

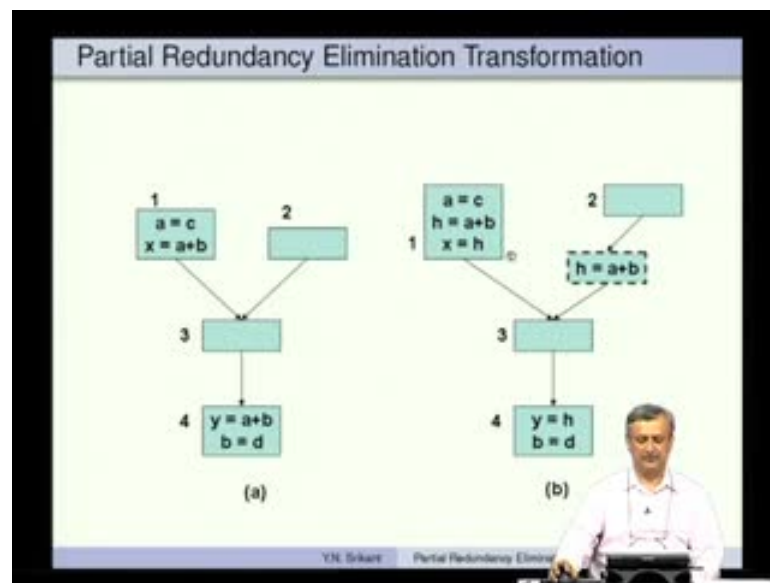
**Compiler Design**  
**Prof. Y. N. Srikant**  
**Department of Computer Science and Automation**  
**Indian Institute of Science, Bangalore**

**Module No. # 12**

**Lecture No. # 20**

**Partial Redundancy Elimination**

(Refer Slide Time: 00:19)

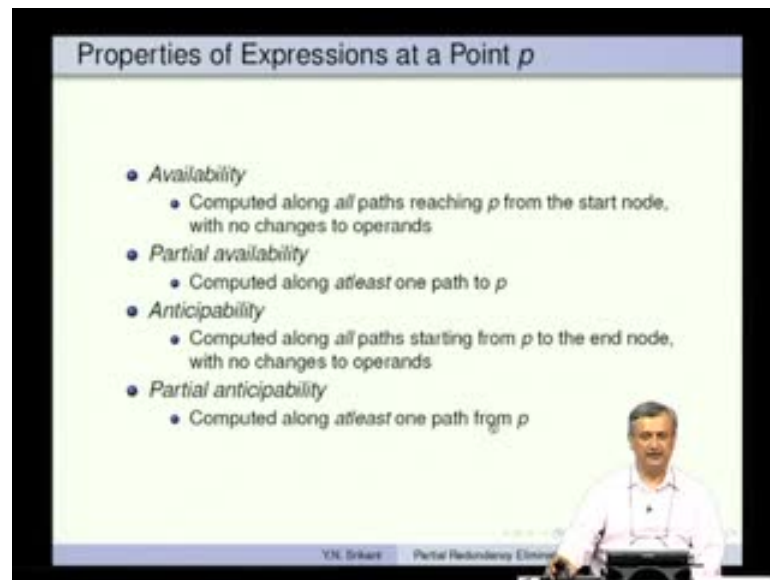


Welcome to the lecture on partial redundancy elimination. In the last lecture, we looked at an example of what exactly is PRE. Just to recapture it little bit. Here is an expression - a plus b and here is the same expression a plus b again and the question is, can we use the computation of a plus b in this basic block 1 and avoid re-computation of a plus b in basic block 4? Under certain circumstances it is possible (Refer Slide Time: 00:37).

So please observe that, the expression a plus b is not completely available at this point (Refer Slide Time: 01:00). It is available partially along this path but, it is not available along this path. That is why this is a partial redundancy; it is not a full redundancy. If it is full redundancy, we can simply apply global common sub expression elimination but, since it is partial redundancy, we have to insert a computation h equal to a plus b along this arc by breaking it.

Now, the expression  $a + b$  is fully redundant at this point, it is available along this path and also this path. We can replace this  $a + b$  here by  $h$  (Refer Slide Time: 01:38); where  $h$  is the variable you were know, storing the value of  $a + b$  along these two paths.

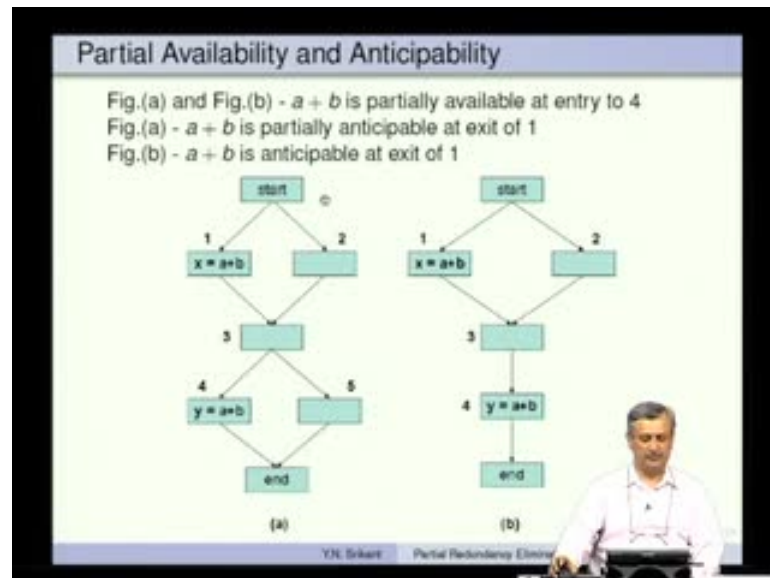
(Refer Slide Time: 01:50)



Now, we were looking at properties of expressions at a point. So availability is a well-known property. We already have learnt about it, so it says computed along all paths reaching  $p$  from the start node with no changes to operands.

Partial availability says, computed along at least one path. So, that is the difference not all paths only at least one path. Anticipability, we have studied this as well under very busy expressions, computed along all paths starting from  $p$  to the end node. So it is useful later on, with no changes to operands. Partial anticipability would be computed along at least one path from  $p$  to the end of the flow graph.

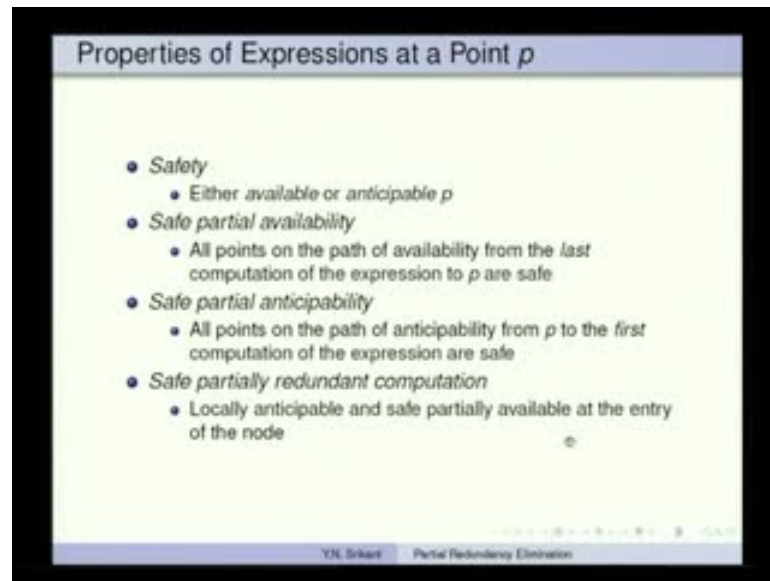
(Refer Slide Time: 02:41)



Here is an example of both: so in a and b, in this two figures (Refer Slide Time: 02:50),  $a + b$  is partially available at entry to 4. Here it is partially available because from here you can use this path and it becomes partially available and for this point 4 we can use this path and it is partially available, it is not fully available in either of that.

In a,  $a + b$  is partially anticipable at exit of 1. So exit of 1 is here, so if we take this path - it is computed and if we take this path - it is not computed. So that is why it is partially anticipable and in figure b;  $a + b$  is anticipable at exit of 1 again; so is fully anticipable because from here it does not matter there is only one path to take and then that path  $a + b$  is computed.

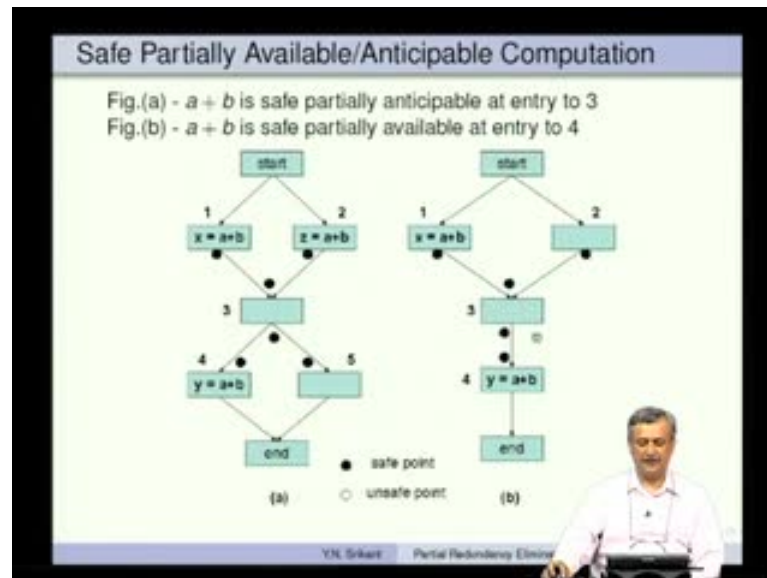
(Refer Slide Time: 03:34)



There are more expression properties: one is safety - this is very important for us. It is defined as either available or anticipable at  $p$  and safe partial availability is you take the path of availability and from the last computation of the expression to  $p$  all the points must be safe.

Remember, for safe partial availability you do not have to look at the path from the start node to  $p$ , and make sure that all these are safe; no, that is not the point. Point is, from the last computation of the expression to the point  $p$  and the safe partial anticipability is similar, all points on the path of anticipability from  $p$  to the first computation of the expression are safe. What is a safe partially redundant computation? It must be locally anticipable that is computed within the basic block and safe partially available at the entry to the node.

(Refer Slide Time: 04:39)



Let us look at some examples. So,  $a + b$  is safe partially anticipable at entry to 3, so the dark spots are all safe points. Let us see, why they are safe.

Safety is either availability or anticipability; so at this point, it is safe because it is available, it is computed here and it becomes immediately available at this point. So, we do not have to worry about anticipability. Of course, it is not anticipable, it is only partially anticipable. The same is here,  $a + b$  is computed here and it is available right here (Refer Slide Time: 05:10).

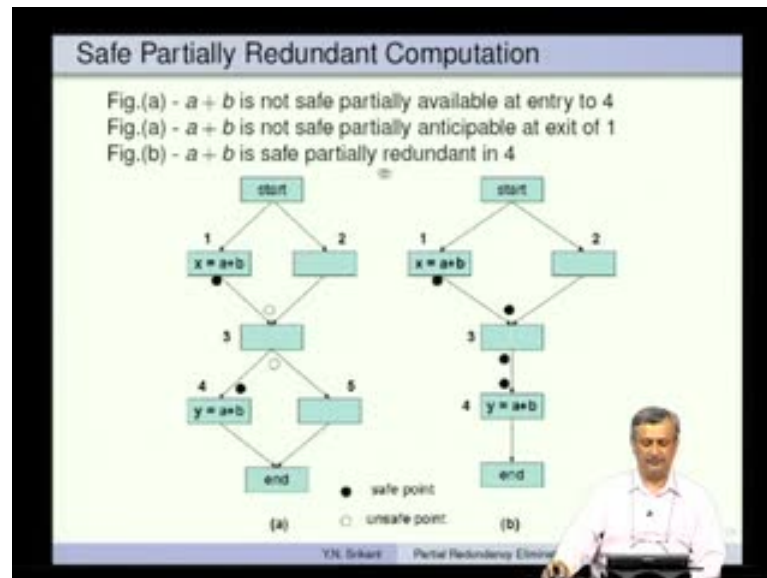
At this point (Refer Slide Time: 05:14), it is available along these two paths, so it is available. At this point same thing, it is available along both the paths and at this point it is available along both the paths and of course, it is anticipable also. At this point again, it is available along all the paths.

So even though, anticipability is not true in many cases, the availability makes these points safe but, remember this point is not really consider we do not have to worry too much about this particular point (Refer Slide Time: 05:50). Even this will be safe, simply because even though availability is false, the anticipability is true along both this paths; it is being computed, so this point will also be really safe.

Now in figure b,  $a + b$  is parse safe partially available at entry to 4; so all these points are safe, now  $a + b$  is partially available because of this point, this is the first

computation of the expression; along this path of safety and all these points are safe from here this, this, this and this (Refer Slide Time: 06:26) and therefore, it is safe partially available at entry to 4.

(Refer Slide Time: 06:42)



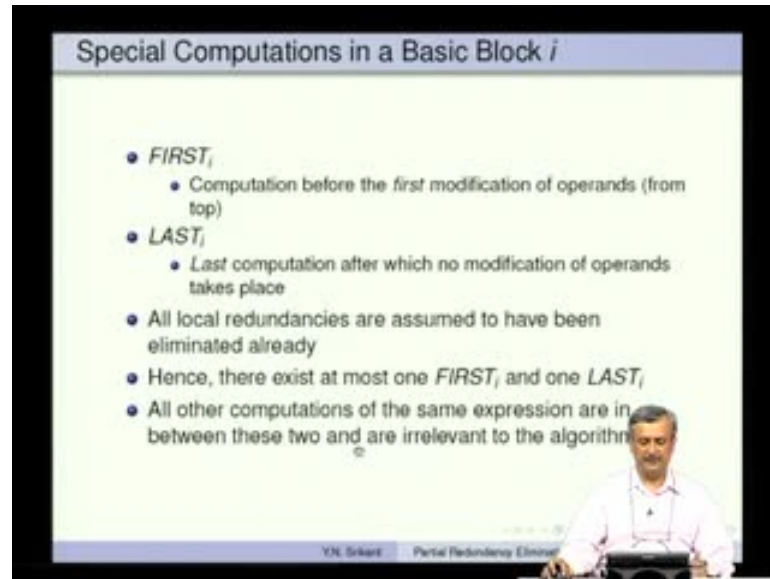
Here is an example to show, what is not safe partially available. So,  $a + b$  is not safe partially available at entry to 4. Take this example; the computation here from the previous example is missing (Refer Slide Time: 06:56). Here, this point is safe because of availability but, this point is not safe because neither available, nor anticipable; only partial availability and anticipability is shown at this point.

This point is similarly unsafe but, this is safe because anticipability is true right here, the expression is being computed, but we want all the points from this first computation to this point. All these 4 points to be safe, they are not; therefore, this expression is not safe partially available at entry to 4.

In the same figure, it is not safe partially anticipable at exit of 1. So exit of 1 and then the computation is here; so, all these points should have been safe, they are not and therefore, it is not safe partially anticipable either.

In figure b,  $a + b$  is safe partially redundant in 4. So, this computation is safe partially redundant, it is locally anticipable (Refer Slide Time: 08:00); so it is computed locally and safe partially available along this path. So, it is safe partially redundant.

(Refer Slide Time: 08:14)



The slide is titled "Special Computations in a Basic Block  $i$ ". It contains the following bullet points:

- $FIRST_i$ 
  - Computation before the first modification of operands (from top)
- $LAST_i$ 
  - Last computation after which no modification of operands takes place
- All local redundancies are assumed to have been eliminated already
- Hence, there exist at most one  $FIRST_i$  and one  $LAST_i$
- All other computations of the same expression are in between these two and are irrelevant to the algorithm

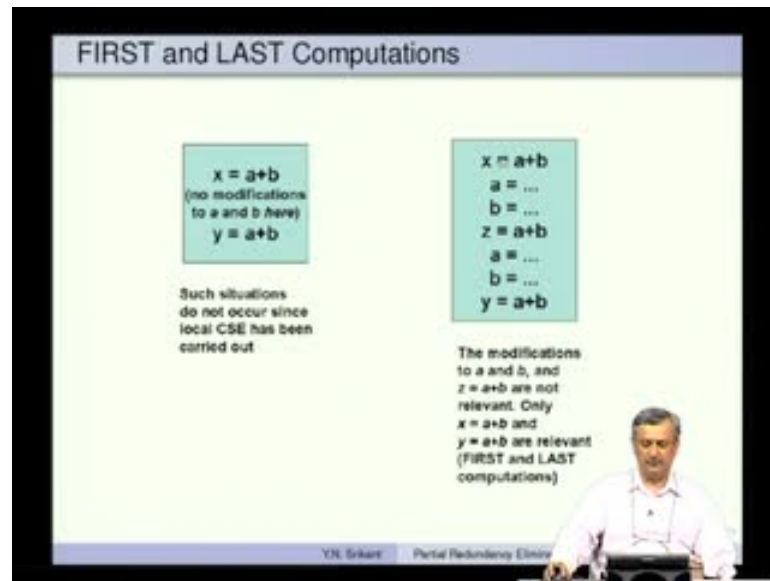
In the bottom right corner of the slide, there is a small inset image of a man with grey hair, wearing a light-colored shirt, sitting at a desk. At the bottom of the slide, there is a footer that reads "Y.N. Srikant Partial Redundancy Elimination".

There are some special computations in a basic block that we need to take note. One of them is the first computation and the other one is the last computation. The first computation, as the name indicates is the computation before the first modification of operands from the top. We have not modified the operands, they take values which were given to them before and then you evaluate the expression.

The last computation in the basic block  $i$  is last computation after which no modification of operands takes place; so you do not modify them, the same value is going to be available at the output of the basic block.

So, all the local redundancies are assumed to have been eliminated already; so we will see this very soon. Hence there exist at most, one FIRST and one LAST computation for a basic block, all other computations of the same expression or in between these two computations and are irrelevant.

(Refer Slide Time: 09:25)



Here is a pictorial explanation of what I just now told you,  $x$  equal to  $a$  plus  $b$  is here, then there are no modifications to  $a$  and  $b$ , then you have  $y$  equal to  $a$  plus  $b$ . Such a situation cannot arise simply because now, this  $a$  plus  $b$  is a common sub expression; we can use  $x$  here, so we are assuming that local CSE has been carried out. This situation will never arise; we would have added actually  $y$  equal to  $x$  here.

Take the other situation, you have  $x$  equal to  $a$  plus  $b$ , then you have modified  $a$  and  $b$ , then you have  $z$  equal to  $a$  plus  $b$  another computation of  $a$  plus  $b$ , again you have modify  $a$  and  $b$ , then you have  $y$  equal to  **$a$  plus  $b$** , third computation of  $a$  plus  $b$ .

What is relevant was just,  $x$  equal to  $a$  plus  $b$  and  $y$  equal to  $a$  plus  $b$ . The reason is before this  $a$  plus  $b$  operands are not modified and after this  $a$  plus  $b$  operands are not modified (Refer Slide Time: 10:26). So, this expression is anticipable here and this expression is available at this point and in-between if there is a computation we need to do it, we will have to evaluate  $a$  plus  $b$  as many times as necessary because operands have been modified. That is why these are not relevant to us, only the first and the last first computations are relevant to us.



(Refer Slide Time: 10:51)

The slide is titled "Outline of the Algorithm" and contains the following text:

Our *PRE* algorithm identifies all safe *PRCs* and makes them totally redundant by suitable insertions

- 1 Compute the predicates,  $AV_i$ ,  $ANT_i$ ,  $SAFE_i$ ,  $SPAV_i$ , and  $SPANT_i$  at entry and exit points of all nodes
- 2 Mark all points which have both  $SPAV$  and  $SPANT$  true and consider the paths formed by connecting such adjacent marked points
- 3 Insertion points: just before *LAST* in starting points of these paths
- 4 Insertion edges: those that enter junction nodes on these paths
- 5 Replacements are for *LAST* and *FIRST* computation the starting and ending points of these paths

At the bottom of the slide, there is a small video inset showing a man in a white shirt speaking. The text "Y.N. Srikant" and "Partial Redundancy Elimination" is visible at the bottom of the slide.

Here is our algorithm in an informal manner, it identifies all the safe partially redundant computations and makes them totally redundant by suitable insertions. What are these suitable insertions? We saw an example before,  $x$  equal to  $a + b$  was replaced by  $h$  equal to  $a + b$  and  $x$  equal to  $h$  and along an arc, and we broke the arc and introduced a computation.

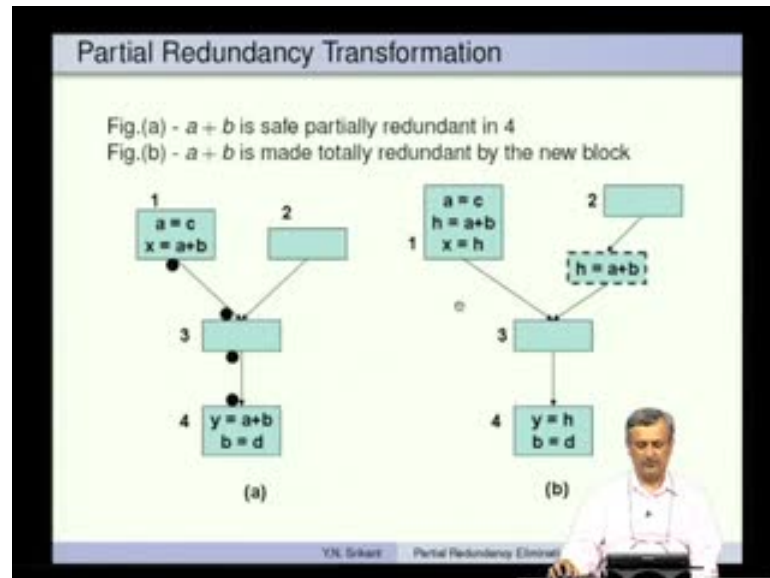
First step is we need to compute several predicates, we will see these -  $AV_i$  is availability,  $ANT_i$  is anticipability,  $SAFE_i$  is safety,  $SPAV_i$  is safe partial availability,  $SPANT_i$  is safe partial anticipability, at the entry and exit points of all nodes. The difference here, from the previous dataflow analysis is we have actually one expression and a bit vector of basic blocks for that expression.

At the entry and exit point of the basic block, we are going to compute these properties. So, one bit for each basic block and one vector for each expression mark all points which have both  $SPAV$  and  $SPANT$  true. So, safe partial available and safe partially anticipable points are all marked and consider the paths formed by connecting such adjacent marked points.

We are going to look at the path and all these points have will be safe; that is because of this guaranty so safe  $SPAV$  and  $SPANT$ . We connect the adjacent points form the path but, all this is we are not going to do it in the algorithm, we are going to compute the predicates directly but, this is the conceptual understanding.

The insertion points are just before last computation in starting points of these paths and then insertion edges are those that enter the junction nodes on these paths. It is as simple as that, replacements are always for the last and first computations, in the starting and ending points of these paths.

(Refer Slide Time: 13:11)



Let us demonstrate what we are doing. We say that a plus b safe partial redundant in 4, in this case. This is the path from the first computation of a plus b to this particular expression a plus b (Refer Slide Time: 13:28). All these points are safe, this is the path that we are talking about and this is the first point on the path and this is the last point on the path. In this, we identify the last computation in this basic block, so that happens to be, this a plus b and this of course is the first computation in the last basic block.

So, because all these points are safe, we can replace this computation by x equal to h after we introduce h equal to a plus b, rest remains the same. We also break this arc, this is the arc that enters this junction, this is at the junction, so it is entering this path (Refer Slide Time: 14:15); so we have to break this particular arc and introduce h equal to a plus b.

Now, this a plus b will be replaced by h, because h has the value of a plus b. This is precisely what we are talking about, replacements for last and first computations in the starting and ending points of these basic blocks.

(Refer Slide Time: 14:43)

**Local Properties**

- $TRANSP_i$  (transparency)
  - True for an expression in a node  $i$ , if its operands are not modified by the execution of statements in node  $i$
- $COMP_i$  (locally available)
  - True if there is atleast one computation of the expression in  $i$  and no modification of operands takes place during and after the computation
- $ANTLOC_i$  (locally anticipable)
  - True if there is atleast one computation of the expression in  $i$  and no modification of the operands takes place before the first computation

YN, Srikant Portal Redundancy Elimination

Let us look at the details of the algorithm little more. Now, we define what is availability, anticipability, etc in the form of dataflow equations; before that some local basic block properties have to be understood. For example, what is transparency of an expression in a basic block? This is a predicate, it is true for an expression in a node  $i$ , if its operands are not modified by the execution of statements in node  $i$ .

(Refer Slide Time: 15:25)

**Local Properties**

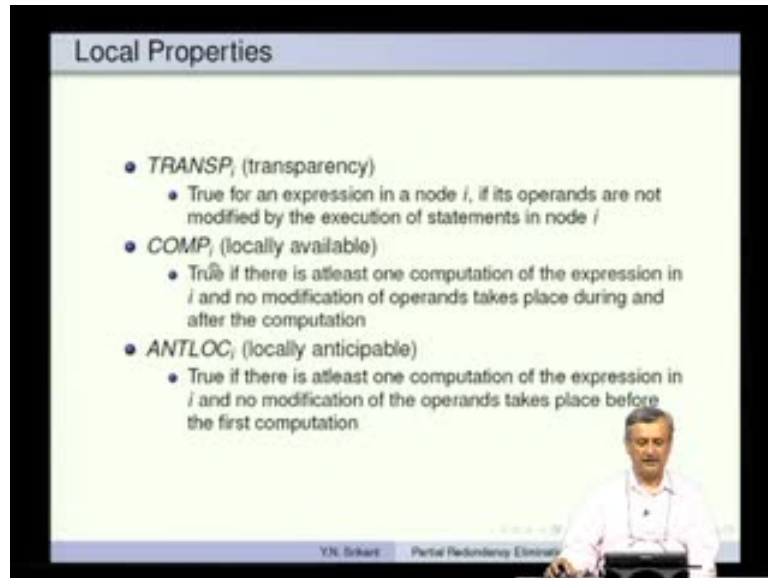
TRANS	COMP	ANTLOC
No assignments to $a$ and $b$ here	$x = a + b$ No assignments to $a$ and $b$ here $x$ cannot be $a$ or $b$	No assignments to $a$ and $b$ here $x = a + b$

$a + b$  is the expression under consideration

YN, Srikant Portal Redundancy Elimination

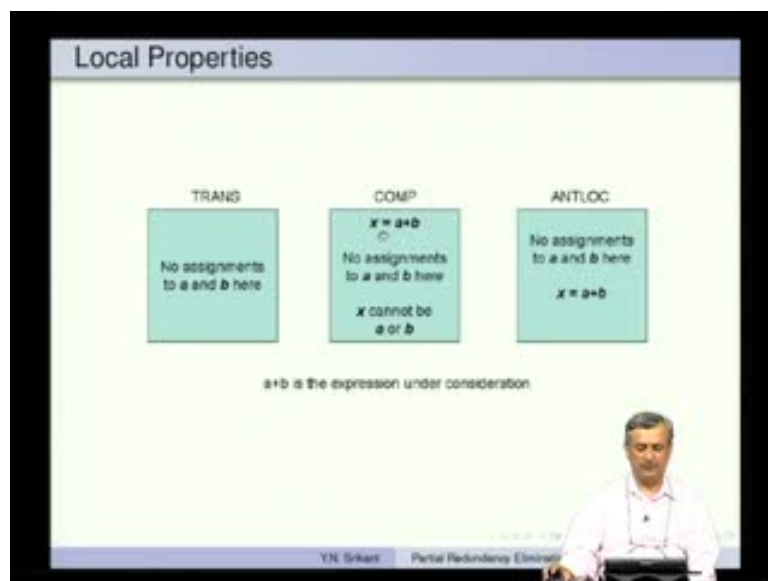
Here is a picture, no assignments to a and b. So, a plus b supposed to be the expression under consideration; no assignments to a and b, so the expression value cannot be modified in anyway.

(Refer Slide Time: 15:39)



What is locally available or locally computed expression?  $COMP_i$  that is the property;. True if there is at least 1 computation of the expression in  $i$  and no modification of the operands takes place during and after the computation.

(Refer Slide Time: 15:59)



This is the situation (Refer Slide Time: 16:00),  $x$  equal to  $a$  plus  $b$  and then there are no assignments to  $a$  and  $b$  after this and  $x$  cannot be  $a$  or  $b$  otherwise,  $x$  is being modified by this assignment itself. This is computed, transparency to an expression locally computed. So, after this computation  $a$  plus  $b$  is available at this point, at the output of the basic block.

(Refer Slide Time: 16:27)

**Local Properties**

- $TRANSP_i$  (transparency)
  - True for an expression in a node  $i$ , if its operands are not modified by the execution of statements in node  $i$
- $COMP_i$  (locally available)
  - True if there is atleast one computation of the expression in  $i$  and no modification of operands takes place during and after the computation
- $ANTLOC_i$  (locally anticipable)
  - True if there is atleast one computation of the expression in  $i$  and no modification of the operands takes place before the first computation

YN, Sarkar Partial Redundancy Elimination

What is locally anticipable property?  $ANTLOC_i$  for a basic block  $i$ . It is true, if there is at least one computation of the expression in  $i$  and no modification of the operands takes place before the first computation of the expression.

(Refer Slide Time: 16:48)

**Local Properties**

TRANS

No assignments  
to a and b here

COMP

$x = a+b$

No assignments  
to a and b here

$x$  cannot be  
a or b

ANTLOC

No assignments  
to a and b here

$x = a+b$

$a+b$  is the expression under consideration

YN. Srikant    Partial Redundancy Elimination

There are no assignments to a and b here, in this case that is not important. In the case of, COMP there could have been assignments to a and b before x equal to a plus b for transparency none at all but here, there are no assignments to a and b and then you have x equal to a plus b. That means, at this point this is anticipable, no modifications to a and b. So, this expression a plus b is anticipable, I can say anticipability is true; that is why it is a locally anticipable property.

(Refer Slide Time: 17:24)

**Global Properties**

**Availability**

$$AVIN_i = \begin{cases} FALSE & \text{if } i = s \\ \prod_{j \in \text{pred}(i)} AVOUT_j & \text{otherwise} \end{cases}$$

$$AVOUT_i = COMP_i + AVIN_i \cdot TRANSP_i$$

**Anticipability**

$$ANTOUT_i = \begin{cases} FALSE & \text{if } i = e \\ \prod_{j \in \text{succ}(i)} ANTIN_j & \text{otherwise} \end{cases}$$

$$ANTIN_i = ANTLOC_i + ANTOUT_i \cdot TRANSP_i$$

**Safety**

$$SAFEIN_i = AVIN_i + ANTIN_i$$

$$SAFEOUT_i = AVOUT_i + ANTOUT_i$$

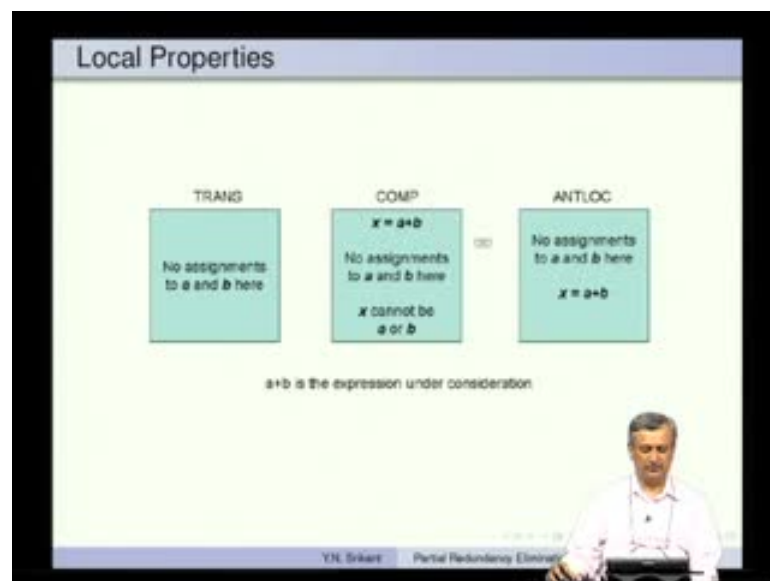
YN. Srikant    Partial Redundancy Elimination

Then we have the dataflow equations: availability - AVIN and AVOUT at the input and output of the basic blocks. So AVIN is false, if it is the start block, this is as before for the start block availability of no expression is really true, no expressions are available at the entry block, input of the entry block, and then the AVIN at any other input point of basic block, you take the AVOUTs of the predecessors and do a AND operation. This is similar to the intersection operation in the available expression analysis problem.

Here, there is a slight modification, we are computing the properties of a single expression with respect to many basic blocks whereas, in the other case we were looking at the input and output points of basic blocks and computing how many expressions available etc.

So, availability of the predecessors is seen here. Why is that we are looking at AND operation on the predecessors? Well, availability says, it must be available along all paths that is why the AND operation is necessary here.

(Refer Slide Time: 19:17).



What about the availability at the output point of a basic block? That is very easy, it is either computed in the basic block or it is available from the top and not modified. That is easy to see here, it is computed in the basic block and then of course, COMP is true implies there are no modifications here. At this point (Refer Slide Time: 19:26), the  $a + b$  is available or if it is available from the top and not modified at all, then also  $a + b$  will be available here. So, these are the two situations that this handles.

(Refer Slide Time: 19:30)

**Global Properties**

**Availability**

$$AVIN_i = \begin{cases} FALSE & \text{if } i = s \\ \prod_{j \in \text{pred}(i)} AVOUT_j & \text{otherwise} \end{cases}$$
$$AVOUT_i = COMP_i + AVIN_i \cdot TRANSP_i$$

**Anticipability**

$$ANTOUT_i = \begin{cases} FALSE & \text{if } i = e \\ \prod_{j \in \text{succ}(i)} ANTIN_j & \text{otherwise} \end{cases}$$
$$ANTIN_i = ANTLOC_i + ANTOUT_i \cdot TRANSP_i$$

**Safety**

$$SAFEIN_i = AVIN_i + ANTIN_i$$
$$SAFEOUT_i = AVOUT_i + ANTOUT_i$$

YN. Srikant Partial Redundancy Elimination

What about anticipability? For a basic block  $i$ , it is false if it is the end block. This is very similar to that, very busy expressions; so the end block. Otherwise, you look at the successors of the basic block and take the AND operation of the ANTIN values. Why are we taking AND operation of the ANTIN values? We are looking at the successors, so there may be many successors like that and then all these successors actually must have anticipability true; otherwise anticipability will not be true at the output point of  $i$ , along all paths there must be anticipability that is why you want the AND operation.

(Refer Slide Time: 20:37)

**Local Properties**

TRANS	COMP	ANTLOC
No assignments to $a$ and $b$ here	$x = a + b$ No assignments to $a$ and $b$ here $x$ cannot be $a$ or $b$	No assignments to $a$ and $b$ here $x = a + b$

$a + b$  is the expression under consideration

YN. Srikant Partial Redundancy Elimination



(Refer Slide Time: 20:43)

**Global Properties**

**Availability**

$$AVIN_i = \begin{cases} FALSE & \text{if } i = s \\ \prod_{j \in \text{pred}(i)} AVOUT_j & \text{otherwise} \end{cases}$$
$$AVOUT_i = COMP_i + AVIN_i \cdot TRANSP_i$$

**Anticipability**

$$ANTOUT_i = \begin{cases} FALSE & \text{if } i = e^{\oplus} \\ \prod_{j \in \text{succ}(i)} ANTIN_j & \text{otherwise} \end{cases}$$
$$ANTIN_i = ANTLOC_i + ANTOUT_i \cdot TRANSP_i$$

**Safety**

$$SAFEIN_i = AVIN_i + ANTIN_i$$
$$SAFEOUT_i = AVOUT_i + ANTOUT_i$$

YN. Srikant Partial Redundancy Elimination

What about  $ANTIN_i$ ? It is locally anticipable, so that means here, a plus b is computed and there are no assignments to a and b, so locally anticipable at this point or anticipable at the output point of i and not modified in the basic block.

(Refer Slide Time: 20:50)

**Local Properties**

TRANS	COMP	ANTLOC
No assignments to a and b here	$x = a + b$ No assignments to a and b here x cannot be a or b	No assignments to a and b here $x = a + b$

a+b is the expression under consideration

YN. Srikant Partial Redundancy Elimination

So, anticipable at this point and not modified makes it anticipable at this point also (Refer Slide Time: 20:56), that is what we mean. Then safety is, either available or anticipable both for input and output of the basic block.

(Refer Slide Time: 21:07)

**Global Properties**

**Safe Partial availability**

$$SPAVIN_i = \begin{cases} FALSE & \text{if } i = s \text{ or } \neg SAFEIN_i \\ \sum_{j \in pred(i)} SPAVOUT_j & \text{otherwise} \end{cases}$$

$$SPAVOUT_i = \begin{cases} FALSE & \text{if } \neg SAFEOUT_i \\ COMP_i + SPAVIN_i, TRANSP_i & \text{otherwise} \end{cases}$$

**Safe Partial anticipability**

$$SPANTOUT_i = \begin{cases} FALSE & \text{if } i = e \text{ or } \neg SAFEOUT_i \\ \sum_{j \in succ(i)} SPANTIN_j & \text{otherwise} \end{cases}$$

$$SPANTIN_i = \begin{cases} FALSE & \text{if } \neg SAFEIN_i \\ ANTLOC_i \\ + SPANTOUT_i, TRANSP_i & \text{otherwise} \end{cases}$$

YN, Srikant Partial Redundancy Elimination

Now, we have safe partial availability  $SPAVIN_i$ ; this is false if  $i$  equal to  $s$  of course, starting point as before, in the case of AVIN or if the input point of  $i$  is not safe, because we also have safety as another parameter  $i$  equal to  $s$  or  $SAFEIN$  not of  $SAFEIN_i$ . In both cases,  $SPAVIN$  will be false and the other part for other nodes it is the OR operation, sigma of the predecessors of the block  $i$ . Take all the predecessors  $j$  of the block  $i$ , take the OR operation on the  $SPAVOUT$  values and that gives us  $SPAVIN$ .

(Refer Slide Time: 22:16)

**Global Properties**

**Safe Partial availability**

$$SPAVIN_i = \begin{cases} FALSE & \text{if } i = s \text{ or } \neg SAFEIN_i \\ \sum_{j \in pred(i)} SPAVOUT_j & \text{otherwise} \end{cases}$$

$$SPAVOUT_i = \begin{cases} FALSE & \text{if } \neg SAFEOUT_i \\ COMP_i + SPAVIN_i, TRANSP_i & \text{otherwise} \end{cases}$$

**Safe Partial anticipability**

$$SPANTOUT_i = \begin{cases} FALSE & \text{if } i = e \text{ or } \neg SAFEOUT_i \\ \sum_{j \in succ(i)} SPANTIN_j & \text{otherwise} \end{cases}$$

$$SPANTIN_i = \begin{cases} FALSE & \text{if } \neg SAFEIN_i \\ ANTLOC_i \\ + SPANTOUT_i, TRANSP_i & \text{otherwise} \end{cases}$$

YN, Srikant Partial Redundancy Elimination

In the previous case here, for AVIN we look at the predecessors but, we looked at an AND operation whereas, here we are looking at the OR operation. The reason is, we want partial availability that means any one of the paths will do that is why this is OR operation.

SPAVOUT is false, if the output point of the basic block  $i$  is not safe. Otherwise, it is either computed or available from the top and transparent, this is as before. The safe partial anticipability at the output point of  $i$  is similar, it is false if  $i$  is either the end block or the output point is not safe. In other cases, again you take the union, the rather the OR operation of  $SPANTIN_j$ ; where  $j$  are the successors. In the previous case, we took successors but, we took AND operation. In this case, again we take OR operation because we are looking at partial anticipability; any one of the paths will do, that is why OR operation.

SPANTIN is simple, false if the input point is not safe. Otherwise, it is locally anticipable or anticipable at the output point and not modified. So, this is as in this case (Refer Slide Time: 23:39), so not much difference between these two.

(Refer Slide Time: 23:43)

**Global Properties**

- **Safe Partial Redundancy**
  - For  $FIRST_i$  (at entry of  $i$ )
 
$$SPREDUND_i = ANTLOC_i \vee SPAVIN_i$$
  - $LAST_i$ , when it is distinct from  $FIRST_i$ , cannot be safe partially redundant, because the computations of the expression between these makes  $ANTLOC_i$  false
- **Total Redundancy**
  - For  $FIRST_i$ 

$$REDUND_i = ANTLOC_i \vee AVIN_i$$
  - For  $LAST_i$ 

$$REDUND_i = COMP_i \vee AV_p$$

where  $p$  is the point just before  $LAST_i$

YN, Srikar Partial Redundancy Elimination

What is safe partial redundancy? There are again two points, which we need to keep track of: one is the FIRST at the entry of  $i$  and the LAST. FIRST and LAST computations need to be considered specially.

For the FIRST computation, in the basic block  $i$ , at the entry of  $i$  safe partially redundant  $i$  f, because it is the FIRST computation we are looking at it as  $i$  f, is locally anticipable and safe partially available at the input point. This is safe partial redundancy and  $LAST_i$  when it is distinct from  $FIRST_i$  cannot be safe partially redundant because the computations of the expression between these makes  $ANTLOC_i$  false.

(Refer Slide Time: 24:53)

**FIRST and LAST Computations**

$x = a+b$   
(no modifications to  $a$  and  $b$  here)  
 $y = a+b$

Such situations do not occur since local CSE has been carried out

$x = a+b$   
 $a = \dots$   
 $b = \dots$   
 $z = a+b$   
 $a = \dots$   
 $b = \dots$   
 $y = a+b$

The modifications to  $a$  and  $b$ , and  $z = a+b$  are not relevant. Only  $x = a+b$  and  $y = a+b$  are relevant (FIRST and LAST computations)

YN. Sikari Partial Redundancy Elimination

So, we have the FIRST and LAST computation here (Refer Slide Time: 24:53). We are really looking at FIRST and LAST being very different. So, if they are same, there is no problem at all; if they are different that means, after the first computation there have been modifications of the operands.

(Refer Slide Time: 25:46)

**Global Properties**

- **Safe Partial Redundancy**
  - For  $FIRST_i$  (at entry of  $i$ )
$$SPREDUND_i = ANTLOC_i.SPAVIN_i$$
  - $LAST_i$ , when it is distinct from  $FIRST_i$ , cannot be safe partially redundant, because the computations of the expression between these makes  $ANTLOC_i$  false
- **Total Redundancy**
  - For  $FIRST_i$ 
$$REDUND_i = ANTLOC_i.AVIN_i$$
  - For  $LAST_i$ 
$$REDUND_i = COMP_i.AV_p$$
where  $p$  is the point just before  $LAST_i$

YN. Srikant Partial Redundancy Elimination

Therefore, this computation needs to be present; this is not at all redundant whereas, this can be redundant because there could be something available from the top. So, if  $FIRST$  and  $LAST$  do not coincide that means, there are modifications of  $a$  and  $b$  in-between and therefore, this last computation actually cannot be seen as redundant computation of and replace with some variable which stores  $a$  plus  $b$  here, we have to do a fresh computation of this again. That is what this really saying, so cannot be safe partial redundant because computations of the expression between these make  $ANTLOC_i$  false.

Now, what is total redundancy? Total redundancy means locally anticipable and available from the top. That is very easy to understand and then for the last  $i$  computation it is computed in the basic block and available just before that particular point  $p$ ; where the last computation is involved.

Total redundancy allows us to do common sub expression elimination and that is precisely what happens, every safe partial redundant computation is converted to a totally redundant computation and then CSE happens to get applied there.

(Refer Slide Time: 26:40)

**Global Properties**

- **Isolatedness**  
A computation is *isolated*, if it is neither safe partially available nor safe partially anticipable at that point

$$ISOLATED_i = ANTLOC_i \wedge \neg SPAVIN_i \wedge \neg (TRANSP_i \vee SPANTOUT_i)$$
$$ISOLATED_i = COMP_i \wedge \neg SPANTOUT_i \wedge \neg (TRANSP_i \vee SPAVIN_i)$$

YN. Srikant Partial Redundancy Elimination

Isolatedness is not a very important property for us today, because it is only needed for proofs but, we will just look at it and understand. A computation is isolated if it is neither safe partially available nor safe partially anticipable at that point. This is not needed for us so we will skip the description of this.

(Refer Slide Time: 27:03).

**Predicates for Insertion**

**INSERT<sub>i</sub>**

- True if the point just before the LAST computation in block *i* is an insertion point
- Interpretation of **INSERT<sub>i</sub>**:  
(*expr* should be computed in *i*) AND (*expr* should be useful later) AND ((operands should be modified in *i*) OR (*expr* should not be available from above))
- This is possible only for the first node on the path and those intermediate nodes where the operands of the *expr* are modified and the *expr* is recomputed

$$INSERT_i = COMP_i \wedge SPANTOUT_i \wedge (\neg TRANSP_i \vee \neg SPAVIN_i)$$

**INSERT<sub>(i,j)</sub>**

- True if a computation should be inserted by splitting edge (*i, j*)

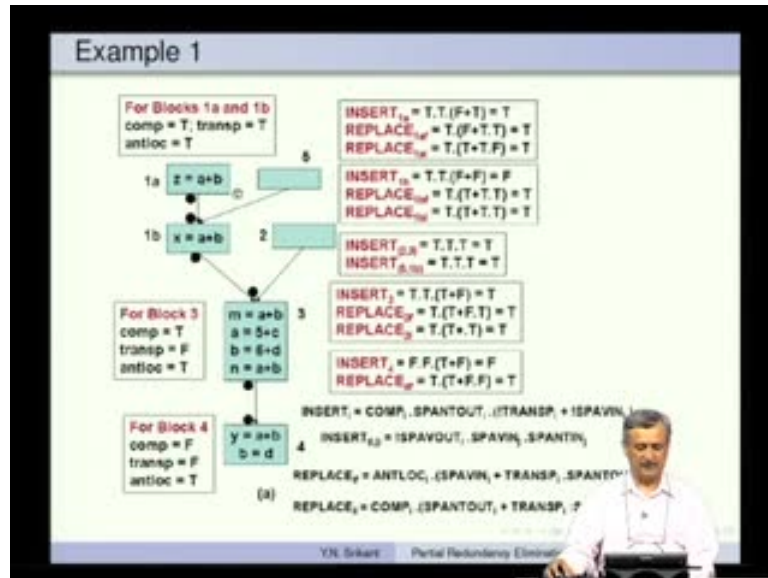
$$INSERT_{(i,j)} = \neg SPAVOUT_i \wedge SPAVIN_j \wedge SPAN_{(i,j)}$$

YN. Srikant Partial Redundancy Elimination

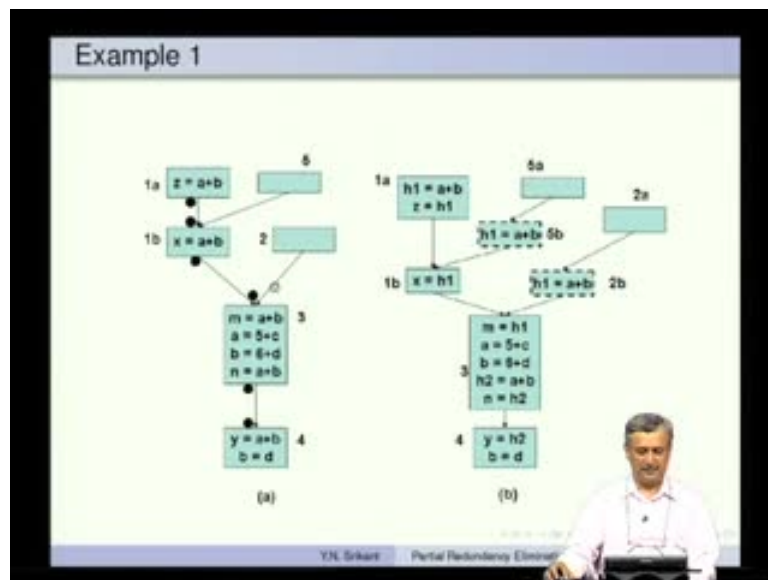
Let us understand the various predicates; we have seen so far the various dataflow values availability, anticipability, safe partial availability, safe partial anticipability and then safe partially redundant computation etc.

Now, the main task is to compute the predicates which tell you whether the computation has to be h equal to a plus b has to be inserted.

(Refer Slide Time: 27:42)



(Refer Slide Time: 27:48)



Let me give you an example. Here is z equal to a plus b, if we replace this then, z equal to a plus b would be replaced by h1 equal to a plus b and then z equal to h1. We call this as insertion of a new computation and this has replacement. For example, x equal to a plus b in 1b has been replaced by x equal to h1. Again, this computation is inserted by breaking the edge, so this is an insertion on the edge (Refer Slide Time: 28:21), h1 equal



to a plus b and again, this is also a computation inserted on the edge by breaking it. So, it is h1 equal to a plus b; whereas this m equal to a plus b is only a replacement and then a b are modified and n equal to a plus b recompute a plus b.

Here observe that, there is another variable h2, because there is a new computation this cannot take to be the same as m equal to a plus b. The value of a plus b here and here are different (Refer Slide Time: 28:55), we have to do a recomputation of a plus b and that is done in the different variable h2. So, h2 equal to a plus b is an insertion and n equal to h2 is a replacement. This y equal to a plus b is replaced by y equal to h2.

(Refer Slide Time: 29:14)

**Predicates for Insertion**

**$INSERT_i$**

- True if the point just before the LAST computation in block  $i$  is an insertion point
- Interpretation of  $INSERT_i$ :  $\Leftrightarrow$   
*(expr should be computed in  $i$ ) AND (expr should be useful later) AND ((operands should be modified in  $i$ ) OR (expr should not be available from above))*
- This is possible only for the first node on the path and those intermediate nodes where the operands of the expr are modified and the expr is recomputed

$$INSERT_i = COMP_i \cdot SPANTOUT_i \cdot (\neg TRANSP_i + \neg SPAVIN_i)$$

**$INSERT_{(i,j)}$**

- True if a computation should be inserted by splitting the edge  $(i, j)$

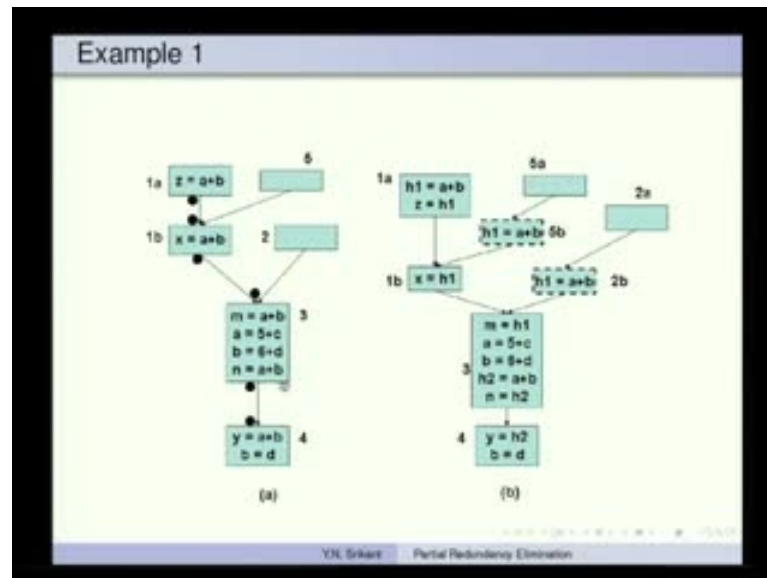
$$INSERT_{(i,j)} = \neg SPAVOUT_i \cdot SPAVIN_j \cdot SPANTIN_j$$

YN, Srikant Partial Redundancy Elimination

This is what we mean, so there is an insert i predicate which says, should I insert h equal to a plus b in this block? In place of just before the last computation in basic block i. Remember ,we are going to do an insertion for the LAST computation and we are going to a do only a replacement for the FIRST computation, that is very clear in this example.



(Refer Slide Time: 29:41)



Here, FIRST and LAST computations are different, so for this computation we have done a replacement and for this computation we are doing an insertion (Refer Slide Time: 29:50). In this case, FIRST and LAST computations are identical it is the same. We are doing both insertion and replacement. In this case also, FIRST and LAST computations are identical but, we do not need an insertion for because the insert predicate will be false but, the replacement is been done.

(Refer Slide Time: 30:13)

### Predicates for Insertion

**INSERT<sub>i</sub>**

- True if the point just before the LAST computation in block *i* is an insertion point
- Interpretation of **INSERT<sub>i</sub>**:  
(*expr* should be computed in *i*) AND (*expr* should be useful later) AND ((*operands* should be modified in *i*) OR (*expr* should not be available from above))
- This is possible only for the first node on the path and those intermediate nodes where the operands of the *expr* are modified and the *expr* is recomputed

$$INSERT_i = COMP_i \cdot SPANTOUT_i \cdot (-TRANSP_i + -SPAVIN_i)$$

**INSERT<sub>(i,j)</sub>**

- True if a computation should be inserted by splitting the edge (*i, j*)

$$INSERT_{(i,j)} = -SPAVOUT_i \cdot SPAVIN_j \cdot SPANTIN_j$$

Here is the INSERT predicate,  $INSERT_i$  equal to  $COMP_i$  dot - that is AND operation -  $SPANTOUT_i$  safe partially anticipable  $OUT_i$  then AND with an OR operation,  $TRANSP_i$  not of  $TRANSP_i$  plus not of  $SPAVIN_i$  - safe partially available IN.

This is not understandable straight away. Let us look at an explanation to understand this.  $COMP_i$  says expression should be computed in the basic block  $i$  of course,  $a$  and  $b$  should not be modified later. Then  $SPANTOUT_i$  it says, expression should be useful later, so there is a computation of that expression later on. Safety is always over through the entire predicates, so everything is safe, safe etc.

So, leaving out that safety, expression should be useful later, is what anticipability is all about. Then this AND operation, then this predicate with OR  $TRANSP_i$  not of  $TRANSP_i$  is operand should be modified in  $i$  and not of  $SPAVIN_i$  says OR expression should not be available from the top or above.

Let us see, why this makes sense. So expression should be computed in  $i$ ; obviously, if we are not computing something there is no way you can insert and replace anything there; so, that is why computation is necessary. If you have  $x$  equal to  $a$  plus  $b$  then,  $i$  can insert a computation  $h$  equal to  $a$  plus  $b$ , if there is no computation of  $a$  plus  $b$  in that block, there is nothing to insert. Expression should be useful later, if this happens to be the last expression and there is no use after that expression later on.

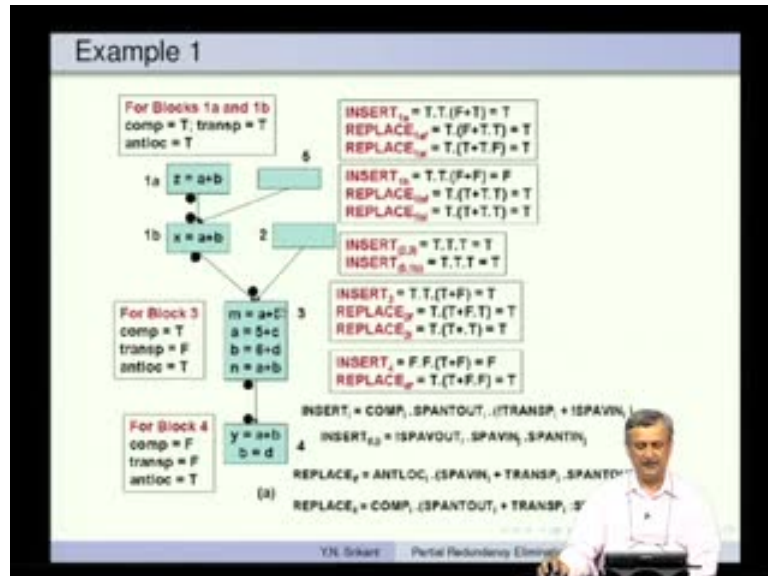
Then there is no point in inserting  $h$  equal to  $a$  plus  $b$  and replacing  $x$  equal to  $a$  plus  $b$  by  $x$  equal to  $h$ ; that is why, anticipability later on is very important for us. Then this says, operands should be modified in  $i$  OR operand expression should not be available from the top.

So, if the operands are not modified in  $i$  then, it is also available from the top, whatever is available from the top will just pass through; there is no need to actually compute that  $a$  plus  $b$  again. So, insertion need not be carried out but, if the operands are modified in the basic block then, there is a need to recompute that expression and that is why, this not of  $TRANSP_i$  make sense.

If the expression is not available from the top either then also, we need to compute that expression; you cannot simply take a value which already exist there isn't anything. This  $INSERT_i$  is true only for the first node on the path and those intermediate nodes where

the operands of the expression are modified and the expression is recomputed. That is taken care of by this part; the first node is taken care of by these things approximately, not exactly.

(Refer Slide Time: 34:04)



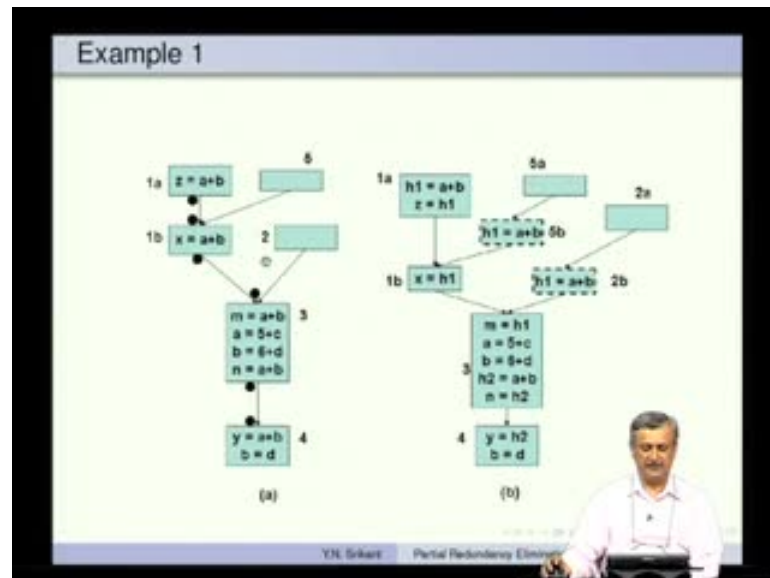
Let us look at an example, here is the node 1a, you have z equal to a plus b. As I already told you, here is a plus b and then modification. This part is a second expression computation of a plus b, so let us just look at this part first.

This is computation beginning and this is the usage and here is another usage in-between, so all these points are safe. Safety is very easily computed, so available here, this is anticipable here, available here and anticipable here (Refer Slide Time: 34:36), so for different reasons all these points are safe.

For this first point, the FIRST and LAST computations go inside, so INSERT<sub>i</sub> is true; then the replace is also true. For this point, this is an intermediate computation, now we get this value from here (Refer Slide Time: 35:05); there is no need to compute this value again provided we break this edge. That is why for this particular case, assuming that, we break this edge, this 1b INSERT<sub>i</sub> is false and replacement happen to be true. We are only going to do a replacement; if we insert something here, it becomes total fully available and then we have this a plus b, this is the FIRST computation in this block.

For this,  $INSERT_3$  it is true and replace is also true, but the insert part is for this LAST computation and not for the FIRST computation. For the first computation, it is only the replacement that we look at the 3F that is true; we are not going to do any insertion for the FIRST computations. We always do the insertions for the LAST computation; observe that here, for the last computation.

(Refer Slide Time: 36:15)



This will be replaced by m equal to h that can be seen here (Refer Slide Time: 36:14), h1 equal to a plus b, z equal to h1, so insertion and then replacement. Here we break this (Refer Slide Time: 36:23), we break this also and then m equal to h1. Why this is necessary to be broken? because otherwise, availability will be pass along on this part.



(Refer Slide Time: 37:12)

### Predicates for Insertion

**INSERT<sub>i</sub>**

- True if the point just before the LAST computation in block *i* is an insertion point
- Interpretation of **INSERT<sub>i</sub>**:  
(*expr* should be computed in *i*) AND (*expr* should be useful later) AND ((operands should be modified in *i*) OR (*expr* should not be available from above))
- This is possible only for the first node on the path and those intermediate nodes where the operands of the *expr* are modified and the *expr* is recomputed

$$INSERT_i = COMP_i \cdot SPANTOUT_i \cdot (\neg TRANSP_i + \neg SPAVIN_i)$$

**INSERT<sub>(i,j)</sub>**

- True if a computation should be inserted by splitting the edge (*ij*)

$$INSERT_{(i,j)} = \neg SPAVOUT_i \cdot SPAVIN_j \cdot SPANTIN_j$$

YN, Srikant    Partial Redundancy Elimination

(Refer Slide Time: 37:37)

### Example 1

**For Blocks 1a and 1b**  
comp = T, transp = T  
antloc = T

**For Block 3**  
comp = T  
transp = F  
antloc = T

**For Block 4**  
comp = F  
transp = F  
antloc = T

**Block 1a:** z = a+b

**Block 1b:** x = a+b

**Block 2:** (empty)

**Block 3:** m = a+b, a = b+c, b = b+d, n = a+b

**Block 4:** y = a+b, b = d

**Block 1a:**  
INSERT<sub>1a</sub> = T, T, (F+T) = T  
REPLACE<sub>1a</sub> = T, (F+T, T) = T  
REPLACE<sub>1a</sub> = T, (T+T, F) = T

**Block 1b:**  
INSERT<sub>1b</sub> = T, T, (F+F) = F  
REPLACE<sub>1b</sub> = T, (T+T, T) = T  
REPLACE<sub>1b</sub> = T, (T+T, T) = T

**Block 2:**  
INSERT<sub>2,3</sub> = T, T, T = T  
INSERT<sub>2,4</sub> = T, T, T = T

**Block 3:**  
INSERT<sub>3</sub> = T, T, (T+F) = T  
REPLACE<sub>3</sub> = T, (T+F, T) = T  
REPLACE<sub>3</sub> = T, (T+, T) = T

**Block 4:**  
INSERT<sub>4</sub> = F, F, (T+F) = F  
REPLACE<sub>4</sub> = T, (T+F, F) = T

**Block 1a/1b to 2:**  
INSERT = COMP, SPANTOUT, (¬TRANSP + ¬SPAVIN)

**Block 2 to 3/4:**  
INSERT<sub>2,3</sub> = SPAVOUT, SPAVIN, SPANTIN

**Block 3 to 4:**  
REPLACE<sub>3</sub> = ANTLOC, (SPAVIN + TRANSP, SPANTOUT)

**Block 4 to exit:**  
REPLACE<sub>4</sub> = COMP, (SPANTOUT, + TRANSP, SPAVIN)

YN, Srikant    Partial Redundancy Elimination

When is  $INSERT_{ij}$  true? It is true, if a computation should be inserted by splitting the edge *ij* and  $INSERT_{ij}$  is not of  $SPAVOUT_i \cdot SPAVIN_j \cdot SPANTIN_j$ . Let us understand what this is. If there is an edge, let us look at this example again. Let us say we take this edge, we are going to do this computation for all edges and find out that it is not necessary to break this or this and so on and so forth; only for these 2 we need to break them.

So, this is  $i$  and this is  $j$  (Refer Slide Time: 37:57). It says  $INSERT_{ij}$  not of  $SPAVOUT_i$ , in other words, here it should not be safe partially available from the top. If it is already available from the top, there is a need to insert something here that can be used, so safe partial availability is false at this point.

$SPAVIN_j$  this  $j$  point, the computation must be partially available from another path; along this path it is not available but, from along another path it should be available; if it is not available along this path also there is the point, there is nothing to be done at all, it is partially available, so we need to break this and make it fully available.

Then it says  $SPANTIN_j$ , at this point the computation must be done later on, it should be useful later on, expression should be use later, it is recomputed later. Again, if there is no recomputation why should we insert anything here? For example, this is being computed, it is recomputed, this is one computation, and this is another computation. Along this path it is partially available; along this path  $a + b$  is not partially available at all. We need to break this edge and then insert  $h$  equal to  $a + b$ .

The same is true along this part have this edge as well. Here, it is not safe partially available, along this path it is safe partially available and at this point it is safe partially anticipable, so we break this edge. So that is what, these three really mean.

(Refer Slide Time: 39:35)

**Predicates for Insertion**

**$INSERT_i$**

- True if the point just before the LAST computation in block  $i$  is an insertion point
- Interpretation of  $INSERT_i$ :  
(*expr should be computed in  $i$* ) AND (*expr should be useful later*) AND (*operands should be modified in  $i$* ) OR (*expr should not be available from above*)
- This is possible only for the first node on the path and those intermediate nodes where the operands of the expr are modified and the expr is recomputed

$$INSERT_i = COMP_i \cdot SPANTOUT_i \cdot (\neg TRANS_i + \neg SPAVIN_i)$$

**$INSERT_{(i,j)}$**

- True if a computation should be inserted by splitting the edge  $(i, j)$

$$INSERT_{(i,j)} = \neg SPAVOUT_i \cdot SPAVIN_j \cdot SPAN_{(i,j)}$$

YN. Srikant    Perla Padmanabha Chinnappa



(Refer Slide Time: 39:38)

**Predicates for Replacement**

$REPLACE_i$  (respectively  $REPLACE_{if}$ )

- True if  $FIRST_i$  (respectively  $LAST_i$ ) should be replaced

$REPLACE_{if} = ANTLOC_i.(SPAVIN_i + TRANSP_i.SPANTOUT_i)$

$REPLACE_i = COMP_i.(SPANTOUT_i + TRANSP_i.SPAVIN_i)$

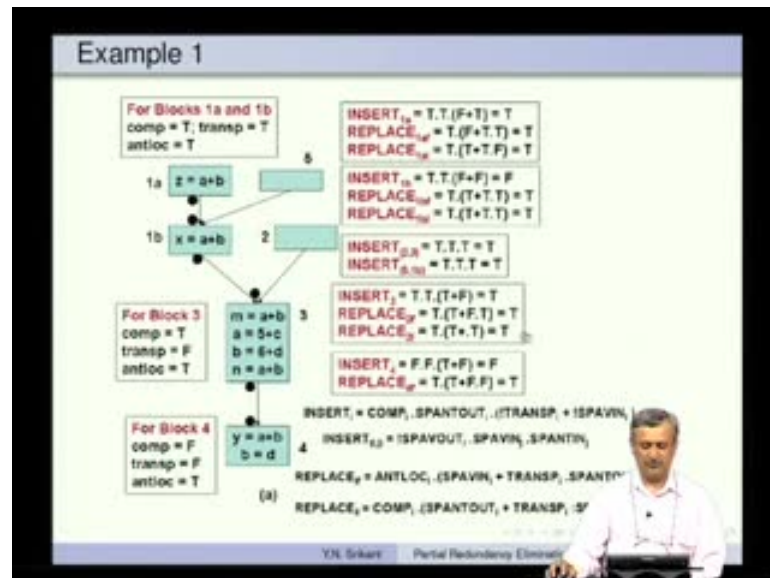
YN. Srikant Partial Redundancy Elimination

When is a replacement to be done? So, replacement of FIRST and replacement of LAST - if and il. It is true if the FIRST respectively, LAST computation should be replaced.  $REPLACE_{if}$  is  $ANTLOC_i$  SPAVIN<sub>i</sub> plus  $TRANSP_i$  dot  $SPANTOUT_i$ . Let us see what this is, when do you replace the first computation? locally computed there is the first computation itself, that is obviously true.

Hence, SPAVIN<sub>i</sub> available from the top, that is why we want to replace it by x equal to h. Otherwise, it is useful later on and of course, not modified. So, either this or that, one of these two most obviously is true. Locally computed make sure that, we have the computation locally available and then partial availability from the top or usefulness later on, one of these must be satisfied; then we can actually replace the FIRST computation by x equal to h.



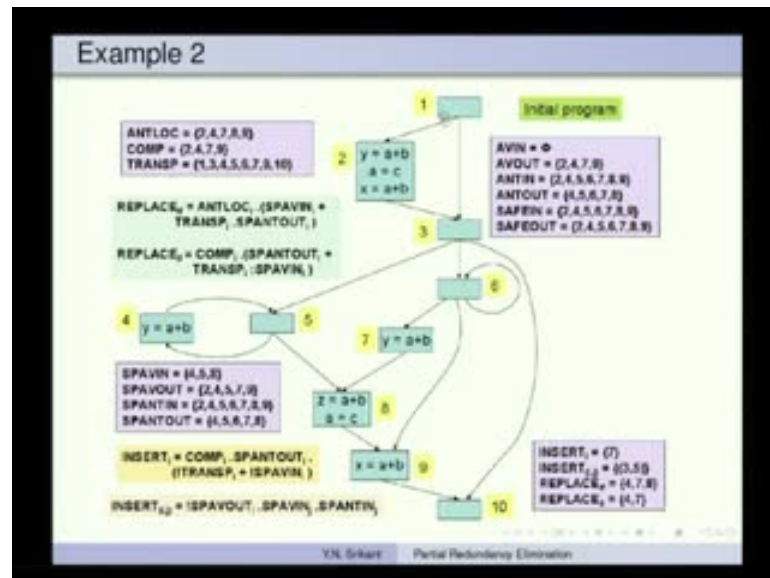
(Refer Slide Time: 41:15)



The LAST computation can be replaced, provided it is computed in the block and then very similar useful later on, or available from the top and not modified. Even though, predicates look a bit difficult it is very easy to look at the diagram and then see what is happening.

Here both the replacement is and insertions are true (Refer Slide Time: 41:22). Here only replacement is true and only replacement is true, here both insertion and replacement and here only replacement. That is how; these predicates really give you the value. For example, why should we replace this? It is available from the top. What about this? It is going to be use later on and we would have inserted something just now. That is why these need to be replaced.

(Refer Slide Time: 42:03)



What about this (Refer Slide Time: 41:54)? again available from the top. We are going to break this and make it available; that is how these replacements happen.

Now, let us take a slightly bigger example and see how this works? This has many loops and many paths and so on and so forth and a plus b is again the expression (Refer Slide Time: 42:15). Let us look at each of these properties one by one: these are local properties – ANTLOC, COMP, and TRANSP. So, ANTLOC locally anticipable is true at 2 4 7 8 9. Why is it true here? So a plus b here, at this point it is true. What about 4? At this point a plus b is computed here.

What about 7? Same thing, this is 7 so a plus b is computed here. What about 8? Again a plus b is computed right here so it is true here (Refer Slide Time: 43:00). What about 9? Again a plus b is computed here. So, COMP is true at the 2 4 7 and 9. In 2 because a plus b is here and then not modified later on; 4 a plus b is here not modified, then 7 a plus b is here not modified and 9 a plus b is here and not modified. Here it is not true because a plus b is here but, a is modified (Refer Slide Time: 43:24).

Similarly, TRANSP these all are empty blocks so nothing much happens, no modification here (Refer Slide Time: 43:34) but there is modification here. There is modification, no modification, no modification. So, except for 8 and 2 all others are transparent.

AVIN is phi because expression  $a + b$  is not available at the entry point of any basic block. Here it is not available (Refer Slide Time: 43:54), easy to see here, it is not available because there is nothing here, here it is not available because there is nothing here and again this point, this point, all these there is partial availability but, there is no availability at the entry point.

AVOUT is true for 2 4 7 9. So, AVOUT is true here because available of the low computation, here also, here also and then 4 at this point also available.

Anticipability is true at the input points of 2 4 5 6 7 8 9. Why is it true here? Because  $a + b$  is computed right here, let us take 7, again true because  $a + b$  is computed right here and 9 also the same thing.

Safety is true for 2 4 5 6 7 8 9 because safety says either available or anticipable. In most cases, local anticipability is true even though availability is not true. For the output points 7 for example, availability is true, even though anticipability is also true. At this point 9 the availability part is not true, only partially available, but anticipability becomes true.

Safe partial availability is similar, this is a tricky one, and others are very easy 4 5 and 8 SPAVIN is true. Let us take the input point of 4. Why is safe partial availability true at the input point of 4? It is not the path alone that matters, if you take only the path partial availability, we can always trace this path like this and then it is path of availability. When you look at safe partial availability, you must make sure that from the point of computation to this point, all the intermediate points are safe. That means, if we consider this as the computation which is available partially at this point.

We have to make sure that this point (Refer Slide Time: 46:18) this point, output of 3, input of 5, output of 5 are all safe, but unfortunately 3 input and output are not safe. Neither available, nor anticipable, so you can go out straight away.

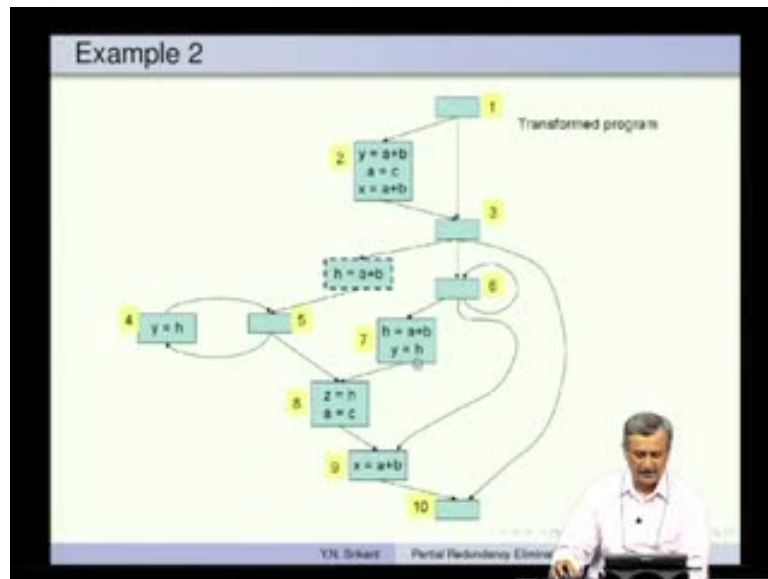
Safe partial availability says, from the point of computation to the point under consideration. We are going to take this as the point of computation (Refer Slide Time: 46:46) that is the output of 4, go through this part 5, come out and then go to input of 4. This is the path of availability that will be consider to make sure that safe SPAVIN of 4 is true.

So, it is computed here and then brought back; that is say partial availability again. This is true for this path also, I can always take this path like this and then like this (Refer Slide Time: 47:14) compute and then go out and then come again, so that is the path. Since, we are going to take from the computation point to the point under consideration we need to consider only this path. So, output of 4, input of 5, outputs of 5 all these are safe. So, SAFEIN SAFEOUT has 5 and 4 no problem.

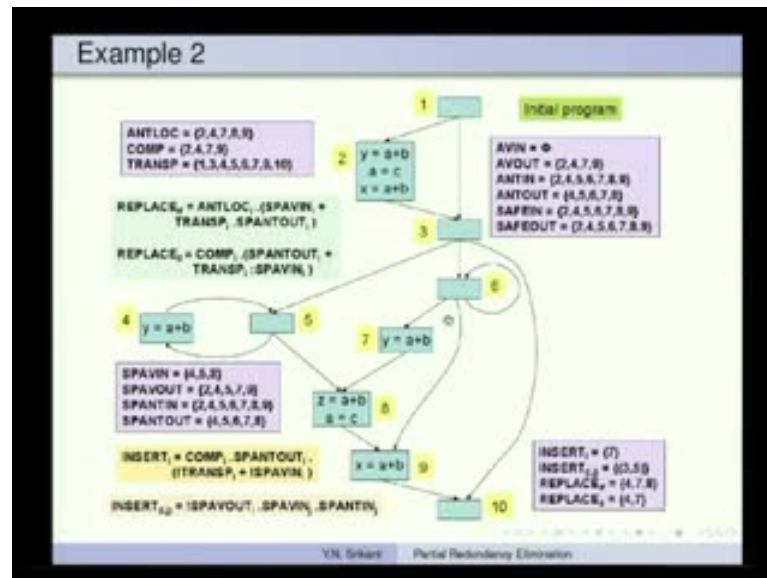
SPAVOUT is heavy straight forward; let us take 5, SPAVOUT of 5 is true because I get something here and this is transparent. Of course, safety is true for all these blocks. SPANTIN is simple because there are too many blocks here.

Let us take, SPANTOUT of say 6. We are considering block 6, then output point of block 6. It does not matter, how we want to go out; this is the only path available to us; a plus b is computed here, so anticipability becomes true; this so far as safety etc.

(Refer Slide Time: 43:32)



(Refer Slide Time: 49:06)



Now, what about insertion and replacement? In fact, in the next slide, this cannot be neither replacing, nor used as redundant computation. We have to break this edge and insert a computation here. This computation gets a replacement (Refer Slide Time: 48:48) this computation gets both a replacement and an insertion, here this gets a replacement but, there is nothing we can do for this computation either, why? See the point is, if you look at this edge, this is the critical edge, so at this point, we do not have safe partial availability and at this point, we have safe partial availability. This is the  $INSERT_{ij}$  is safe partial availability of this point is false, this point is true and  $SPANTIN_j$  is also true at this point.

This is the critical edge when we need to break it but, the immediate question will be breaking this is fine, if I break this edge here, then I am inserting an a plus b here and that is also all my problems, it makes it available for everything that goes out. So I can replace this (Refer Slide Time: 49:53) I can replace this, I can replace this, I can replace this also, I cannot replace this because a is modified of course at this point.

(Refer Slide Time: 50:08)

**Example 2**

Solution:

- Insertion just before the *last* computation in node 7
- Insertion on edge (3, 5)
- Replacement of the *first* computation in nodes 4, 7, and 8
- Replacement of the *last* computations in nodes 4 and 7

Question:

- Why should we not split edge (1,3) and place the computation  $h = a + b$ ? Why only on the edge (3,5)?

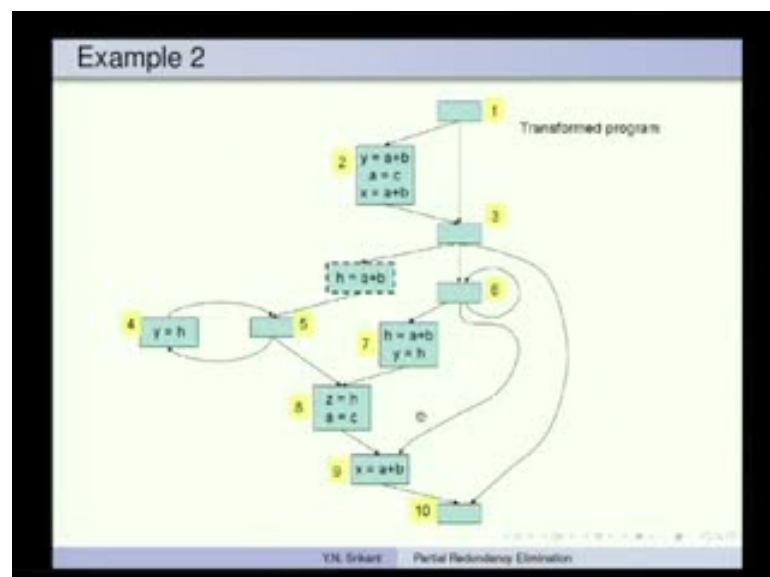
Answer:

- It is not safe. The path 1-3-10 had no computation of  $a + b$  before transformation and by placing a computation on the edge (1,3), we are introducing one
- However, this solution works for all "valid" inputs

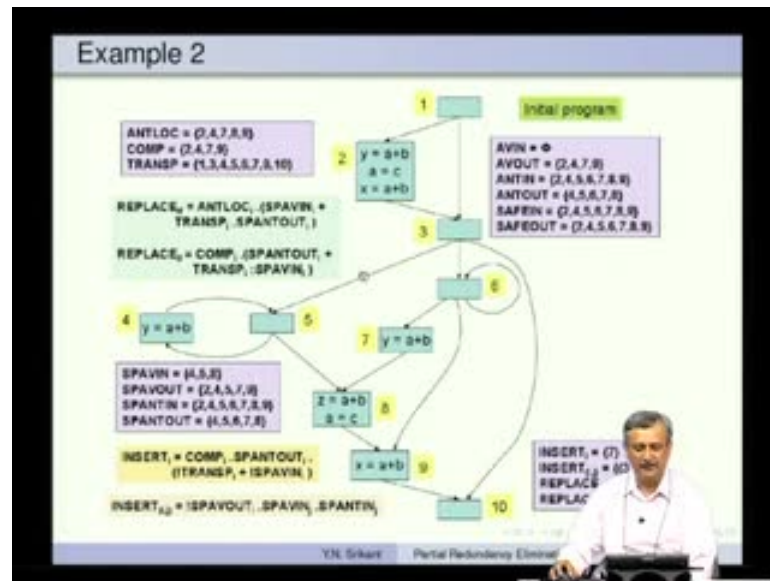
YN, Srikant Partial Redundancy Elimination

So I can replace all these by inserting a computation right here. Why we are not doing it? Why should we not split the edge 1 3 and replace the computation  $h$  equal to  $a$  plus  $b$ ? Why only on the edge (3, 5)? The answer is it is not safe. The path 1-3-10 had no computation of  $a$  plus  $b$  before transformation and by placing a computation on the edge 1-3 we are introducing 1.

(Refer Slide Time: 50:28)



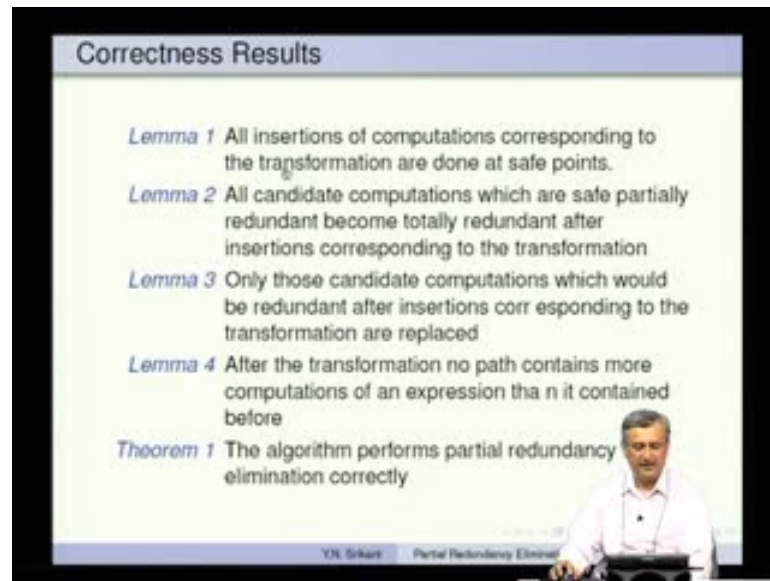
(Refer Slide Time: 50:43)



So, 1, 3 and 10 had no computation. Whereas, if we introduce 1 here this gets 1 computation and this becomes unsafe. Whereas, if we introduce 1 here, this already had a computation for example, if you had taken this path it already had this computation, there was no problem. That is the reason why we insert the computation here and not at this point and I cannot replace this a plus b by a previous computation because a is being modified here. So, that is why it fails for this point as well.

That is how, we do the computations and this is the inside for, why certain computations are replaced? Why are adjust broken? Etc.

(Refer Slide Time: 51:23)



The final solution is something we already said, so let us not waste time there. There are certain correctness results that we need to be aware of. So, couple of lemmas and theorems, we are not going to prove them but, let us state them.

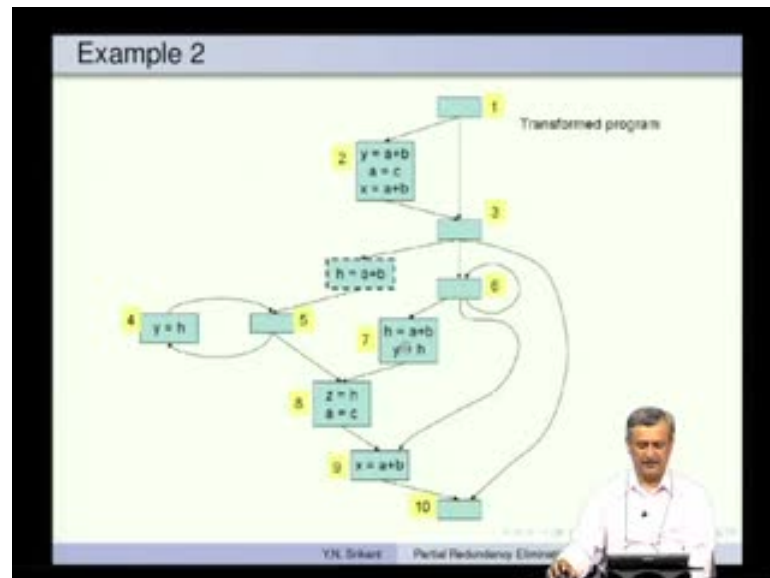
Lemma 1 says all insertions of computations corresponding to the transformation are done at safe points. By definition of that insert and  $INSERT_{ij}$  etcetera, we always use SPANTIN, SPANTOUT etc and SPAVIN, SPAVOUT. That means the transformations are automatically carried out at safe points, this is the intuitive explanation.

Lemma 2 says, all candidate computations which are safe partially redundant, become totally redundant after insertions corresponding to the transformation. Something may become partially redundant but, it may not be safe partially redundant. In such a case, we will not be replacing it, only when safe partially redundant computation is available, we replace it.

Lemma 3 says, only we have woven this safety completely into the insert and replace predicates and that is how this becomes true only for safe partially redundant computations. Only those candidate computations which would be redundant after insertions corresponding to the transformation are replaced.

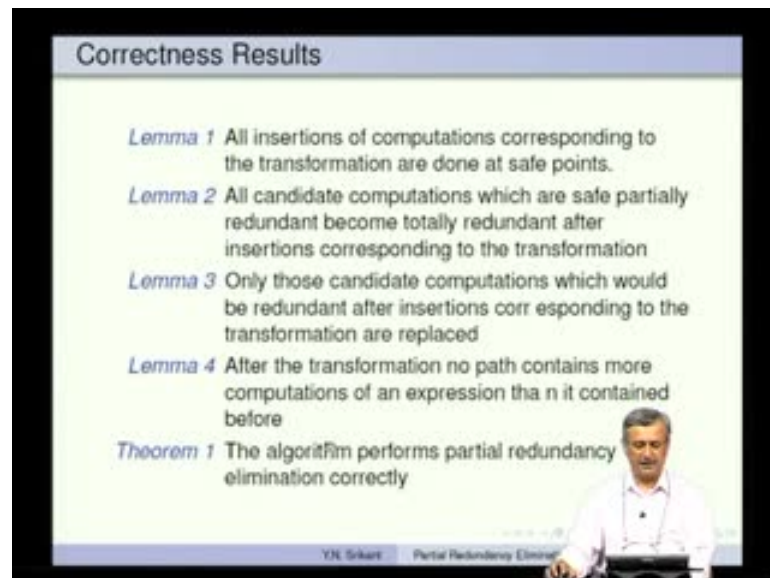


(Refer Slide Time: 53:07)



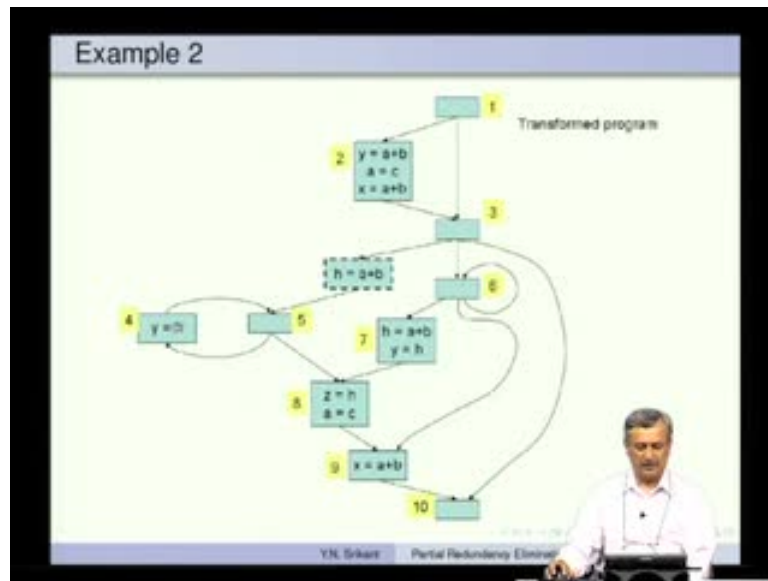
This is again true **because** otherwise, we would have replaced this also, that is not true. They should become completely redundant, here this does not become redundant, a equal to c make sure that we have to recomputed this all over again. This does not become redundant because a equal to c is modifying a.

(Refer Slide Time: 53:27)



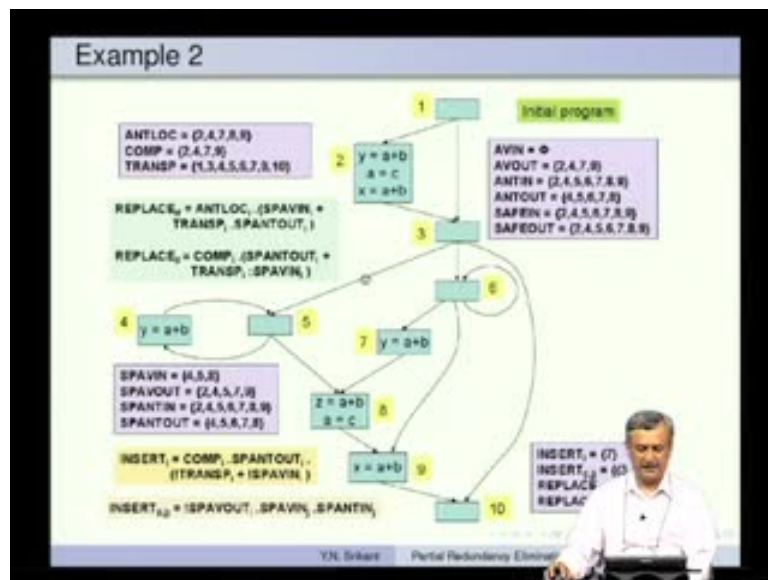
After the transformation, no path contains more computations of an expression than it contained before, so this is something we already assured. We did not want to insert anything by breaking this edge. Otherwise it would have had an extra computation.

(Refer Slide Time: 53:49)



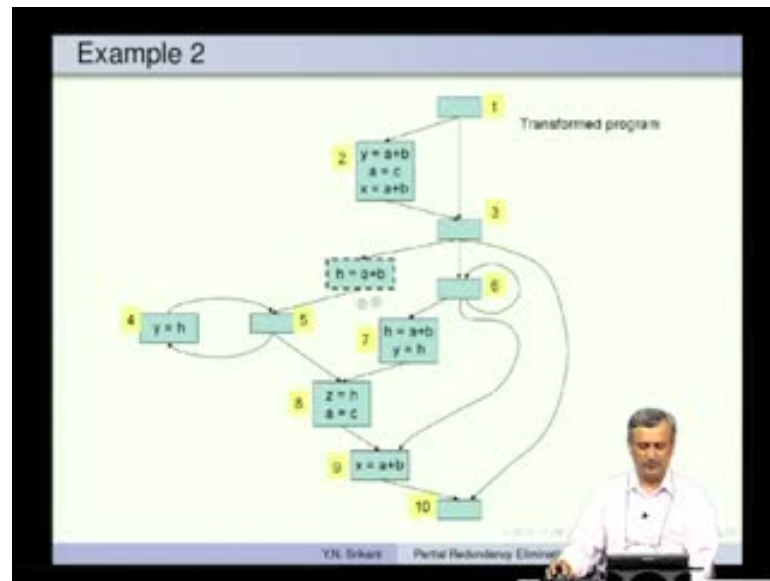
Of course, not only that, when we inserted something here, we took away something at this point.

(Refer Slide Time: 53:55)



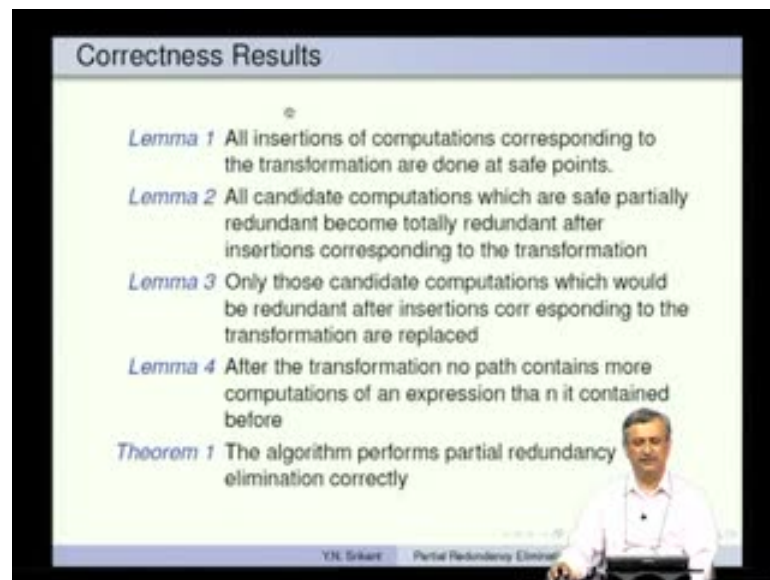
We had a computation here along this path by breaking this edge (Refer Slide Time: 53:59) we introduce a computation here but, we took away that computation.

(Refer Slide Time: 54:00)



This computation remind, its only question of making it h equal to a plus b from x equal to a plus b.

(Refer Slide Time: 54:13)



The theorem says the algorithm performs partial redundancy elimination correctly; it is based on these lemmas.

(Refer Slide Time: 54:20)

**Optimality Results**

*Lemma 5* A candidate computation is not replaced by the transformation if and only if it is an isolated computation

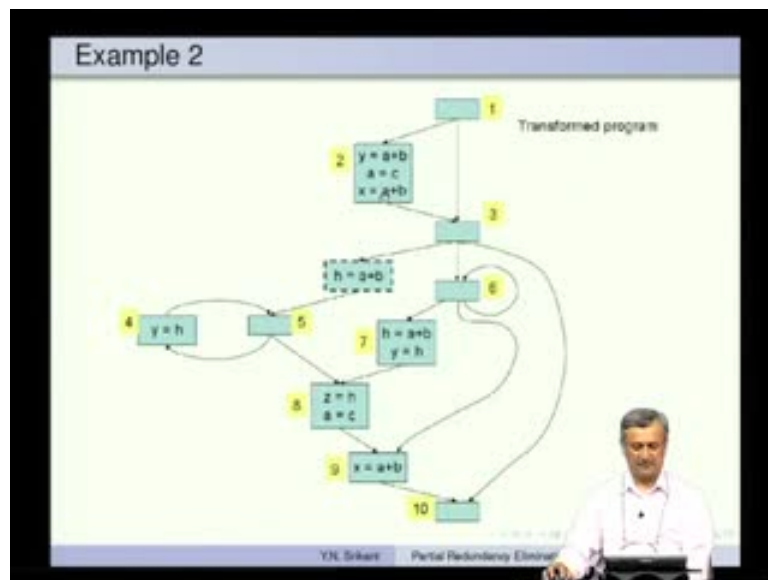
*Theorem 2* The transformation is computationally optimal, i.e., there does not exist any other correct transformation with less number of computations of an expression on any path

*Theorem 3* The transformation is lifetime optimal, i.e., the transformation keeps the live ranges of the newly introduced temporaries to the minimum

Y.N. Srikant Partial Redundancy Elimination

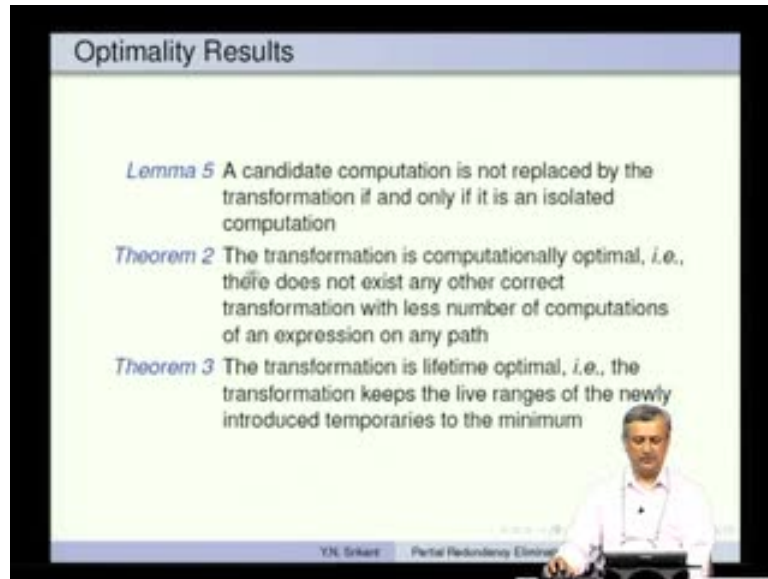
The fifth lemma says, a candidate computation is not replaced by the transformation, if and only if it is an isolated computation, this is what I was saying. So, unnecessary computations are not really replaced.

(Refer Slide Time: 54:40)



In this case for example, this computation is not replaced; this computation is not replaced etcetera.

(Refer Slide Time: 54:47)



**Optimality Results**

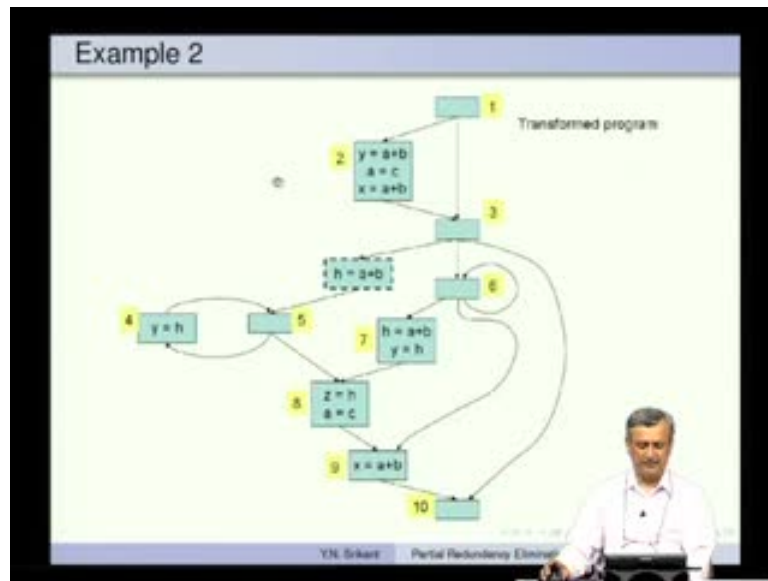
- Lemma 5* A candidate computation is not replaced by the transformation if and only if it is an isolated computation
- Theorem 2* The transformation is computationally optimal, i.e., there does not exist any other correct transformation with less number of computations of an expression on any path
- Theorem 3* The transformation is lifetime optimal, i.e., the transformation keeps the live ranges of the newly introduced temporaries to the minimum

Y.N. Srikant Partial Redundancy Elimination

Theorem 2 says the transformation is computationally optimal. This is an important result that is, there does not exist any other correct transformation with less number of computations of an expression on any path. We have introduced the minimum number of computations necessary. We have not introduced even one extra computation, everything is optimal.

Finally, the theorem 3 says the transformation is lifetime optimal that is, the transformation keeps the live ranges of the newly introduced temporaries to the minimum.

(Refer Slide Time: 55:23)



What this really says, is we have introduced these temporaries  $h$ ,  $h_1$ ,  $h_2$ , and etcetera. At the points where its live range becomes the smallest, I could not have pushed this to this point or this point without sacrificing either safety or any other property.

(Refer Slide Time: 55:46)

The slide, titled "Optimality Results", contains the following text:

- Lemma 5** A candidate computation is not replaced by the transformation if and only if it is an isolated computation
- Theorem 2** The transformation is computationally optimal, i.e., there does not exist any other correct transformation with less number of computations of an expression on any path
- Theorem 3** The transformation is lifetime optimal, i.e., the transformation keeps the live ranges of the newly introduced temporaries to the minimum

That is why this property becomes important, the theorem becomes important. It is lifetime optimal, so live ranges are kept to the minimum.

So, this is an overview of the partial redundancy elimination algorithm with examples. We have not really proved any theorems, but I hope it has given you an insight into how the algorithm really works. Thank you very much; this is the end of the lecture.