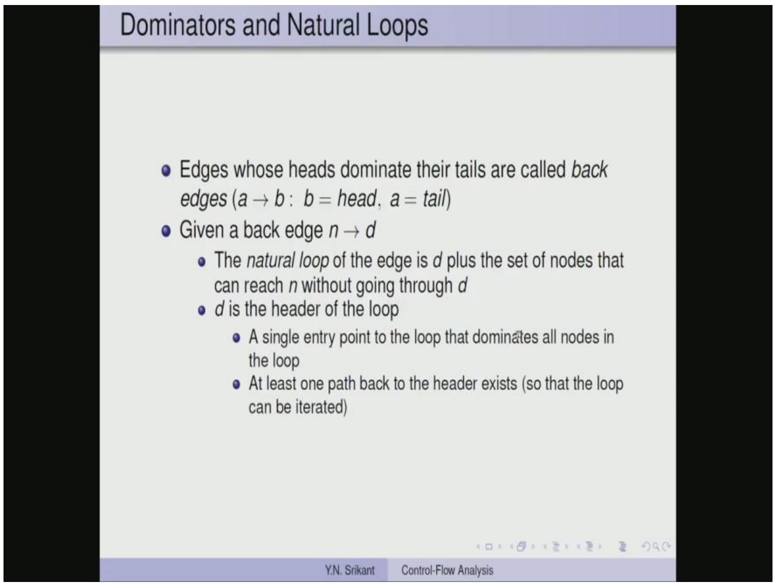
Principles of Compiler Design

Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 35
Introduction to Machine-Independent Optimizations Part – 5

Welcome to part 5 of the lecture on Machine Independent Optimizations. Today, we will continue our discussion on control flow analysis.

(Refer Slide Time 00:33)



Dominators and Natural Loops

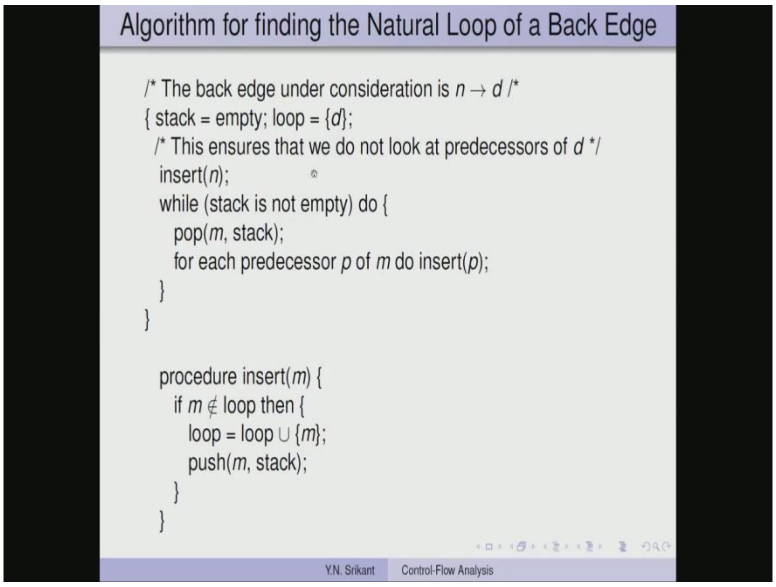
- Edges whose heads dominate their tails are called *back edges* ($a \to b$: $b = head$, $a = tail$)
- Given a back edge $n \to d$
  - The *natural loop* of the edge is d plus the set of nodes that can reach n without going through d
  - d is the header of the loop
    - A single entry point to the loop that dominates all nodes in the loop
    - At least one path back to the header exists (so that the loop can be iterated)

Y.N. Srikant Control-Flow Analysis

So, we defined dominators in the last part, so and we were discussing the natural loop structure which is defined by a back edge, so just to do a bit of recap edges whose heads dominate their tails are called back edges. So, if there is an edge from a to b, b is the head and a is the tail, so given a back edge n to d the natural loop of the edge is the node d plus the set of nodes that can reach n without going through d again.

So, the property of the header is that you know of course, the rather the head is that d is the header of the loop, so it is a single entry point to the loop that dominates all the nodes in the loop and at least one path back to the header exists, so that the loop can be iterated. Let us now consider the algorithm to find the natural loops structure based on this definition.

(Refer Slide Time 01:49)



Algorithm for finding the Natural Loop of a Back Edge

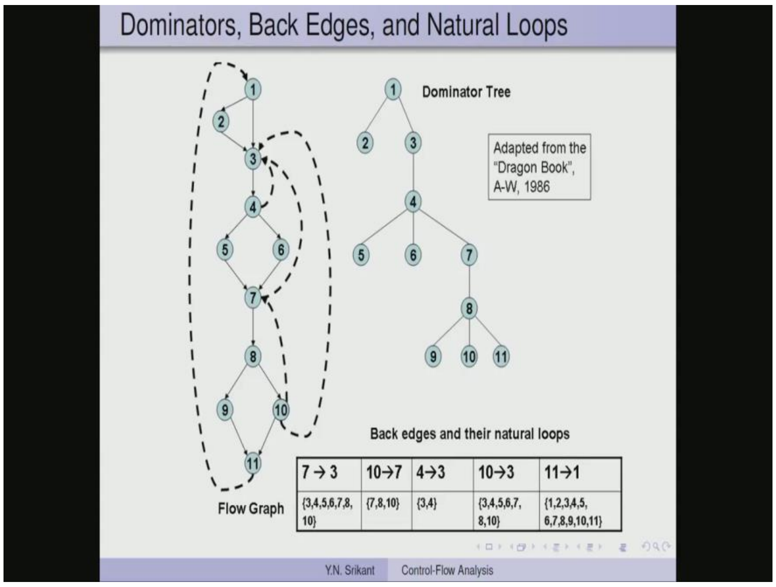
```
/* The back edge under consideration is $n \rightarrow d$ /*
{ stack = empty; loop = {$d$};
  /* This ensures that we do not look at predecessors of $d$ */
  insert($n$);
  while (stack is not empty) do {
     pop($m$, stack);
     for each predecessor $p$ of $m$ do insert($p$);
  }
}

  procedure insert($m$) {
     if $m \notin$ loop then {
        loop = loop $\cup$ {$m$};
        push($m$, stack);
     }
  }
```

Y.N. Srikant Control-Flow Analysis

This is a fairly straight forward algorithm, it uses a stack, so let us say the back edge under consideration is n to d to begin with the stack has been initialized to empty and the set of nodes in the loop has been initialized to d, because that is the header. Once we initialize the loop to d, it ensures us that we do not look at the predecessors of d which is you know we should not be doing that otherwise we would be going outside the loop. It calls the function insert n which just checks whether the node that is passed as a parameter is in the loop set.

So, if it is not in the loop set it is added to the loop set and then it is pushed on to the stack, the reason we do this is to trace the predecessors of each of the nodes, which are on the stack. So, now to begin with we have inserted n, so that has also been added to the loop and it has been pushed on the stack. So, to begin with we have only n on the stack, so while stack is not empty do pop it and for each predecessor of m do insert p, so we go on doing this until the stack becomes empty. So, let us understand this algorithm with an example.

(Refer Slide Time 03:27)



Dominators, Back Edges, and Natural Loops

Back edges and their natural loops

7 → 3	10→7	4→3	10→3	11→1
{3,4,5,6,7,8, 10}	{7,8,10}	{3,4}	{3,4,5,6,7, 8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

Y.N. Srikant Control-Flow Analysis

So, here we have many back edges, so the back edges are 7 to 4, 10 to 7, 4 to 3, 10 to 3 and 11 to 1. So, let us just take one of the back edges and understand how the loop can be computed. So, let us say we take the back edge from 7 to 4, why is it a back edge that is because 4 dominates 7, that can be easily checked here 4 dominates 7, in the algorithm we put 4 into the loop which is the header of this loop corresponding to the back edge 7 to 4.

So, there is always a back edge to which a natural loop corresponds, so we are considering tracing the loop structure of 7 to 4, so we add 4 to it and then we do an insert on 7 which is also added to the loop and it is also pushed on to the stack. So, inside the you know while loop, which keeps popping nodes from the stack we first pop 7, so then we look at the predecessors of 7, so these are the two predecessors of 7. So, let us say we add we go to 5, so 5 is not in the loop, so we add 5 to the loop structure and we also push 5 on to the stack then we look at 6 we also push 6 on to the stack and we add it to the loop.

But once we reach pop 5 and then we find that it is already added to the loop structure, so nothing is done nothing is done for 6 as well, but then from 7, 5 and 6 are not the only predecessors, we also have another predecessor which is 10. So, we actually add 10 also to the loop and it is pushed on to the stack and once it is popped its predecessor is 8, so 8 is put into the loop and again pushed on to the stack, but nothing more can be added from 8 because 7 and 8 are already in the loop.

So, we get 4, 5, 6, 7, 8, 10 as the loop structure of the back edge 7 to 4, so 4, 5, 6, 7, 8, 10, so this is the loop structure of the back edge 7 to 4. So, let us consider the back edge 10 to 7 and see what happens, so we add a 7 to the loop, we add 10 to the loop and also push it on to the stack, when we pop 10 its predecessor is 8, so we add that to the loop and push it on to the stack and for 8, there are no other nodes to be added because 8 itself has been added to the loop.

So, 7 and 8 and 10 happen to be the happen to be nodes in the loop corresponding to 10 to 7, so far it is, but the most unintuitive result is for the loop from 4 to 3. So, we start with 3 add it to the loop, then we add 4 to the loop structure and once we add 4 the only predecessor of 4 is 7 in this case. So, we add 7, 6, and 5 to the loop structure the predecessor of 7 is 10, so we add 10 and 8 also to the loop structure.

So, for this tiny you know seemingly single loop, we have added 3, 4, 5, 6, 7, 8 and 10, the reason why this is correct even though it looks unintuitive is once we start from 3, we can not only go to 4 and go back to 3, we can actually go to 4, then 5, then 7, then we can traverse 7 to 4 and then 4 to 3. Remember, that to reach the node number three we will have to traverse the back edge, if we do not do that then the this must be the only way reach 3, otherwise it is incorrect.

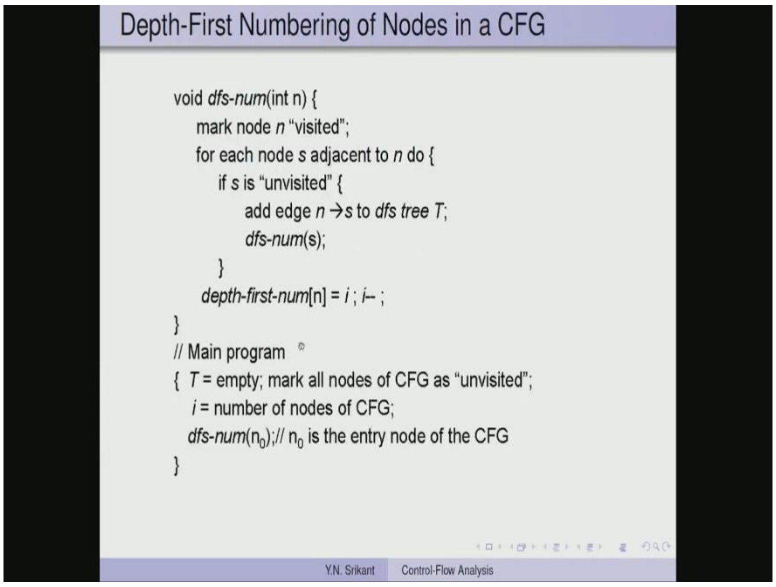
So, for example, we can therefore, go to 4, 5, 7, 8 and 10, but then we can also reach 3 via the other back edge, but that is not considered as a part of this loop, for reaching 3 for the loop corresponding to the back edge 4 to 3, we must always traverse the back edge 4 to 3. So, we go down to 10, then we go to 7, then we go to 4 and then to 3, so this is how the entire loop structure this entire thing is added to the loop structure for 4 to 3.

So, now, it is easy to see you know the loop structure for 10 to 3; obviously, we would add 3, then 10, then 8, gets added 7, gets added 5 and 6 and 4, all these get added to the loop structure automatically, because they are all and for the back edge from 11 to 1, the entire flow graph would be all the nodes in the flow graph would be added to the loop structure.

Let us change the back edge structure a little bit, so instead of the back edge from 7 to 4, we actually change it to become the back edge from 7 to 3, now the loop structure of 10 to 3 does not change, 11 to 1 does not change nor does the loop structure of 10 to 7 changes. The two loops which change are 4 to 3 and 7 to 3, 7 to 3, now we add three then we add 7, we add 5, we add 6, then we add 10, 8, so this is going to be our loop structure so for.

Of course, we also add 4, so 3, 4, 5, 6, 7, 8, 10 becomes the loop structure for 7 to 3, but for the back edge 4 to 3 there is a drastic cut, so we add 3, then we add 4 to the loop, there are no other you know predecessors of 4 apart from 3, so the loop structure is just this particular small loop 4 to 3, 3 to 4 and 4 to 3. So, by the back edge you know when we change the loop structure, we have changing we are changing the back edge and once we change the back edge the number of nodes in the loop also get changed. So, this is the implication of the back edge on the loop structure.

(Refer Slide Time 10:47)



Depth-First Numbering of Nodes in a CFG

```
void dfs-num(int n) {
    mark node n "visited";
    for each node s adjacent to n do {
        if s is "unvisited" {
            add edge n → s to dfs tree T;
            dfs-num(s);
        }
    depth-first-num[n] = i ; i-- ;
}
// Main program
{  T = empty; mark all nodes of CFG as "unvisited";
   i = number of nodes of CFG;
   dfs-num(n_0);// n_0 is the entry node of the CFG
}
```

Y.N. Srikant   Control-Flow Analysis

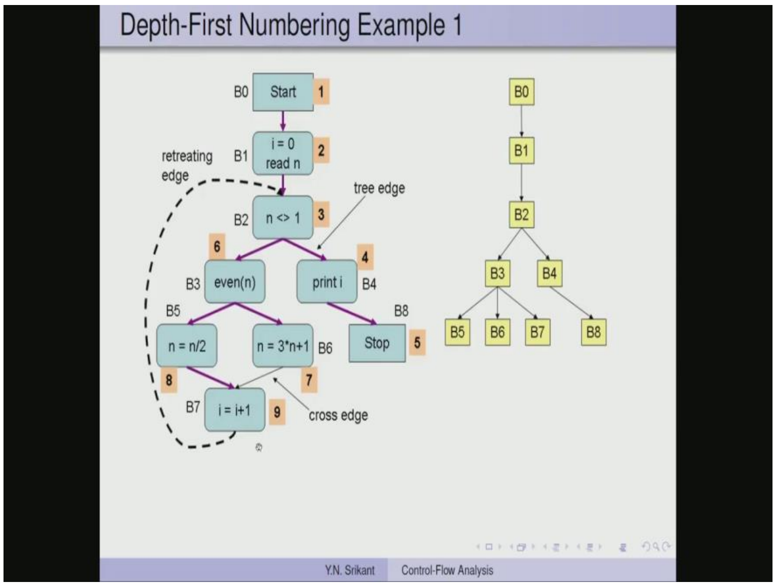
So, the next you know concept that we need to learn is the depth first numbering of nodes in a control flow graph why is this important, in the case of data flow analysis we know that we can visit the nodes of the control flow graph in any order, but I also mentioned that you know visiting the nodes in the depth first search order actually gives us fewer number of iterations compared to any other visit order.

So, to understand the d f s order, let us actually understand how to do the d f s numbering itself, we are really going back to the data structure course. So, this is a depth first search on the graph just that the numbering of the nodes is going to be slightly different. So, d f s num takes the node as a parameter, then it marks the node n as visited, as in any depth at first search.

So, for the for each node s adjacent to n, so is exactly the same as in the depth first search, if s is unvisited then we add the edge n to s to the depth first search tree t, so we are constructing the d f s tree and then we call d f s number. So, remember that we have not yet numbered the nodes, so after all the nodes adjacent to n have been visited, we come out of this loop.

And then number depth first num n as i to begin with i has been initialized to the number of nodes of the control flow graph and then once we do this we decrement i. So, this is the way it gets numbered now a node gets numbered after all its dependents, descendants or dependents get numbered the initial call on d f s num is through the entry node of the control flow graph.

(Refer Slide Time 13:05)



Depth-First Numbering Example 1

Y.N. Srikant Control-Flow Analysis

So, let us understand this algorithm with an example, so here is the same simple control flow graph that we had before, so we start our depth first search numbering from the block b 0. So, the counter has been initialized to 9, because we have 8, so that is because if we start the numbering from 1, then we have 9 nodes, if we start the numbering from

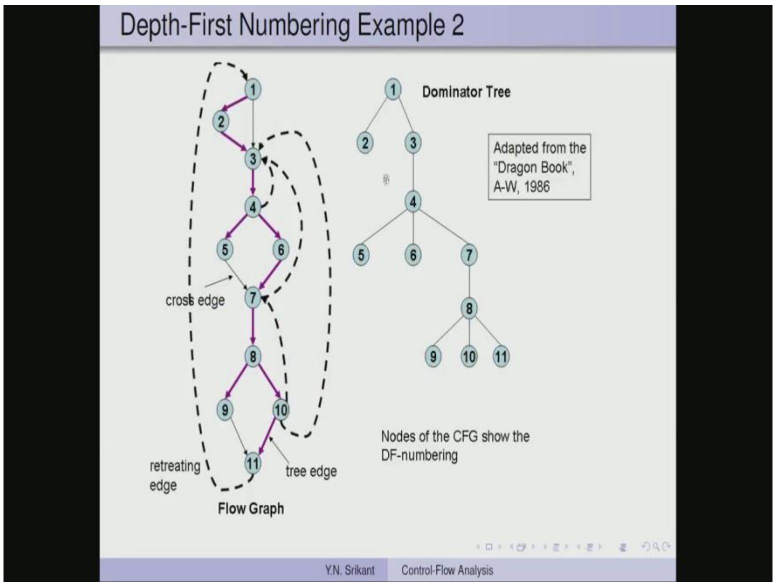
0, then we have you know 0 to 8 again 9 nodes, but the numbering is going to be different.

So, let us assume that the numbering happens from 1, so we have initialized the counter to nine this is the number of nodes in the control flow graph, so we start our depth first search from b 0; its only neighbor is b 1, so this is visited. The next neighbor is b 2, that will also be visited then we visit you know say b 3 and then we visit b 5 and then we visit b 9, there are no more in the you know adjacent nodes for 9, everything has been taken care of.

So, because this is the next node which is adjacent, but we have already visited it, so node b 7, gets the number 9, then we return to b 5, that would get its we have exhausted the adjacency list of b 5, so it gets the number 8, then we go to node number b 3. Now, there is one more neighbor to be visited that is node number b 6, so we visit that and then we find that all its neighbors that is only this has been visited already, so we number this as 7 and then we return to b 3, that now has exhausted all its neighbors, so it gets the number 6.

Then, we return to b 2, we still have to visit b 4 and b 8, before we actually number b 2, so we go to b 4, then we go to b 8, so this has no neighbor. So, this gets the number 5, b 4 gets the number 4, then we return to b 2 which gets the number 3, then b 1 gets 2 and b 0 gets one, so this is the ordering in the depth first search numbering scheme.

(Refer Slide Time 15:47)

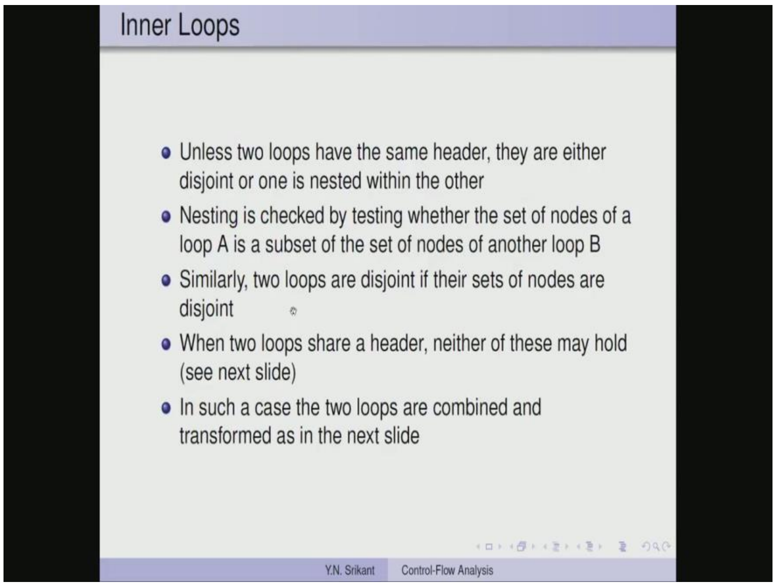

Depth-First Numbering Example 2
cross edge
retreating edge
tree edge
Flow Graph
Dominator Tree
Adapted from the "Dragon Book", A-W, 1986
Nodes of the CFG show the DF-numbering
Y.N. Srikant
Control-Flow Analysis

If, we apply the same algorithm to this example, we start with the node number one. So, the numbering actually that already shows the depth first search numbering, but it is possible to number the nodes in a slightly different order also. So, in this example suppose we had chosen to from here after visiting 3 instead of going this way, if we had chosen this path we would have come to b 4 and then b 8.

So, this node would have been numbered 9, then this would have been numbered 8 and then we would have gone further to this part of the node this part of the graph. So, the order in which we visit the nodes will also change the depth first search numbering a same is true here, so we start here then we go to 2, then we go to 3, 4, then we visit say 6, then we visit 7, 8, then we visit 10 and 11 no more.

So, this gets the number 11, this gets number 10, we go back and then visit this gets 9, this gets 8, this gets 7, then we go back to 6 and then this will be this will cannot be numbered right now, so visit this give it 5 and then 4, then 3, then 2 and 1, whatever has been marked as marked in purple it corresponds to the d f s tree, it is a spanning tree. So, these are the tree edges, then the back edges are also called as retreating edges and whatever is neither a tree edge nor a cross you know would be cause called as a cross edge, so this is a cross edge here.

(Refer Slide Time 17:48)



Inner Loops

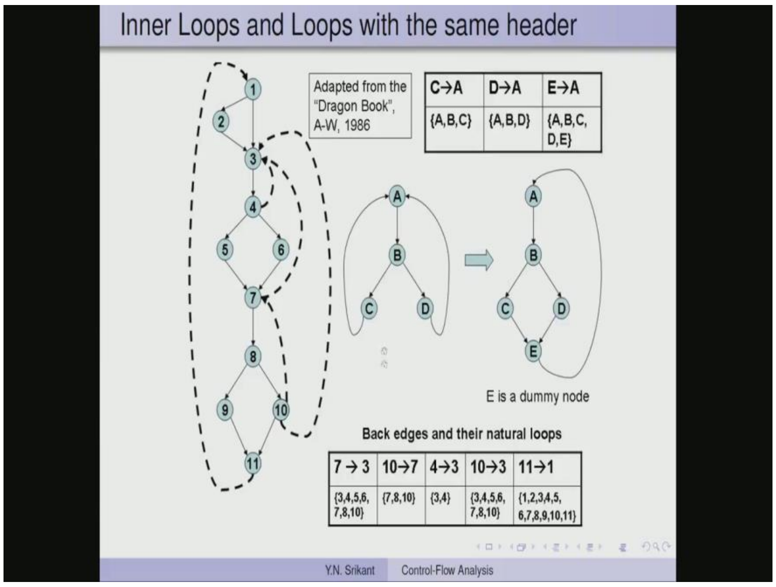
- Unless two loops have the same header, they are either disjoint or one is nested within the other
- Nesting is checked by testing whether the set of nodes of a loop A is a subset of the set of nodes of another loop B
- Similarly, two loops are disjoint if their sets of nodes are disjoint
- When two loops share a header, neither of these may hold (see next slide)
- In such a case the two loops are combined and transformed as in the next slide

Y.N. Srikant Control-Flow Analysis

Now, what exactly is an inner loop, so let us understand the inner loop there are unless two loops have the same header, they are either disjoint or one is nested within the other, so this is a simple property. So, if, but if they have the same header then they need to be neither nested you know nor disjoint I will give you an example of this, but otherwise assuming that this is not so we check whether the loops you know, whether the nesting can be checked very simply, by testing whether the nodes of a loop a or is A, subset of the nodes of another loop B, so if it is a subset then there is nesting.

Similarly, if the loops have no common nodes then the loops are disjoint, but when the loops share a header neither of these may hold, I will show you an example of this, and if this happens loops share a header and neither nesting nor disjointness is a property, we will have to actually combine the loops and transform it.

(Refer Slide Time 19:15)



Inner Loops and Loops with the same header

Adapted from the "Dragon Book", A-W, 1986

C→A	D→A	E→A
{A,B,C}	{A,B,D}	{A,B,C, D,E}

E is a dummy node

Back edges and their natural loops

7→3	10→7	4→3	10→3	11→1
{3,4,5,6, 7,8,10}	{7,8,10}	{3,4}	{3,4,5,6, 7,8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

Y.N. Srikant Control-Flow Analysis

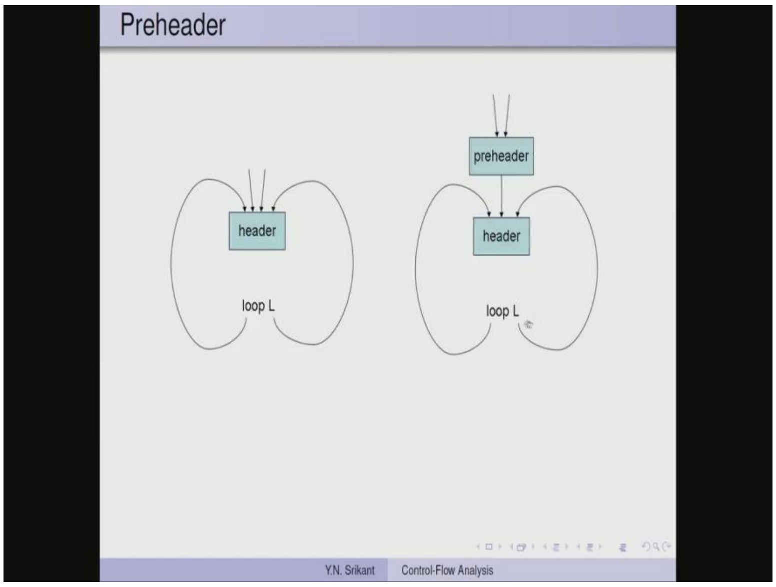
So, let us take this a this big example, so we have many loops here 7 to 3, 10 to 7, 4 to 3, 10 to 3 and 11 to 1, so if you look at it 7, 8, 10 is a subset of this set, the loop structure of 7 to 3, so 10 to 7 is nested within 7 to 3. Even, though it does not appear so this is 10 to 7 and this is 7 to 3, so even though it does not appear so it is indeed nested, similarly is 7, 8, 10 is nested in 10 to 3 and 11 to 1, as well that is you know visible.

Then 4, 3 is nested in 7, 3 it is nested in 10, 3 and it is nested in 11, 1 as well, now it is also correct that 4 to 3 and 10 to 7 are disjoint loops, so these two have no nodes in common, so they are disjoint loops. Suppose, we had a structure such as this so here for example, we have a loop which is this, there is another loop which is this, so for in the loop structure, the loop corresponding to the back edge C to A is A, B, C and the loop corresponding to D to A is A, B, D you cannot say that one loop is nested in the other.

And you cannot say that the two are disjoint because they still share a node, in such a case we actually want to transform this to this type of a control flow graph by adding a dummy edge. So, once we add a dummy edge we see that and we can shift the two back edges to rather combine the two back edges into one and make it edge from E to A, so once we do that the for the back edge E to A, the entire loop becomes a single loop.

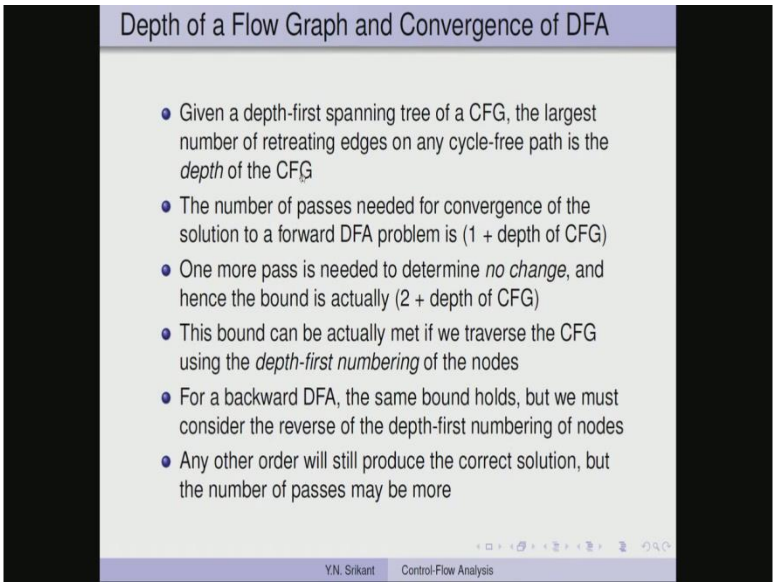
So, here also you know there are difficulties of that kind, for example 10 to 3, of course, there is no structure of this kind which appears, but nesting and disjointness definitely happen to be the case. So, this type of sharing headers but neither being disjoint nor being nested is not true in this case.

(Refer Slide Time 22:01)


Preheader

We also need to understand the concept of a preheader, this will be required the algorithm for loop invariant code motion suppose, we have a loop structure with a header they and there are many paths coming into this header it is usually convenient to make the header get just one input by actually separating all other inputs to go into a preheader. So, the semantics of this and this are the same it is just that the loop has now become very clean, there is only one input from the outside and the rest of the edges are all only the back edges, so whereas, here there are number of inputs coming from outside.

(Refer Slide Time 22:44)



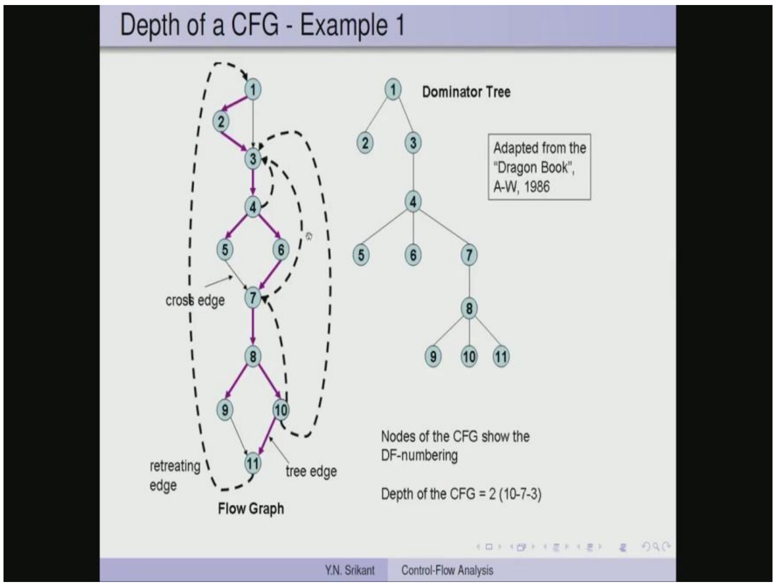
Depth of a Flow Graph and Convergence of DFA

- Given a depth-first spanning tree of a CFG, the largest number of retreating edges on any cycle-free path is the *depth* of the CFG
- The number of passes needed for convergence of the solution to a forward DFA problem is (1 + depth of CFG)
- One more pass is needed to determine *no change*, and hence the bound is actually (2 + depth of CFG)
- This bound can be actually met if we traverse the CFG using the *depth-first numbering* of the nodes
- For a backward DFA, the same bound holds, but we must consider the reverse of the depth-first numbering of nodes
- Any other order will still produce the correct solution, but the number of passes may be more

Y.N. Srikant Control-Flow Analysis

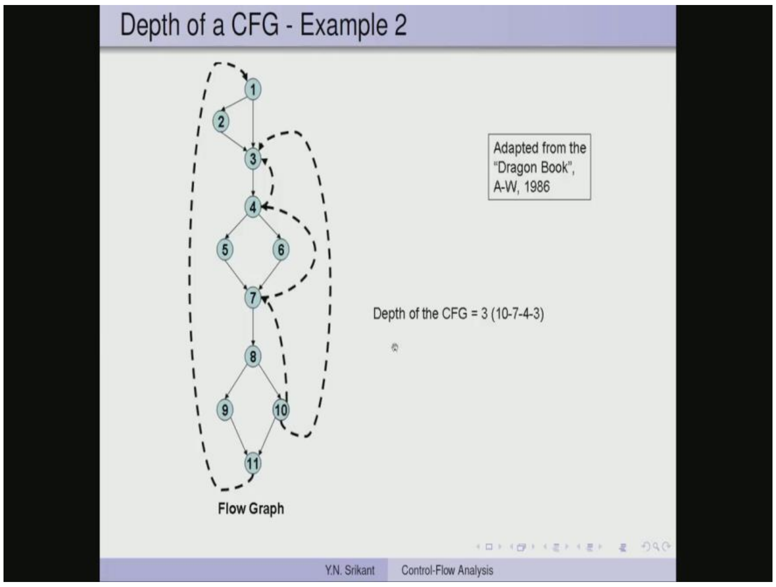
The next thing which is very important is to understand the convergence of a data flow algorithm, I promise this in the last part of our lecture. So, what can we say about the number of iterations that the data flow algorithm iterative data flow algorithm takes, so let us say we are given a depth first spanning tree of a control flow graph. So, we know how to construct the depth first tree, we do the depth first numbering automatically we get the depth first tree. The largest number of retreating edges on any cycle free path in the spanning tree is the depth of the control flow graph, so let us understand that and then continue.

(Refer Slide Time 23:33)


Depth of a CFG - Example 1
1
2
3
4
5
6
7
8
9
10
11
cross edge
retreating edge
tree edge
Flow Graph
Dominator Tree
Adapted from the "Dragon Book", A-W, 1986
Nodes of the CFG show the DF-numbering
Depth of the CFG = 2 (10-7-3)
Y.N. Srikant
Control-Flow Analysis

So, here the depth of the control flow graph is 2 that is because when we take the spanning tree which is shown in the picture and we consider the sequence of back edges 10 to 7, and then 7 to 3, this is the maximum number of back edges which can be traversed in the tree. All others would be just this is just one and this is again just one, so whereas, here we have two back edges which can be traversed, so this is the depth of the control flow graph in this case.

(Refer Slide Time 24:17)



Depth of a CFG - Example 2

Adapted from the "Dragon Book", A-W, 1986

Depth of the CFG = 3 (10-7-4-3)

Y.N. Srikant Control-Flow Analysis

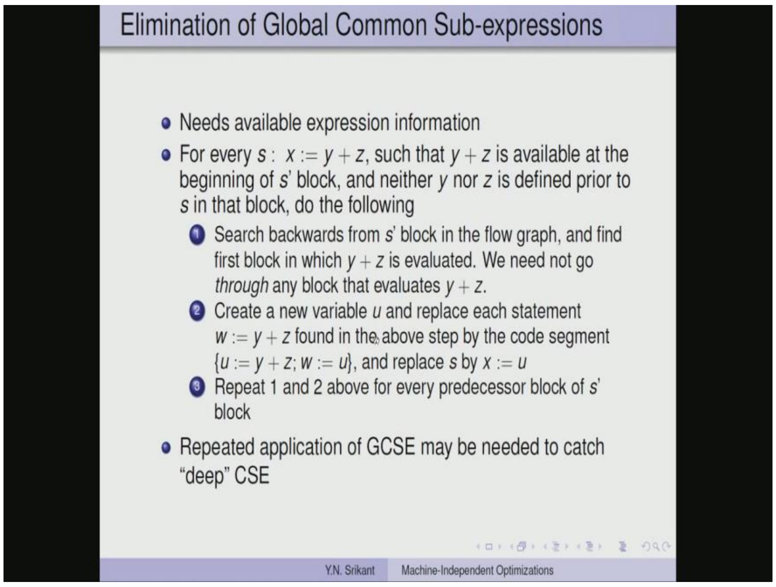
Whereas, in this case the control flow graph has a depth of 3, so the reason is we go from 10 to 7, then we go from 7 to 4 and then from 4 to 3, so the depth of this control flow graph is 3, what has the depth of the control flow graph got to do with the convergence of the algorithm. It is here, the number of passes needed to for the convergence of the solution to a forward data flow analysis problem is 1 plus the depth of the control flow graph, so this is the basic result.

So, once we know the depth of the control flow graph, we can say that this is the number of iterations needed for the convergence and this convergence; this bound can actually be achieved, if we traverse the control flow graph using the depth first numbering of the nodes. Any other order may actually take a few more iterations, one more pass is needed to determine no change and actually therefore, the bound becomes 2 plus depth of CFG and for a backward data flow analysis problem the same bound holds, but we need to reverse the depth first search you know rather we need to consider the reverse of the depth first numbering of the nodes.

So, instead of doing the traversal in the depth first search order, we do it in the reverse order, any other order will still produce the correct solution, but the number of passes may be more than what is actually predicted. So, that is about some fundamentals regarding control flow analysis, so now we are ready to discuss the algorithms for machine independent optimizations.

So, we are going to consider a few optimizations which are very common the first one; obviously, would be the global common sub expression elimination and when we do this we will also need copy propagation and constant propagation is a very simple algorithm. So, we will look at that also and loop invariant code motion is something very different which is an application of the dominator relationship.

(Refer Slide Time 26:52)



Elimination of Global Common Sub-expressions

- Needs available expression information
- For every s : $x := y + z$, such that $y + z$ is available at the beginning of s' block, and neither y nor z is defined prior to s in that block, do the following
  1. Search backwards from s' block in the flow graph, and find first block in which $y + z$ is evaluated. We need not go *through* any block that evaluates $y + z$.
  2. Create a new variable u and replace each statement $w := y + z$ found in the above step by the code segment $\{u := y + z; w := u\}$, and replace s by $x := u$
  3. Repeat 1 and 2 above for every predecessor block of s' block
- Repeated application of GCSE may be needed to catch "deep" CSE

Y.N. Srikant Machine-Independent Optimizations

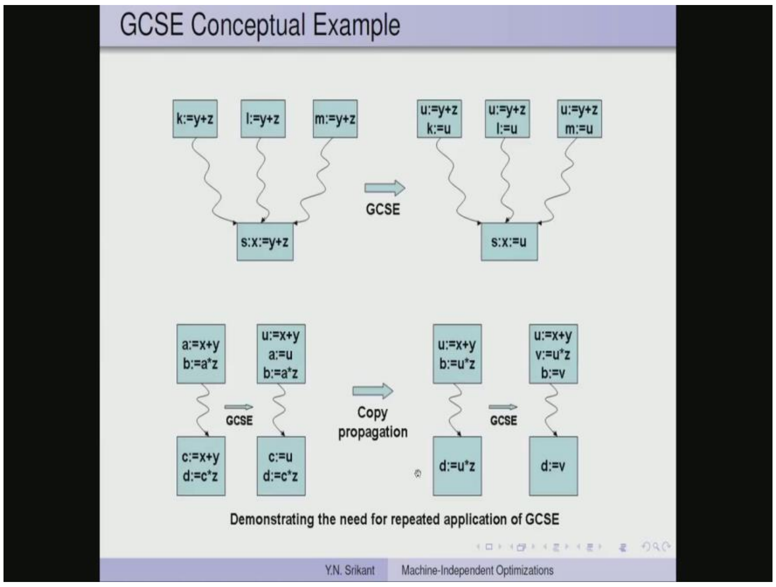
So, we have already seen global common elimination a few times, but we did not discuss the algorithm formally, so let us do that first, it needs available expression information, so we know how to compute the available expressions you by data flow analysis. So, that would be a forward flow problem with the, you know confluence operator being intersection.

So, basically for every statement S which is of the form X equal to Y plus Z, such that Y plus Z is available at the beginning of S s block, so the statement S is in a particular block, so the you know rather Y plus this is in a particular block and we want to make sure that Y plus Z is available at the entry of this particular block. Only, then we can actually do some replacement and we must also make sure that neither Y nor Z is defined prior to S in that particular block.

So, S this is defined, but before that we should not have any definitions of either Y or Z otherwise Y plus Z which is reaching the entry point of this block you know will not be useful within the block and we cannot do any elimination of the common sub expression because Y and Z have changed their value. So, in such a case if these conditions are met we search backwards from S s block in the control flow graph and find the first block in which Y plus Z is evaluated, so we actually have to do this for all the paths which go backwards from S.

So, we need not go through any block that evaluates Y plus Z, we just have to go until Y plus Z is evaluated that is it and that makes sure that Y plus Z is available, so availability is already satisfied now, we are finding the blocks in which Y plus Z is evaluated. So, once we find these blocks, so we are actually going to replace you know this rather, we are going to create a new variable u and replace each statement W equal to Y plus Z by u equal to Y plus Z and W equal to u, we saw this already this is just a formal statement of that and then we replace S by X equal to u.

(Refer Slide Time 30:00)

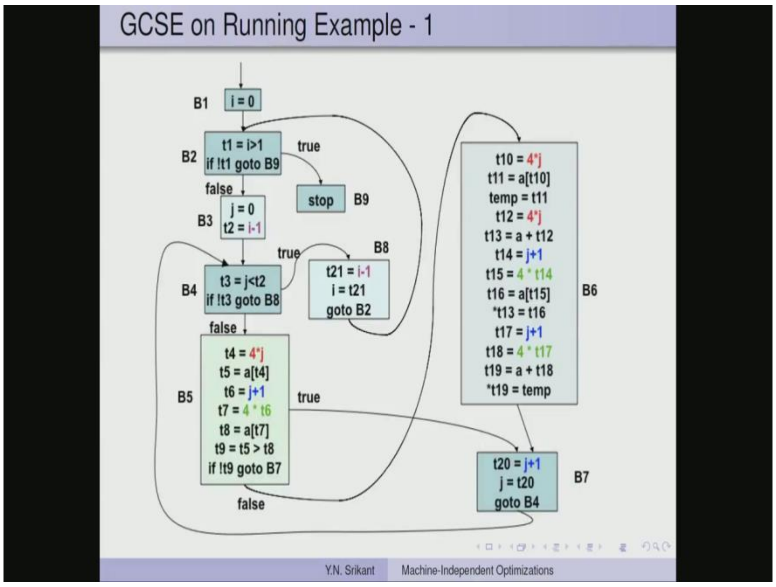


So, this we repeat for every predecessor of the block S and we have already seen that repeated application of GCSE may be required to catch deep common sub expression; they perform deep common sub expression elimination. So, let me again show you the same example, so here is Y plus Z, assume that Y plus Z is available at the entry point of this basic block, so these are the three paths emerging out of S.

So, along each path we go up to the block which evaluates Y plus Z here, here and here, then create the same temporary u equal to u and then insert u equal to Y plus Z, u equal to Y plus Z and u equal to Y plus Z and of course, k equal to u, l equal to u and m equal to u, so now this statement can be replaced by x equal to u. So, we have eliminated the common sub-expression here you know this is it is evaluated only once here and then reused.

So, what we mean by a deep common sub-expression is one which surfaces only after one round of GCSE and one round of copy propagation, so we have seen this example before, so briefly let us see what it does, so we have x plus y and x plus y here. So, first round of GCSE makes this u equal to X plus Y and this as c equal to u, then a round of copy propagation actually you know makes u star Z visible as a common sub-expression. So, we can eliminate that also by the same algorithm and thereby improve the efficiency of the code.

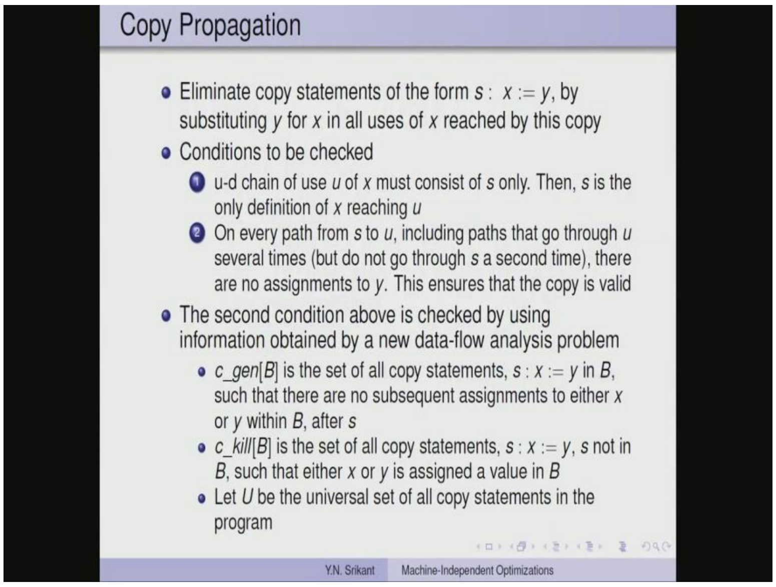
(Refer Slide Time 31:51)



So, this is what we mean by a deep CSE, deep CSE as ACS rather common sub-expression and deep common sub-expressions, actually surface only after one round of GCSE and copy of propagation. On the running example, there are many places where GCSE is possible, so here is i minus 1 and here we have 4 star j and then we have you know this 4 star t 6 which will surface later.

So, 4 star j has been eliminated you know and then we have other expressions which are possible, so j plus 1 is possible here and so on and so forth. Then so after 1 round of GCSE, we get this we have still not studied copy propagation defer applying copy propagation to the stage after studying this algorithm for copy propagation.

(Refer Slide Time 32:42)



Copy Propagation

- Eliminate copy statements of the form $s: x := y$, by substituting y for x in all uses of x reached by this copy
- Conditions to be checked
  1. u-d chain of use u of x must consist of s only. Then, s is the only definition of x reaching u
  2. On every path from s to u, including paths that go through u several times (but do not go through s a second time), there are no assignments to y. This ensures that the copy is valid
- The second condition above is checked by using information obtained by a new data-flow analysis problem
  - *c_gen*[*B*] is the set of all copy statements, $s : x := y$ in B, such that there are no subsequent assignments to either x or y within B, after s
  - *c_kill*[*B*] is the set of all copy statements, $s : x := y$, s not in B, such that either x or y is assigned a value in B
  - Let U be the universal set of all copy statements in the program

Y.N. Srikant   Machine-Independent Optimizations

So, the next algorithm that we want to understand is the copy propagation algorithm, this is a very important algorithm and it is nontrivial to solve, the purpose of copy propagation is to eliminate copy statements of the form X equal to Y. So, what is this copy propagation we have a large number of uses of X, lets say neither X nor Y change in before, we reach the usage of X, so in such a case we can replace all these uses of X by the variable Y.

So, thereby we have actually you know eliminated this copy statement, if X is not used later after all these copy propagation has taken place, then we can eliminate the copy statement X equal to Y, to do this there are two major conditions which need to be checked. The first major condition is the use definition chain of use u of X must consist of S only, so here in other words S is the only definition of X reaching u. So, there is a usage of X, so this copy must be the only definition of X reaching u, if there is one more then; obviously, we do not know which value is valid at u.

So, we cannot actually replace X by this Y whereas, if this is the only copy which is reaching this u, then we can replace the use of X here by Y, this is fairly straight forward to do we can use the use definition chain of that is that can be constructed using the reaching definitions. The second condition is more complex on every path from S to u including paths that go through u several times, so cycles are for u, but they do not go through S, a second time.

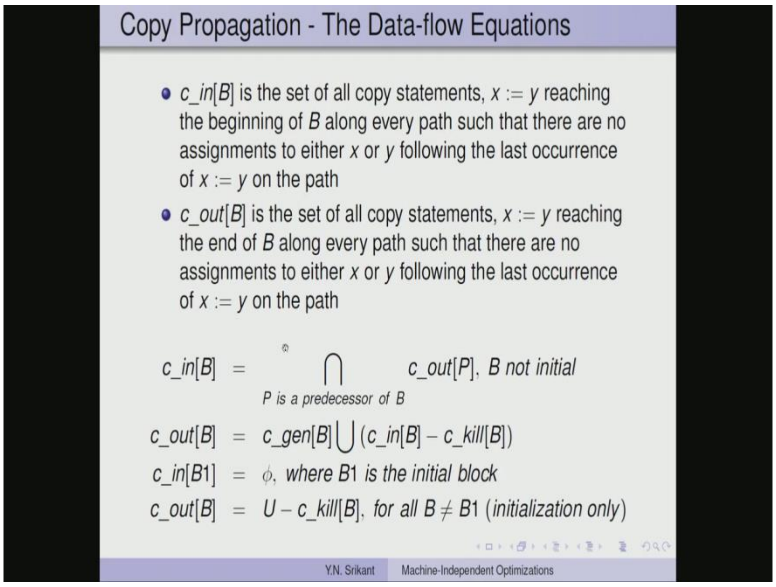
So, definitions cannot be gone through a second time, there are no assignments to Y, so if there is an assignment to Y then the value of Y changes, so we cannot reuse it this ensures that the copy is valid. So, again this is the reason why we cannot go through the definition a second time, but we can go through the usages any number of times, that is not an issue and the value of Y must not change, the value of X should also not change, but that will be automatically taken care of when check this.

So, we will see that now to check the second condition is nontrivial, so we need to formulate a new data flow problem, so let us, formulate the problem of reaching copies as we can call it. So, this problem we again define c gen and c kill and c gen is the set of all copy statements X equal to Y in B, such that there are no subsequent assignments to either X or Y within B after the statement S.

So, this is generation of copies which copies reach the end of the basic block B, so this is very trivial almost if a copy X equal to Y has to reach the end of the basic block then neither X nor Y must be modified within the block after this copy. So, what is the c kill set, it is a set of all copy statements X equal to Y and S is not in B, so very similar to the reaching definitions and available expressions problem.

So, there is a you know an assignment to either X or Y assigned a value in B, so an assignment to either X or Y not necessarily copy just assignment to either X or Y, so this X or Y assignment kills the copy involving X or Y. So, if there is a copy involving X and Y, then assignment to either X or Y in the basic block B will be killed. So, we are considering the copy statements all over the program, but the assignments within the basic block. This is what we did in the reaching definitions and available expressions problem as well, let u be the universal set of copy statements in the program.

(Refer Slide Time 37:55)



Copy Propagation - The Data-flow Equations

- $c\_in[B]$ is the set of all copy statements, $x := y$ reaching the beginning of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path
- $c\_out[B]$ is the set of all copy statements, $x := y$ reaching the end of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path

$$\begin{aligned} c\_in[B] &= \bigcap_{P \text{ is a predecessor of } B} c\_out[P],\ B \text{ not initial} \\ c\_out[B] &= c\_gen[B] \bigcup (c\_in[B] - c\_kill[B]) \\ c\_in[B1] &= \phi,\ \text{where } B1 \text{ is the initial block} \\ c\_out[B] &= U - c\_kill[B],\ \text{for all } B \neq B1 \text{ (initialization only)} \end{aligned}$$

Y.N. Srikant Machine-Independent Optimizations

So, what is the c in of a basic block, c in of a basic block is a set of all copy statements X equal to Y, reaching the beginning of the basic block along every path such that there are no assignments to either X or Y following the last occurrence of X equal to Y on the path. So, this is something intuitive we just want to make sure that when a copy statement reaches the beginning of the block either X or Y have not been assigned a value on that path.

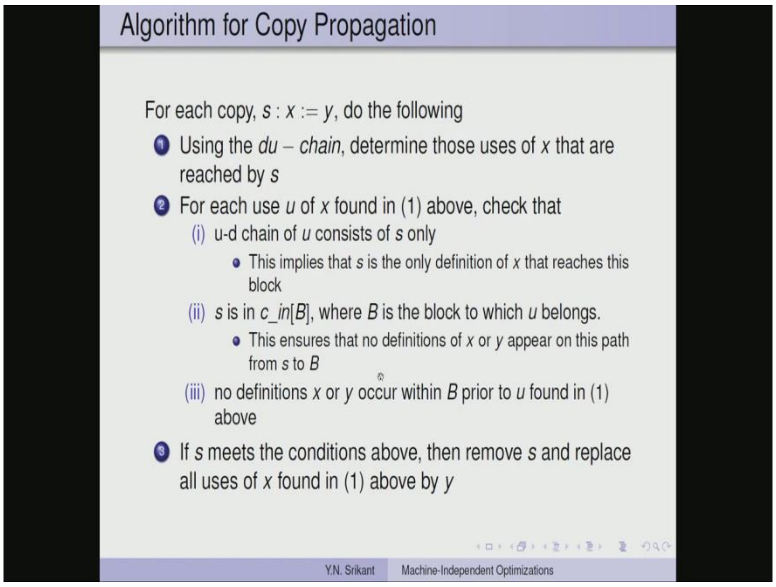
So, the same is true for c out, so it is the set of all copy statements X equal to Y reaching the end of the basic block, along every path such that there are no assignments to either X or Y following the last occurrence of X equal to Y on the path. So, after the copies statement we should have no modification of either X or Y, so that is the meaning of both c in and c out, here are the data flow equations for computing the reaching copies.

So, just like the available expressions problem the confluence operator is intersection and this is a forward flow problem, because out is being computed in terms of in and because of the confluence being intersection. We have the same initialization as in the available expressions problem, c in of B 1 is phi permanently and c out of B is u minus c kill or B not equal to B 1, so initialization is using the universal set. So, we could have also have set c in of b equal to u, so for b not equal to B 1, but this is also fine.

So, now c out of B is simple, so take all the copies which are generated in the block and then the copies which come in to the block through at the entry point remove what is killed in the block. So, that is what c out B is this is very intuitive just like the reaching definitions and available expressions problem and in the case of c in, it is an intersection of all the copies of the predecessors.

So, again this is intuitive because the same copy must reach the input point via all the predecessors of the basic block B, so I will show you in the example that mere format of the statement being X equal to Y will not make the copy the same, it may be a different copy, so let us see how to actually do the copy propagation.

(Refer Slide Time 40:54)



Algorithm for Copy Propagation

For each copy, $s : x := y$, do the following

1. Using the *du – chain*, determine those uses of x that are reached by s
2. For each use u of x found in (1) above, check that
   - (i) u-d chain of u consists of s only
     - This implies that s is the only definition of x that reaches this block
   - (ii) s is in $c\_in[B]$, where B is the block to which u belongs.
     - This ensures that no definitions of x or y appear on this path from s to B
   - (iii) no definitions x or y occur within B prior to u found in (1) above
3. If s meets the conditions above, then remove s and replace all uses of x found in (1) above by y

Y.N. Srikant Machine-Independent Optimizations

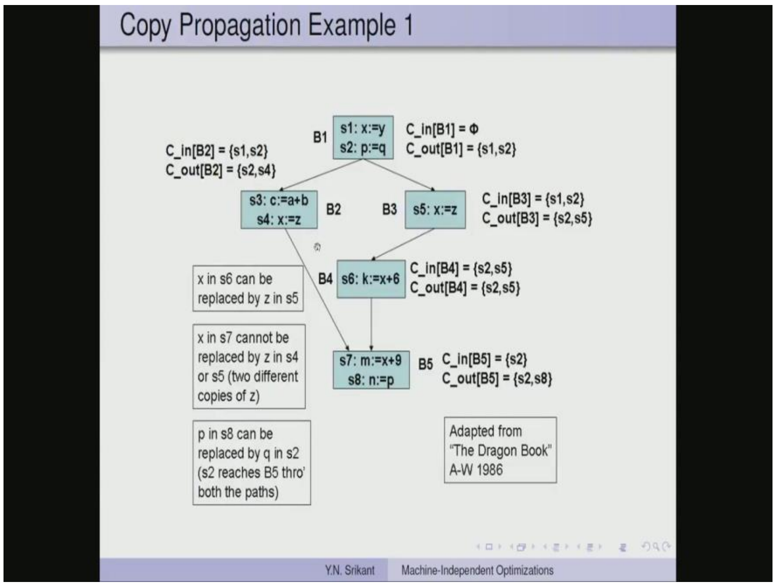
So, we have computed the reaching copies for each of the basic blocks, so for each copy statement X equal to Y, we use the definition use chain which is nothing, but a link list of you know linking a definition to all its uses. So, determine those uses of X that are reached by S, so to compute that d u chain, it is a different problem very similar to the live variable analysis and that is left as an exercise.

So, using the du chain determine those uses of X that are reached by S, so this is just to make the you know checks easy that is all, so we look at all the uses of X, these are the potential places where replacement by y can be made. Then for each use u of X found in 1, we need to check several conditions u d chain of u consists of S only, so this implies that S is the only definition of X that reaches the block. So, I already mentioned this if there is more than one definition reaching the use u then we are not sure whether it is the copy or the other definition.

So, we cannot get rid of you know X in this usage u, S is in the c in of the basic block where B is the block to which u belongs, the usage u, so this make sure that no definitions of X or Y appear on this path from S to B, so this was taken care of in the definition of in itself. So, that is why this is a restatement of a repeated statement the same property no definitions of X or Y occur within be prior to u found in 1, so this is within the basic block this is across the basic block, so both have to be made sure off.

If S meets the conditions above then remove S, replace all uses of X found in 1, above by Y, so we have to be very careful here we must make sure that every usage of X satisfies these properties, even if one of them does not then we cannot get rid of this copy. Only, when all usages of X satisfy these properties we can replace those usages by Y otherwise we cannot get rid of the copy statement.

(Refer Slide Time 43:50)



Copy Propagation Example 1

x in s6 can be replaced by z in s5

x in s7 cannot be replaced by z in s4 or s5 (two different copies of z)

p in s8 can be replaced by q in s2 (s2 reaches B5 thro' both the paths)

Adapted from "The Dragon Book" A-W 1986

Y.N. Srikant Machine-Independent Optimizations

Here is an example of copy propagation adapted from the book, so we have S 1 which is a copy statement X equal to Y, S 2 is another copy statement p equal to q, S 3 is not a copy statement, S 4 is a copy statement X equal to Z, S 5 is a copy statement X equal to Z. So, here I want to emphasize that that copy X equal to Z in the block B 2 and the copy X equal to Z in the block B 3 are two different copies; they are not the same even though their form is similar.

The assignment to X happens in both, the right side is the same, but these two copies are not the same, so once we have two different statements, then the two are different copies.

Unfortunately, the copy propagation algorithm is not powerful enough to capture, the effect of these two copies being the same. Even though in this case they are indeed the same our algorithm cannot capture this effect, then S this S 6 is not a copy statement, S 7 is also not a copy statement, but S 8 is.

So, to compute the gen and kill I have not shown it here, but it is very trivial, so here X equal to Y and p equal to q are both generated by this basic block and when we consider the kill look at X equal to Y, so all the copies involving X are killed by this copy, so this is killed, this is killed, so and that is it these two are killed. So, that is similarly for others as well.

So, we will not worry about the gen and kill because it is very simple to compute them lets understand what the structure of c in and c out are, so this basic block has no incoming statements. So, c in of B 1 is always phi, in the case of c out; obviously, S 1 and S 2 reach this point, so they are included in the outset. For B 2 c in is nothing but the outset of B 1, so that would be S 1, S 2 and c out will we actually have X equal to Z.

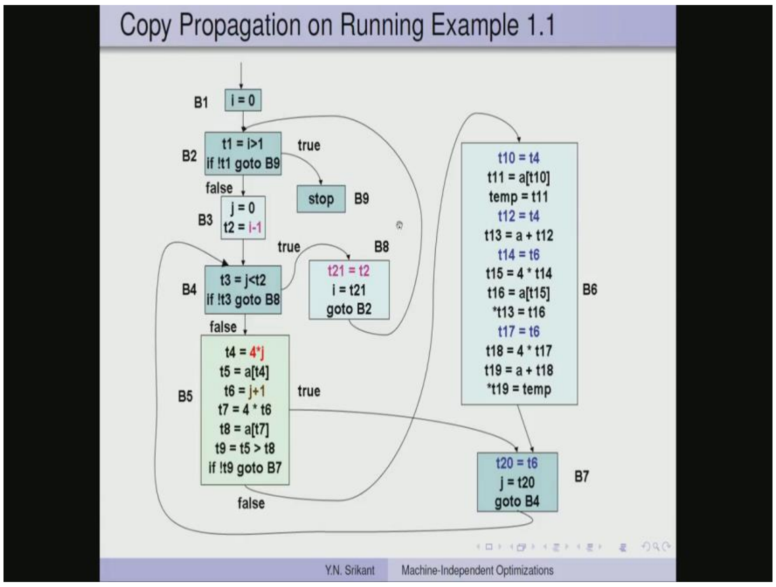
So, this is a copy which goes out, so S 2, S 4 is included and since S 4 is assigning to X, the this copy X equal Y would be killed, so that is removed from this set. So, we have S 2and S 4; obviously, statements which are not copies are not included this set for this block again c in would be the out of this. So, that is S 1, S 2 and out of this block would be; obviously, S 5 is included and it is assigning to X, so this S 1 would be removed, so S 2 and S 5.

So, here you can see that S 4 here and S 5 here are being maintained as two different copies in this the particular you know block b 4 which has S 6, there is a usage of X here. So, since our copy which reaches from here is nothing but X equal to Z and all the conditions of the copy propagation are satisfied here X and Z are not modified here they are not modified here. So, this X has only this particular copy as the reaching rather the defining occurrence.

So, this X can be replaced by this Z, so this can really become Z plus 6, absolutely no problem over that, when we come here this has another X unfortunately, the X which comes from here is S 5 and the X which comes from here is S 4, even though these two are really the same copies, they are being you know represented as different copies. So, since there are two definitions or two copies reaching this usage of X we cannot actually replace it by Z.

Now, there is n equal to p here, p has a you know definition in S 2 and that reaches along this path and as well with no modifications to either p or q, so we can replace n equal to p, the p in n equal to p by q and this would essentially become n equal to q. So, this is how the copy propagation algorithm works and let me stress again that even though syntactically these two copies are the same, they are being treated differently by the algorithm.

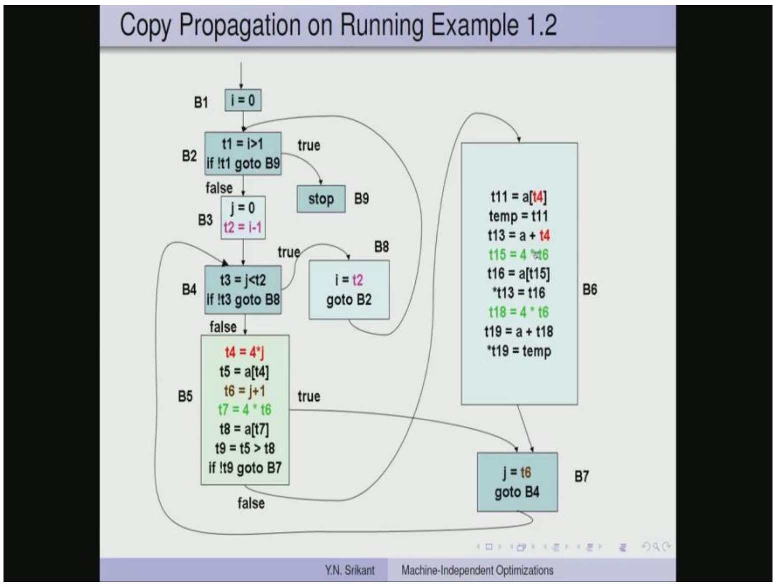
(Refer Slide Time 49:31)



Copy Propagation on Running Example 1.1

On our running example there are many copies here, so for example, there is t 10 which can be replaced by t 4 and there is t 12 here which can be replace by t 4 and then there is t 14 here which can be replace by t 6, etcetera.

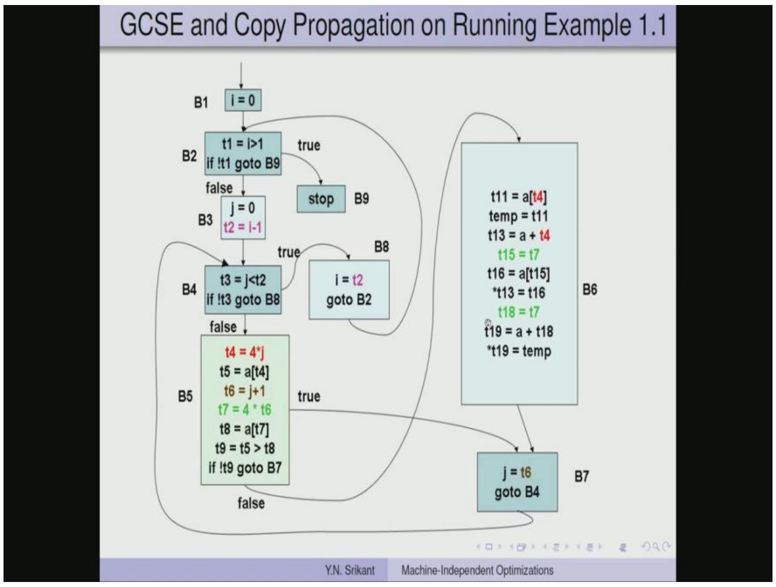
(Refer Slide Time 49:50)



Copy Propagation on Running Example 1.2

So, once we do that we get this and it exposes the next level of common sub-expressions which can be eliminated. So, we get this.

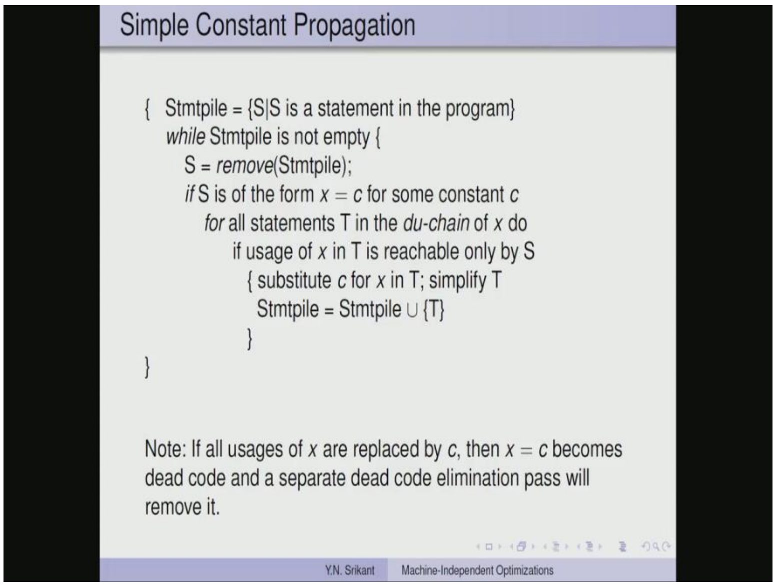
(Refer Slide Time 50:01)



GCSE and Copy Propagation on Running Example 1.1

So, we can now do another copy propagation on this and finally, we get a very condensed piece of code.

(Refer Slide Time 50:14)



Simple Constant Propagation

```
{   Stmtpile = {S|S is a statement in the program}
    while Stmtpile is not empty {
        S = remove(Stmtpile);
        if S is of the form $x = c$ for some constant $c$
            for all statements T in the du-chain of $x$ do
                if usage of $x$ in T is reachable only by S
                  { substitute $c$ for $x$ in T; simplify T
                    Stmtpile = Stmtpile $\cup$ {T}
                  }
}
```

Note: If all usages of x are replaced by c, then $x = c$ becomes dead code and a separate dead code elimination pass will remove it.

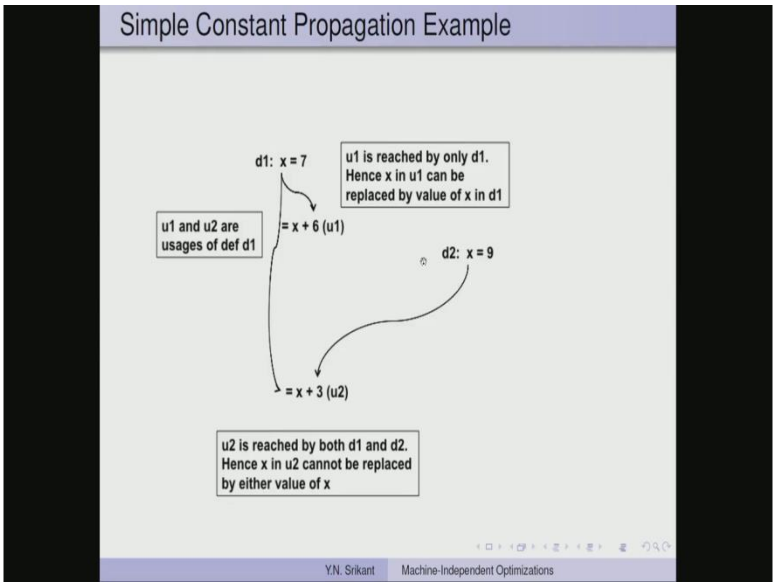
Y.N. Srikant Machine-Independent Optimizations

So, that completes our available the copy propagation algorithm, so now, let us move on to the next optimization which is simple constant propagation and constant folding. In fact, constant propagation you know is very similar to copy propagation it is just that the right hand side of the copy now becomes a constant value. So, the algorithm has the same flavor as before, but it is not necessary to solve a different you know data flow analysis problem here.

So, we use a statement pile which consists of all the statements in the program, so we take one statement at a time and then see what to do it, so while statement pile. So, we include all the statements in the program into this, so while statement pile is not empty remove these statement, if the statement is not a you know statement of the form X equal to c, where c is a constant then we ignore the statement.

And we actually go to the next statement in the pile, if it is indeed a statement of the form X equal to c, then we check all the statements t in the definition use chain of X, so that means, we check the X is a definition, because it is of the form X equal to c. We see all the usages of X and this d u chain is a very convenient data structure to examine all such usages, now if the usage of X in t is reachable only by S, so this can be checked using the u d chain, so let me show you what this really means.

(Refer Slide Time 52:23)



Simple Constant Propagation Example

d1: x = 7

= x + 6 (u1)

d2: x = 9

= x + 3 (u2)

u1 is reached by only d1. Hence x in u1 can be replaced by value of x in d1

u1 and u2 are usages of def d1

u2 is reached by both d1 and d2. Hence x in u2 cannot be replaced by either value of x

Y.N. Srikant Machine-Independent Optimizations

So, here is a constant definition X equal to 7, let us say there are two usages for this definition, one is here u 1 and the other is here u 2, so the d u chain consists of both these usages. In the case of u 1, this is the only definition which is reaching this u 1, so we can simply replace this X by 7 and also simplify this expression 7 plus 6 as 13, so there is no problem with that.

But, if you consider u 2 there is another definition d 2, X equal to 9 which is reaching this X, so is the value of X here is it 7 or is it 9, that cannot be determined at compile time. And therefore, we actually do not replace this X either by 7 or by 9, we just leave it as it is and if we do that then you know it is not possible to remove either this statement X equal to 7 or X equal to 9, so this is what we mean by the statement if usage of X in t is reachable only by S.

So, if it is true then substitute c for X in t, then we simplify the statement t, so I showed you this already, here we could have simplified this to 7 plus 6 why should we do this, suppose this was Y equal to X plus 6 and now after simplification this becomes Y equal to 13, so this is another constant assignment, so now all usages of Y can be actually potentially replaced by 13, so that is the reason why we require this simplification of t.

And after simplification we add it to the statement pile this is again required, because if this had become Y equal to 13, this is a new statement which was not present before, so we must add it to the statement pile, so that we examine all the d u chain of Y etcetera as well. So, the statement pile now gets another extra statement, now the same loop is repeated until the statement pile becomes empty.

So, in this case after the usages of X are replaced by c, then X equal to c possibly becomes dead code if there is still you know some usage of X which is not replaced, then it is not dead code. Otherwise, it is dead code and a separate dead code elimination pass can remove such code, so that is not an issue what have we not done here actually, we have not performed what is known as conditional constant propagation. This is only simple constant propagation in the case of conditional constant propagation.

We consider conditions statements and if the conditions evaluate to a constant value then we need to choose either the true branch or the false branch at compile time itself, the other branch becomes redundant. So, this type of conditional constant propagation is a little more complicated than the simple constant propagation and when we consider the static single assignment form which is more effective for conditional constant propagation. We will consider examples and the algorithm to perform it, thank you, we will stop here today and continue the next part.

Thank you.