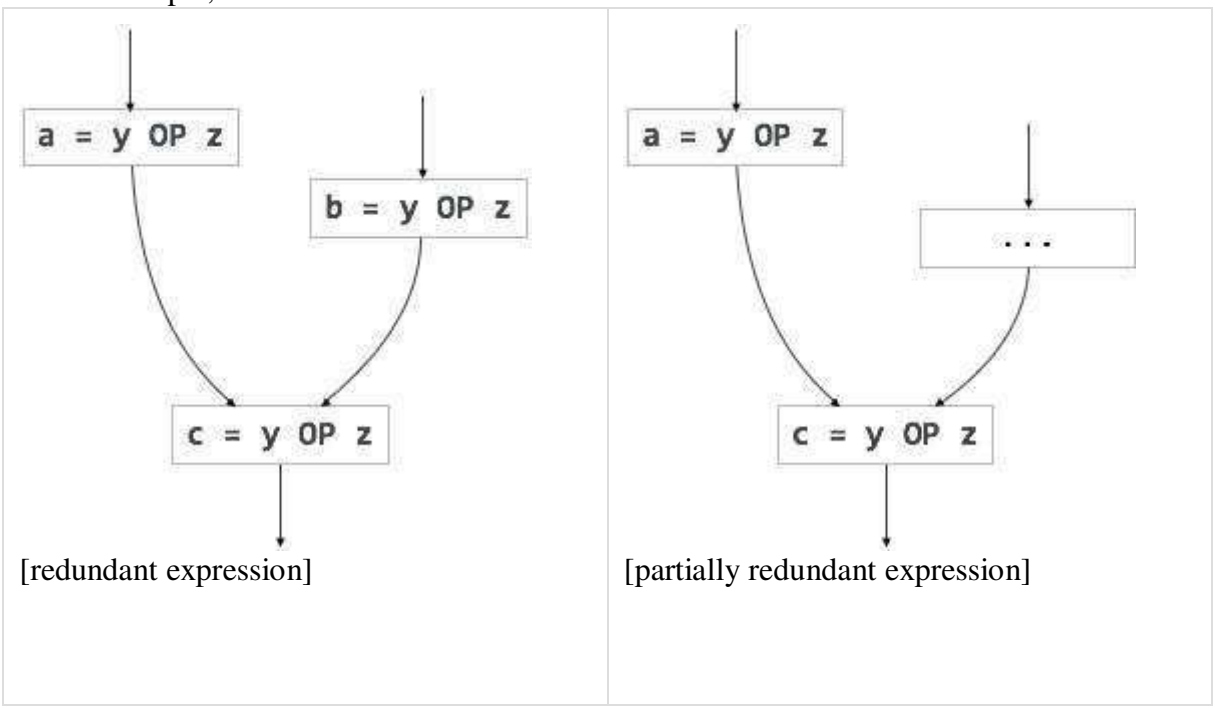

Figure 5.3: Partial Redundancy

Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique. Another example of a partially redundant code can be:

```
If (condition)
{
  a = y OP z;
}
else
{
  ...
}
c = y OP z;
```

We assume that the values of operands (**y** and **z**) are not changed from assignment of variable **a** to variable **c**. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{
  ...
  tmp = y OP z;
  a = tmp;
  ...
}
else
{
  ...
```

tmp = y OP z;
}
c = tmp;
Here, whether the condition is true or false; y OP z should be computed only once.

6. Introduction To Global Dataflow Analysis

- In order to do code optimization and a good job of code generation, compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of "reaching definitions" , such as knowing where a variable like debug was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data-flow information can be collected by setting up and solving systems of equations of the form :

- out [S] = gen [S] U (in [S] – kill [S])

- This equation can be read as "the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement."

The details of how data-flow equations are set and solved depend on three factors.

- The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].
- Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
- There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

6.1 Points and Paths:

Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.

Now let us take a global view and consider all the points in all the blocks. A path from p1 to pn is a sequence of points p1, p2,….,pn such that for each i between 1 and n-1, either

1. Pi is the point immediately preceding a statement and pi+1 is the point immediately following that statement in the same block, or
2. Pi is the end of some block and pi+1 is the beginning of a successor block.

6.2 Reaching definitions:

A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x. These statements certainly define a value for x, and they are referred to as unambiguous definitions of x. There are certain kinds of statements that may define a value for x; they are called ambiguous definitions.

The most usual forms of ambiguous definitions of x are:

1. A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.
2. An assignment through a pointer that could refer to x. For example, the assignment *q:=y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.

We say 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. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the appearing later along one path.

6.3 Data-flow analysis of structured programs:

Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

S->id: = E| S; S | if E then S else S | do S while E
E->id + id| id

Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.

We define a portion of a flow graph called a region to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header. The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

We say that the beginning points of the dummy blocks at the statement's region are the beginning and end points, respective equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S. gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.

7. CODE IMPROVING TRANSFORMATIONS

- Algorithms for performing the code improving transformations rely on data-flow information. Here we consider common sub-expression elimination, copy propagation and transformations for moving loop invariant computations out of loops and for eliminating induction variables.

- Global transformations are not substitute for local transformations; both must be performed.

7.1 Elimination of global common sub expressions:

The available expressions data-flow problem discussed in the last section allows us to determine if an expression at point p in a flow graph is a common sub-expression. The following algorithm formalizes the intuitive ideas presented for eliminating common sub expressions.

ALGORITHM: Global common sub expression elimination. INPUT: A flow graph with available expression information.

OUTPUT: A revised flow graph.
METHOD: For every statement s of the form x := y+z6 such that y+z is available at the beginning of block and neither y nor r z is defined prior to statement s in that block, do the following.

To discover the evaluations of y+z that reach s's block, we follow flow graph edges, searching backward from s's block. However, we do not go through any block that evaluates y+z. The last evaluation of y+z in each block encountered is an evaluation of y+z that reaches s.

- Create new variable u.
- Replace each statement w: =y+z found in (1) by
- u : = y + z
- w : = u
- Replace statement s by x:=u.
- Some remarks about this algorithm are in order.
- The search in step(1) of the algorithm for the evaluations of y+z that reach statement s can also be formulated as a data-flow analysis problem. However, it does not make sense to solve it for all expressions y+z and all statements or blocks because too much irrelevant information is gathered.

Not all changes made by algorithm are improvements. We might wish to limit the number of different evaluations reaching s found in step (1), probably to one.
Algorithm will miss the fact that a*z and c*z must have the same value in
a :=x+y c :=x+y
vs
b :=a*z d :=c*z

Because this simple approach to common sub expressions considers only the literal expressions themselves, rather than the values computed by expressions.

7.2 Copy propagation:

- Various algorithms introduce copy statements such as x :=copies may also be generated directly by the intermediate code generator, although most of these involve temporaries local to one block and can be removed by the dag construction. We may substitute y for x in all these places, provided the following conditions are met every such use u of x.
- Statement s must be the only definition of x reaching u.
- On every path from s to including paths that go through u several times, there are no assignments to y.
- Condition (1) can be checked using ud-changing information. We shall set up a new dataflow analysis problem in which in[B] is the set of copies s: x:=y such that every path from initial node to the beginning of B contains the statement s, and subsequent to the last occurrence of s, there are no assignments to y.

ALGORITHM: Copy propagation.
INPUT: a flow graph G, with ud-chains giving the definitions reaching block B, and with c_in[B] representing the solution to equations that is the set of copies x:=y that reach block B along every path, with no assignment to x or y following the last occurrence of x:=y on the path. We also need ud-chains giving the uses of each definition.

OUTPUT: A revised flow graph.
METHOD: For each copy s : x:=y do the following:

Determine those uses of x that are reached by this definition of namely, s: x: =y.

Determine whether for every use of x found in (1) , s is in c_in[B], where B is the block of this particular use, and moreover, no definitions of x or y occur prior to this use of x within B. Recall that if s is in c_in[B]then s is the only definition of x that reaches B.
If s meets the conditions of (2), then remove s and replace all uses of x found in (1)
by y.

7.3 Detection of loop-invariant computations:
Ud-chains can be used to detect those computations in a loop that are loop-invariant, that is, whose value does not change as long as control stays within the loop. Loop is a region consisting of set of blocks with a header that dominates all the other blocks, so the only way to enter the loop is through the header.
If an assignment x := y+z is at a position in the loop where all possible definitions of y and z are outside the loop, then y+z is loop-invariant because its value will be the same each time x:=y+z is encountered. Having recognized that value of x will not change, consider v := x+w, where w could only have been defined outside the loop, then x+w is also loop-invariant.

ALGORITHM: Detection of loop-invariant computations.
INPUT: A loop L consisting of a set of basic blocks, each block containing sequence of three-address statements. We assume ud-chains are available for the individual statements.
OUTPUT: the set of three-address statements that compute the same value each time executed, from the time control enters the loop L until control next leaves L.
METHOD: we shall give a rather informal specification of the algorithm, trusting that the principles will be clear.

- Mark “invariant” those statements whose operands are all either constant or have all their reaching definitions outside L.
- Repeat step (3) until at some repetition no new statements are marked “invariant”.
- Mark “invariant” all those statements not previously so marked all of whose operands either are constant, have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is a statement in L marked invariant.

Performing code motion:
Having found the invariant statements within a loop, we can apply to some of them an optimization known as code motion, in which the statements are moved to pre-header of the loop. The following three conditions ensure that code motion does not change what the program computes. Consider s: x: =y+z.
The block containing s dominates all exit nodes of the loop, where an exit of a loop is a node with a successor not in the loop.
There is no other statement in the loop that assigns to x. Again, if x is a temporary assigned only once, this condition is surely satisfied and need not be changed.
No use of x in the loop is reached by any definition of x other than s. This condition too will be satisfied, normally, if x is temporary.

ALGORITHM: Code motion.
INPUT: A loop L with ud-chaining information and dominator information.
OUTPUT: A revised version of the loop with a pre-header and some statements moved to the pre-header.
METHOD:

- Use loop-invariant computation algorithm to find loop-invariant statements.
- For each statement s defining x found in step(1), check:

i) That it is in a block that dominates all exits of L,
ii) That x is not defined elsewhere in L, and
iii) That all uses in L of x can only be reached by the definition of x in statement s.

- Move, in the order found by loop-invariant algorithm, each statement s found in (1) and meeting conditions (2i), (2ii), (2iii) , to a newly created pre-header, provided any operands of s that are defined in loop L have previously had their definition statements moved to the pre-header.

To understand why no change to what the program computes can occur, condition (2i) and (2ii) of this algorithm assure that the value of x computed at s must be the value of x after any exit block of L. When we move s to a pre-header, s will still be the definition of x that reaches the end of any exit block of L. Condition (2iii) assures that any uses of x within L did, and will continue to, use the value of x computed by s.

8. Symbolic Debugging Scheme For Optimized Code:

Symbolic debuggers are system development tools that can accelerate the validation speed of behavioral specifications by allowing a user to interact with an executing code at the source level. In response to a user query, the debugger retrieves the value of a source variable in a manner consistent with respect to the source statement where execution has halted.
Symbolic debuggers are system development tools that can accelerate the validation speed of behavioral specifications by allowing a user to interact with an executing code at the source level [Hen82]. Symbolic debugging must ensure that in response to a user inquiry, the debugger is able to retrieve and display the value of a source variable in a manner consistent with respect to a breakpoint in the source code. Code optimization techniques usually makes symbolic debugging harder. While code optimization techniques such as transformations must have the property that the optimized code is functionally equivalent to the un-optimized code, such optimization techniques may produce a different execution sequence from the source statements and alter the intermediate results. Debugging un-optimized rather than optimized code is not acceptable for several reasons, including:

- while an error in the un-optimized code is undetectable, it is detectable in the optimized code,
- optimizations may be necessary to execute a program due to hardware limitations, and
- a symbolic debugger for optimized code is often the only tool for finding errors in an optimization tool.

In a design-for-debugging (DfD) approach that enables retrieval of source values for a globally optimized behavioral specification. The goal of the DfD technique is to modify the original code in a pre-synthesis step such that every variable of the source code is controllable and observable in the optimized program.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in