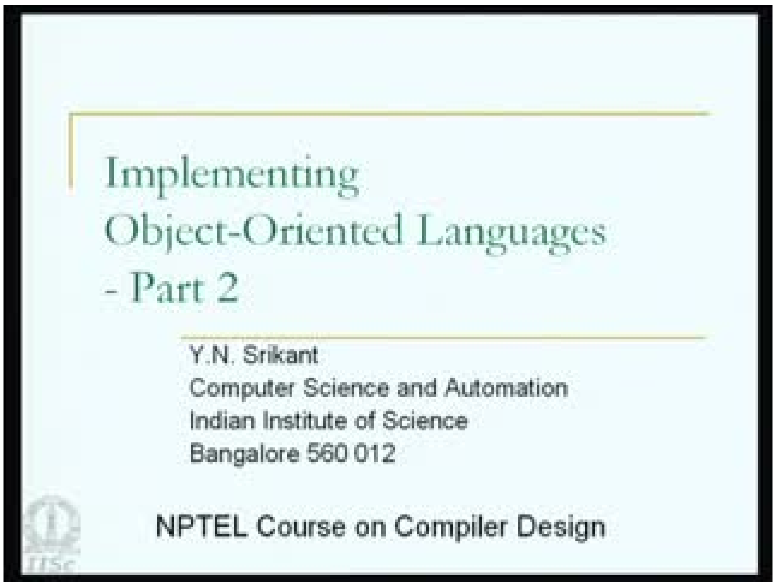
Compiler Design

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

Module No. # 06
Lecture No. # 11
Implementing Object-Oriented Languages-Part 2 and Introduction to Machine-Independent Optimizations

Welcome to part 2 of the lecture on object oriented languages and their implementation.

(Refer Slide Time: 00:21)

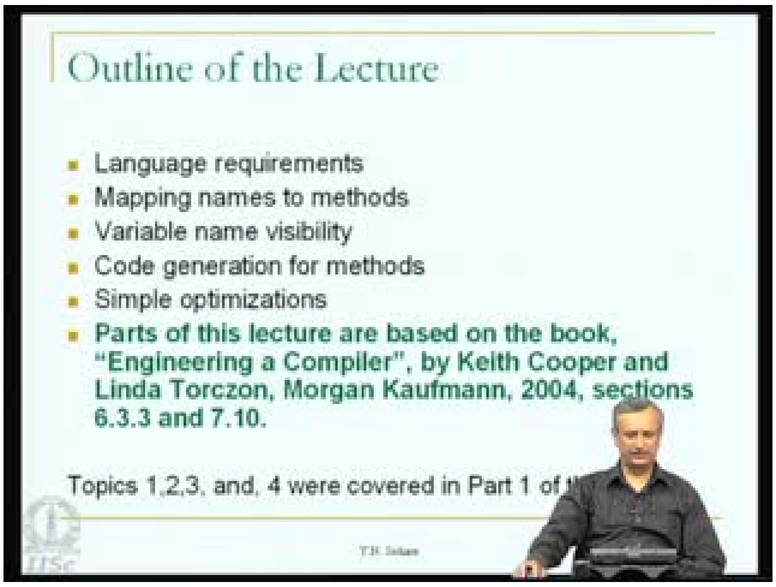


Implementing Object-Oriented Languages - Part 2

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

(Refer Slide Time: 00:27)



Outline of the Lecture

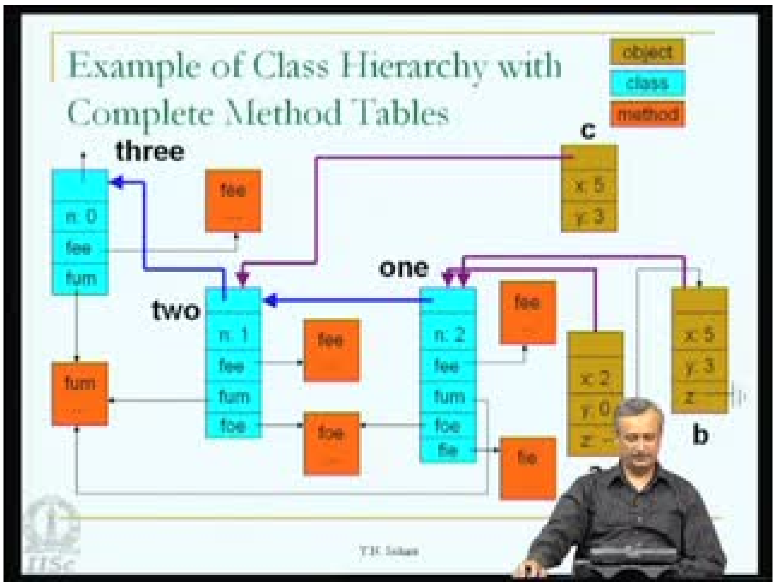
- Language requirements
- Mapping names to methods
- Variable name visibility
- Code generation for methods
- Simple optimizations
- Parts of this lecture are based on the book, "Engineering a Compiler", by Keith Cooper and Linda Torczon, Morgan Kaufmann, 2004, sections 6.3.3 and 7.10.

Topics 1,2,3, and, 4 were covered in Part 1 of t

Last lecture - covered topics from 1, 2, 3 and 4 that are language requirements, how to map names to methods, visibility of variable names and code generation for methods.

Today, after a very quick review of the code generation we will look at simple optimizations.

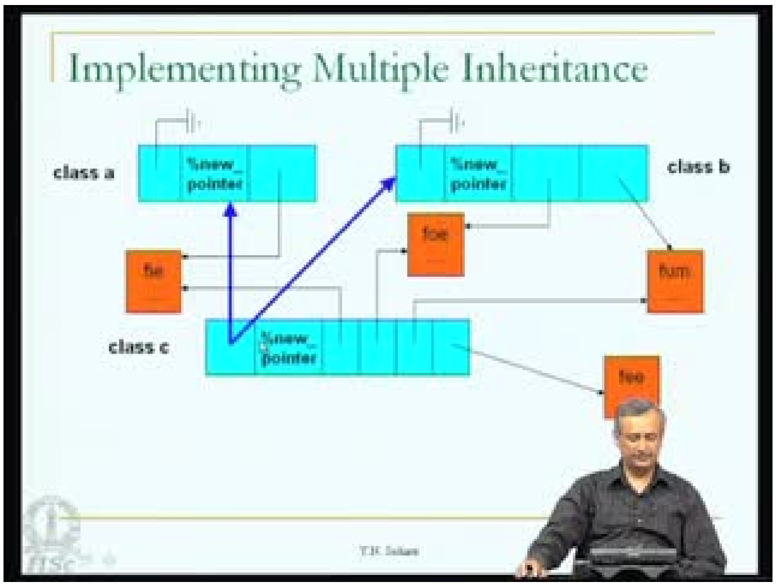
(Refer Slide Time: 00:47)



Example of Class Hierarchy with Complete Method Tables

We saw this picture in the last lecture, it just shows example of a class hierarchy with complete method tables. So, the idea is to make the method invocation faster, each class stores a complete method table. Whenever, an object wants to invoke a method, the method table is looked up and the offset of the method is applied and then the method is called. So, this works whether the method is inherited or the method is implemented right, in the present class.

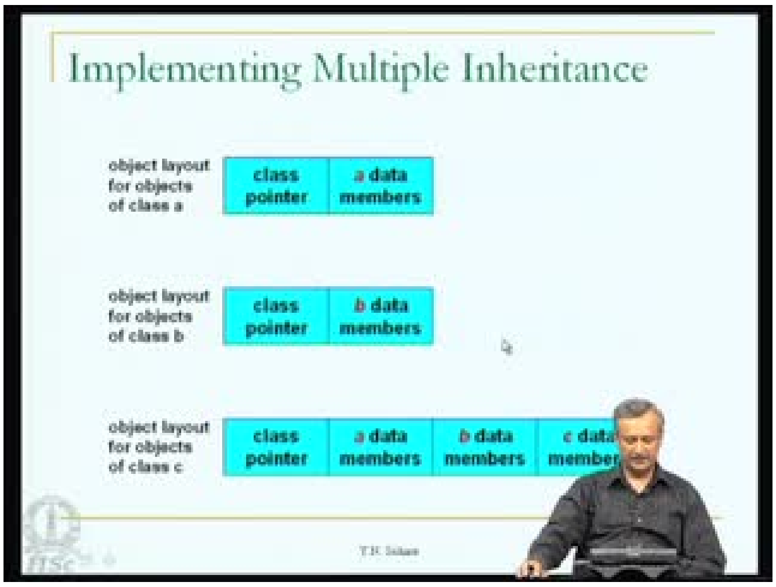
(Refer Slide Time: 01:28)



Implementing Multiple Inheritance

There are no problems in implementing single inheritance. It is very simple but, implementing multiple inheritance - that is, a single class inheriting from two other classes; for example, class c inheriting from class a, class b creates some difficulties.

(Refer Slide Time: 01:52)

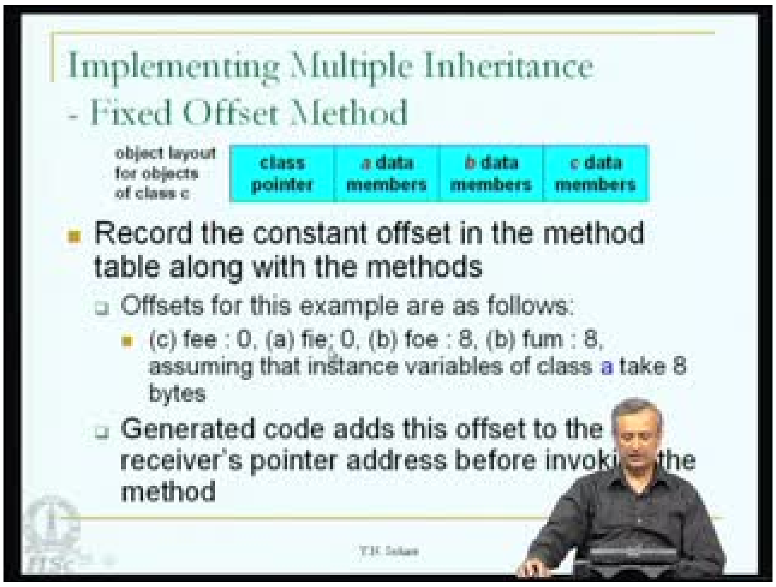


Implementing Multiple Inheritance

object layout for objects of class a	class pointer	a data members		
object layout for objects of class b	class pointer	b data members		
object layout for objects of class c	class pointer	a data members	b data members	c data members

For example, let us say the object records of class a have just a data members, class b have b data members but, class c which inherits both from a and b has space for a data members, b data members and c data members. The difficulty is in calling methods and how to address this difficulty is what we studied in the last class. Let us review it very quickly.

(Refer Slide Time: 02:15)



Implementing Multiple Inheritance - Fixed Offset Method

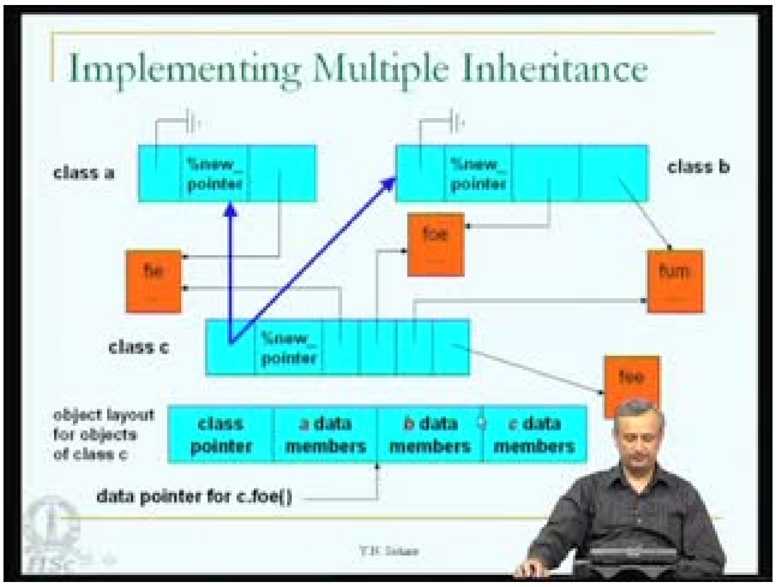
object layout for objects of class c	class pointer	a data members	b data members	c data members

- Record the constant offset in the method table along with the methods
  - Offsets for this example are as follows:
    - (c) fee : 0, (a) fie: 0, (b) foe : 8, (b) fum : 8, assuming that instance variables of class a take 8 bytes
  - Generated code adds this offset to the receiver's pointer address before invoki the method

Now, the difficulty is when we are actually calling some methods from class c if the methods belong to class a, then there is no problem because the members of class a are in the right position. If the methods belong to class c, then also there is no problem because class a and b have been compiled with class c; the problem starts, when we invoke methods of class b from objects of class c. The problem is when we invoke methods of class b from objects of type class b, the data members of class b are supposed to be right here, in place of a data member but, they are now at an offset.

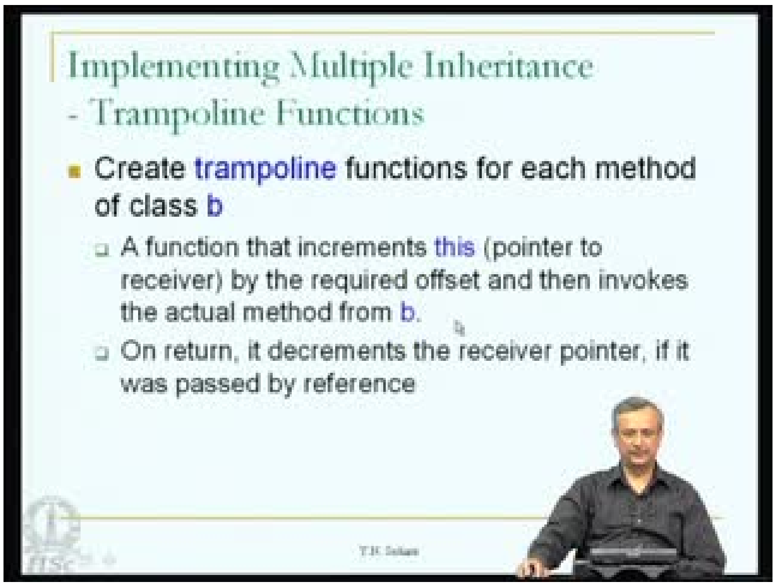
We need to actually provide this offset information; this can be done by 2 methods. We can store the constant offset in the method table along with the methods and the generated code adds this offset to the receiver's pointer address before invoking the method.

(Refer Slide Time: 03:33)



This is what I was saying, so the data pointer for c dot foe should be here, because it belongs to class b if it points to here or here it is wrong (Refer Slide Time: 03:43).

(Refer Slide Time: 03:45)



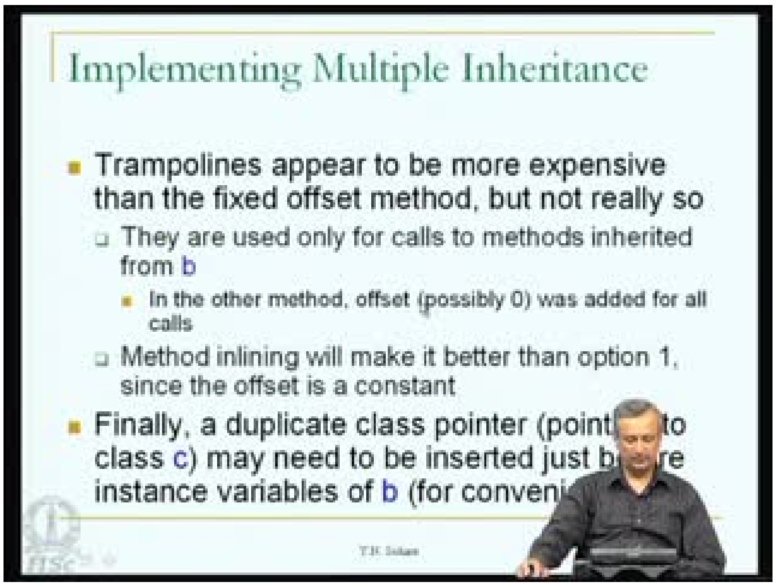
Implementing Multiple Inheritance - Trampoline Functions

- Create trampoline functions for each method of class b
  - A function that increments this (pointer to receiver) by the required offset and then invokes the actual method from b.
  - On return, it decrements the receiver pointer, if it was passed by reference

Y.N. Srikant

The other method is using trampoline function. So, these are functions created for each method of class b and they increment this pointer by the required offset and then invoke the actual method and of course, there is a corresponding decrement after the return.

(Refer Slide Time: 04:04)



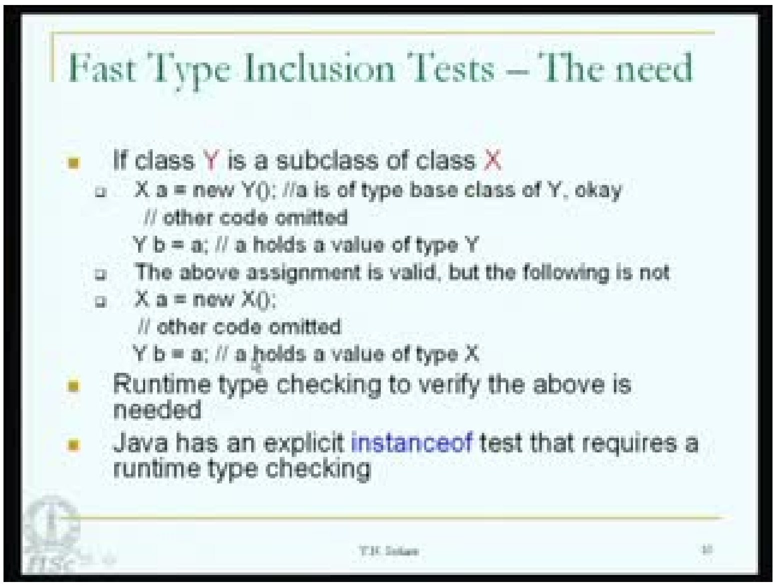
Implementing Multiple Inheritance

- Trampolines appear to be more expensive than the fixed offset method, but not really so
  - They are used only for calls to methods inherited from b
    - In the other method, offset (possibly 0) was added for all calls
  - Method inlining will make it better than option 1, since the offset is a constant
- Finally, a duplicate class pointer (point to class c) may need to be inserted just b re instance variables of b (for conveni

Y.N. Srikant

So, there is nothing much to choose between the 2 methods. It may appear trampolines are very expensive but, they are not because they can be inlined and the offsets are all constant and so on and so forth.

(Refer Slide Time: 04:21)



Fast Type Inclusion Tests – The need

- If class Y is a subclass of class X
  - X a = new Y(); //a is of type base class of Y, okay
    // other code omitted
    Y b = a; // a holds a value of type Y
  - The above assignment is valid, but the following is not
  - X a = new X();
    // other code omitted
    Y b = a; // a holds a value of type X
- Runtime type checking to verify the above is needed
- Java has an explicit instanceof test that requires a runtime type checking

Y.N. Srikant

We now look at simple optimizations, which are required in implementing object oriented languages. First one is the type inclusion tests must be made very fast, let us see why? If class Y is a subclass of class X, let us assume so. Then, suppose we declare an object a of class X and initialize it with an object of type Y; so class Y is subclass of class X, a happens to be base class pointer and this is a object of type Y and there is nothing wrong, this is perfectly ok.

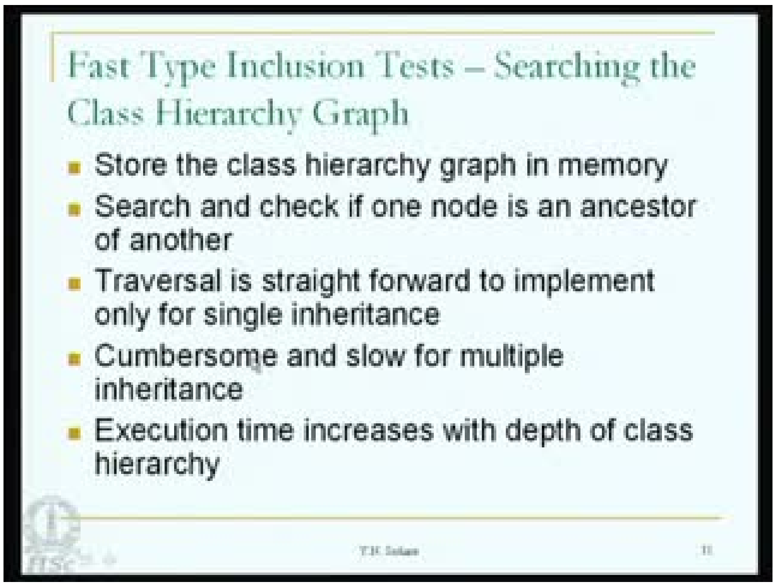
So, a base class pointer can point to any derived type objects, then we can also say Y b equal to a. We are declaring another variable b of type Y and assigning the value a to it which is fine. If a holds a value of type Y because of this previous one, a was pointing to some object of type Y. The assignments above these two are perfectly valid; there is nothing wrong, but the following assignment is not valid.

Suppose, we declare a variable a of type X and assign it an object of type X but then, we declare another variable b of type Y and assign a. Now, this a was holding an object of type X and this b is of type Y; Y is a subclass of class X, so you cannot assign b equal to a. The other way, when a actually was new Y and it was pointing to some object of Y, this assignment would have been perfectly ok; now it is not.

When we make such assignments either allow them or do not allow them. It is necessary to conduct runtime type checking to verify that the above is correct or this assignment is needed and so on and so forth, this is the problem. This cannot be done at compile time, because there could be conditional code, which cannot be completely evaluated at compile time.

So, Java also has an explicit instance of test that requires runtime type checking. This can be only checked at runtime. The type of b is known only at runtime if there is conditional code and all that. Type of a will also be known only at runtime; whether this b and a are compatible actually is to be checked at runtime.

(Refer Slide Time: 07:25)



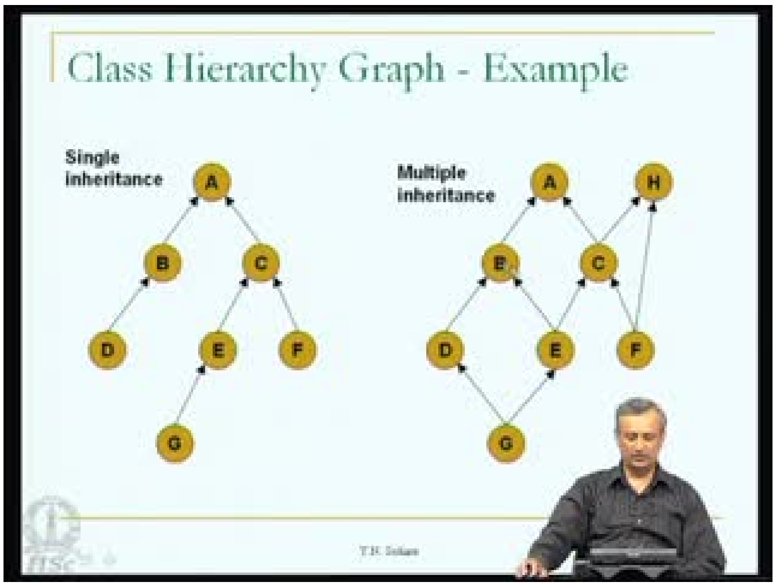
Fast Type Inclusion Tests – Searching the Class Hierarchy Graph

- Store the class hierarchy graph in memory
- Search and check if one node is an ancestor of another
- Traversal is straight forward to implement only for single inheritance
- Cumbersome and slow for multiple inheritance
- Execution time increases with depth of class hierarchy

Y.N. Srikant 11

How do we do such tests? Whether you know, a class, an object of a particular class is correct in making this type of an assignment is to be checked. So, to do this we need to check whether something belongs in the class hierarchy; whether a is a subclass of b or b is a subclass of a etc.

(Refer Slide Time: 08:03)



Class Hierarchy Graph - Example

There are many methods; we will look at one or two of them. We can store the entire class hierarchy graph in memory. Here is an example, of how a class hierarchy graph looks like; this is for single inheritance, this is for multiple inheritances. A is the root, so there is nothing beyond A; A is not going to derive from any other class. B and C derive from A, D derives from B, E and F derives from C, and G derives from E. This is our class hierarchy graph for single inheritance.

For multiple inheritance, there are differences. For example, E derives from B and C, F derives from C and H, C derives from A and H, G derives from D and E etc. So, this is the multiple inheritance part. Now, if you want to check, whether one node is an ancestor of the other; that is A is an ancestor of D, C is an ancestor of E, C is an ancestor of G etc; B is not an ancestor of F, so these are all to be checked. This is very simple to do, we anyway have a pointer to the class record from the object and we have this class hierarchy graph.

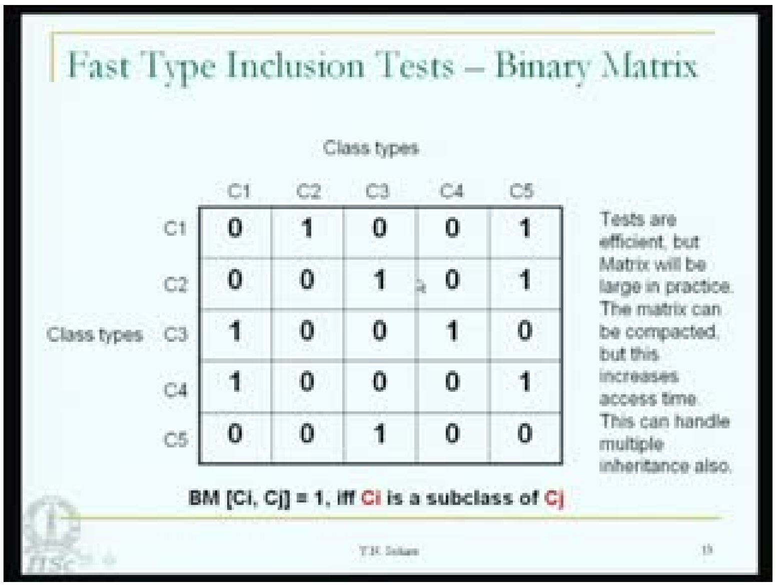
There will be a pointer in the class record to this particular node in the class hierarchy graph (Refer Slide Time: 9:19). We can go on traverse in the class hierarchy graph till the root and check whether one of the ancestors is met the way we wanted it. So, if you want to check whether A is a an ancestor of G, start from G and go upwards of course, we meet A; when we go to the till the root and we do not find any of these on the way then, if we did not find A, A would not be an ancestor of G. For example, we do not find B when we traverse from G towards A. This is a fairly straight forward traversal to implement only for the single inheritance, if it is multiple inheritance there is too much checking and it is not a very profitable issue.

For example here (Refer Slide Time: 10:15), suppose we want to check whether A is an ancestor of G. You know here, we need to check all paths from G towards A and we do not know which one of these go towards A and which one of these do not go towards A. So, we will have to check maybe G D B A, we will be very fortunate if we take this path, because we hit A right now and we can terminate the search; but if we had taken G E C and H we would have wasted a search and then we had to check G E B and A or maybe G E C and A and so on and so forth.

So, with a large inheritance in class hierarchy graph, checking inheritances in a multiple inheritance environment is not straight forward. You may actually have to do a large number of checks before we know whether it is true or false.

The execution time both for single inheritance and multiple inheritance increases with the depth of class hierarchy, this is clear. If there is very deep hierarchy, then you may have to go all the way till the root to determine whether some ((tech)) node is an ancestor of some other node and this ancestor relationship determines the class derivation or otherwise.

(Refer Slide Time: 11:34)



Fast Type Inclusion Tests – Binary Matrix

Class types

Class types		C1	C2	C3	C4	C5
	C1	0	1	0	0	1
	C2	0	0	1	0	1
Class types	C3	1	0	0	1	0
	C4	1	0	0	0	1
	C5	0	0	1	0	0

Tests are efficient, but Matrix will be large in practice. The matrix can be compacted, but this increases access time. This can handle multiple inheritance also.

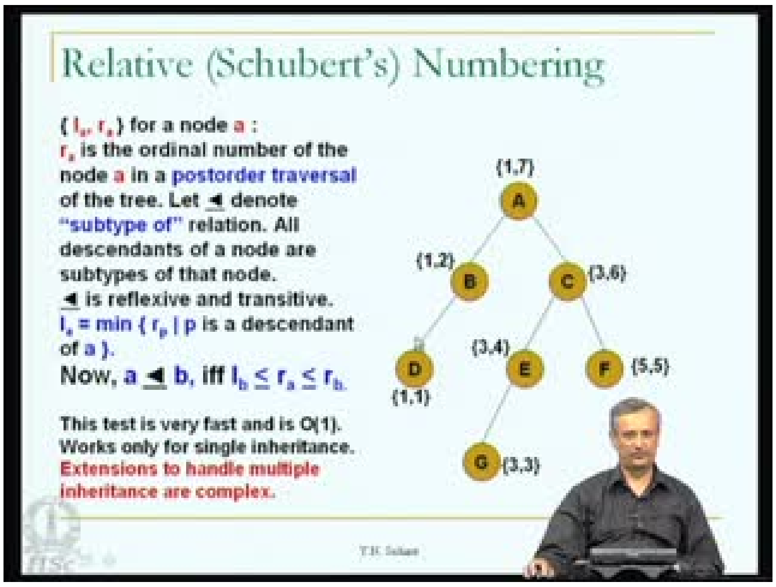
BM [Ci, Cj] = 1, iff Ci is a subclass of Cj

Y.N. Srikant 13

It is also possible to use a binary matrix. For example, consider this matrix you have class types on one side, class types on the other side - C1 C2 C3 C4 C5 and on the other side also. So, BM is a binary matrix of [Ci ,Cj] equal to 1, if and only if Ci is a subclass of Cj. If you take this row, C2 is a subclass of C3; that is why it is 1 but, C2 is not a subclass of C4, so it is zero. C2 is also a subclass of C5 so it is 1, so there is multiple inheritance. So single inheritance, multiple inheritance - all types can be actually shown using the binary matrix.

These tests are quite efficient to check but, matrix will be large in practice. So, if there are a large number of classes in the matrix and the hierarchy is large then, this matrix will also be very large. The matrix can be compacted but, this increases the access time. This is quite fast efficient but, the matrix is large. So, are there other solutions possible?

(Refer Slide Time: 12:45)



Relative (Schubert's) Numbering

{ l_a, r_a } for a node a :
r_a is the ordinal number of the node a in a postorder traversal of the tree. Let $\trianglelefteq$ denote "subtype of" relation. All descendants of a node are subtypes of that node.
$\trianglelefteq$ is reflexive and transitive.
$l_a = \min \{ r_p \mid p \text{ is a descendant of } a \}$.
Now, $a \trianglelefteq b$, iff $l_b \le r_a \le r_b$.

This test is very fast and is O(1).
Works only for single inheritance.
Extensions to handle multiple inheritance are complex.

Fortunately yes, that is for single inheritance the solution is very very efficient one. Let us consider a class hierarchy of this type (Refer Slide Time: 12:55), so A B etc, etc, etc this is a tree. Now let us say, we conduct a post order traversal of the tree. So, what we do is we first visit D and then we have to visit B and then you know we actually cannot go to A but, we have to come all the way to G, then we go to E, then we go to F, then C, then A. In other words, this is 1, this is 2, this is 3, this is 4, this is 5, this is 6 and this is 7 (Refer Slide Time: 13:21). These post order traversal numbers are all indicated as the second component of these pairs, the number pairs, which are indicated on each one of the nodes; so these are the labels.

The labels are (l_a, r_a) for a node A; r_a is the ordinal number of the node A in a post order traversal of the tree. Let this, solid less than equal to denote subtype of relation. So, all the descendants of a node are subtypes of that particular node. This is similar to that class hierarchy graph, so single inheritance only. This subtype is of reflexive and transitive definitely not symmetric, because the subtype of relation is not symmetric. So, it is definitely reflexive because A is a subtype of itself and transitive because if A is a subtype of B, B is a subtype of C and then, in this case for example, B is a subtype of A, D is a subtype of B so D is a subtype of A, so it is transitive.

Now, what is the first component? l_a .So, this is the minimum post order traversal number among the descendants of that particular tree. It so happens that in this case, this is 1, right; so we start this is 1, and this will get a number 1(Refer Slide Time: 14:51). If we had taken this particular node, the minimum number is 3 so that, left component becomes 3 and so on and so forth. That is how this is numbered, so l_a is a minimum of r_p

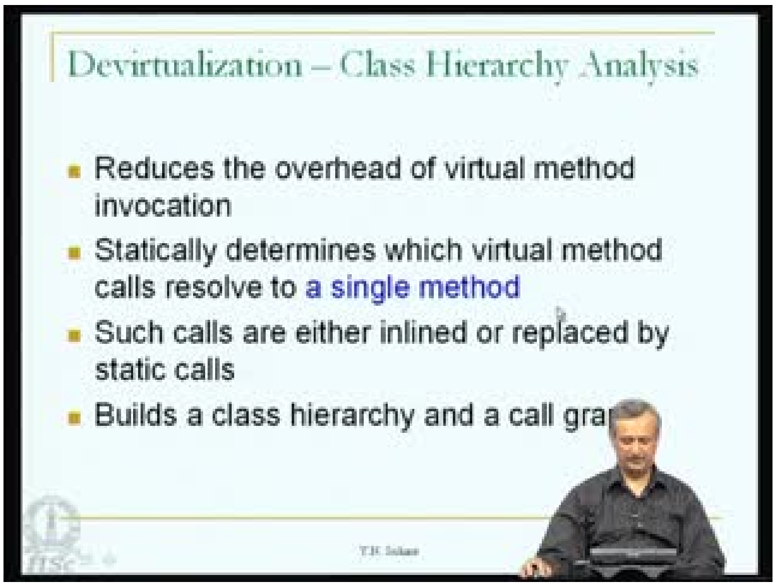
where p is a descendant of a. You take the entire post order numbering, take the minimum of all those descendants post order numberings and give that as the l_a of that particular node.

Now, the subtype of relationship can be very easily determined using this condition: if a is the subtype of b, if and only if l_b is less than or equal to r_a less than or equal to r_b; these are numbers and these can be checked in one time very simple, only 2 tests (Refer Slide Time: 15:37).

Let us verify this; say for example, we take A and we take G. So, G is a subtype of A that is what we need to determine. So l_b is 1 here, r_a is 3, so 1 is less than or equal to 3, r $_a$ is 3 and r_b is 7, so r_a less than or equal to r_b is also satisfied (Refer Slide Time: 15:55). If you are trying the other way, whether A is the subtype of 3, it is automatically not satisfied because in that case l_b is 3 and r_a is 7, so 3 less than or equal to 7 is satisfied but, 7 less than or equal to 3 is not satisfied. So, this makes the inequality not satisfied and violated, so the relationship is not true.

This test is very fast it is O(1) but, it works only for single inheritance. There are many extensions of this using slices of these hierarchy trees and implementing them using what are known as PQ trees etc. These are far more complex and beyond the scope of this particular lecture.

(Refer Slide Time: 16:52)



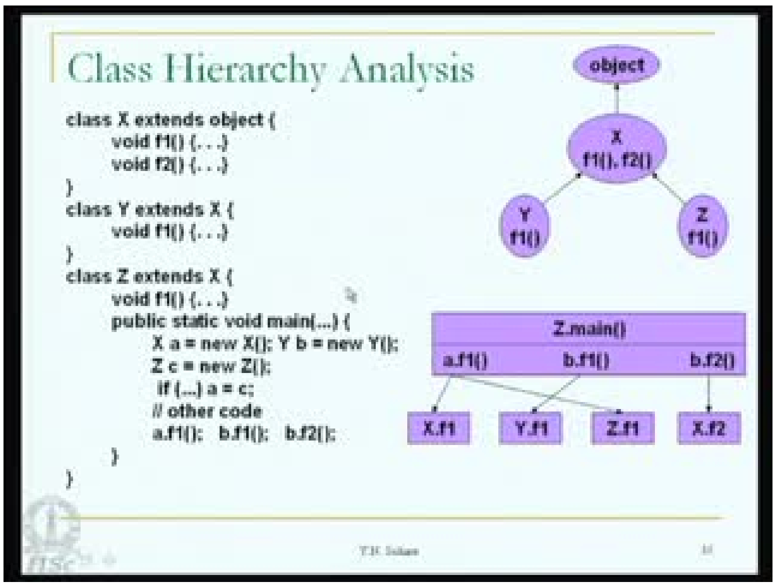
Devirtualization – Class Hierarchy Analysis

- Reduces the overhead of virtual method invocation
- Statically determines which virtual method calls resolve to a single method
- Such calls are either inlined or replaced by static calls
- Builds a class hierarchy and a call gra

The second simple optimization which we need to do, the first one for single inheritance was that Schubert's numbering or relative numbering that we studied. De-virtualization is a very simple optimization but, not so easy to implement very efficiently. If it is implemented in a very simplistic manner, it is simple, very efficiently is a difficult task. So, using class hierarchy analysis this is we are going to study only one method.

What is de-virtualization? It reduces the overhead of virtual method invocation, how? We try to statically determine which virtual method calls resolve to single methods. So, if there is a virtual method, we want to see if that can be replaced by a static call. Then, we do not have to really use the virtual table, method table, and call the method at all; we can actually replace it by a static call by putting the name of the class and the method directly. Such calls are either in-lined or replaced by static calls, so that is whatever resolves to single method. I will show you an example of this and we build a class hierarchy and a call graph along with it.

(Refer Slide Time: 18:21)



Class Hierarchy Analysis

```
class X extends object {
      void f1() {. . .}
      void f2() {. . .}
}
class Y extends X {
      void f1() {. . .}
}
class Z extends X {
      void f1() {. . .}
      public static void main(...) {
            X a = new X(); Y b = new Y();
            Z c = new Z();
             if (...) a = c;
            // other code
            a.f1();   b.f1();   b.f2();
      }
}
```

object
X f1(), f2()
Y f1()
Z f1()
Z.main()
a.f1()
b.f1()
b.f2()
X.f1
Y.f1
Z.f1
X.f2

Take this simple example; object is the Java super class from which all classes actually have to derive. So, X is a class deriving from object, Y derives from X and Z derives from X. So, X has two functions, methods, here f1 and f2, Y has a method f1 which overrides f1of class X, Z also has another method f1 which overrides the method f1 of class X. Here is the declaration for class X and class Y; class Z also has the main, let us see what this is. So, main declares an object a of type X and assigns an object of type X to it. Similarly, b is a method or an object of type Y and is assigned an object of type Y and similarly, C is an object of type Z and it gets a new object of type Z. So these are the initial assignments.

Now if some condition is true, then a is assigned c; so this is perfectly valid because X is the base pointer and C is the derived class pointer, so a equal to c is perfectly justified.

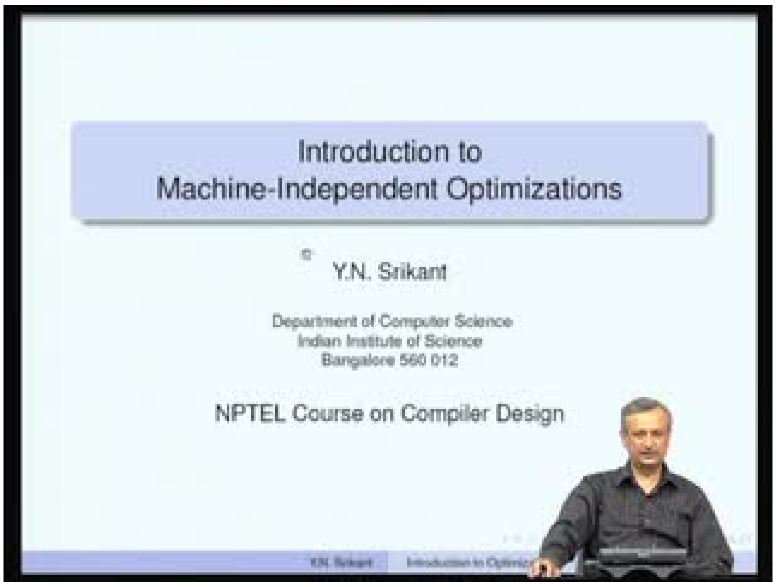
Now there are three calls here and let us see, what they mean? So a dot f1, b dot f1 and b dot f2.

So, this is indicated in the form of the calling graph here (Refer Slide Time: 20:02). So Z dot main as a dot f1, b dot f1 and b dot f2. So b dot f1 is always y dot f1, there is no other function at all and here, there is no question of, b does not point to any other object either. b dot f2 is always x dot f2, because you know b is an object of type Y and Y does not have f2, so it always resolves to the f2 of class X, but the situation is not simple with a dot f1.

If this code were to be false (Refer Slide Time: 20:42), then a would be pointing to an object of type X. In that case, it would be when a dot f1 is invoked it invokes x dot f1. If this condition were to be true, then a gets a pointer to c and therefore, in that case a dot f1 will mean Z dot f1, which is actually local to the class Z. So, there are two branches of this kind; so this particular call does not resolve to a single method it has2 possibilities; whereas, these two calls resolve to single method. We can replace, b dot f1 and b dot f2 by Y dot f1 and X dot f2 respectively and that actually takes care of de-virtualization here; but a dot f1 cannot be de-virtualized.

There are many more complicated schemes available for de-virtualization but, that will be beyond the scope of this lecture. So, we will conclude object oriented language implementation at this point, this is the end of this particular lecture and we will continue with introduction to optimizations very soon.

(Refer Slide Time: 21:53)



Introduction to
Machine-Independent Optimizations

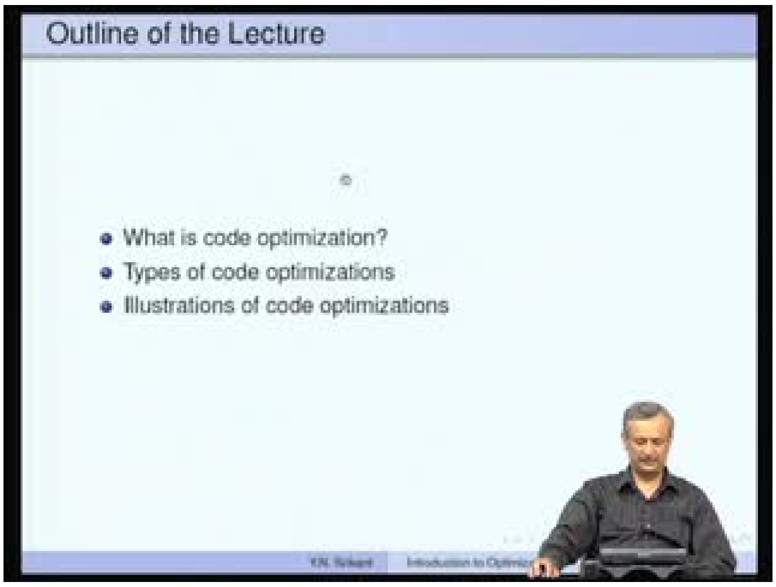
© Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Welcome to the lecture on Introduction to Machine-Independent Optimizations. So far we have seen, code generation runtime organization and then how to implement, register allocation, object orient languages and so on. Now, we start looking at machine independent intermediate code, look at a large number of optimizations that can be implemented on these and then see what information is needed in order to implement such optimizations.

(Refer Slide Time: 22:35)

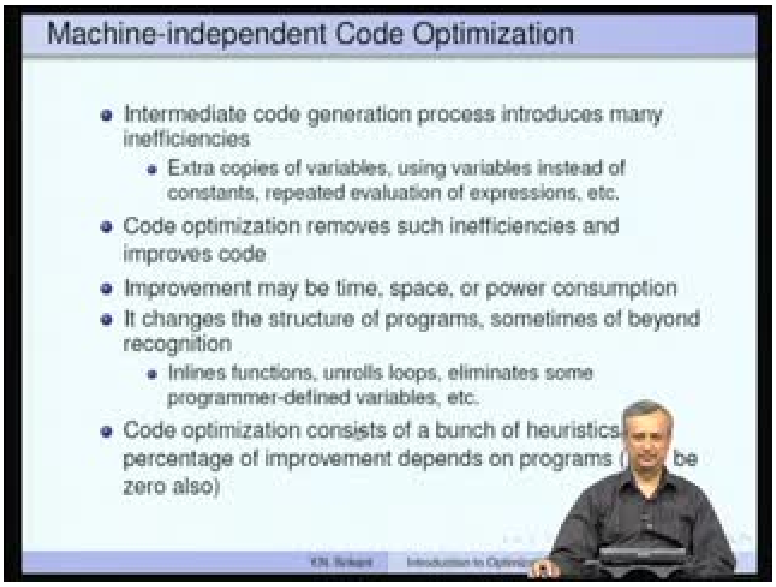


Outline of the Lecture

- What is code optimization?
- Types of code optimizations
- Illustrations of code optimizations

The outline of the lecture is very simple but, we are going to see a large number of examples to begin with and then study the algorithms for such optimizations. We will see what is code optimization? What are the different types of code optimizations? Then look at illustrations of many of these code optimizations.

(Refer Slide Time: 22:58)



Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
  - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics percentage of improvement depends on programs (be zero also)

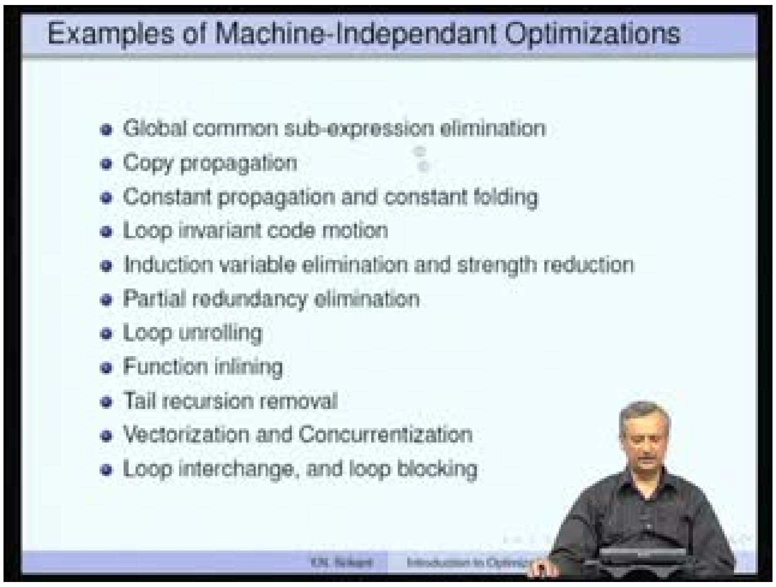
Many of you may recognize that this slide is from the introductory lecture that I gave. So, machine independent code optimization, intermediate code generation process introduces much inefficiency. You know it is a very simple strategy that we use for intermediate code generation; there are extra copies of variables, using variables instead of constants, repeated evaluation of expressions all these happens; everything that you think is very inefficient will happen during intermediate code generation.

Now, the code optimization phase or machine independent code optimization phase is supposed to remove such inefficiencies and improve the code. We will see examples of how this can be done very soon; this improvement may be time improvement, space improvement or power consumption improvement, any one of these can be improved but our emphasis in this lecture will be on improving time.

It changes the structure of the programs sometimes, beyond recognition also. For example, the some of the functions may be in-line, so you will not even have a function you had written one but, it does not exist anymore. It unrolls loops, so you had written i equal to 1 to 100, but now it may be running from i equal to 1to 25 and it maybe unrolled 4times. It eliminates some of the programmer defined variables; for example, induction variable such as loop counters maybe eliminated and some other variable in the loop maybe substituting for that particular loop counter variable.

Code optimization is essentially a heuristic approach. It consists of a bunch of heuristics, so the percentage improvement depends on the programs. A certain program, you may not be able to improve at all and for a certain type of program you may be able improve it by 50 percent; so no guarantee is on how much improvement can be made.

(Refer Slide Time: 25:22)



Examples of Machine-Independant Optimizations

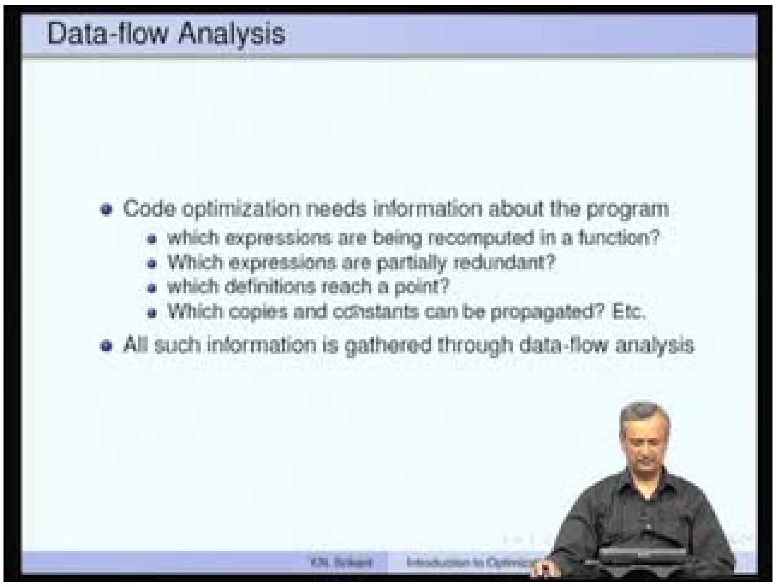
- Global common sub-expression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction
- Partial redundancy elimination
- Loop unrolling
- Function inlining
- Tail recursion removal
- Vectorization and Concurrentization
- Loop interchange, and loop blocking

These are the various types of optimizations that we are going to see through illustrations. It does not mean that, this list is exhaustive. There are many types of optimizations which are probably minor and probably some of which can be carried out with only at the time of machine code generation. We will limit ourselves to these optimizations and see how they work.

So, first one is global common sub-expression elimination, second is copy propagation then constant propagation and constant folding, loop invariant code motion, induction variable elimination and strength reduction, partial redundancy elimination, loop unrolling, function in lining, tail recursion removal, vectorization and concurrentization, loop interchange, and loop blocking. We have seen some of this optimization in different flavors before.

For example, we saw common sub expression elimination, but that was within basic blocks. We know how to tackle it; we build a direct the cycle graph using value numbering technique and constant propagation etc and constant folding can also be done within the basic blocks using the same value numbering technique. So, let us continue and see what other types of, how these things work and so on.

(Refer Slide Time: 26:56)

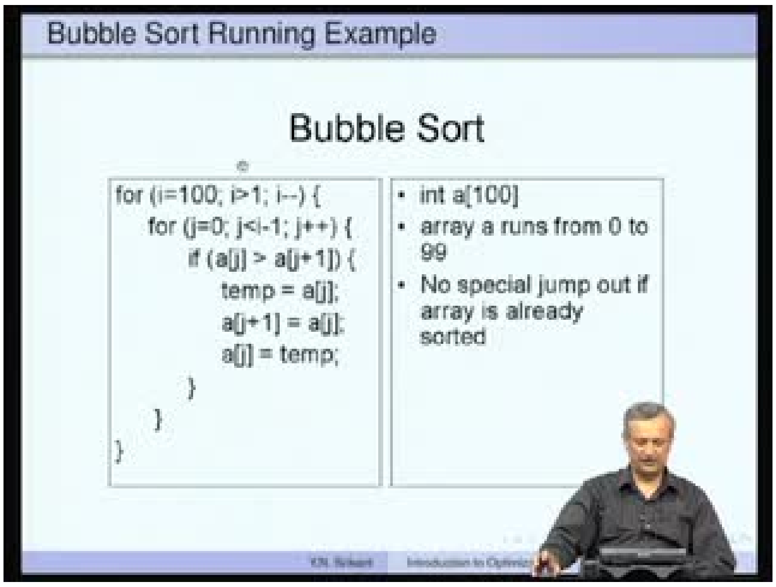


Data-flow Analysis

- Code optimization needs information about the program
  - which expressions are being recomputed in a function?
  - Which expressions are partially redundant?
  - which definitions reach a point?
  - Which copies and constants can be propagated? Etc.
- All such information is gathered through data-flow analysis

Code optimization needs information about the program. For example, which expressions are being recomputed in a function? Which expressions are partially redundant? Which definitions reach a point? partial redundancy will be clear very soon. Which definitions reach a point and which copies and constants can be propagated? Such information is gathered through data flow analysis, and our discussion of data flow analysis will begin after this introductory lecture on optimization.

(Refer Slide Time: 27:34)



Bubble Sort Running Example

Bubble Sort

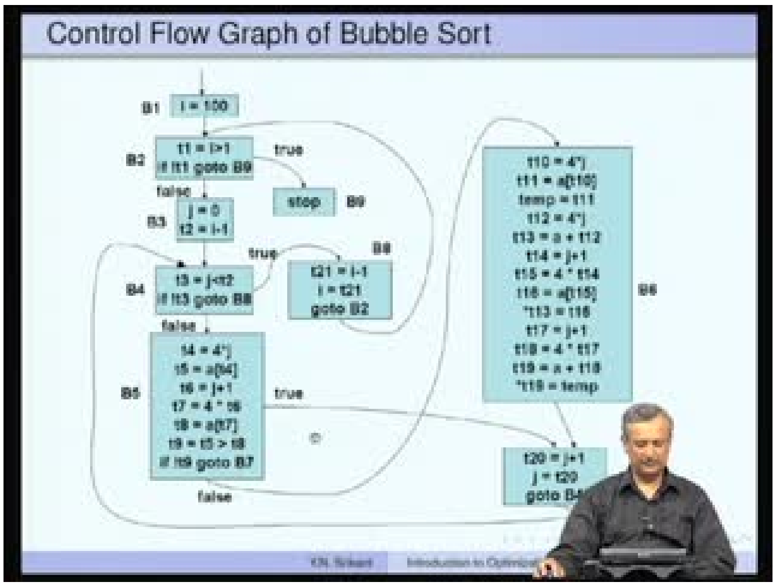
```
for (i=100; i>1; i--) {
    for (j=0; j<i-1; j++) {
        if (a[j] > a[j+1]) {
            temp = a[j];
            a[j+1] = a[j];
            a[j] = temp;
        }
    }
}
```

- int a[100]
- array a runs from 0 to 99
- No special jump out if array is already sorted

Y.N. Srikant | Introduction to Optimiz

For a few examples, we will consider a standard program of bubble sort which everybody understands. Here is a very simple version of that: i equal to 100, i greater that 1, i minus minus; so here is the outer loop and there is another inner loop j equal to 0, j less than i minus 1, j plus plus. So, the array used is of 100 size and it runs from 0 to 99 and we have not used any special jump out if array is already sorted; we did not used any flag and all that. Here, if a j is greater than a j plus1 then, we do a swap operation otherwise we just increment j and go to the next one. At the end of this entire procedure of this j pass, we would have found the largest element and next during the next i pass we find the next largest element and so on and so forth. So, that is really bubble sort.

(Refer Slide Time: 28:45)



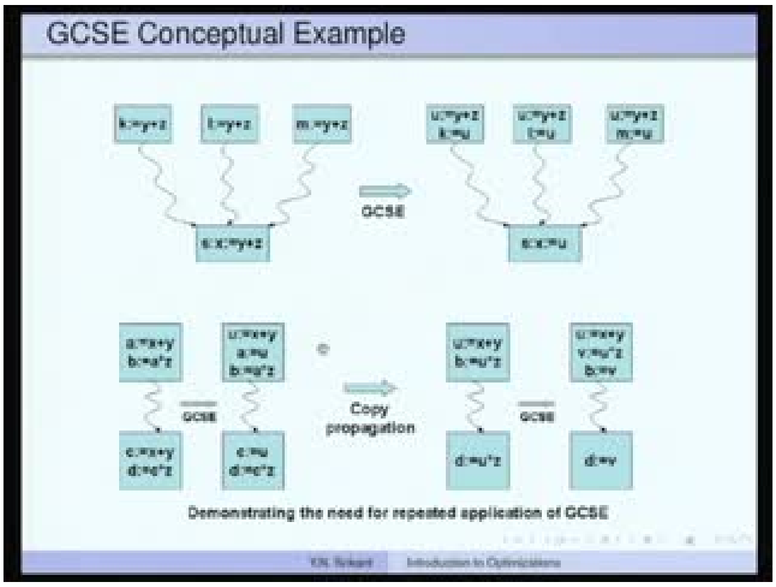
Control Flow Graph of Bubble Sort

```
B1: i = 100

B2: t1 = i>1
    if !t1 goto B9

B3: j = 0
    t2 = i-1

B4: t3 = j<t2
    if !t3 goto B8

B5: t4 = 4*j
    t5 = a[t4]
    t6 = j+1
    t7 = 4 * t6
    t8 = a[t7]
    t9 = t5 > t8
    if t9 goto B7

B6: t10 = 4*j
    t11 = a[t10]
    temp = t11
    t12 = 4*j
    t13 = a + t12
    t14 = j+1
    t15 = 4 * t14
    t16 = a[t15]
    *t13 = t16
    t17 = j+1
    t18 = 4 * t17
    t19 = a + t18
    *t19 = temp

B7: t20 = j+1
    j = t20
    goto B4

B8: t21 = i-1
    i = t21
    goto B2

B9: stop
```

Y.N. Srikant | Introduction to Optimiz

Here is the intermediate code and the control flow graph for bubble sort (Refer Slide Time: 28:48). Start with i equal to 100, then i greater than1 is evaluated into t1, so instead of saying if t1 go to B3 otherwise exit. We have said if not t1 go to B9 and if that is true go to exit, otherwise if it is false you jump you fall through. Then here starts the j loop with j equal to 0 and the upper bound of the loop is i minus 1 (Refer Slide Time: 29:19), it will check the upper bound if it is violated, you jump out; if it is not violated, you continue with the rest of the program.

(Refer Slide Time: 30:21)



GCSE Conceptual Example

Demonstrating the need for repeated application of GCSE

Y.N. Srikant Introduction to Optimizations

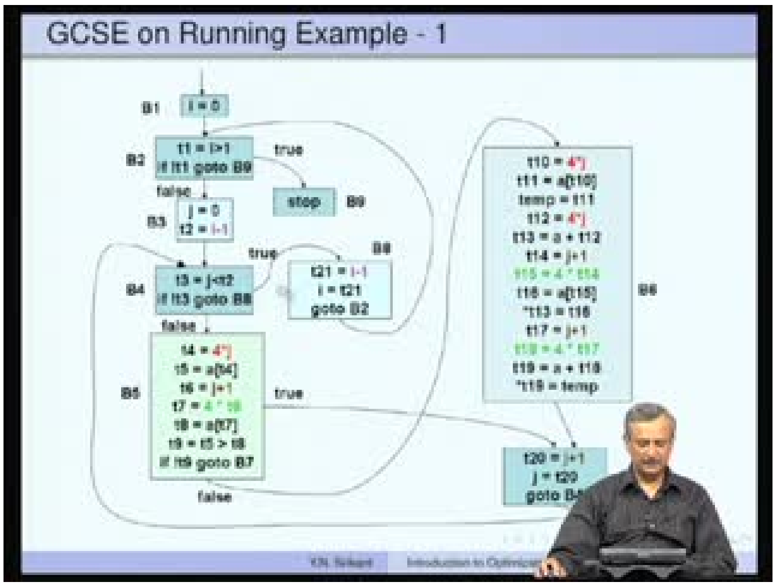
You can now see how much code gets generated just for a very simple check. It is not as if things are trivial, but just to check whether if t5 greater than t8 etc; then here is aj greater than aj plus 1, here is the swap routine (Refer Slide Time: 36:06). So, just for doing some simple things like this; we generate a large number of temporaries and then there are many opportunities for common sub expression elimination, so this will be clear in the next example after we see what GCSE is.

The global common sub-expression elimination conceptually is very simple. Here you have y plus z and it is possible that along all the paths which come to this particular basic block (Refer Slide Time: 30:33). There is an evaluation of y plus z and there is no modification of either y or z along any of these paths. In such a case, it is very simple to see that we do not have to evaluate this y plus z all over again. You store it into the same temporary and then for example, u equal to y plus z , u equal to y plus z and u equal to y plus z and use that u here. So, this is global common sub expression elimination. It is also possible that, because of deep sub expressions a single application of GCSE does not give results to the fullest extent possible; so, let us see. Here is a equal to x plus y, then b equal to a star z and this has a path to c equal to x plus y and d equal to c star z. So, x plus y is a very straight forward common sub-expression which can be eliminated, we do that u equal to x plus y a equal to u and c equal to u. So see that, we have used the same temporary for both for this assignment and for this assignment (Refer Slide Time: 31:48).

Now, this a equal to u is a copy operation (Refer Slide Time: 31:55). Since a has not been modified, we can use u in place of a; so this can become u star z and here also c equal to u is a copy operation, there is no modification of c between this and this (Refer Slide Time: 32:05). We can replace c by u and this becomes u star z. Now see, that because of the copy propagation we found another common sub expression, this is u star z in both places - so we can eliminate that also, introduce another temporary v equal to u tar z and d equal to v.

In this way, we may actually have to do several optimizations and then come back to global common sub-expression elimination. So, the order in which the optimizations have to be carried out is not well known. We do them in a particular order which is heuristically known to be good again and again until no more improvement is possible.

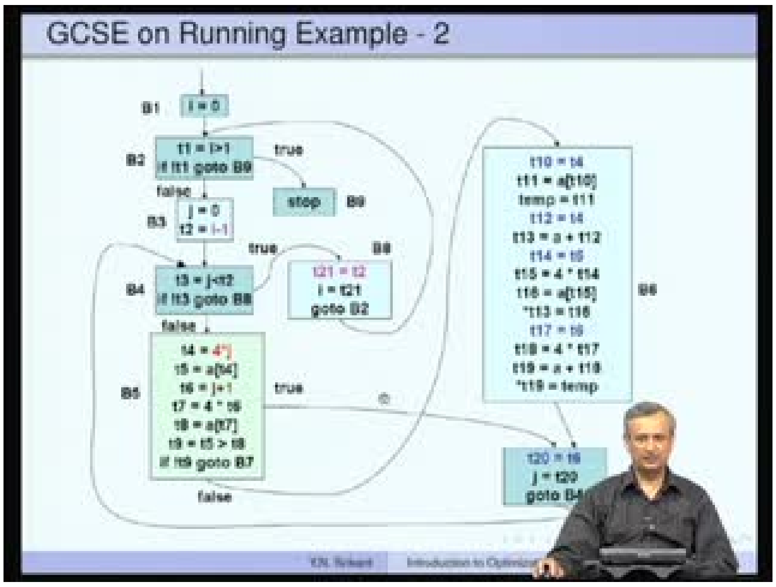
(Refer Slide Time: 32:51)



GCSE on Running Example - 1

So, coming back to the GCSE on the running example, let us see where the opportunities are. Here is i minus 1 for example right, so we have i minus 1 here also; possibly this and this are common sub-expressions. Here is 4 star j, here is 4 star j and here is also 4 star j (Refer Slide Time: 33:20); then 4 star t6 actually will become a common sub-expression only after a round of copy propagation but, for the present let us worry about 4 star j, i minus i and j plus 1 also. Here is j plus 1 and here is j plus1 one more instance of j plus 1.

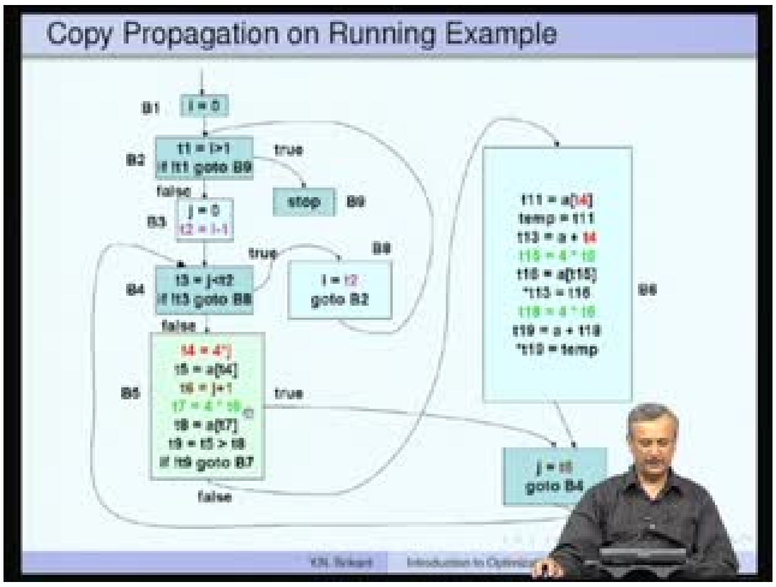
(Refer Slide Time: 33:47)



Now, if we look at it, this t2 is equal to i minus 1 is evaluated once and this also reaches this point (Refer Slide Time: 33:57). So, we can replace this i minus 1 by a simple t2 which was the previous evaluation of i minus 1; t4 is equal to 4 star j, so t4 is going to replace 4 star j here and 4 star j here also; t6 is j plus 1, so t6 is going to replace j plus 1 here and also j plus 1 here. This is very simple common sub-expression elimination on the running example.

Even in this simple example because of the inefficiencies in intermediate code generation we have got 1 2 and 3, if we apply it once more we will very easily be able to do some copy propagation. For example here (Refer Slide Time: 34:50), so this t4; t12 is equal to t4, t13 is equal to a plus t12, so t12 can be replaced by t4 and t14 equal to t6 so we can replace t14 by t6; so this becomes 4 star t6 again. So, this is another 4 star t6, this is one more 4 star t6, here t17 can also be replaced by t6, so this becomes 4 star t6; so both these can be replaced by t7. In this manner, common sub-expression elimination can be done very effectively even on this small example.

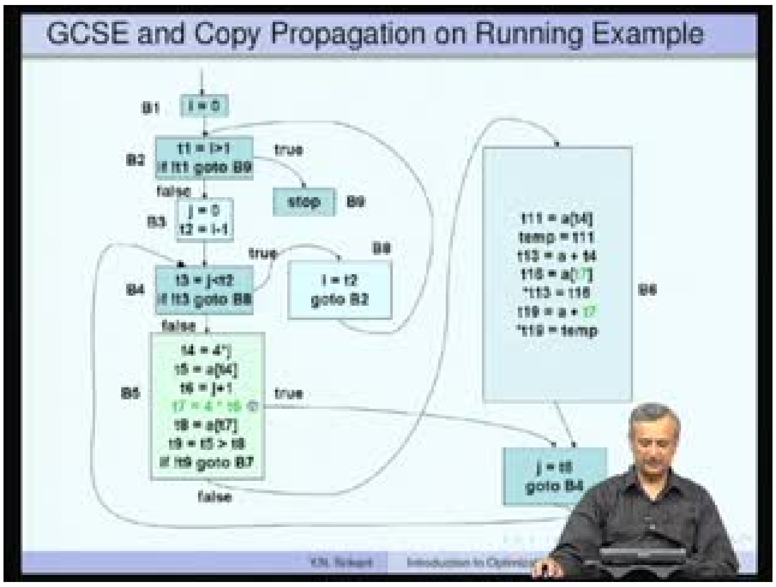
(Refer Slide Time: 35:25)



Copy Propagation on Running Example

That is what we have shown here (Refer Slide Time: 35:28), this becomes copy propagation; so we have eliminated the copies that I showed you. Here, this became t6, this became t6, this became t4 and this became t4 and so on and so forth (Refer Slide Time: 35:36). This is copy propagation on the running example. We did global common sub-expression elimination that was on i minus 1, 4 star j and j plus 1.Now, 4 star t6 is a common sub-expression that can also be eliminated but, that was obtained only after copy propagation.

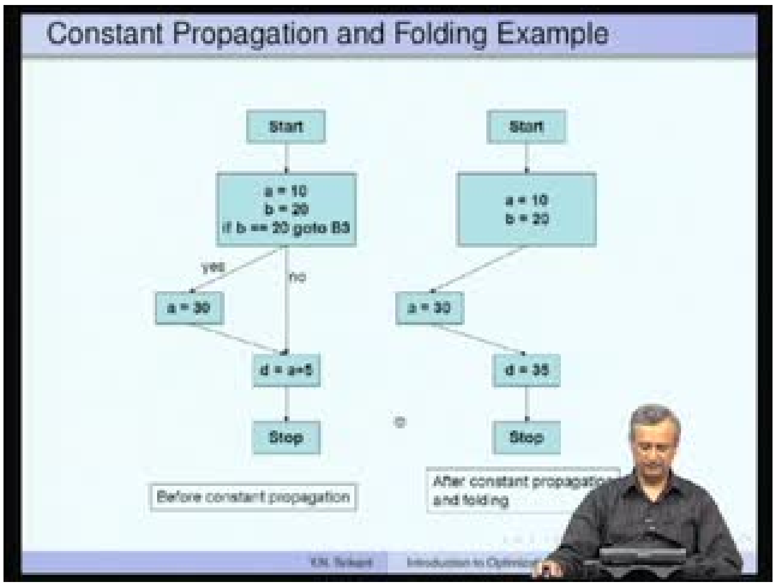
(Refer Slide Time: 36:07)



GCSE and Copy Propagation on Running Example

After the second round of common sub expression elimination the code becomes really efficient. Please see this (Refer Slide Time: 36:14), so we have replaced t7 here which is computed here .So all the common sub-expressions are computed only once and used many times here; so now this is a compact program.

(Refer Slide Time: 36:27)


Constant Propagation and Folding Example
Start
a = 10
b = 20
if b == 20 goto B3
yes
no
a = 30
d = a+5
Stop
Before constant propagation
Start
a = 10
b = 20
a = 30
d = 35
Stop
After constant propagation and folding
Y.N. Srikant
Introduction to Optimizations

What about the next optimization, so we have seen two of them - one is common sub-expression elimination, the other is copy propagation. What about constant propagation and constant folding? This is a very illustrative, nice example to show both. We have a equal to 10, b equal to 20 and a condition if b equal to 20, then go to b3 otherwise jump out; so yes come this side, no go this side. If it is b equal to 20 then a equal to 30, then come back d equal to a plus 5 and stop.

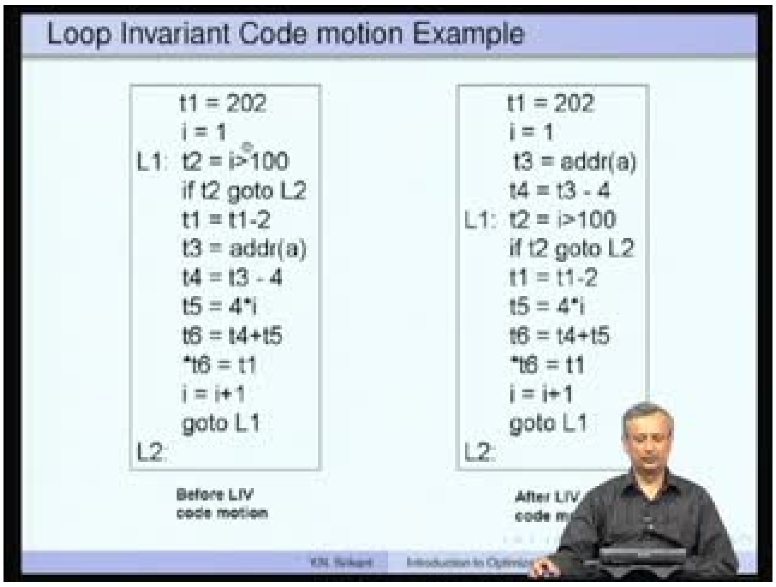
So, just observe that these are all constants which are known to the compiler. a equal to 10, 10 is known, so this compile time assignment is known; b equal to 20, so we know that by value numbering this b can take a value 20 very easy to find. Therefore, by checking b equal to 20 at this point, the conditional constant propagation algorithm knows that it is true, because b already has a value 20. So, this at compile time will evaluate to true (Refer Slide Time: 37:39). This is constant folding; we have evaluated this particular condition. Now, as this is true, the no part can never be executed. So this edge can be removed like this (Refer Slide Time: 37:53); it is only this part, which gets executed; the flow graph gets modified in this fashion.

Now, a gets a value 30 and again it is a constant; so we can replace this a by 30 (Refer Slide Time: 38:08). If we had this flow graph, we did not know whether a would be 10 or a would be 30 at this point but, since we know that this condition is always true, this part does not get executed only this part gets executed. Therefore, this a can get only the value from this particular assignment a equal to 30 (Refer Slide Time: 38:28). This gets

35 and then stops, so the program modified is like this, all the values have been computed; a plus 5 is computed as 35, so that is an example of a constant folding.

So, b getting a value 20 here, checking b 20 equal to 20 is an example of constant propagation. The value of a being replaced by 30, here from this part is another example of constant propagation. Later, we are going to study how exactly all this can be done.

(Refer Slide Time: 39:06)



Loop Invariant Code motion Example

```
     t1 = 202
     i = 1
L1:  t2 = i>100
     if t2 goto L2
     t1 = t1-2
     t3 = addr(a)
     t4 = t3 - 4
     t5 = 4*i
     t6 = t4+t5
     *t6 = t1
     i = i+1
     goto L1
L2:
```

Before LIV code motion

```
     t1 = 202
     i = 1
     t3 = addr(a)
     t4 = t3 - 4
L1:  t2 = i>100
     if t2 goto L2
     t1 = t1-2
     t5 = 4*i
     t6 = t4+t5
     *t6 = t1
     i = i+1
     goto L1
L2:
```

After LIV code m

Y.N. Srikant   Introduction to Optimiz

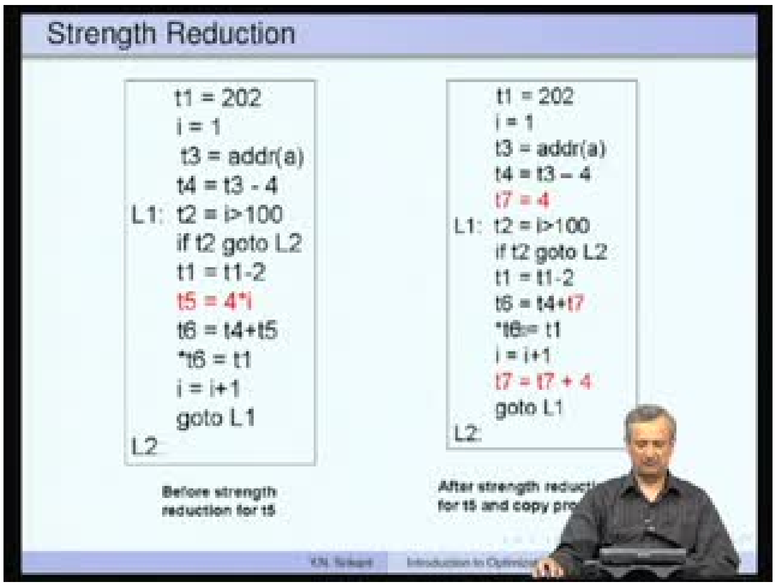
The next optimization that we are going to study is the loop invariant code motion. Take this simple example, t1 equal to 202, i equal to 1, if i greater than 100, if t2 exit. So, we are going to execute this particular loop 100 times. Otherwise fall through, decrement t1; then compute t3 equal to addr a; addr a is a constant that is the base address of array a and t3 equal to t3 minus 4, this is for some alignment. Then we compute 4 star i, add t4 and t5. Now, we go to the location in the array a- it is a of i, which we want to access; start t6 equal to t1 assigns to that particular location, the value t1 and i equal to i plus 1 and go back to L1, so this the loop that we go on executing.

Let us look at some of the assignments here (Refer Slide Time: 40:27). Obviously, the value of i varies here, so t2 equal to i greater than 100 varies with the loop; the same is true of this because t2 varies this particular condition also varies rather cannot be evaluated once for all (Refer Slide Time: 40:40). t1 equal to t1 minus 2 also varies with the loop so it cannot be evaluated once for all; whereas, t3 equal to address of a - address of a is a constant. The same assignment gets executed 100 times within this particular loop but, the value of t3 will always be the same.

So, it is possible to move this particular quadruple outside the loop, then nothing wrong with that and such a computation which does not vary within the loop is called a loop invariant computation and moving such a computation outside the loop is loop invariant code motion.

The next one is t4 equal to t3 minus 4, t3 is a constant, so t3 minus 4 is also a constant and this can also be moved outside the loop, this is loop invariant code again. Once we do this (Refer Slide Time: 41:47), the loop becomes smaller. The idea of all this is if we remove many statements from the loop, the loop execution time becomes lesser and lesser. We want to make the loop as efficient as possible and that is the reason for loop invariant code motion. We try to move statements from within the loop, which do not get modified during the execution of the loop to outside the loop.

(Refer Slide Time: 42:09)



Strength Reduction

```
    t1 = 202
    i = 1
    t3 = addr(a)
    t4 = t3 - 4
L1: t2 = i>100
    if t2 goto L2
    t1 = t1-2
    t5 = 4*i
    t6 = t4+t5
    *t6 = t1
    i = i+1
    goto L1
L2:
```

Before strength reduction for t5

```
    t1 = 202
    i = 1
    t3 = addr(a)
    t4 = t3 – 4
    t7 = 4
L1: t2 = i>100
    if t2 goto L2
    t1 = t1-2
    t6 = t4+t7
    *t6 = t1
    i = i+1
    t7 = t7 + 4
    goto L1
L2:
```

After strength reduct for t5 and copy pr

Y.N. Srikant   Introduction to Optimiz

The next example that we consider is optimization example, which is strength reduction. So here, this is the code which we had before, we have already done loop invariant code motion; so that is why the code is in this fashion and these two are outside the loop. Here is a statement t5 equal to 4 star i, so this actually uses a multiplication.

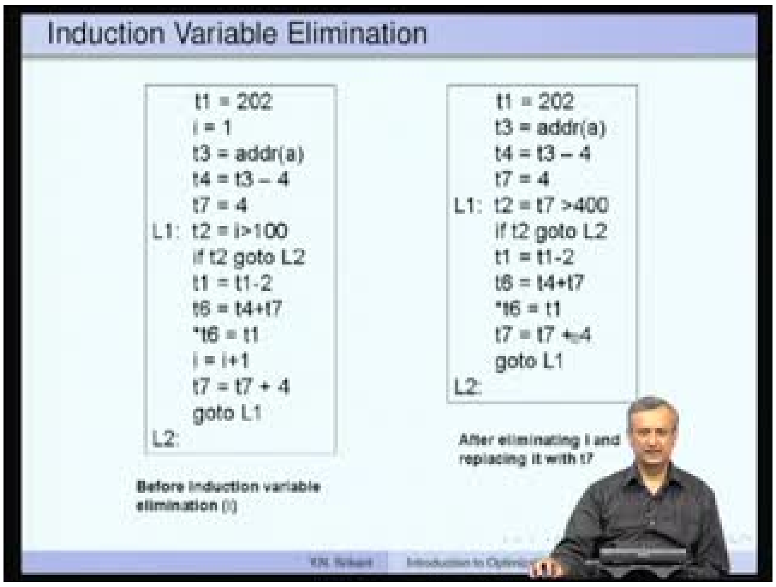
On very small processors and of course, in the olden days multiplication used to take much more time than addition. For example, floating point multiplication takes a long time whereas, fixed point multiplication takes much lesser time even today.

This particular optimization is called strength reduction because an operation which is expensive is actually converted to a number of operations which are lesser expensive. Let us assume that multiplication is more expensive than addition. So, this statement uses multiplication. Let us see what we have done, we have set t7 equal to 4 and we have set t7 equal to t7 plus 4 and in t6 which has t4 plus t5, we have set t4 plus t7. How is this correct?

If you observe the values that t5 takes i equal to 1. So, to begin with it takes the value 4 then i equal to 2, so it takes 8, 12, 16, 20 and so on. In other words, the value of t5 is incremented by 4: 4, 8, 12, 16 etc in every iteration of the loop, it is incremented by 4. Instead of multiplying i by 4, why don't we simply take another variable and add 4 to it. Thereby the effect is the same, t7 equal to t7 plus 4 the values of t7 they start with 4 just like t5 starts with 4, t7 starts with 4 and then it becomes 8, 12, 16, 20 etc just like t5 becomes 8, 12, 16, 20 etc. There is no difference between the sequence of t5 values and the sequence of t7 values; in every sense we can replace t5 by t7 and that is precisely what we have done.

In this process, we have replaced the more costly operation of star by a simpler operation and a less costly operation of plus. Even today on very small embedded processes, multiplication may still be more expensive than addition, because multiplication may typically be done by a subroutine whereas, addition is a hardware instruction. Strength reduction and then of course, we also have done a little bit of copy propagation ; instead of t5, so we would have actually set t5 equal to t7 here because they are equivalent and then t7 value can be replaced for t5 in this particular statement (Refer Slide Time: 45:43). We have done these two and replaced this t6 equal to t4 plus t5 by t6 equal to t4 plus t7 and we have removed the statement t5 equal to 4 star i. This is strength reduction, it requires a little bit of code for initialization and then perhaps even copy propagation.

(Refer Slide Time: 46:04)



Induction Variable Elimination

```
    t1 = 202
    i = 1
    t3 = addr(a)
    t4 = t3 – 4
    t7 = 4
L1: t2 = i>100
    if t2 goto L2
    t1 = t1-2
    t6 = t4+t7
    *t6 = t1
    i = i+1
    t7 = t7 + 4
    goto L1
L2:
```

Before induction variable elimination (i)

```
    t1 = 202
    t3 = addr(a)
    t4 = t3 – 4
    t7 = 4
L1: t2 = t7 >400
    if t2 goto L2
    t1 = t1-2
    t6 = t4+t7
    *t6 = t1
    t7 = t7 + 4
    goto L1
L2:
```

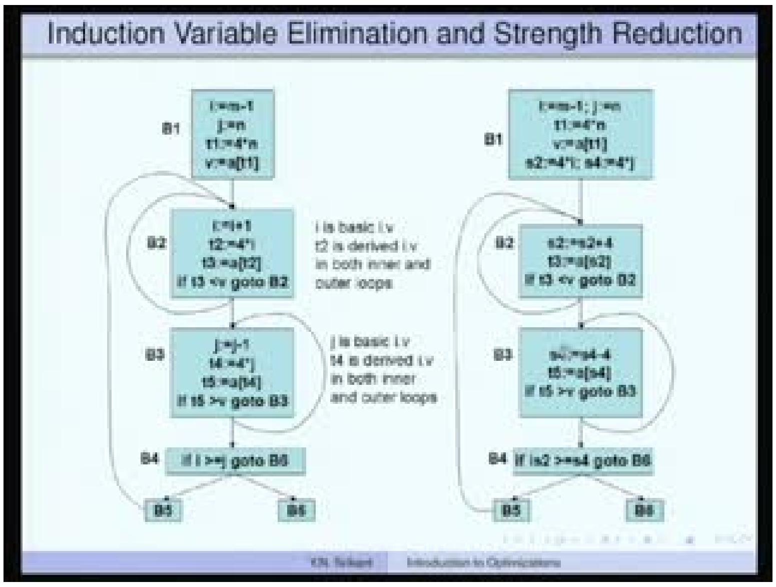
After eliminating i and replacing it with t7

The next example is that of induction variable elimination, again the previous example we carry forward. So, before the induction variable elimination, let us say i is the induction variable we want to eliminate. Why is i called an induction variable? You see the values of i. For example, if you look at it, they start with 1 and then go on in an arithmetic progression 1, 2, 3, 4, 5, 6 etc.

Therefore, all such variables which progress in arithmetic progression are called induction variables. The purpose of i in this particular loop is only to count the number of times the loop is executing and then terminate the loop, once it crosses this particular loop of 100.

If you look at the loop a little more carefully, there is another variable t7 which actually also progresses in an arithmetic progression. We start with 4 and then go to 8, 12, 16, 20 etc this is also an arithmetic progression. It is just that it is not the same as i but, it is 4 times the value of i. So, induction variable elimination tries to replace i with t7 and remove i; that is possible in this case, only thing is in that case instead of i we must use t7 and once we use t7 the test i greater than 100 should become t7 greater than 400. So, with these changes it is possible to remove the induction variable i, change the test to t7 and then the loop becomes even smaller. This is the loop that we get after induction variable elimination (Refer Slide Time: 48:11).

(Refer Slide Time: 48:18)

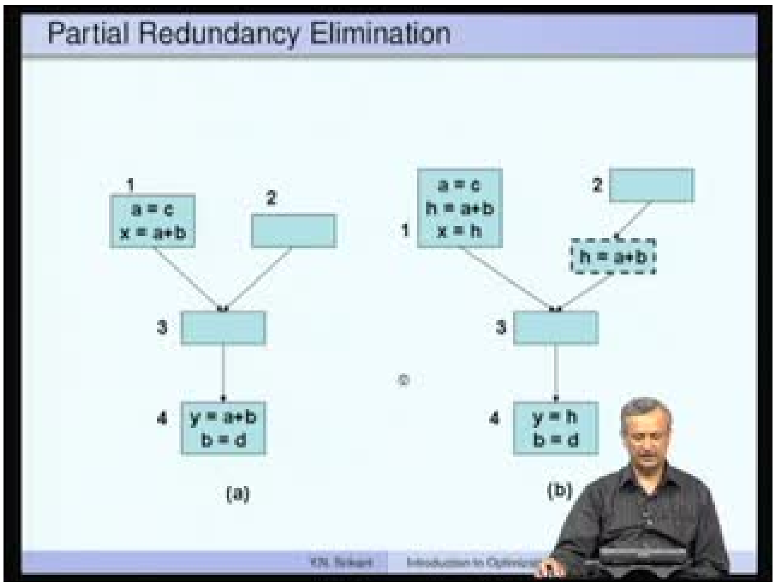


Here, is a bigger example of induction variable elimination and strength reduction and this is from the dragon book. Here are two loops and a third loop which encompasses both of them. This is the outer loop and two inner loops (Refer Slide Time: 48:30). So, it is very easy to see that this i is a counter variable here, j is a counter variable here but, it so happens that i is a counter variable in the outer loop and also j is a counter variable in the outer loop. So i is a basic induction variable and then t2 depends on i, see it is 4 times i as in the previous case; so if i goes as 1, 2, 3, 4; t2 goes as 4, 8, 12, 16 etc. Such variables t2, which depend on i are derived induction variables. There are conditions to be checked for basic and derived condition for induction variables which we learn later. But, for the present it suffices to know that i is a basic induction variable both in this loop, inner loop and the outer loop, t2 is a derived induction variable both in this loop and the next loop.

The same is true of j, it is not necessary that the values of induction variables only increase; they can monotonically decrease also. j starts with n and reduces n minus 1, n minus 2 etc and t4 derives from j, so t4 is 4 times j and it reduces in the same proportion. So both these are induction variables in the inner loop and also in the outer loop.

So, both these i and j can be eliminated, it is possible to introduce a new and t2 of course, can also be eliminated, t4 can also be eliminated. These are replaced by 2 variables s2 and s4, there is a suitable initialization which is carried out here; so s2 becomes s2 plus 4 in the loop, s4 becomes s4 minus 4 in this loop. In the test which as i greater than equal to j, this is the only place why i and j have been used, they are used to control the loop, we use s2 and s4. Once we do this, it is possible to actually convert this program to this program (Refer Slide Time: 50:48); we can replace i greater than equal to j by s2 and s4 and the variables s2 and s4 can be incremented appropriately by 4 and decremented by 4. This is the program in which we have been able to do strength reduction, we have been able to do strength reduction here also and then we have eliminated 2 induction variables i and j replace them by s2 and s4.

(Refer Slide Time: 51:19)



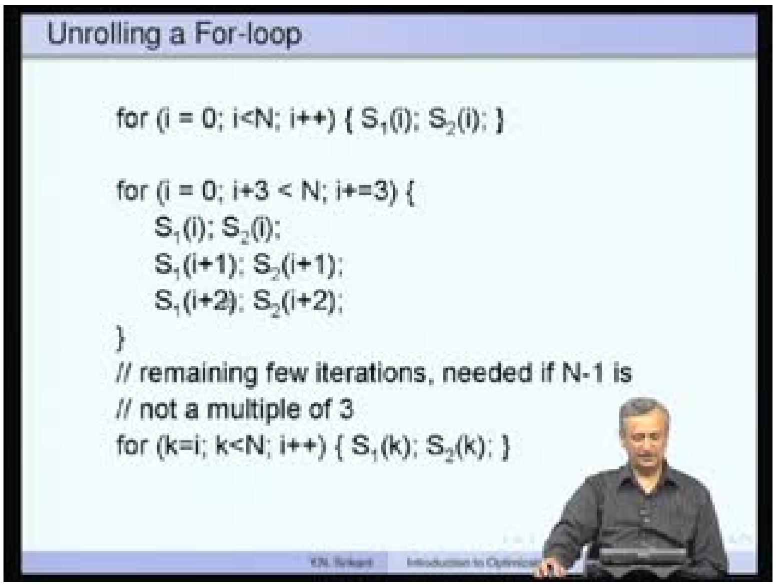
Next, optimization is a very important one called the partial redundancy elimination. There are actually, you know many optimizations which this particular PRE as it is called subsumes. For example, loop invariant code motion gets subsumed in this. If we do this properly, then 2 or 3 types of common sub expression elimination can also gets subsumed by this; so 2 or 3 optimizations can all be subsumed by this particular optimization itself.

Let us understand this, here is a equal to c and then we have x equal to a plus b. Here we have y equal to a plus b and we have b equal to d. There is no computation of a plus b here and there is no computation of a plus b here; so if you had a plus b here also for example, some z equal to a plus b, then you would have detected that this a plus b is a common sub-expression because it is computed here and here. The GCSE algorithm would have been applied but, now it is computed only along this path and it is not computed along this path were before this particular computation. So, such a computation is a partially redundant computation.

If we take this path, there is no need to do twice; maybe this value can be used directly here. If we take this path, then it is necessary to do it; if we are able to detect that such partially redundant computations exist, that is a plus b is partially redundant at this point; so it can be computed here and then used here (Refer Slide Time: 53:11). This can be done provided we introduce h equal to a plus b along this path also, h equal to a plus b is introduced here. Now, what happens is, this is fully redundant this particular a plus b becomes fully redundant with the introduction of this; that is why a equal to c, h equal to a plus b, h is the temporary variable in GCSE and x equal to h; h equal to a plus b again here and this becomes y equal to h. We have converted a partially redundant computation to a fully redundant computation by breaking this particular edge introducing this computation of a plus b.

The challenge is in detecting that this is a partially redundant computation and then taking steps to see, how we can convert this PRE into a fully redundant computation and then apply GCSE. This is a very interesting algorithm, which we are going to see later.

(Refer Slide Time: 54:19)



Unrolling a For-loop

```
for (i = 0; i<N; i++) { S1(i); S2(i); }

for (i = 0; i+3 < N; i+=3) {
    S1(i); S2(i);
    S1(i+1); S2(i+1);
    S1(i+2); S2(i+2);
}
// remaining few iterations, needed if N-1 is
// not a multiple of 3
for (k=i; k<N; i++) { S1(k); S2(k); }
```

Let us look at a simple optimization called unrolling a loop. Here is a simple loop, for i equal to 0, i less than n, i plus plus and then there are two statements S1 and S2. We have put, i in the parenthesis here to indicate that these are instances of the ith iteration. If we unroll the loop, we are going to reduce the number of iterations. Let us say, we have unrolled it thrice that means, there are going to be three instances of S1 and S2 in the body of the loop S1 i; S2 i; S1 i plus one; S2 i plus1; S1 i plus 2 and S2 i plus 2. The loop is actually going to run less number of times, it starts with 0 checks whether i plus 3 less than N and then i is incremented by 3 because we have already included the instances of S1 and S2 corresponding to i plus 1 and i plus 2 also.

At the end, there may be some iterations which may remain because if N minus 1 is not a multiple of 3; say for example, you have N equal to 3 then, N minus1 is 2 so it is not a multiple of 3. So, this cannot be done because we need only this, then it is going to be executed, the rest of the iterations will all be executed by this particular loop which remains.

We start with k equal to I, k less than N, i plus plus and then without any unrolling we execute S1 k and S2 k. The advantage of loop unrolling is it enlarges the body the loop. Now, this body is much larger than before, this was S1 to S1, S2 here we have many instances of S1 and S2 and because of this the other types of machine dependent optimizations such as instruction scheduling, software pipelining etc become very efficient. We will learn about this much later but, it suffices to say that this is a very efficient way of enabling other machine dependent optimizations.

So, with this we will stop this lecture and in the next lecture, we will continue with other examples of optimizations.