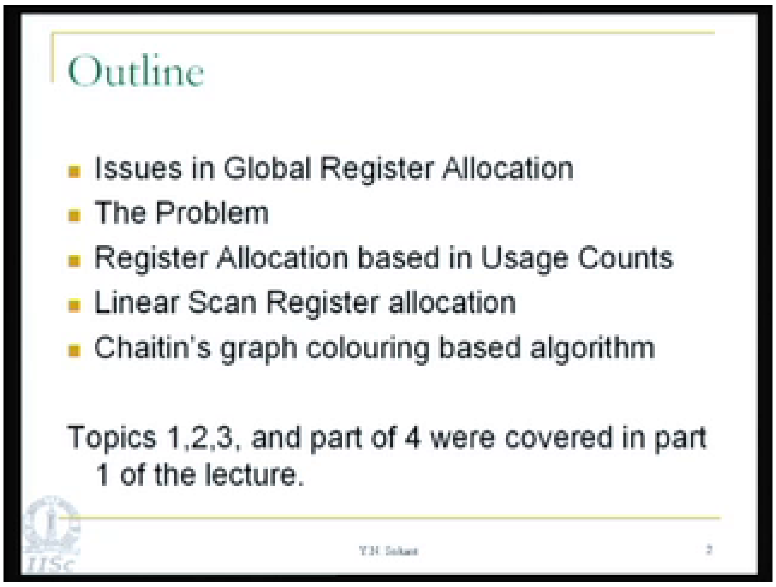
Compiler Design

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

Module No. # 05
Lecture No. # 13
Global Register Allocation-Part2

Welcome to part two of a lecture on global register allocation.

(Refer Slide Time: 00:21)



Outline

- Issues in Global Register Allocation
- The Problem
- Register Allocation based in Usage Counts
- Linear Scan Register allocation
- Chaitin's graph colouring based algorithm

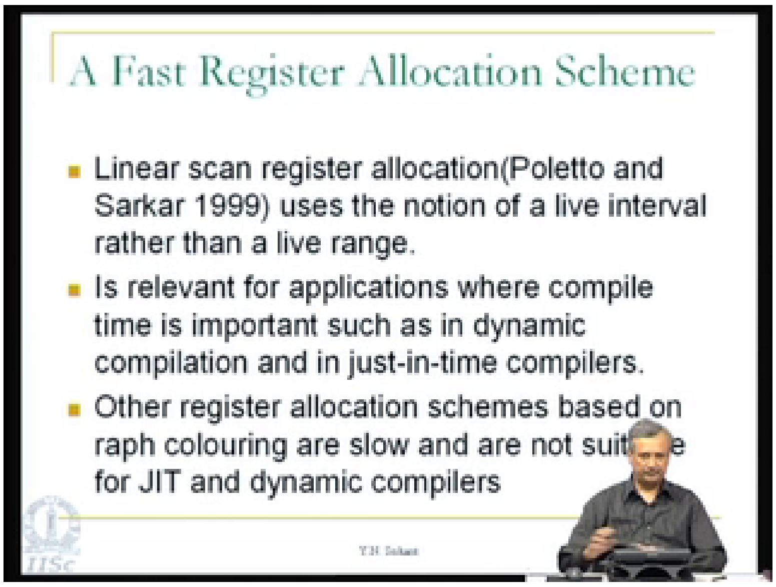
Topics 1,2,3, and part of 4 were covered in part 1 of the lecture.

Y.N. Srikant 2

In the last lecture, we considered the issues in global register allocation. We defined the problem. We looked at the simple register location algorithm based on usage counts. I gave you an introduction to the linear scan register allocation.

Today, we will continue with linear scan register allocation and then go on to Chaitin's graph colouring based algorithm.

(Refer Slide Time: 00:53)



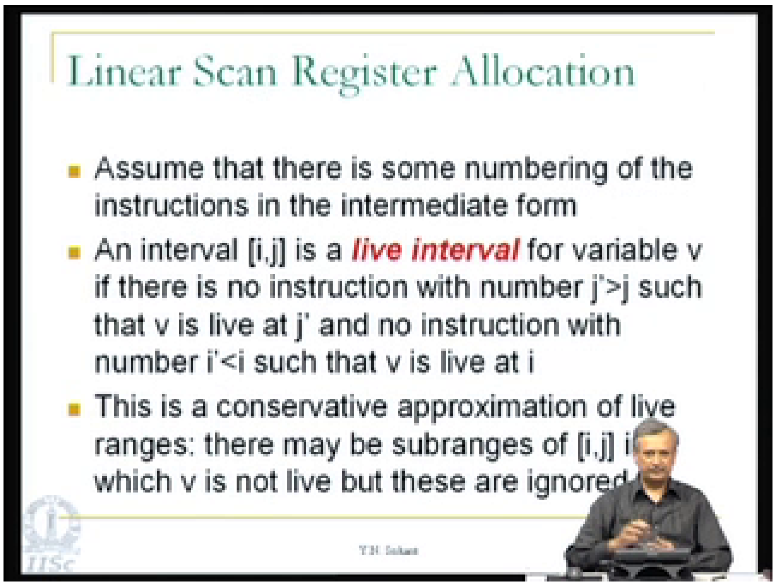
A Fast Register Allocation Scheme

- Linear scan register allocation(Poletto and Sarkar 1999) uses the notion of a live interval rather than a live range.
- Is relevant for applications where compile time is important such as in dynamic compilation and in just-in-time compilers.
- Other register allocation schemes based on raph colouring are slow and are not suit e for JIT and dynamic compilers

Y.N. Srikant

So, linear scan register location is a very fast register location algorithm; it uses the notion of a live interval instead of the live range, as it is usually done. In applications such as dynamic compilers and just-in-time compliers, the speed of register allocation is also very important and in such cases, linear scan register allocation finds a place.

(Refer Slide Time: 01:25)

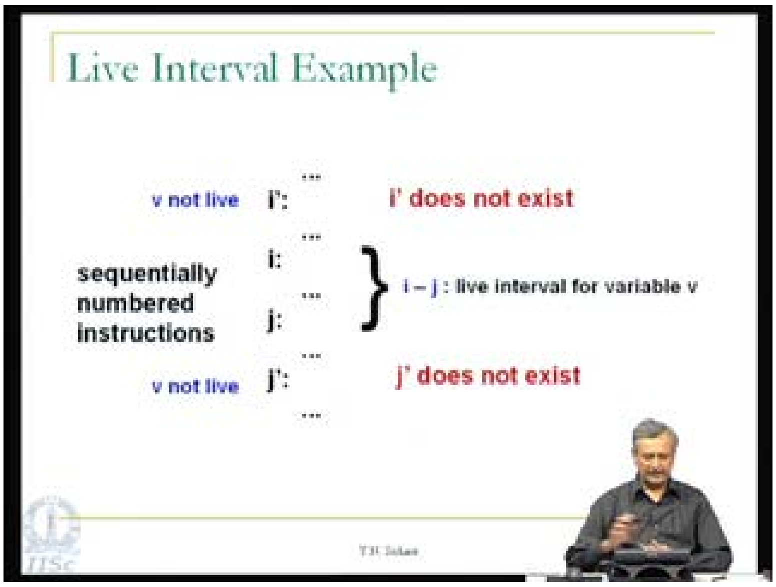


Linear Scan Register Allocation

- Assume that there is some numbering of the instructions in the intermediate form
- An interval [i,j] is a ***live interval*** for variable v if there is no instruction with number j'>j such that v is live at j' and no instruction with number i'<i such that v is live at i
- This is a conservative approximation of live ranges: there may be subranges of [i,j] i which v is not live but these are ignored

Y.N. Srikant

(Refer Slide Time: 01:29)



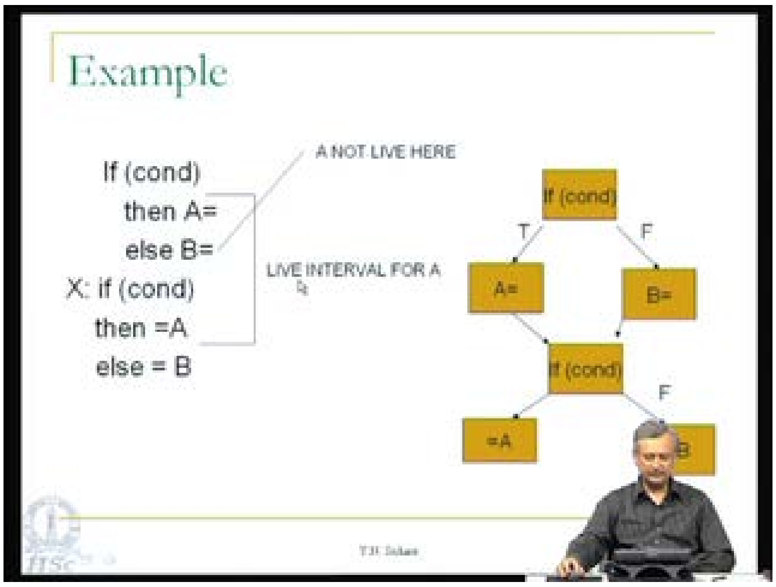
Live Interval Example

		...	
v not live	i':		i' does not exist
		...	
sequentially numbered instructions	i:		i – j : live interval for variable v
		...	
	j:		
		...	
v not live	j':		j' does not exist
		...	

So, we will quickly review the concept of a live interval. If you consider the sequence of instructions, you know, in linear order numbered in this fashion. If you take an interval I J, and if you want this to be a live interval for the variable v, then there should be no I prime at which v is live, and there should be no J prime at which v is live. So, if this is the maximal interval over which v is live, and then this is called as a live interval. It is possible that there are pockets within this interval I J, where v is not live. You know it is possible that v is defined and then it is used for a while then v is not defined. You know used at all for some time then v is defined again and so on.

All these are put together in a single interval. The advantage of doing this is, it is easy to find such live intervals, whereas, live you know, we can do just one pass over the quadrapulse or virtually allocated machine instructions and then find these live intervals; whereas, live variable, live interval, and live range analysis requires a complete data flow analysis.

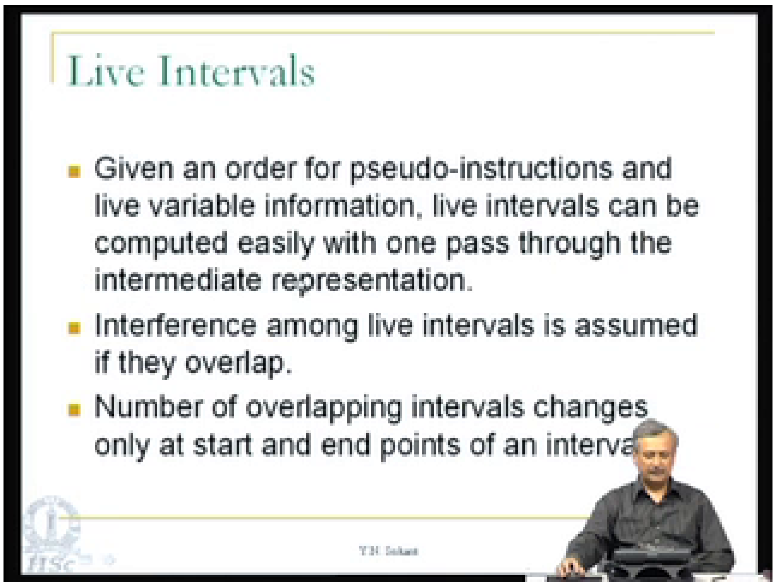
(Refer Slide Time: 02:54)



Example

```
If (cond)
   then A=
   else B=
X: if (cond)
   then =A
   else = B
```

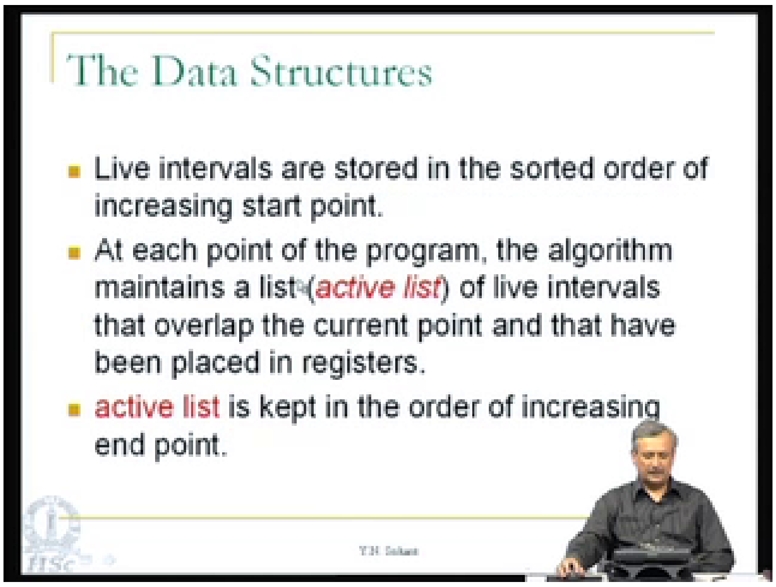
(Refer Slide Time: 02:55)



Live Intervals

- Given an order for pseudo-instructions and live variable information, live intervals can be computed easily with one pass through the intermediate representation.
- Interference among live intervals is assumed if they overlap.
- Number of overlapping intervals changes only at start and end points of an interva

(Refer Slide Time: 02:57)



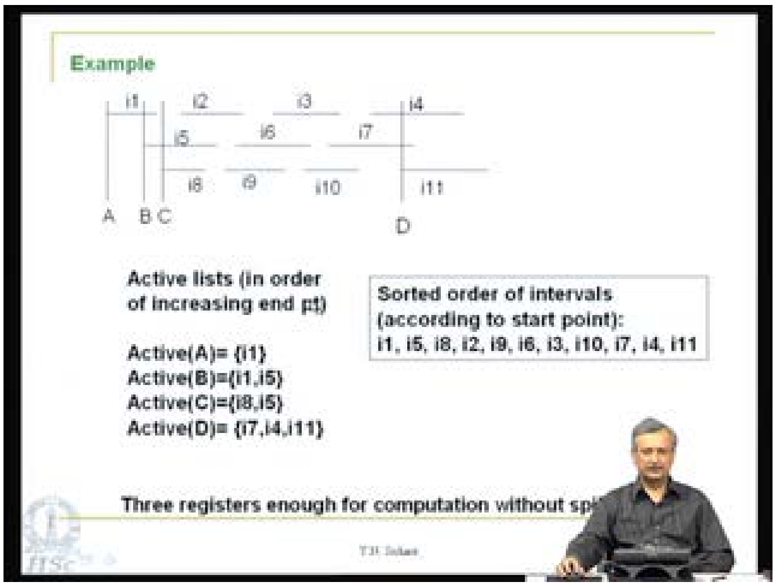
The Data Structures

- Live intervals are stored in the sorted order of increasing start point.
- At each point of the program, the algorithm maintains a list (*active list*) of live intervals that overlap the current point and that have been placed in registers.
- active list is kept in the order of increasing end point.

Y.N. Srikant

So, the data structures which are used are the active list and of course, the list of live intervals.

(Refer Slide Time: 03:05)



Example

Active lists (in order of increasing end pt)

Active(A)= {i1}
Active(B)={i1,i5}
Active(C)={i8,i5}
Active(D)= {i7,i4,i11}

Sorted order of intervals (according to start point):
i1, i5, i8, i2, i9, i6, i3, i10, i7, i4, i11

Three registers enough for computation without sp

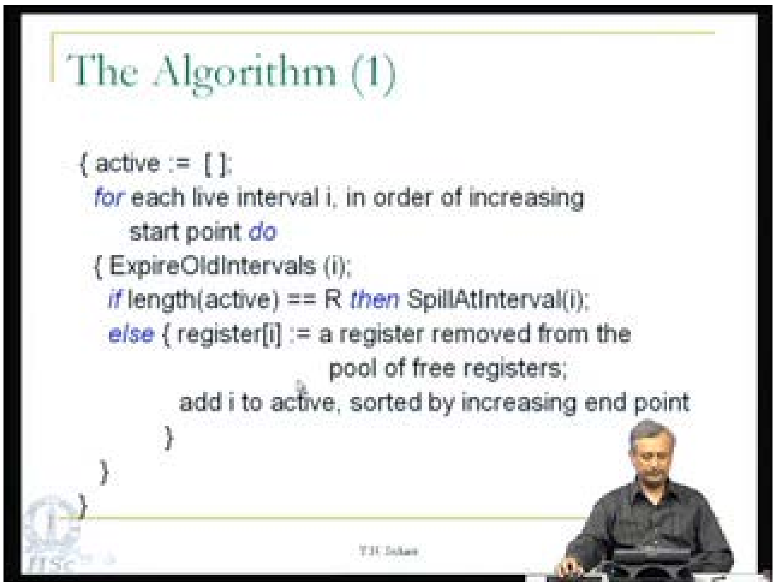
Y.N. Srikant

We considered this example last time. The intervals are stored in a sorted order according to the increasing starting point. There is also a list of active intervals, called active list, in which we store those intervals for which registers have been allocated and are still active.

I explained this example last time. For example, at the point A, there is only one interval i1, which is active and it will be allocated a register; at the point B, i5 becomes active, i1 has already been given a register, and it is not yet dead and therefore, we have more registers then we can give 1 to i5 as well.

Similarly at i8, i1 is dead. It is not live anymore; but i5 is still active, and i8 now gets into the active list, if it is given a register. So, this can go on and at point D, we have i4, i7 and i11 all three of them overlapping i4. You know i7 would have been given registers earlier, if there is one more register available, we can give it to i11 and then place it in the active list.

(Refer Slide Time: 04:33)



The Algorithm (1)

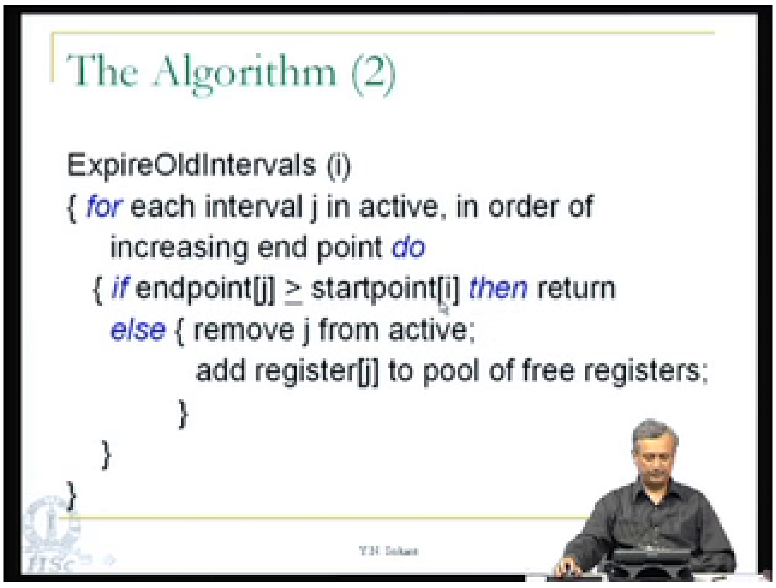
```
{ active := [ ];
  for each live interval i, in order of increasing
      start point do
  { ExpireOldIntervals (i);
    if length(active) == R then SpillAtInterval(i);
    else { register[i] := a register removed from the
                          pool of free registers;
           add i to active, sorted by increasing end point
         }
  }
}
```

Let us look at the linear scan allocation algorithm in some more detail. So, the active list is initialized to null to begin with; and then it says for each live interval i, is in the order of increasing start point. So, this is the list of active live intervals which is stored. I showed this already. So, we take one at a time and then process it.

So, the first step is expired old intervals; we are going to see the details of this very soon. So, that means the intervals which are dead and which are not active any more will be removed from the active list. Among the ones which are left in the active list, if length of active is equal to R, then split at interval i. So, in other words, the number of registers which is given to us is R and the length of the active list is also R; then we cannot allocate anymore registers. So, we either need to push the new live interval in memory or we need to remove one of the intervals from the active list and give that particular register to the new interval. So, this will be decided by the function SpillAtInterval i. So, if the number of registers used is more than the length of active list, then give it a register i, register i is equal to a register removed from the pool of free registers, add i to active, so this is also marked as active and it is sorted by the increasing end point.

Now, what is important is to note that - if the live interval which is not given a register, then it does not get into the active list.

(Refer Slide Time: 06:21)



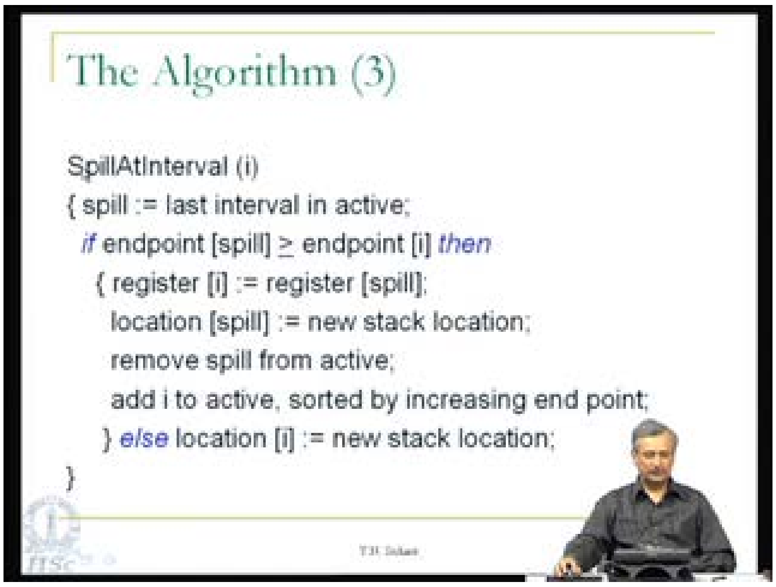
The Algorithm (2)

```
ExpireOldIntervals (i)
{ for each interval j in active, in order of
    increasing end point do
  { if endpoint[j] ≥ startpoint[i] then return
    else { remove j from active;
           add register[j] to pool of free registers;
         }
  }
}
```

Y.N. Srikant

So, let us now look at ExpireOldIntervals. So, this function for each interval j in active, again consider the active list in the order of increasing end point, do if endpoint j greater than or equal to start point i then return. In other words, the interval is still active it is not dead; otherwise, remove j from active and add register j to the pool of free registers. So, this is very simple.

(Refer Slide Time: 06:53)



The Algorithm (3)

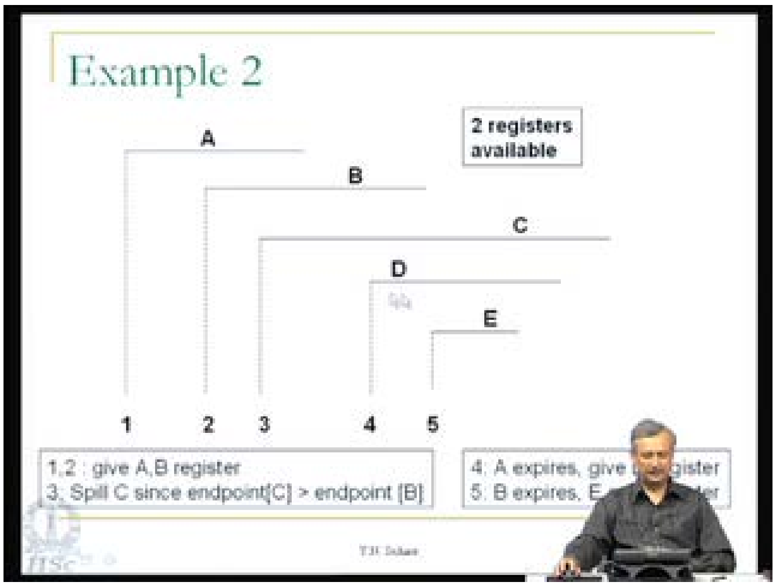
```
SpillAtInterval (i)
{ spill := last interval in active;
  if endpoint [spill] ≥ endpoint [i] then
    { register [i] := register [spill];
      location [spill] := new stack location;
      remove spill from active;
      add i to active, sorted by increasing end point;
    } else location [i] := new stack location;
}
```

Now lets look at SpillAtInterval i. Let us say, last interval is inactive, you know is spill this has its endpoint you know at the which the end point of this particular last interval is the highest right so it ends after all other intervals in the active list end

So, if the endpoint of spill is greater than or equal to endpoint of i, i is our current point, that means, we now have an interval which ends very late much more than the endpoint of the new interval that we are considering. So, in such a case, we spill. So, what we do is, we remove the register from the last interval that is inside spill; so register i equal to register spill, and then the location of spill will be new stack location. In other words, we are going to make this particular last interval which is in spill, reside in memory all the time. So, the implication is that every time we use a variable in this live interval, we are going to load it from memory and immediately whenever we have an assignment to a variable in this live interval, we are going to store it into memory location corresponding to the variable.

So, remove the spill that is, this interval from active, add i to active, because now it has been given a new register. Sort it by increasing endpoint. So, if this is not so, if the new interval that we are considering ends much later than what the last interval in active was, then location i is a new stack location. So, we assign the new interval itself to memory and retain the old interval (Refer Slide Time: 08:57).

(Refer Slide Time: 09:05)



Example 2

1,2 : give A,B register
3: Spill C since endpoint[C] > endpoint [B]

4: A expires, give [illegible]gister
5: B expires, E [illegible]ter

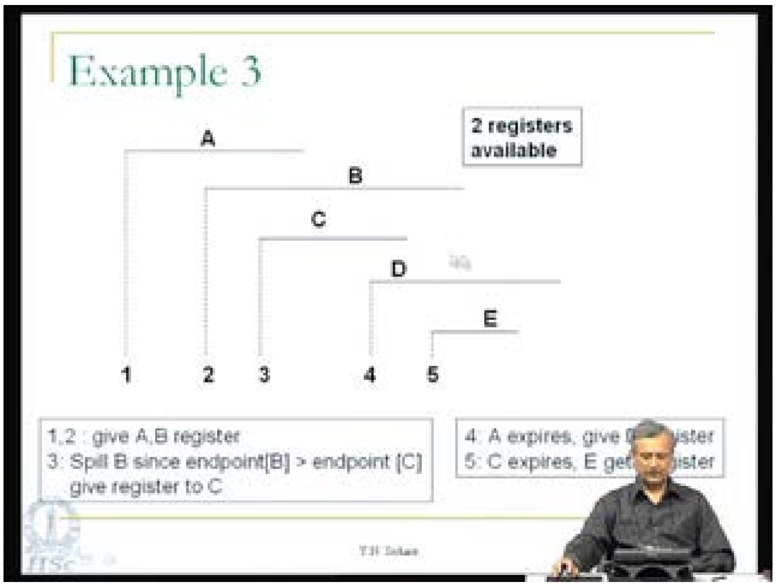
Let us look at this. It does not require any spilling. So, let us consider the second example, which involves some spilling, assume that there are only two registers available now, there are A B C D E are the five intervals, so these are the five starting points of the intervals.

At the point 1, you know a free register is given to A. So, the active list will contain only A, at the point 2, A has not yet finished; B has begun. So, there is one more register available, so we can give B also a register. At the point C, we have a problem. A and B have been given registers, but C requires a new register and we do not have a new register.

So, observe that the end point of C is here, whereas, the last interval in the active list is that of B and its endpoint is right here. So, C ends much later than B. We are going to put C itself into memory spill C; since endpoint of C is greater than endpoint of B.

At point 4, D becomes active, but A has already expired, there is a C that will always be permanently in memory. We are not going to consider C anymore. So, B is the only one which has taken a register; D is also given a register now, so at this point, B and D are the two variables, live intervals which are given registers. At the point E, B has just expired, so the register of B is free, only D is in the register. So, a new register becomes available and that is given to E. Let us see how the situation changes, if the endpoint of C is different - for example, it is here.

(Refer Slide Time: 10:53)


Example 3

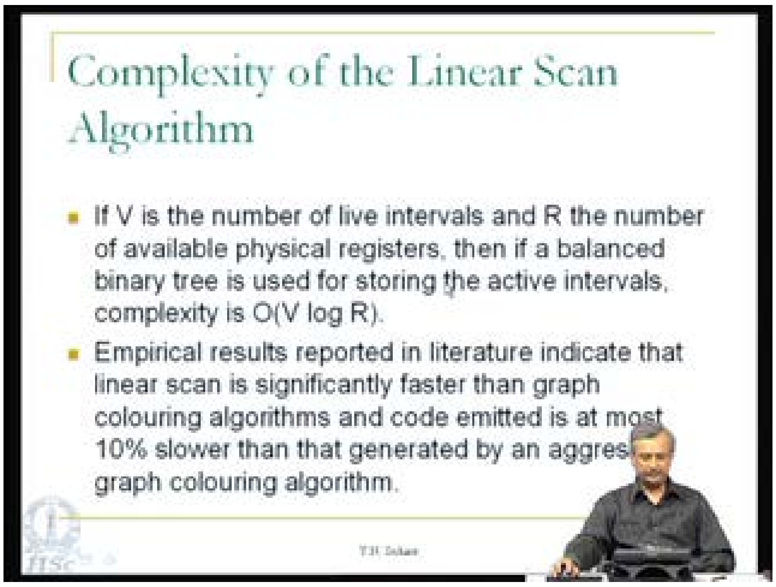
2 registers available

1,2 : give A,B register
3: Spill B since endpoint[B] > endpoint [C] give register to C

4: A expires, give D register
5: C expires, E gets register

In this you know, in these two it is the same. Give A, B a register at 1 and 2. When you consider C, again A and B have been given registers but the endpoint of C is much before the endpoint of B, the last interval in the active list. So, it is B which will be spilled and not C. So, spill B, since endpoint of B is more than the endpoint of C and give register to C, now A and C will reside in registers at the point D. A will retire, so give D the new register. At endpoint E, C expires, so E will be given the new register. So, this is how register allocation and the spilling happens in the linear scan allocation algorithm. It is a very simple algorithm.

(Refer Slide Time: 11:48)

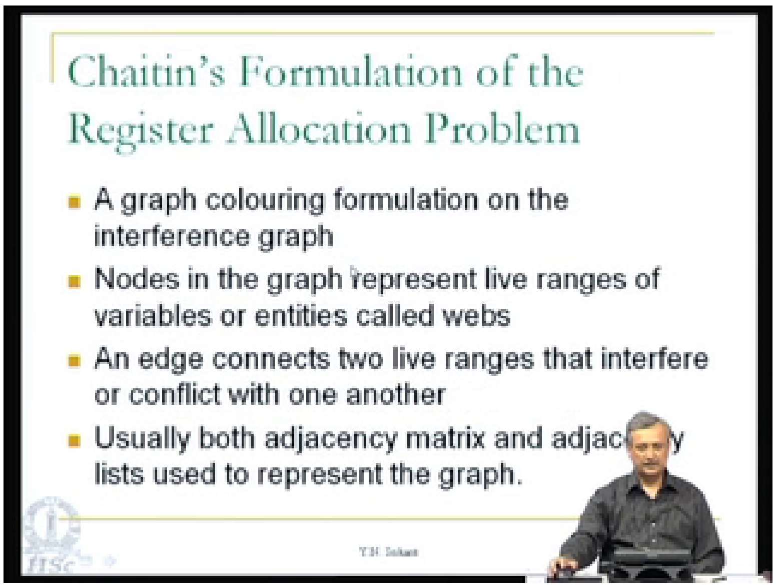

Complexity of the Linear Scan Algorithm

- If V is the number of live intervals and R the number of available physical registers, then if a balanced binary tree is used for storing the active intervals, complexity is O(V log R).
- Empirical results reported in literature indicate that linear scan is significantly faster than graph colouring algorithms and code emitted is at most 10% slower than that generated by an aggressive graph colouring algorithm.

So, what is the complexity of the linear scan algorithm? If V is the number of live intervals, and R is the number of available physical registers, then if a balanced binary tree is used for storing the active intervals, the complexity has been stated as O of V log R. Remember, we want to use a balanced binary tree for storing the active intervals. This is in the increasing order of endpoints.

Empirical results reported in literature indicate that linear scan is significantly faster than graph colouring algorithms and code emitted is at most 10 percent slower than that generated by an aggressive graph colouring algorithm. So, it is not very slow just about 10 percent, and this binary tree will help us in getting to know the intervals at minimum cost.

(Refer Slide Time: 12:55)



Chaitin's Formulation of the Register Allocation Problem

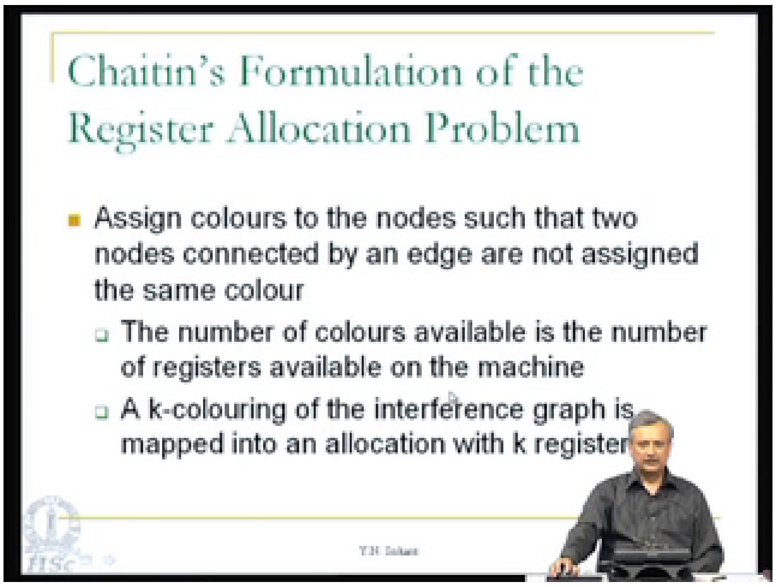
- A graph colouring formulation on the interference graph
- Nodes in the graph represent live ranges of variables or entities called webs
- An edge connects two live ranges that interfere or conflict with one another
- Usually both adjacency matrix and adjacency lists used to represent the graph.

Y.N. Srikant

We move on to the algorithm due to Chaitin. This is the most celebrated register allocation algorithm. What is Chaitin's formulation of the register allocation problem? The problem is actually transformed to a graph colouring problem. The graph which is considered here, is known as an interference graph. What are the nodes in the interference graph? The nodes in the interference graph represent the live ranges of variables or sometimes called webs. We have already known what a live range is and we will see what a web is, a little later.

So, a live range is the set of points where a variable is live. What are the edges in this interference graph? An edge connects two live ranges that interfere or conflict with each other. In other words, if you consider two live ranges, they correspond to two variables. So, if the two live ranges overlap, that is, they run through the same points, then we cannot give the same register to both the live ranges and two variables. Therefore, they are set to conflict or interfere with each other. Then, we put an edge between these two live ranges. Usually both adjacency matrix and adjacency lists are required to represent the graph, because manipulations require both representations.

(Refer Slide Time: 14:40)



Chaitin's Formulation of the Register Allocation Problem

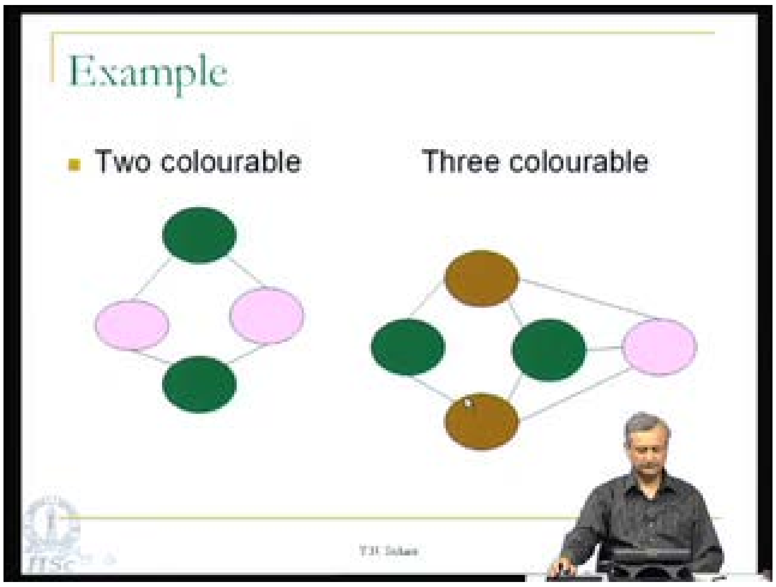
- Assign colours to the nodes such that two nodes connected by an edge are not assigned the same colour
  - The number of colours available is the number of registers available on the machine
  - A k-colouring of the interference graph is mapped into an allocation with k register

Y.N. Srikant

After this, what assigns colours to the nodes? We are doing the graph colouring such that two nodes connected by an edge are not assigned the same colour. So, this is as usual in the case of the colouring problem. So, if there are two nodes which are connected by an edge, then we do not give the two nodes, the same colour.

The number of colours available is the number of registers available on the machine and the k colouring of the interference graph is mapped to an allocation with k registers. So, this is the formulation of the problem. Now, let us see how to solve it.

(Refer Slide Time: 15:17)



Example

- Two colourable

Three colourable

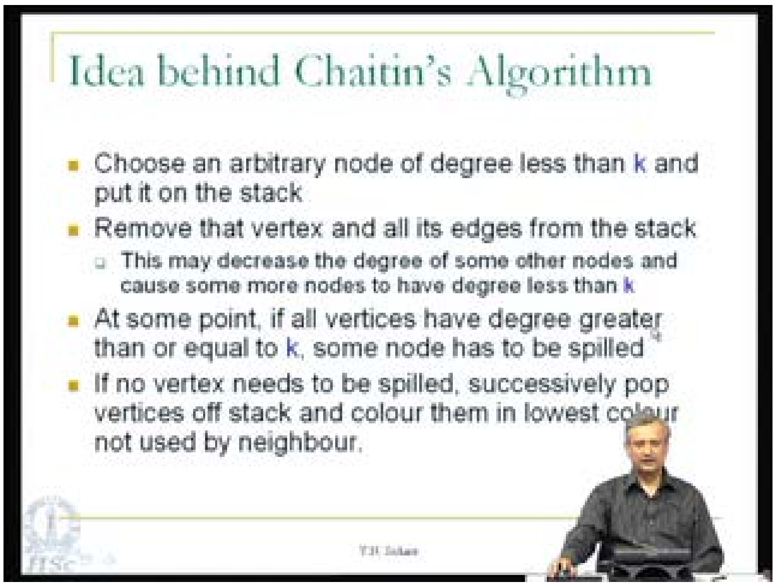
There are two examples here. This graph has four live ranges. These two live ranges interfere; so they are active at the same time. There is an interference edge between these two. Similarly, between these two, between these two, and between these two.

These two live ranges do not interfere and these two live ranges do not interfere. So, there are no edges between these; so, it is very easy to see that we can colour this with two colours. These two are given in one colour and these two are given the other colour and there is no violation of the restrictions.

This graph is similar to this. It is just that we have added one node here. So, this node is connected to these three nodes by three edges. Because of that, we cannot give any of these two colours to this particular node; we need a three colour. So, this graph cannot be coloured with two colours; it can only be covered with three colours.

Suppose, we did not have this particular edge, you assume that we did not have this particular edge, then we could have given this green colour, this could have been coloured green, but because of this edge, these two cannot share the green colour.

(Refer Slide Time: 16:43)


Idea behind Chaitin's Algorithm

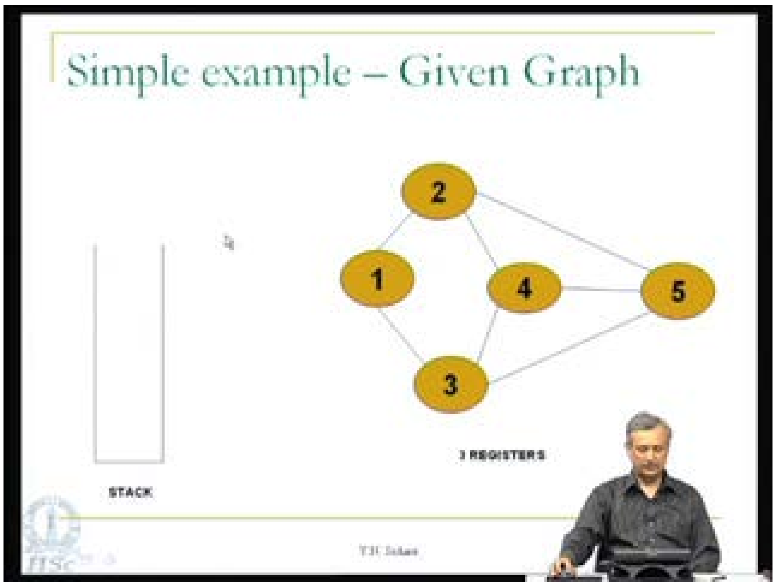
- Choose an arbitrary node of degree less than k and put it on the stack
- Remove that vertex and all its edges from the stack
  - This may decrease the degree of some other nodes and cause some more nodes to have degree less than k
- At some point, if all vertices have degree greater than or equal to k, some node has to be spilled
- If no vertex needs to be spilled, successively pop vertices off stack and colour them in lowest colour not used by neighbour.

So, what is the idea behind Chaitin's algorithm? The basic idea is that of graph reduction. What do you mean by graph reduction and how do we do it?

So, choose an arbitrary node of degree less than k and put it on the stack. The basic idea is, if you consider nodes of degree less than k, k is the number of registers or colours available to us, remove that particular node and all the edges associated with it, then continue with such reduction until you cannot do so. This process is called as reduction. Let us see, how it helps us.

Remove that vertex and all its edges from the graph. This may decrease the degree of some other nodes and cause some more nodes to have degree less than k. So, this is obvious. We will see an example soon. If you remove a node and some edges, obviously, some other nodes will have degree less now. At some point, if all vertices have degree greater than or equal to k, there is no way you can remove any node, because we wanted nodes of degree less than k - some node has to be spilled. We will see what spilling is. It implies loading and storing at various points in the live interval, live range. If no vertex needs spilling, then you know successively pop vertices off stack and colour them in lowest colour not used by the neighbor. Let us look at an example and then see what happens, if there is spilling.

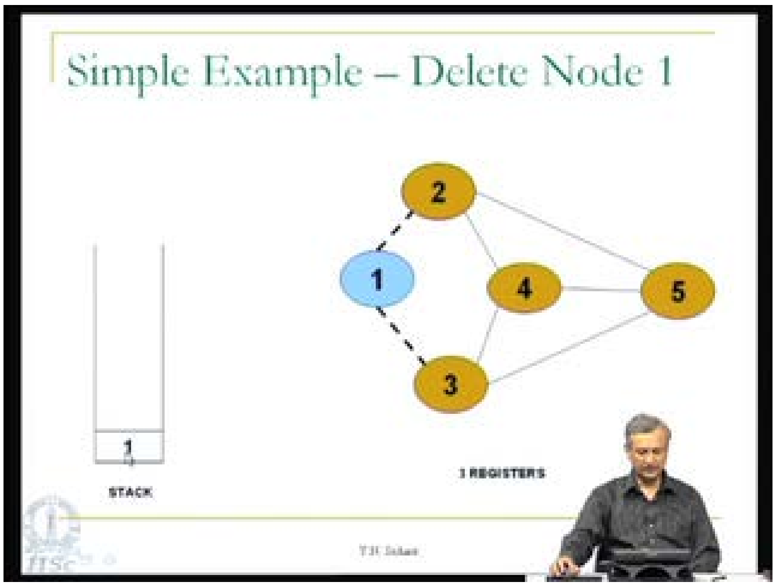
(Refer Slide Time: 18:39)



Simple example – Given Graph

So, here is a stack, here is a very simple graph, which requires three colours and these are the numbers,. Node 1 is the only one which has degree two, which is less than 3. All others have degree 3.

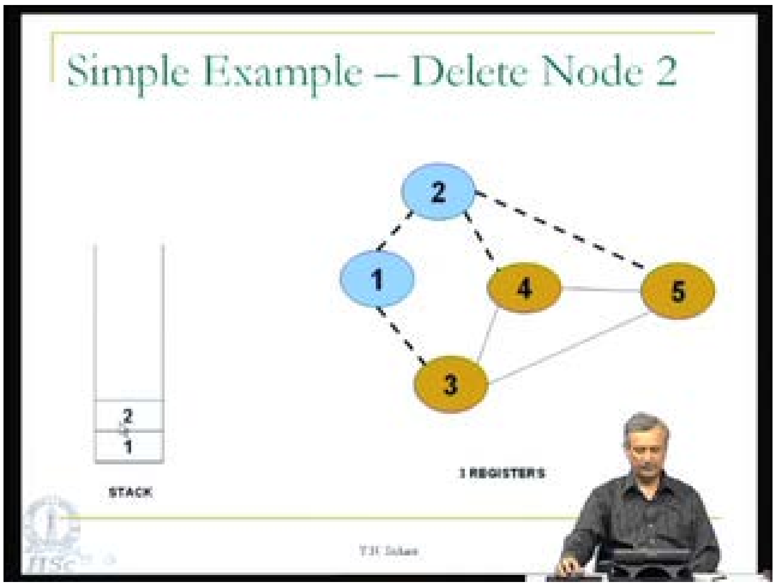
(Refer Slide Time: 18:59)



Simple Example – Delete Node 1

So, let us pick 1 and delete that; so, it goes to the stack. This is the deleted node. Now, these two edges have also been removed; that is why they are shown as dotted. So, if you remove these edges and the node, then node 2, node 3 also have degree 2, whereas, these 3, 4 and 5 still have degree 3.

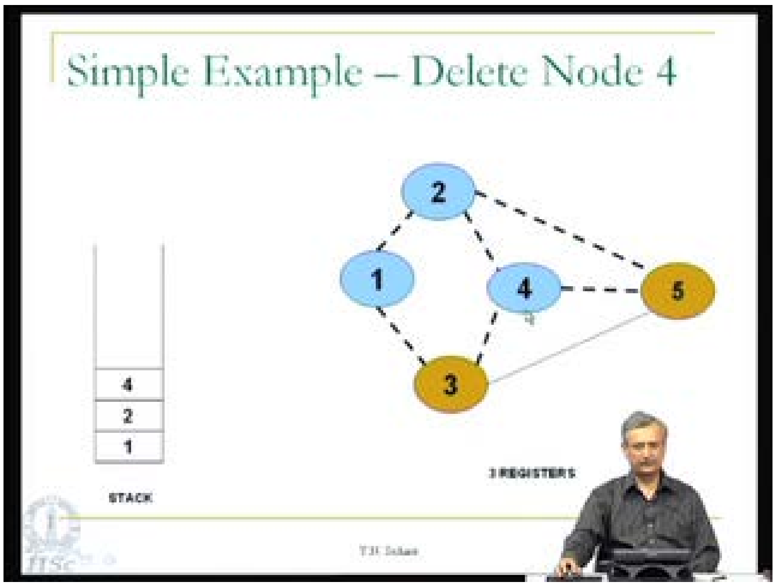
(Refer Slide Time: 19:26)



Simple Example – Delete Node 2

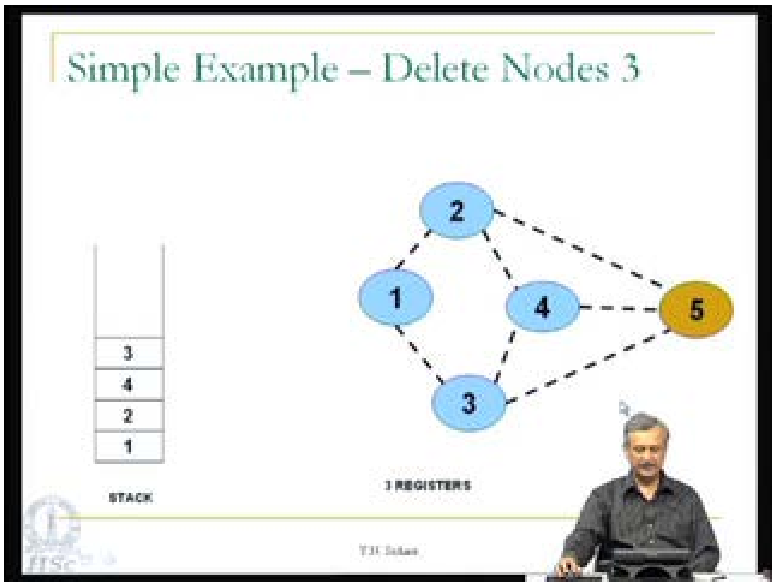
So, we can delete one of these two, either 2 or 3. Let us delete node number 2. If we delete that, then the edges corresponding to these, connecting the node 2 to 4 and 5 are also removed. So, the graph which is left out is just 3, 4, 5, and all three nodes in this graph have degree 2. So, we can remove any one of them.

(Refer Slide Time: 19:43)

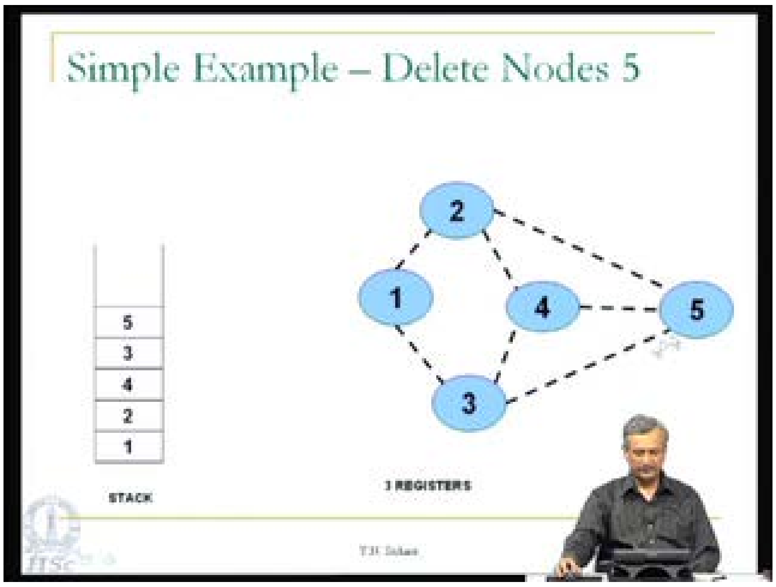


Simple Example – Delete Node 4

(Refer Slide Time: 19:58)

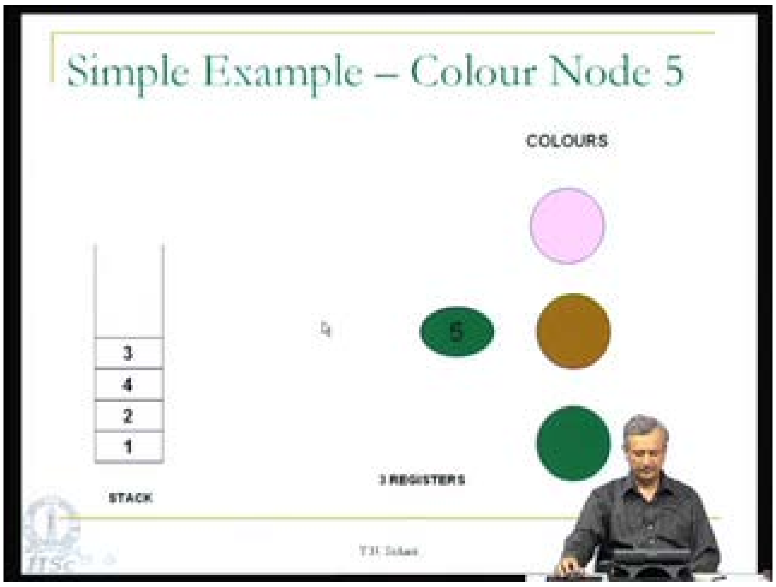

Simple Example – Delete Nodes 3

(Refer Slide Time: 20:08)


Simple Example – Delete Nodes 5

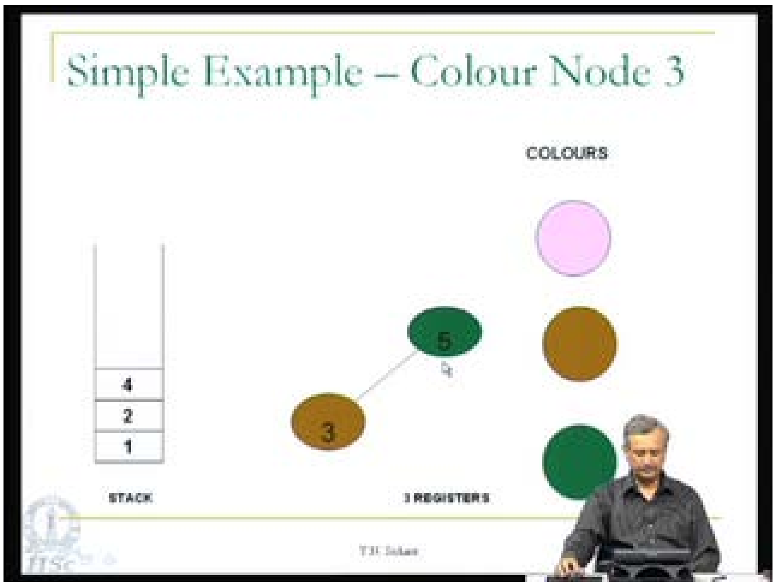
Let us say, we choose to remove four instead of three, why choose numerical order all the time, so, if we remove four and the two edges associated with it, we are left with three and five connected by an edge. Let us remove node number 3; so, we remove three, then this edge is also gone. Only node 5 remains. There are no more edges, so, we remove node five and the entire graph has been deleted. The stack stores these nodes in a particular order. Colouring of the nodes of the graph is actually done in this particular reverse order. We will see how that happens.

(Refer Slide Time: 20:24)



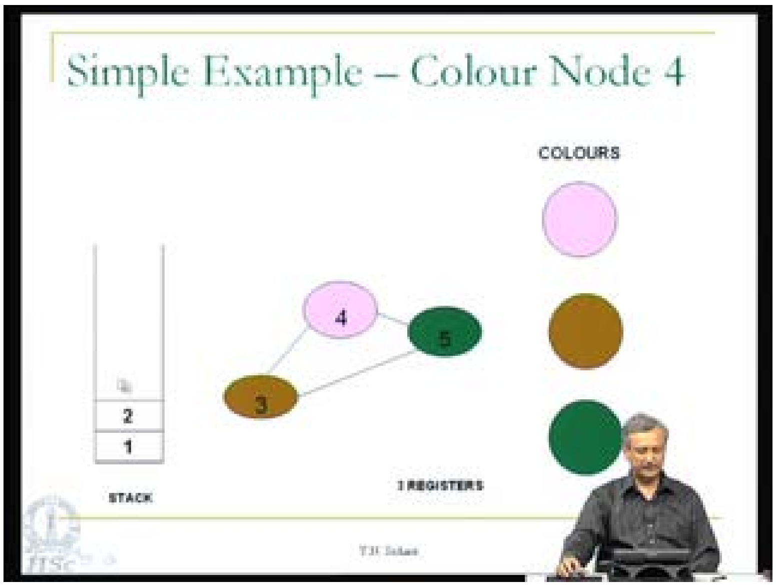
Simple Example – Colour Node 5

(Refer Slide Time: 20:35)



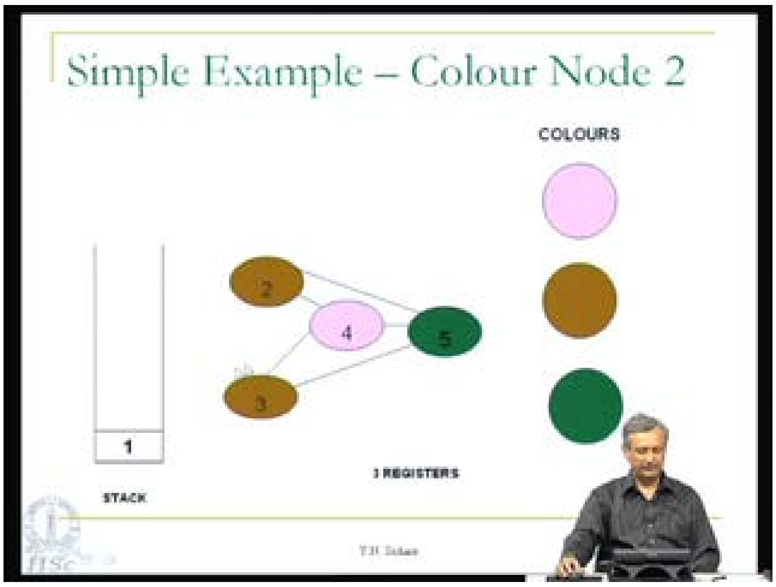
Simple Example – Colour Node 3

(Refer Slide Time: 20:49)



Simple Example – Colour Node 4

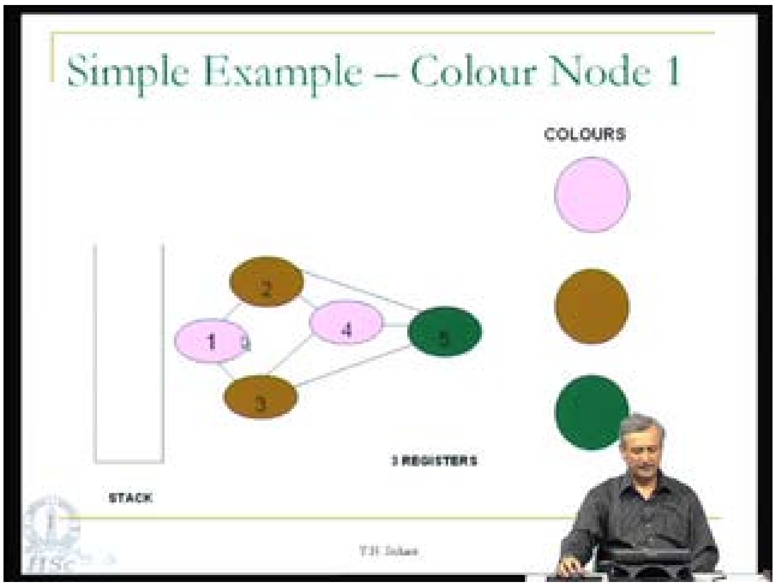
(Refer Slide Time: 21:01)



Simple Example – Colour Node 2

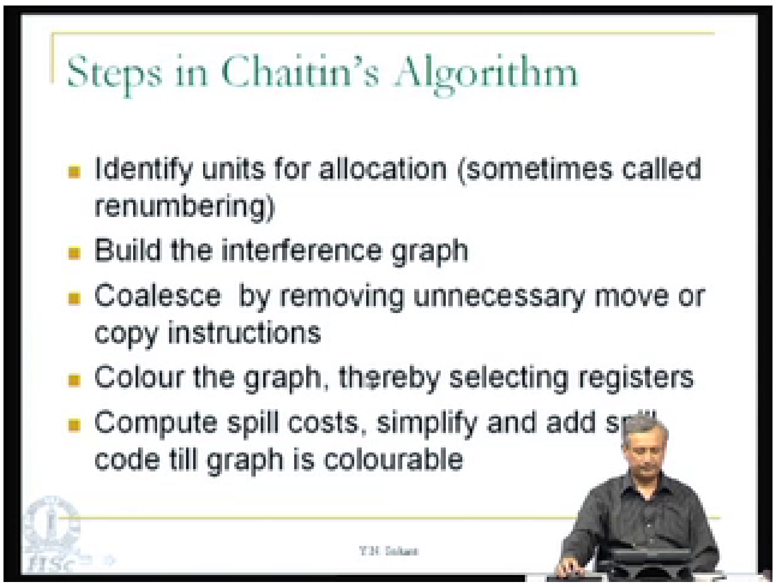
So, there are three colours given to us. Take node number 5, which is the top of stack and give it green colour. Then you have remove 3 and it cannot be given colour green. So, let us give it brown and the edge between these two is also restored. Then, 4 is removed from the stack, the edges of 4 are also restored, 4 cannot be given any of these two colours, so, it is given a new colour. Then, we remove node two, its edges are all restored. So, once we do that, we check the neighbors, which are 4 and you know this particular 5, we cannot give either this light violet colour or this green colour; so, we will have to give it a different colour; it is not connected to this particular node, so we can give the brown colour, which is remaining free.

(Refer Slide Time: 21:31)


Simple Example – Colour Node 1

Now, we add one. So, 1 is connected to 2 and 3. We have to give a colour which is different from that of 2 and 3. We just chose to give 1, we could possibly have given it green also, this is the colouring of the graph. So, we can give the registers corresponding to these colours to these live ranges and appropriately modify the program corresponding to it.

(Refer Slide Time: 21:59)


Steps in Chaitin's Algorithm

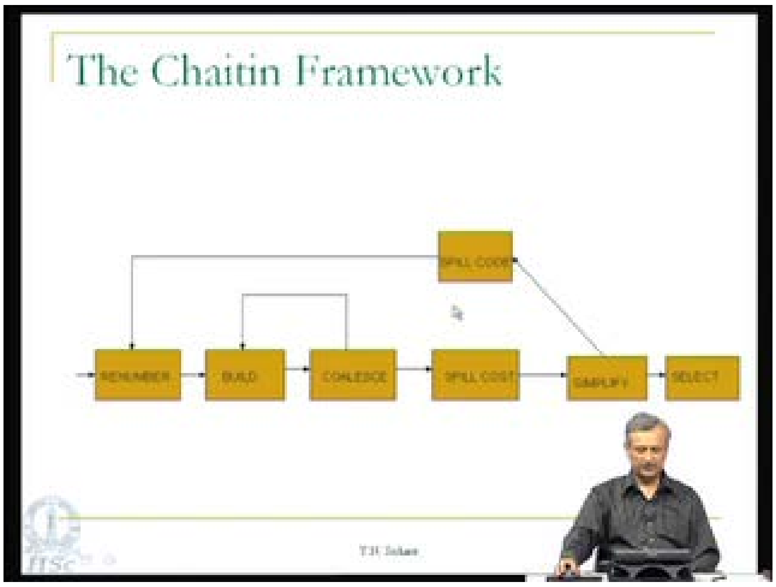
- Identify units for allocation (sometimes called renumbering)
- Build the interference graph
- Coalesce by removing unnecessary move or copy instructions
- Colour the graph, thereby selecting registers
- Compute spill costs, simplify and add spill code till graph is colourable

Y.N. Srikant

Let us see the steps in Chaitin's algorithm and include spilling, identify the units for allocation. These are the live ranges or webs. We will see webs very soon. Webs build the interference graph by including the edges interfering between these live ranges.

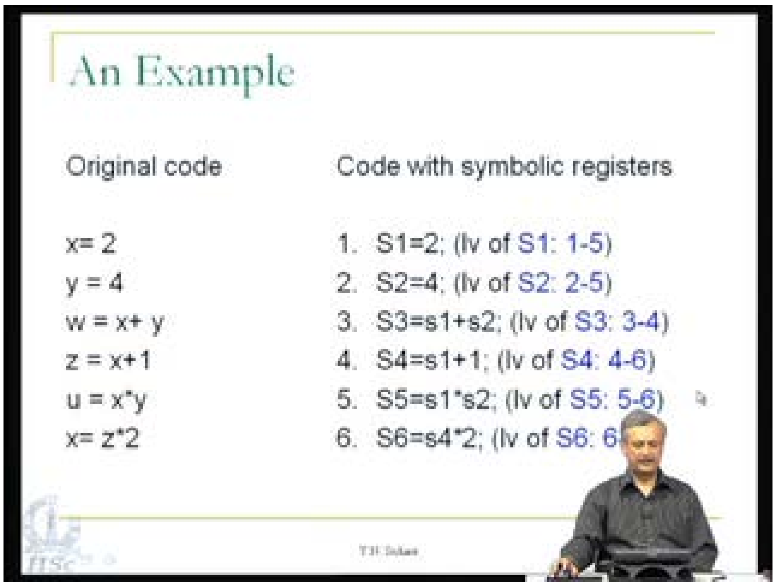
Third step is called coalescing. Coalesce by removing unnecessary move or copy instructions. So, we will see what happens during this. Then colour the graph after this coalescing is over. Colour the graph thereby selecting registers. So, if there is to be some spilling, then compute spill costs, simplify. We will see all this. Add spill code and then see whether the graph is colourable, we go on doing this until the whole becomes colourable. This will eventually happen because eventually by spilling nodes, you would have actually reduced the requirement on registers.

(Refer Slide Time: 23:10)


The Chaitin Framework

This is the framework for Chaitin's register location algorithm. This is the stage, which constructs the live ranges or webs. This is the stage which builds the interference graph, then coalescing the live ranges, compute, this keeps happening until there is no improvement possible, then you compute the spill cost, then simplify the graph and then introduce spill code if necessary. Go back to step 1 and keep going until you cannot change it anymore. Finally select the registers and colour.

(Refer Slide Time: 23:48)



An Example

Original code	Code with symbolic registers
x= 2	1. S1=2; (lv of S1: 1-5)
y = 4	2. S2=4; (lv of S2: 2-5)
w = x+ y	3. S3=s1+s2; (lv of S3: 3-4)
z = x+1	4. S4=s1+1; (lv of S4: 4-6)
u = x*y	5. S5=s1*s2; (lv of S5: 5-6)
x= z*2	6. S6=s4*2; (lv of S6: 6

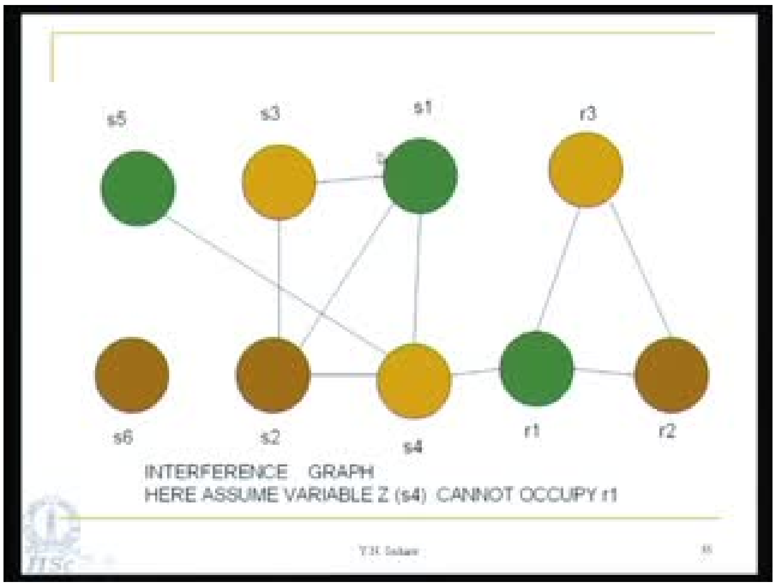
Y.N. Srikant

Here is a very simple example. This is the original code- x equal to 2, y equal to 4, w equal to x plus y, z equal to x plus 1, u equal to x star y and x equal to z star 2, so, observe that x is defined twice, all others are not defined twice. Let us number the instructions, and then let us assume that there are virtual registers assigned to these variables. So, whatever is defined again, will be actually given a new virtual register. So, S1 equal to 2, S2 equal to 4, S3 equal to s1 plus s2, S4 equal to s1 plus 1, S5 equal to s1 star s2, and S6 equal to s4 star 2.

The live range of S1 is from 1 to 5. Why? It is defined in 1 and then it is used in 3, 4 and 5 and afterwards it is not used. So, the last usage is 5, so 1 to 5 is the live range of S1, live range of S2 is from 2 to 5. Observe that S2 is used here, and S2 is used here and afterwards it is not used.

So, live range of S3 is 3 to 4, and then you know S3 is not used anymore. We are assuming that it is just one instruction. S4 is between 4 and 6. It is used here and here. Then, we are assuming that it is not used anymore. S5 is 5, 6. Again, this is the only one which is used and S6 is 6 onwards. These are the live ranges of these virtual or symbolic registers. Let us look at the interference graph for this particular example.

(Refer Slide Time: 25:42)



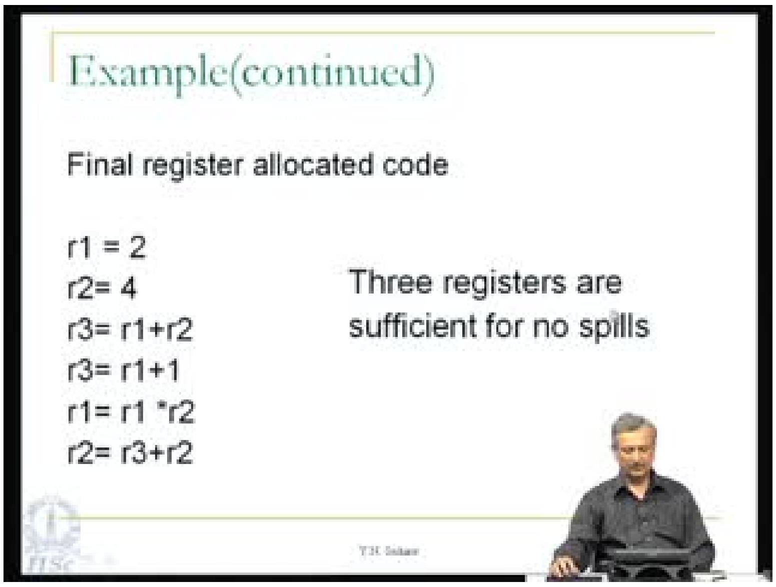
So, now the specialty of this graph is that we have introduced the three registers available r1, r2 and r3, which are three colours also into this graph, the reason is, if there are restrictions, that one of the physical registers cannot be used for any variable, then we can actually introduce these nodes and then add an edge between them. For example, assume variable z which corresponds to s4 cannot occupy the register 1, so there is an edge between these two- that means this s4 node can never get green colour, so, thereby it will get different register.

If there is no such restriction, then this edge will not be present and these three variables have an edge between these three registers which are three colours will have edges between them indicating that one register cannot occupy the place of the other, so, they interfere with each other If this edge is not present, then there is no connection between this interference graph and this register interference graph. (Refer Slide Time: 27:03) (Refer Slide Time: 27:22)

So, there are edges between these two, for example, between s 1 and s 3, there is an edge. Let us see why? So, the live range of s1 is 1 to 5, live range of s3 is 3 to 4, so, 1 to 5 and 3 to 4 have overlap and that is the reason why there is an edge. Similarly, between other pairs of variables or edge, the nodes are already coloured. It is very easy to see how to colour this. There are three registers, so you can push this, and then this, and this. You will have this particular node this and this, all these, have variables. These are all corresponding to variables, so we can spill this, and then this, so on and so forth. Some order can be used. We can spill some of these also, the same stack algorithm not spill sorry the colour some of these also and then delete these nodes and so on and so forth not spilling.

So, the stack can be used for pushing these nodes and then you can pop the nodes from the stack and color them as we did before. So, this is one coloring possible. There are many colorings which can be seen to happen. Three registers are sufficient and no spills are necessary in this particular code. Nothing more to it in this example.

(Refer Slide Time: 28:26)



Example(continued)

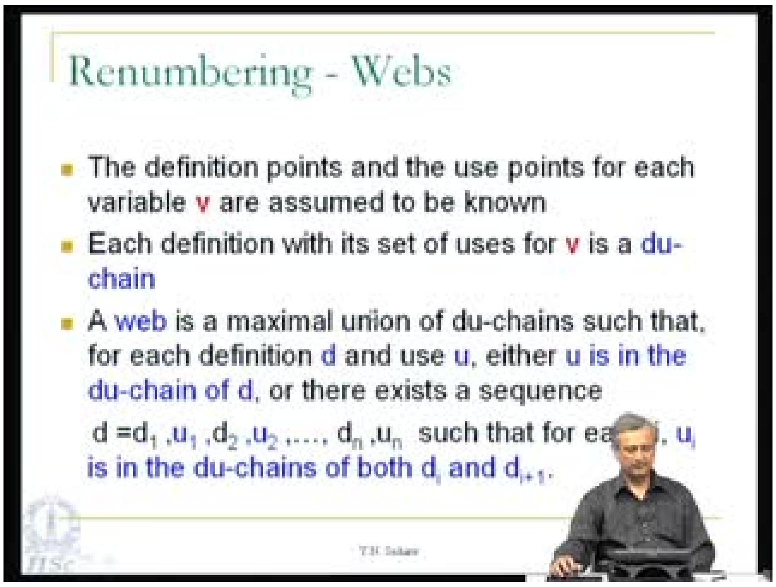
Final register allocated code

```
r1 = 2
r2= 4
r3= r1+r2
r3= r1+1
r1= r1 *r2
r2= r3+r2
```

Three registers are sufficient for no spills

Y.N. Srikant

(Refer Slide Time: 28:33)



Renumbering - Webs

- The definition points and the use points for each variable v are assumed to be known
- Each definition with its set of uses for v is a du-chain
- A web is a maximal union of du-chains such that, for each definition d and use u, either u is in the du-chain of d, or there exists a sequence $d = d_1, u_1, d_2, u_2, \ldots, d_n, u_n$ such that for ea[illegible], u_i is in the du-chains of both d_i and d_{i+1}.

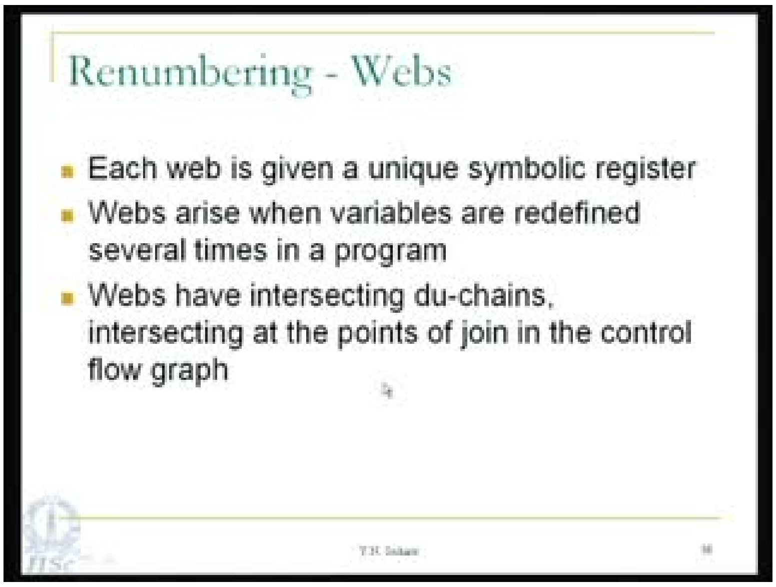
Y.N. Srikant

Now, let us see what webs are. These are slightly more complicated than live ranges. The definition points and the use points for each variable v are assumed to be known. That means the du-chain. Each definition with its set of uses for a variable v is called as a du-chain or definition use chain. A du-chain is a very useful data structure, which will be used later for many optimizations also and it is useful as well.

So, what is a web? A web is a maximal union of du-chains, such that for each definition d and use u, either u is in the du-chain of d, so then it is a web or there exists a sequence d equal to d1, u1, d2, u2 etcetera dn, such that for each i, ui is in the d u chains of both d i and d i plus 1, so, you are looking at this u1, it is in the d u chain of d1 and also d2. If you look at d2, it is in the d u chain of d2 and d3 etcetra. Except for this un, every one of the others will have du-chains before and after.

So, we are going to collect all such du-chains, such that the usage is in the previous du-chain and the succeeding du chain and such a maximal union is called as a web.

(Refer Slide Time: 30:18)



Renumbering - Webs

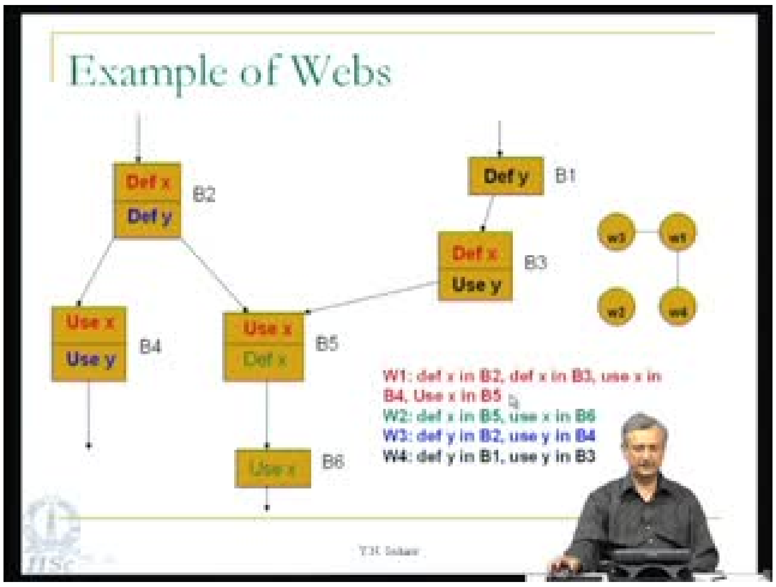
- Each web is given a unique symbolic register
- Webs arise when variables are redefined several times in a program
- Webs have intersecting du-chains, intersecting at the points of join in the control flow graph

Y.N. Srikant

So, each web is given a symbolic register, which is unique and webs arise when variables are redefined several times in a program. That we will see very soon. Webs have intersecting du-chains intersecting at the points of joint in the control flow graph. Here is an example.

(Refer Slide Time: 30:36)

Example of Webs



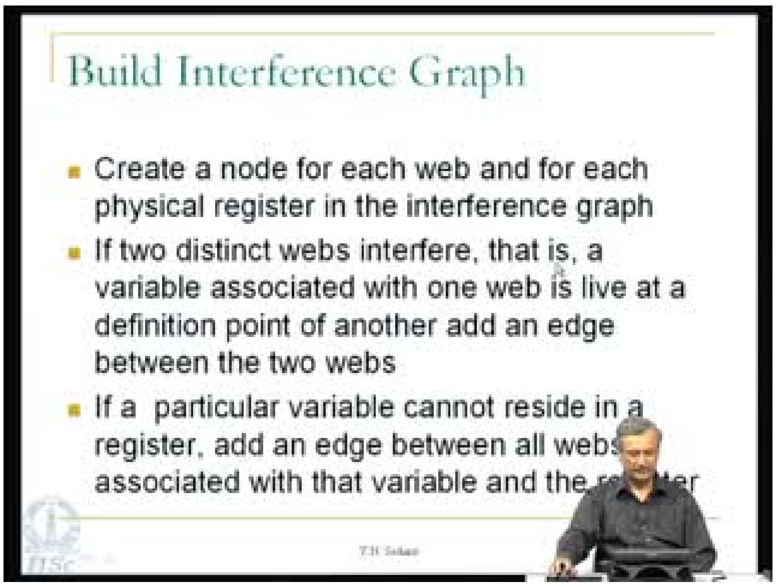
W1: def x in B2, def x in B3, use x in B4, Use x in B5
W2: def x in B5, use x in B6
W3: def y in B2, use y in B4
W4: def y in B1, use y in B3

Y.N. Srikant

So, there is a definition of the variable x here, there is a definition of the variable x here also, either this or this will be used here depending on the control flow. So, these are the two definitions used. Chain of this definition will consist of this and this, whereas, the definition chain of this will consist of only this, but you can see that these intersect. So, w1, the web number 1 is def x in B2, def x in B3, use x in B4, and use x in B5. So, all these four together form a single web.

Similarly, w2 consists of def x in B5 and use x in B6 - just this simple du-chain. w3 consists of def y in B2, and use y in B4, so again a very simple du-chain. That is a web on its own. w4 is this - def y in B1, and use y in B3. So, if you place these webs as nodes and then you can look at the interference between these webs also. For example, w1 and w3 interfere with each other; so w1 is the red one and w3 is the blue one. What it says is, these two are overlapping each other, so, it says you cannot give the same register to both w1 and w3. Similarly, w1 and w4 also interfere. w1 is the red one, and w4 is the black one. These two also have overlapping, you know overlapping webs, so, these two are active at the same time, so we cannot give the same register to y and x. Then, there is no interference between w2 and w3. w2 is the green one ,and w3 is the blue one. These two do not interfere. So, they can be given the same register as well. That is what is indicated by this interference graph. Once we construct the d u chains, and then construct the interference graph from the webs, then rest of the colouring algorithm is identical as before.

(Refer Slide Time: 33:25)



Build Interference Graph

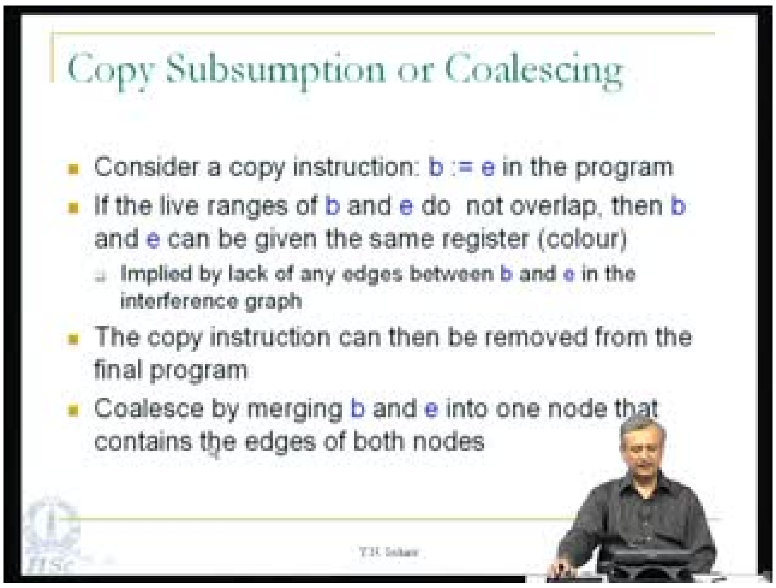
- Create a node for each web and for each physical register in the interference graph
- If two distinct webs interfere, that is, a variable associated with one web is live at a definition point of another add an edge between the two webs
- If a particular variable cannot reside in a register, add an edge between all webs associated with that variable and the register

Y.N. Srikant

Building the interference graph. Create a node for each web, and for each physical register in the interference graph, if two distinct webs interfere, that is a variable associated with one web is live at a definition point of another, add an edge between the two webs. So, it is very important. A variable associated with one web is live at a definition point of another. This is a very important thing that you need to keep in mind, when the interference is considered.

If a particular variable cannot reside in a register, add an edge between all webs associated with that variable and the registers. So, these are the restrictions on physical registers not being allowed to reside in other in a particular physical register.

(Refer Slide Time: 34:22)



Copy Subsumption or Coalescing

- Consider a copy instruction: b := e in the program
- If the live ranges of b and e do not overlap, then b and e can be given the same register (colour)
  - Implied by lack of any edges between b and e in the interference graph
- The copy instruction can then be removed from the final program
- Coalesce by merging b and e into one node that contains the edges of both nodes

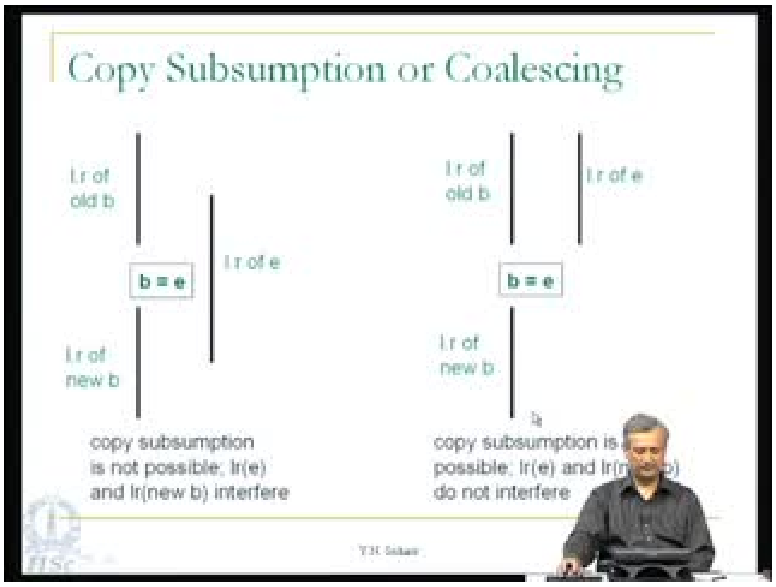
Y.N. Srikant

We have built the graph now, so we come to the stage where there is copy Subsumption or Coalescing. Let us understand what this is? Consider an instruction such as b equal to e, this is known as copy instruction, and we are copying the value of e into b.

If the live ranges of b and e do not overlap, we will see an example of how this can happen. b and e can be given the same register, same color. This is implied by the lack of any edges between b and e in the interference graph.

So, now the copy instruction can be removed and we can merge the two nodes b and e into one node and the edges corresponding to both of them will be attached to this particular node. Let us see an example.

(Refer Slide Time: 35:19)



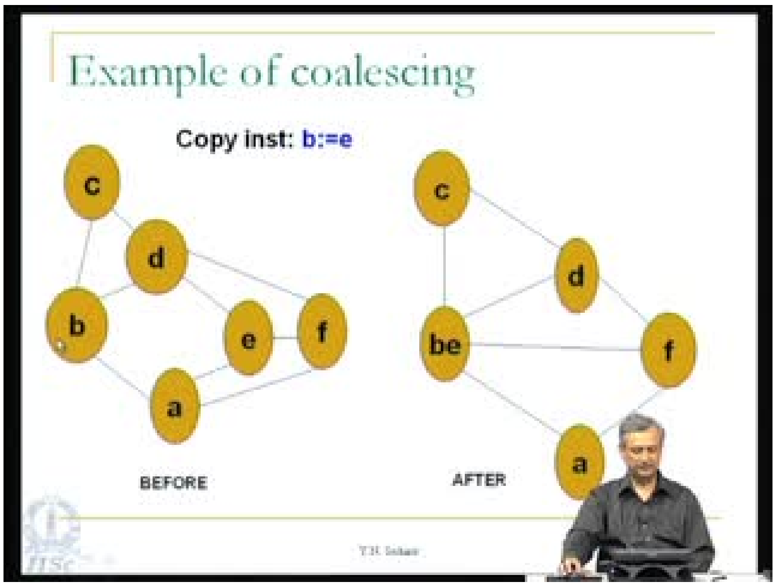
copy subsumption is not possible; lr(e) and lr(new b) interfere

copy subsumption is possible; lr(e) and lr(new b) do not interfere

Here is the copy instruction, b equal to e, this is the live range of b, before this copy instruction, let us call it as the old b. Now b gets a new value. This is the live range of the new b. This is the live range of e. So, e is active throughout here. It is very easy to see that this new live range of new b and the live range of e interfere. Both are active at the same time, therefore, we cannot merge these two. There is no way you can merge these two. There will be an edge between these two live ranges.

Whereas, if the situation were to be like this, this is the old b live range and the live range of e ends here; it does not continue anymore. There are no more uses, this is the last usage of e and then onwards there is no more usage of e. In other words, here is e then it becomes b. So, there is no overlap between these two, the new b and the e. So, we can assign the same register to these two and there would be no edge between these two, because there is no interference.

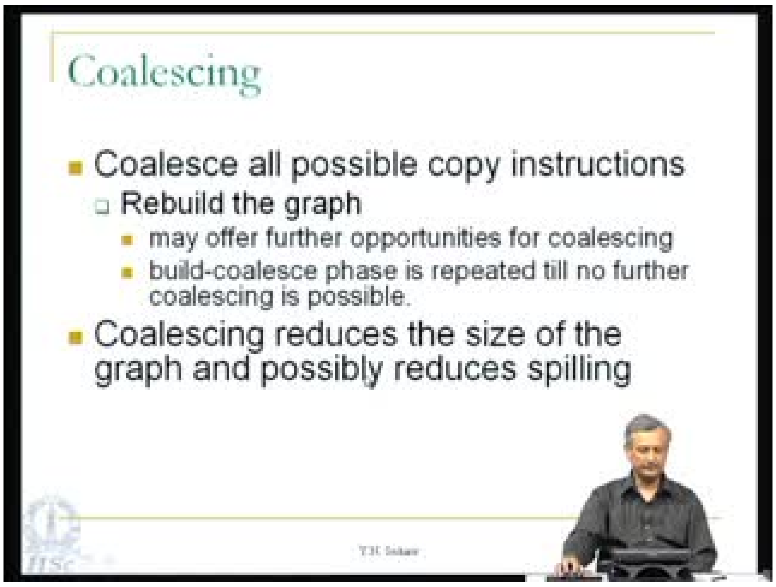
(Refer Slide Time: 36:37)



That is what is shown here. Here is b, here is e, there is no edge between these two and there is indeed, b equal to e, a copy instruction in the source code. If there were to be no copy instruction in the source code, then, there is no point in trying to merge these two. This is possible only when we have a copy instruction. So, there is a copy instruction and between b and e, there is no edge, so we can merge these two into a single node called b e. So, all the edges which go into b that is from d and from a, so from d and from a go to b, and all the edges which are going into e, that is from f, from d ,and a, will now go into b e, so, f goes into b e, the other two are already there.

So, this is known as copy coalescing. After this, if there were more copy instructions, they are again removed and we try to reduce the graph as far as possible by merging nodes. Once we cannot do any more merging, we proceed to the next step.

(Refer Slide Time: 37:44)

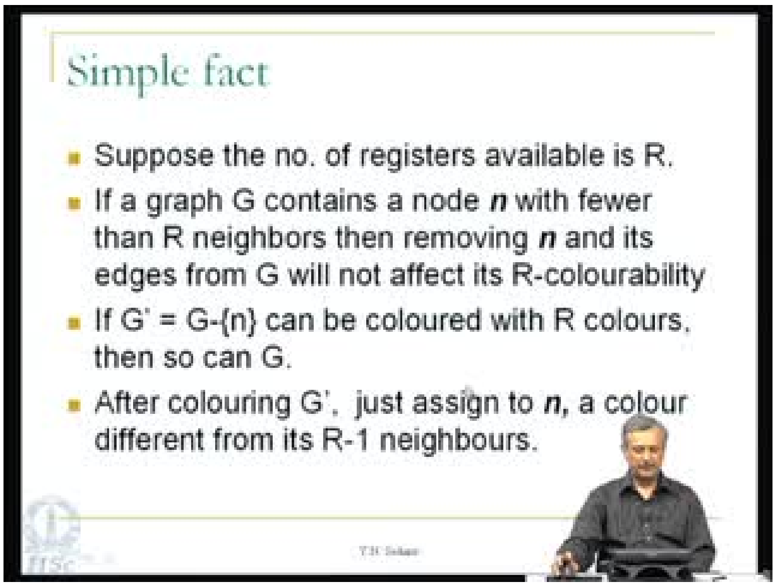


Coalescing

- Coalesce all possible copy instructions
  - Rebuild the graph
    - may offer further opportunities for coalescing
    - build-coalesce phase is repeated till no further coalescing is possible.
- Coalescing reduces the size of the graph and possibly reduces spilling

Coalesce all possible copy instructions, rebuild the graph, this may offer further opportunities for coalescing. So, the build coalesce phase is repeated till no further coalescing is possible. So, coalescing reduces the size of the graph and possibly reduces spilling as well. Now, since it reduces the size of the graph, possibly, you know it is just a hunch there is a need to probably spill fewer nodes.

(Refer Slide Time: 38:17)



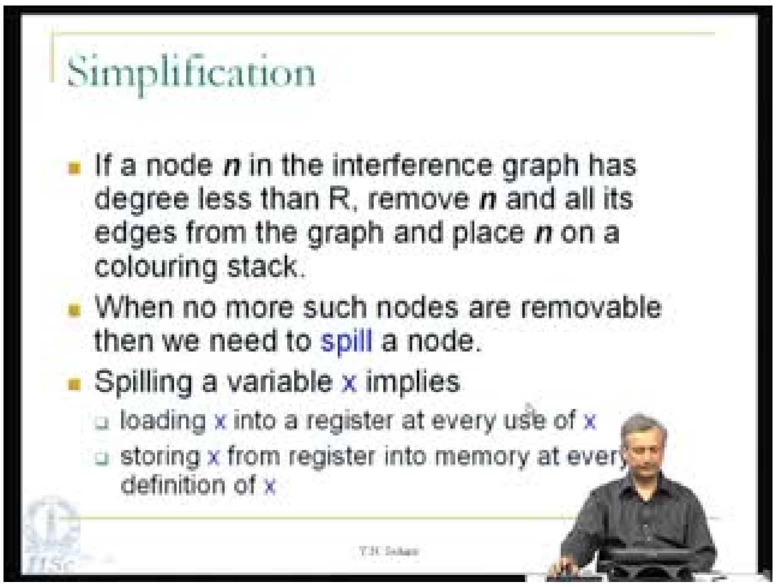
Simple fact

- Suppose the no. of registers available is R.
- If a graph G contains a node ***n*** with fewer than R neighbors then removing ***n*** and its edges from G will not affect its R-colourability
- If G' = G-{n} can be coloured with R colours, then so can G.
- After colouring G', just assign to ***n***, a colour different from its R-1 neighbours.

Here is a simple fact regarding the colouring. Suppose the number of registers available is R, if a graph G contains a node n with fewer than R neighbors, then removing n and its edges from G will not affect its R colourability. This is a very profound result. This is a result on which the colouring algorithm is based. This is the reduction, so, a node n with fewer than R neighbors, n is removed and then the colouring of G will not be affected. Its colourability is not affected.

So, if G prime equal to G minus n can be coloured with R colours, can G be actually demonstrated? How is it possible? After colouring G prime, just assign to a colour different from its R minus one neighbors, so that is over. Since the degree of the node n which was removed is less than R, there is a colour which is available and that can be assigned to n, so, that is how this is justified.

(Refer Slide Time: 39:36)



Simplification

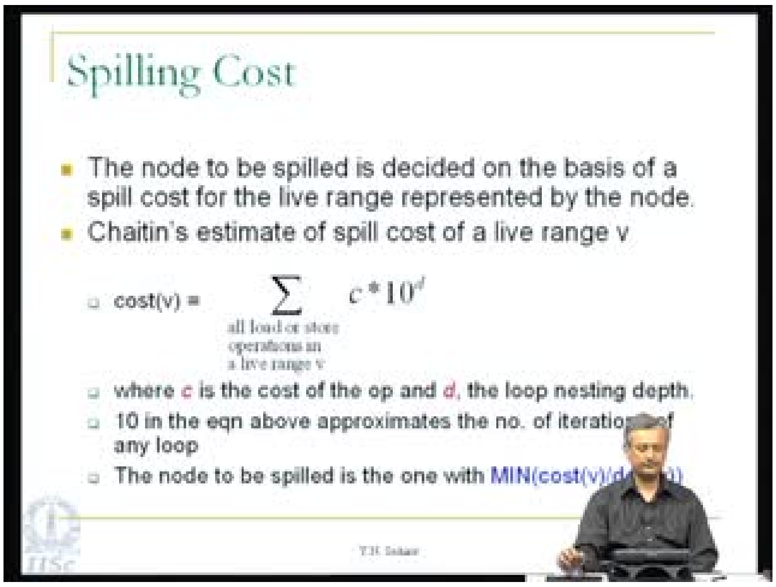
- If a node *n* in the interference graph has degree less than R, remove *n* and all its edges from the graph and place *n* on a colouring stack.
- When no more such nodes are removable then we need to spill a node.
- Spilling a variable x implies
  - loading x into a register at every use of x
  - storing x from register into memory at every definition of x

Y.N. Srikant

So, what is the simplification? If a node in the interference graph has degree less than R, remove n and all the edges from the graph and place n on a colouring stack. This is something we already studied.

When no more such nodes are removable from the graph, we need to spill a node. So, all the nodes have degree equal to R or greater than R. Spilling a variable x. What does it really imply? So, spilling implies that you load the value of x into a register every time we want to use x. Storing the value from register into memory at every definition of x. If there are many uses and definitions of x, at every point that variable will have to be either loaded or stored from memory. So, there is a penalty associated with such spilling. So, we have to be careful on what we spill, how we spill, and so on and so forth.

(Refer Slide Time: 40:38)



Spilling Cost

- The node to be spilled is decided on the basis of a spill cost for the live range represented by the node.
- Chaitin's estimate of spill cost of a live range v
  - cost(v) = $\sum_{\substack{\text{all load or store} \\ \text{operations in} \\ \text{a live range v}}} c*10^d$
  - where c is the cost of the op and d, the loop nesting depth.
  - 10 in the eqn above approximates the no. of iteratio f any loop
  - The node to be spilled is the one with MIN(cost(v)/d)

Y.N. Srikant

So, how do we compute the spilling cost? The node to be spilled is decided on the basis of a spill cost for the live range represented by the node. Chaitin's estimate of the spill cost of a live range is here. Cost of v is sigma of c into 10 to the power d. What is the sigma? It is overall load or store operations in a live range v.

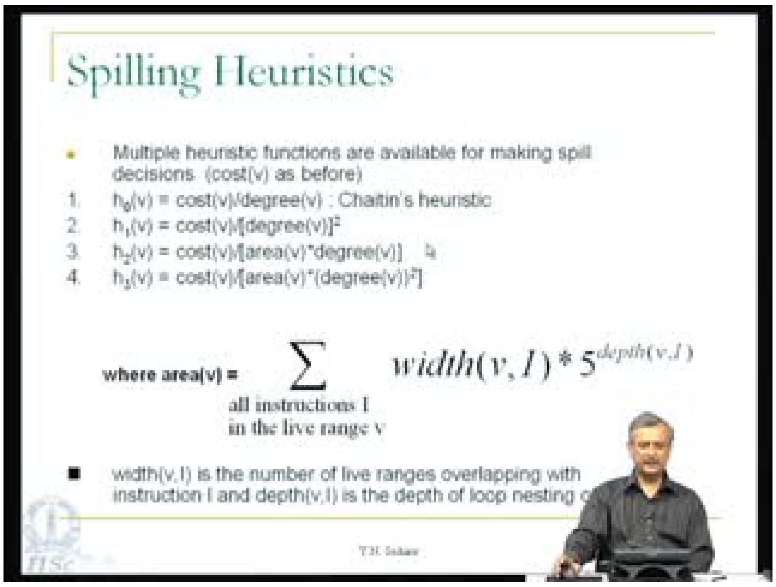
So, why are we doing this? I already mentioned that spilling implies loading from memory or storing into memory at every usage or definition point of the variable in the live range or in the web. Now, consider all the load and store operations in a live range, the cost of c is the cost of the operation, load or store. D is the loop nesting depth. So, obviously if the loop iterates only once, then the cost will be computed only once for each usage of or definition of the variable. If the loop iterates twice, then we have to do it twice and so on. Now, the question is how do we know how many times the loop iterates. We may not know this.

The equation uses 10 as the approximation for the number of iterations of any loop. How is this justified? The point is, if there is a loop, we just want to make the cost of loads and source inside the loop much higher than the cost of loads and stores outside the loop.

So, for each nesting of the loop, if there is a nesting of only one, then the cost becomes 10 times that of outside. If there is a nesting of two, the inner most loop, let us say is inside two loops, then d would have become two. So, the cost now becomes 100 for the inner loop, for the load and store operations inside the inner loop. Now, there is a difference of 10 for each nesting level. This you know in practice has been shown to take adequately into consideration the variation in cost at the outer level and the inside level. People have also observed that making this 10 as 100 or even replacing this by the exact value of the number of iteration of the loop does not really change the spilling or colourability of the graph.

So, the node to be spilled is the one with minimum cost per degree for every node. We compute this cost for every live range or node, and then compute the minimum cost per degree, degree v is also known. What is the degree? For example, it interferes with many live ranges. There are as many edges as the number of live ranges which interfere with it. So, if you remove this particular node, then a number of other nodes which interferes with it will be relieved. So, we want to pick that particular node, which will help many other nodes. That is why, the degree is in the denominator and we want to compute the minimum of cost over degree.

(Refer Slide Time: 44:57)



Spilling Heuristics

- Multiple heuristic functions are available for making spill decisions. (cost(v) as before)

1. $h_0(v)$ = cost(v)/degree(v) : Chaitin's heuristic
2. $h_1(v)$ = cost(v)/[degree(v)]^2
3. $h_2(v)$ = cost(v)/[area(v)*degree(v)]
4. $h_3(v)$ = cost(v)/[area(v)*(degree(v))^2]

$$\text{where area(v)} = \sum_{\substack{\text{all instructions } I \\ \text{in the live range } v}} width(v, I) * 5^{depth(v,I)}$$

- width(v,I) is the number of live ranges overlapping with instruction I and depth(v,I) is the depth of loop nesting o

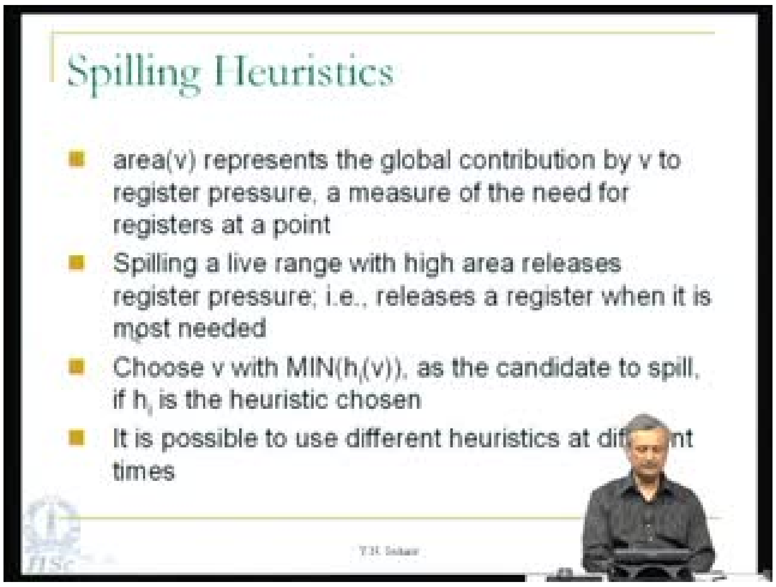
Y.N. Srikant

This is not the only way to pick up a node for spilling. There are many heuristics which are reported in the literature. Multiple heuristic functions are available for making spill decisions. Cost v is as before. For example, h0 v is cost per degree which is the Chaitin's heuristic, $h_{1,}$ the function h_1 v, is cost v by degree v whole square. People have found that instead of degree you use degree whole square it works slightly better. Again these are heuristics and can only be justified empirically.

The third one is h2 v, h2 v is cost v divided by area v into degree v. So, this needs some explanation. The fourth one, h3 v equal to cost v divided by area v into degree v whole square, so, this thing multiplied again squared. What is area? Area v is width of v comma I multiplied by 5 to the power depth of v comma I. What is I? All instructions I in the live range v, so, the point is, this 5 is a replacement for 10 in the previous equation and depth is the value of the nesting depth of the instruction I in the live range v. What is width here? Width v,I is the number of live ranges overlapping with instruction I. We are looking at each live range and each instruction in the live range v, so, look at the number of live ranges overlapping at that particular instruction. It could become different in each case.

So, the width v I is the number of live ranges overlapping with instruction I and depth v I is the depth of loop nesting of the instruction I in the range v.

(Refer Slide Time: 47:02)



Spilling Heuristics

- area(v) represents the global contribution by v to register pressure, a measure of the need for registers at a point
- Spilling a live range with high area releases register pressure; i.e., releases a register when it is most needed
- Choose v with MIN(h_i(v)), as the candidate to spill, if h_i is the heuristic chosen
- It is possible to use different heuristics at dif[illegible]nt times

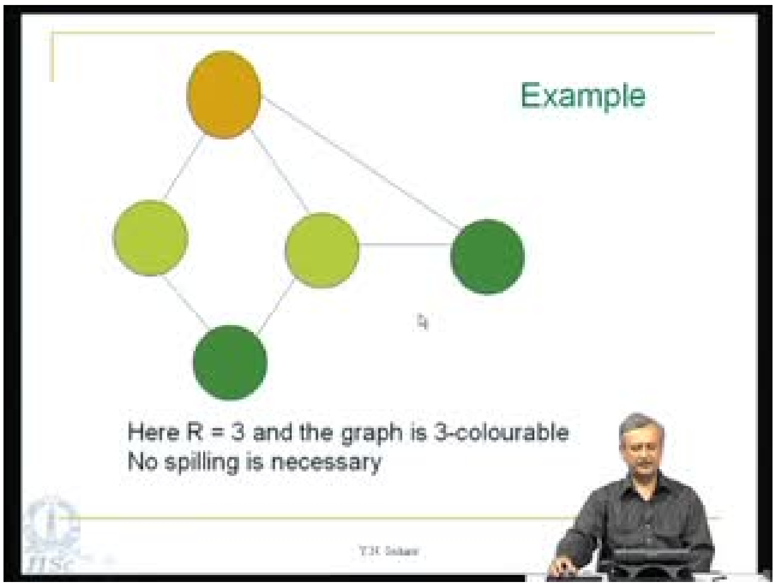
Y.N. Srikant

There is some explanation necessary for area v. What is area v? Area v represents the global contribution of v to the register pressure. What is a register pressure? It is a measure of the need for registers at a point. You see there are many registers, which are used at a point and some points in the program require many registers, some other points require very few registers. So, the need for registers at a particular point is the register pressure.

So, this area v represents how much is the contribution by v to register pressure and obviously, spilling a live range with a very high area releases register pressure, that is, it releases a register, when it is most needed. So, this is very useful, obviously. Choose v with minimum hi v as the candidate to spill, if h i is the heuristic chosen. We can also use different heuristics at different points. Why are we using minimum again? You know the area part is in the denominator of the heuristic function. So, higher the area, the value of hi v will be smaller. If we choose a node with very high area, then it obviously corresponds to choosing the node with minimum hi of v.

So, we can actually use different heuristics at different points in time absolutely no problem as far as heuristics are concerned.

(Refer Slide Time: 50:09)



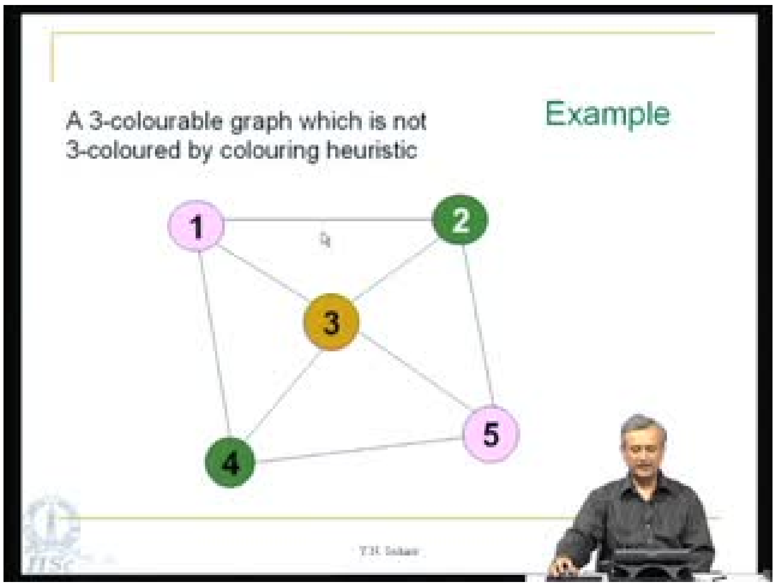
Example

Here R = 3 and the graph is 3-colourable
No spilling is necessary

It is also seen that a single heuristic actually does not serve all programs. People actually tend to evaluate the heuristics within the program, within the compiler, then depending on the heuristic which gives the minimum value, that particular heuristic is used at that point in the program. At a different point in the program, a different heuristic is employed.

Here is an example. This is R equal to 3 and the graph is three colourable. We already saw before and no problem as far as colouring is concerned.

(Refer Slide Time: 50:55)



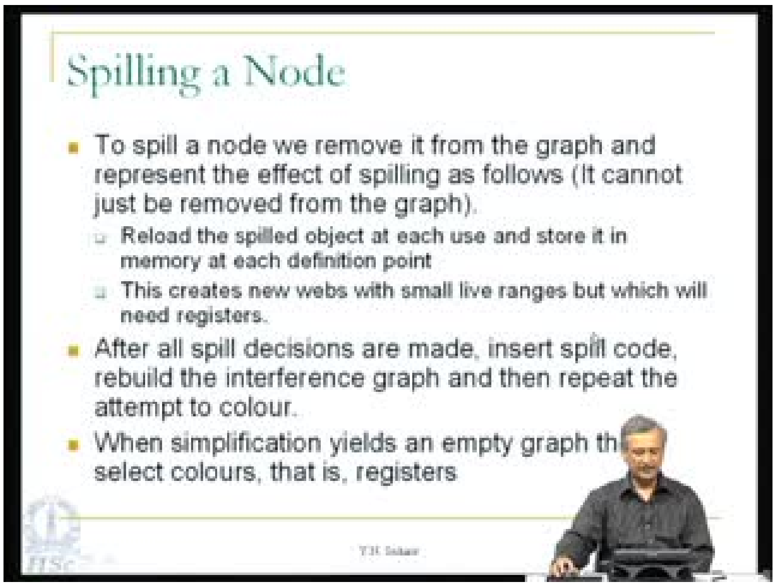
A 3-colourable graph which is not 3-coloured by colouring heuristic

Example

Here is a very interesting example. This is a three colourable graph, but this cannot be three-coloured by colouring heuristic. You can easily see that, this is three colourable and the nodes have been assigned values; so that, only three colours are used. Just look at the graph structure, these four nodes have degree 3. This particular node has degree 4. So, if you have only three colors available to you, you will be forced to spill one of the nodes. It is not possible to colour this particular graph without any spilling, if we use the algorithm due to Chaitin.

Here is where, others have improved the Chaitin's colouring heuristic and those actually will be very helpful in colouring such graphs with three colours.

(Refer Slide Time: 52:04)



Spilling a Node

- To spill a node we remove it from the graph and represent the effect of spilling as follows (It cannot just be removed from the graph).
  - Reload the spilled object at each use and store it in memory at each definition point
  - This creates new webs with small live ranges but which will need registers.
- After all spill decisions are made, insert spill code, rebuild the interference graph and then repeat the attempt to colour.
- When simplification yields an empty graph th select colours, that is, registers

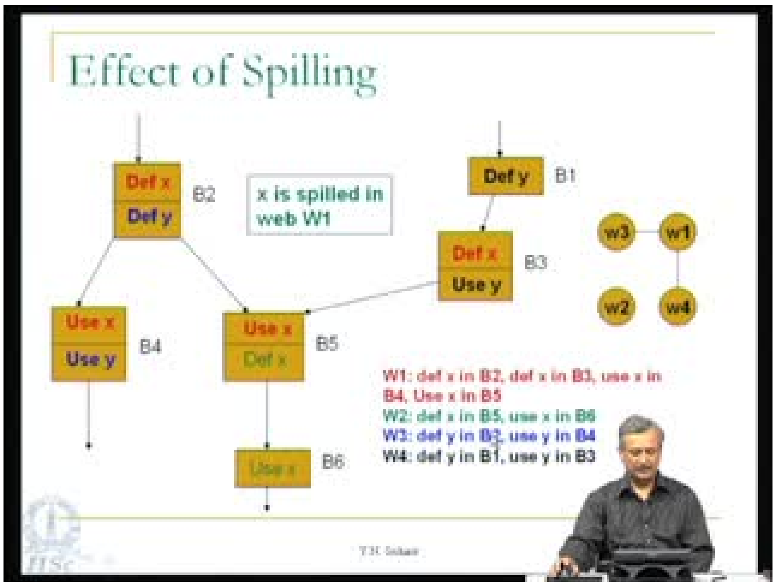
Y.N. Srikant

So, what exactly does spilling a node imply? To spill a node, we need to remove it from the graph and represent the effect of spilling as follows. We just cannot remove it from the graph and be done with it. It also implies as I already told you, reloading the spilled object at each use and we need to store it in memory at every definition point. So, there is a lot of load and store happening in the live range.

So, this set of loads and stores actually create new webs. We will see what happens. Live range and web has been used interchangeably in my lecture now; because once you create the webs and the interference graph, there is no difference as far as algorithm is concerned. So, the spilling creates new webs with small live ranges, but which will need registers. After all the spill decisions are made, then we need to insert the spill code physically. So, spill decisions have been made; but we have not inserted spill code. We just removed the node from the graph and went ahead with the reduction of the graph. We did not introduce any spill code, but instructions have to be physically inserted into the code. So, insert spill code, now the program has changed completely. We rebuild the interference graph and then repeat the procedure and attempt it to colour again.

So, when the simplification yields an empty graph, then select registers. This is the process of spilling. You know this is a cycle. It says try to reduce the graph. At some point, you cannot do so. Now spill a node, but do not insert physical code right now, continue with your reduction, so, at some point, the spilling and reduction yields an empty graph. Now at that point, you actually have to insert the spill code, recolour, rebuild the interference graph and so on and so forth.

(Refer Slide Time: 54:36)



Effect of Spilling

W1: def x in B2, def x in B3, use x in B4, Use x in B5
W2: def x in B5, use x in B6
W3: def y in B2, use y in B4
W4: def y in B1, use y in B3

We will stop at this point and in next lecture, we will study the effect of spilling on webs and see how it affects colouring. Thank you!