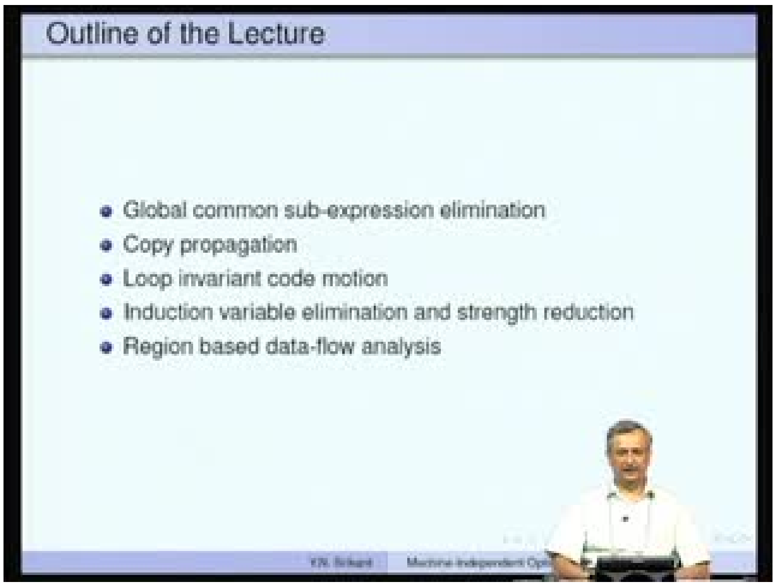
Compiler Design

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

Module No. # 10
Lecture No. # 16
Machine-Independent Optimizations

Welcome to the lecture on Machine-Independent Optimizations.

(Refer Slide Time: 00:21)

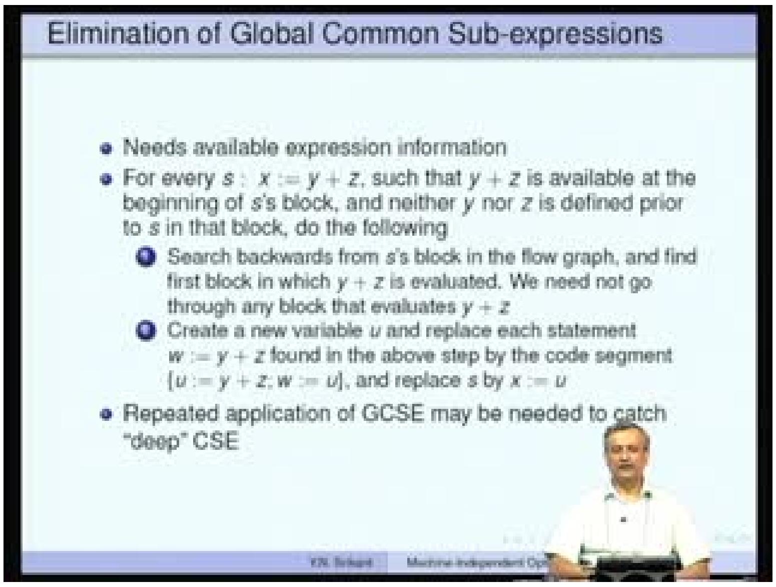


Outline of the Lecture

- Global common sub-expression elimination
- Copy propagation
- Loop invariant code motion
- Induction variable elimination and strength reduction
- Region based data-flow analysis

The outline of the lecture is - it consists of the following topics. We will look at the global common sub-expression elimination. We have seen examples of this already. Now, we will look at the algorithm itself. We will look at copy propagation. Again, we have seen examples, but not the algorithms. So, we are going to discuss that now. Loop invariant code motion, induction variable elimination and strength reduction, and then a new type of data-flow analysis called region based data-flow analysis. Partial redundancy elimination is also a very important optimization. We will discuss that as a separate lecture because it is fairly detailed and involved.

(Refer Slide Time: 01:12)

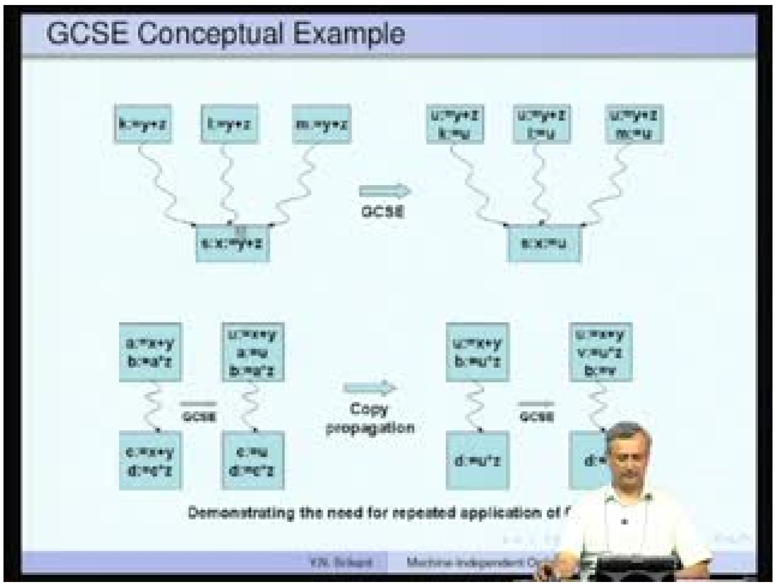


Elimination of Global Common Sub-expressions

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

What is global common sub-expression elimination?

(Refer Slide Time: 01:25)



GCSE Conceptual Example

Basically, we have expressions, which are defined in various basic blocks of the program. Now, in a particular block such as this for example, there is an expression y plus z. Is it necessary to recompute this expression y plus z? Or can we use the recomputed value, which already is available in this particular basic block? That is the question. If we are going to provide conditions to check whether the previously computed values of the expression can be used here, otherwise, we are going to recompute it.

This requires the available expression information (Refer Slide Time: 02:14). Recall that we discussed available expression data-flow analysis couple of lectures ago. An expression is available at a particular point if it is computed along a path starting from the starting point to that particular point and the operands are not modified in between. If the operands are modified, obviously the value of the expression changes, though the expression is not set to be available. So, we assume that the data-flow analysis of available expressions has been carried out and the information is already available to us at in and out points of the various basic blocks.

What is the algorithm? For every statement x equal to y plus z such that, y plus z is available at the beginning of s's block. If the expression y plus z is computed in a block and we want to reuse the previous computation of y plus z, obvious condition to be satisfied is that y plus z is available at the beginning of s's block. Then, the expression value should not have changed. So, neither y nor z is defined prior to s in that block. So, this is also very important; otherwise, the value would have changed. If these conditions are satisfied, then the availability of the expression y plus z at the entry point of the basic block tells us that (Refer Slide Time: 04:02) the expression is available along every path that reaches this particular block. Recall that the confluence operator is intersection. We would have made sure at the time of data-flow analysis that the expression is available along every one of these paths.

The available expression information only (Refer Slide Time: 04:28) tells us that it is available, but in which block has it been computed? That is not yet known. To determine this, we need to search backwards from the s's block in the flow graph and find the first block in which y plus z is evaluated. This has to be done for every path that actually re parts backwards in the flow graph.

We have this particular basic block (Refer Slide Time: 05:06). There are three paths that we have shown here. We must make sure that along each path, we search backwards and come to the block that evaluates y plus z. Along this path, again search backwards and come to the block that evaluates y plus z. Same along this path as well, but what is important is that we do not have to cross this block and go beyond; that is not even necessary. We just have to locate the first computation along this path (Refer Slide Time: 05:34) in the backward direction.

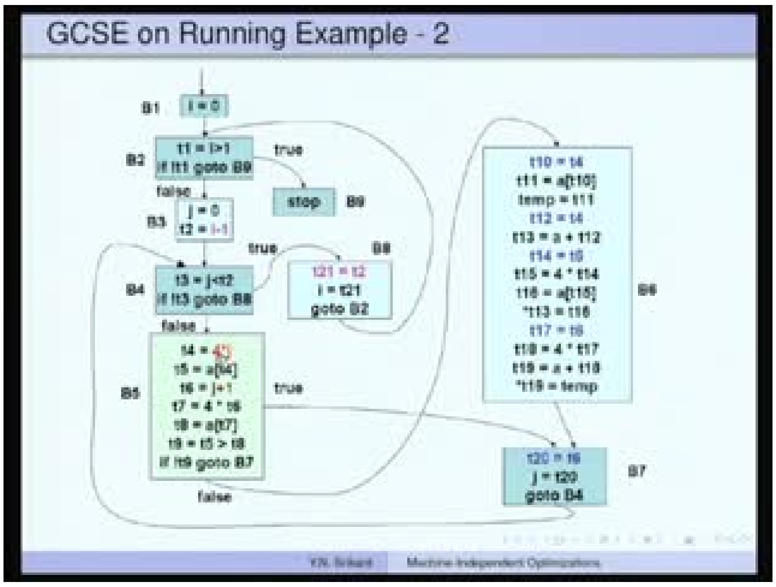
If that is done, then we would have actually located the three of these (Refer Slide Time: 05:44) for the three paths. Each one of them actually has evaluated y plus z. The common sense (( )) to do now is – use the same temporary into which we want to evaluate y plus z. So, u equal to y plus z, u equal to y plus z, and u equal to y plus z. Here, we can say x equal to u, but the variable k, l, and m cannot be disposed of. So, we also have statements k equal to u, l equal to u, and m equal to u.

That is what he said here (Refer Slide Time: 06:18) – Create a new variable u and replace each statement w equal to y plus z found in the above step by the code segment, u equal to y plus z and w equal to u. Then finally, replace the statement s by x equal to u. This takes care of one particular common sub-expression. It is probably necessary to apply GCSE again and again to catch deep common sub-expressions. Let me explain that now.

Here is another example (Refer Slide Time: 06:55), which we have seen before, but in the light of GCSE. Let us look at it again. There is a block here containing a equal to x plus y and b equal to a star z. It leads to another block containing c equal to x plus y and d equal to c star z. Clearly, x plus y is a common sub-expression. So, let us eliminate it. We do that u equal to x plus y and a equal to u and here, it is c equal to u. The other two statements remain as they are.

Now, suppose we do copy propagation, we find this a equal to u is a copy. So, the value of a here (Refer Slide Time: 07:42) can be replaced by u and the value of c here can also be replaced by this u. So, we get b equal to u star z and d equal to u star z, which actually are common sub-expressions again. Elimination of these uses the more compact code, u equal to x plus y, v equal to u star z, b equal to v, and d equal to v. So, this is what we mean by application of GCSE repeated number of times. Usually, we apply GCSE copy propagation and GCSE again. In fact, it may happen that we apply all the optimizations once, but then again apply it once more and so on and so forth. However, GCSE in copy propagation are so closely linked to each other that we may have to apply them in pairs until no more GCSE can be performed.

(Refer Slide Time: 08:39)

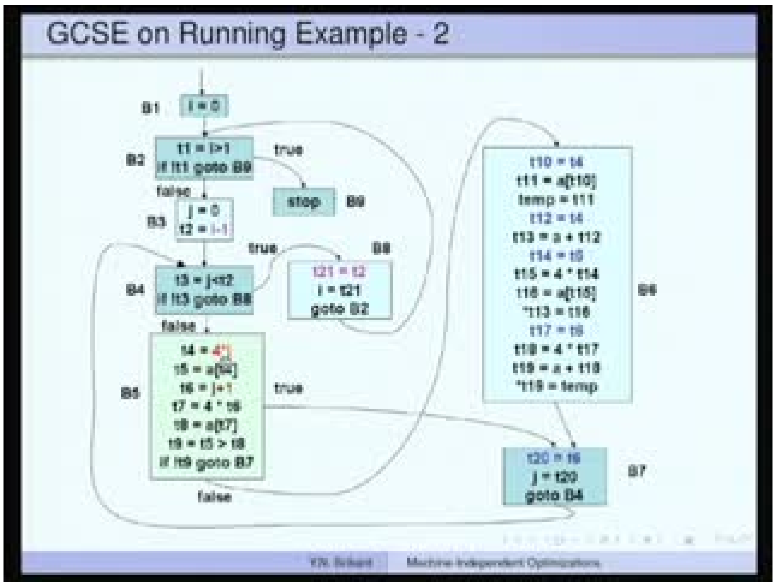


GCSE on Running Example - 2

```
B1  i = 0
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 = t4
    t11 = a[t10]
    temp = t11
    t12 = t4
    t13 = a + t12
    t14 = t6
    t15 = 4 * t14
    t16 = a[t15]
    *t13 = t16
    t17 = t6
    t18 = 4 * t17
    t19 = a + t18
    *t19 = temp
B7  t20 = t6
    j = t20
    goto B4
B8  t21 = t2
    i = t21
    goto B2
B9  stop
```

Let us look at a bigger example. We use this as the running example in our introduction to optimizations lecture as well. There are many common sub-expressions here for example, t2 equal to i minus 1 here and t21 equal to i minus 1. So, i minus 1 is a common sub-expression. t4 equal to 4 star j here and then t10 equal to 4 star j and t12 equal to 4 star j; one more. Then, we have t6 equal j plus 1 here and then t14 equal to j plus 1, t17 equal to j plus 1, t20 equal to j plus 1. So, there are many of… Then, 4 star t6 will be caught as a common sub-expression in the second round.

(Refer Slide Time: 09:34)



GCSE on Running Example - 2

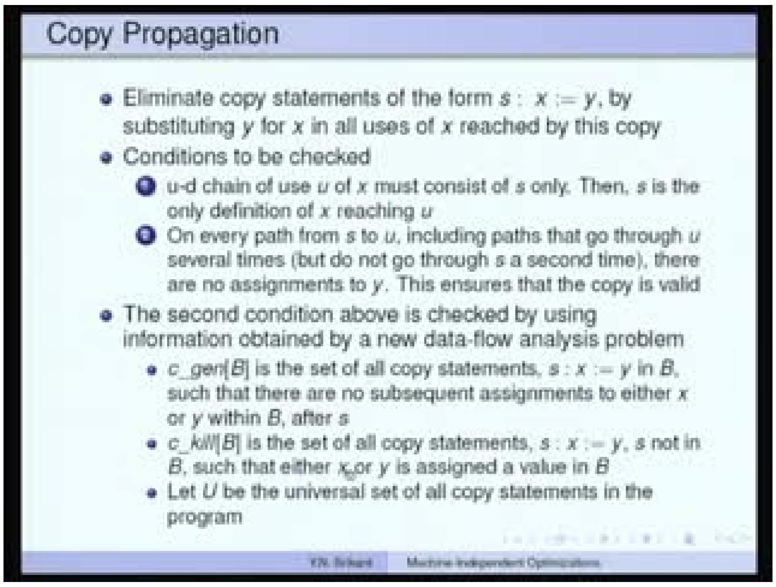
```
B1  i = 0
B2  t1 = i>1
    if t1 goto B9
B3  j = 0
    t2 = i-1
B4  t3 = j<t2
    if t3 goto B8
B5  t4 = 4*j
    t5 = a[t4]
    t6 = j+1
    t7 = 4 * t6
    t8 = a[t7]
    t9 = t5 > t8
    if t9 goto B7
B6  t10 = t4
    t11 = a[t10]
    temp = t11
    t12 = t4
    t13 = a + t12
    t14 = t6
    t15 = 4 * t14
    t16 = a[t15]
    *t13 = t16
    t17 = t6
    t18 = 4 * t17
    t19 = a + t18
    *t19 = temp
B7  t20 = t6
    j = t20
    goto B4
B8  t21 = t2
    i = t21
    goto B2
B9  stop
```

Let us look at the primary sub-expressions and once they are eliminated, we get this picture. This i minus 1 is a common sub-expression. So, we have caught that here and replaced by t2. So, t2 equal to i minus 1 is actually computed only once. Then, 4 star j is computed in t4. So, t10 equal to t4.

We have not computed 4 star j again and again. j plus 1 is computed into t6 (Refer Slide Time: 10:02) here. We have this; j plus 1 is t6. So, t14 equal to t6 and t17 equal to t6. Finally, t20 equal to t6. So, we are reusing the same value again and again. We did not introduce yet another temporary for 4 star j like in the algorithm here (Refer Slide Time: 10:30); see for example, here we said u equal to x plus y and then a equal to u. We did not do that here because these are already temporaries. So, there was no need to… If there were global variables, we would have had to do it over and over again, but now this is not necessary. So, we straight away use the same temporary.

Once we do this (Refer Slide Time: 10:53), there is a need to do some copy propagation because you can see easily that t10 can be replaced by t4 here and then t12 can be replaced by t4 here and so on and so forth. So, t14 can be replaced by t6 and so on and so forth.

(Refer Slide Time: 11:13)



Copy Propagation

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

Y.N. Srikant Machine-Independent Optimizations

If we do that, we actually would have done some copy propagation. So, let us study the algorithm for copy propagation and continue this example (Refer Slide Time: 11:24) after the copy propagation is understood.

What is copy propagation? Statements of the form x equal to y are called copies. Copy of y is maintained in x. Elimination of such copy statements to the extent possible is copy propagation because instead of x, we are going to replace it by y itself. So, we do not have to go to x and then copy the value from x again into some other location. We are going to straight away use the value of y in the place of x. So, that is propagating the copy from this point to some other point; substituting y for x in all uses of x reached by this particular copy.

There are a couple of conditions to be checked to make sure that copy propagation can be performed. First condition: the u-d chain; the use-definition chain of the use u of x; so, this is the copy (Refer Slide Time: 12:34), x equal to y; x would have many uses. If you look at the u-d chain of that particular use u of x, it must consist of s only. Suppose the u-d chain of some use of x had two definitions in it, then this copy s would be one of them and there would be some other statement teaching that particular use as well. In such a case, we are not certain that copy propagation must be carried out. So, we do not do it. We must be certain that s is the only definition of x reaching u. If that is so, then copy propagation can be performed, but that is not all.

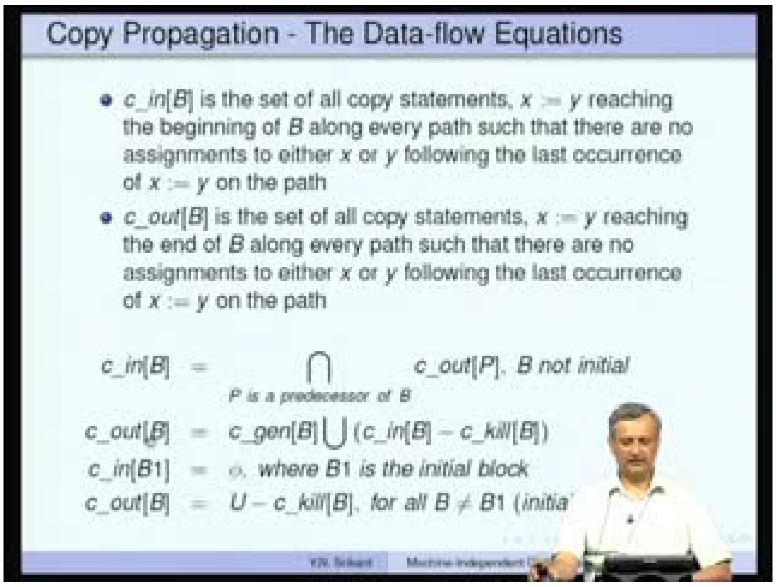
On every path from s to u, including paths that go through u several times, there could be a loop, which involves u, but not s; but, do not go through s a second time. So, you need not go through s a second time, but you can go through u any number of times. There are no assignments to y. So, this is very important. We do not want to have any assignments to y because the value of y would change in that case. This ensures that the copy is valid. If y has not changed, then there is no problem. To check this, it is necessary that we formulate a data-flow analysis problem. When we formulate the data-flow analysis problem, condition 1 automatically gets tackled.

Here is an optimization problem (Refer Slide Time: 14:14) – copy propagation. We are trying to solve that problem using another data-flow analysis. This is not something very special because to perform partial redundancy elimination, which is an optimization, we are going to use data-flow analysis as a tool again. So, what is this new data-flow analysis problem? Here, the values in the domain of the data-flow analysis problem are the sets of copy statements. The domain itself consist of sets of copy statements. c gen B is the set of all copy statements, s x equal to y in B, such that there are no subsequent assignments to either x or y within B after the statement, s.

Again, we have a gen kill in and out; c gen is the equivalent of gen that we have seen so far. So, what are the copies generated by a certain basic block B? Those copies x equal to y, such that x and y are not modified after s. So, that is the statement, which will be in the c gen. If you take all such copies within the basic blocks, whose left hand side or the right hand side are not modified, then that is the gen function.

What is the kill function? This is straight forward. Suppose you have either x or y assigned in a basic block then this assignment (Refer Slide Time: 16:02) to either x or y kills copies in other basic blocks. For example, c kill B is the set of all copy statements, s x equal to y, s is not in the basic block B; we are killing it, such that either x or y is assigned a value in B. So, it make sense because if you are assigning some value to x or some value to y, then any copy, which comes to the basic block will not be useful because it is being destroyed within the basic block. It cannot pass through the basic block and cannot retain that copy because these assignments will kill that particular copy. Then finally, let u be the universal set of all copy statements in the program.

(Refer Slide Time: 16:57)



Copy Propagation - The Data-flow Equations

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

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

Y.N. Srikant | Machine-Independent

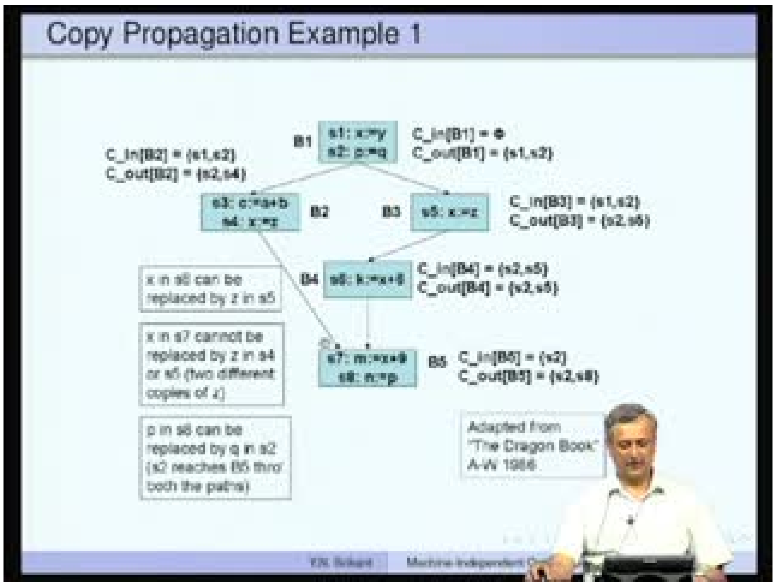
What are the data-flow equations? Here are the data-flow equations. Let us take a look at this structure of these equations and find out the usual parameters. We have the usual c gen, c kill, c in, and c out. Then, we have intersection of c out p; p being a predecessor of B. That means, the confluence operator is an intersection here. Is this a backward problem or a forward problem? out B is being defined in terms of in B. So, this is a forward-flow analysis problem.

This has the same flavor as the available expression problem, which also was a forward-flow problem with intersection as the confluence operator. As in that case, c in of B1 is phi, where B1 is the initial block; so, no copies are assumed to enter the basic block B1, which is the initial block. The initialization for c out B will be u minus c kill B, for all B not equal to B1. So, you could have also said c in of B equal to u, where B not equal to B1; that is also possible.

We make sure that it is the universe of all statements so that we get the maximum benefit in the data-flow analysis. What is c in B? c in B is the set of all copy statements, x equal to y; obviously reaching the beginning of the basic block; that is why it is c in B; along every path such that there are no assignments to either x or y following the last occurrence of x equal to y on the path. You are looking at intersection of course (Refer Slide Time: 18:48); so, you have look at all paths leading to the basic block; the copy must come to the basic block along all paths; that is why the intersection. Along such paths, there should be no assignments to either x or y; so, if there were, then we would have eliminated such copies from the set c in B.

C out B is similar. It is at the end of the basic block. Copy statements, x equal to y reaching the end of the block along every path such that there are no assignments to either x or y following the last occurrence of x equal to y on the path; same thing. So, this is out (Refer Slide Time: 19:25) and that is in. Since we want the copy to reach along all paths leading to basic block, we are taking the intersection. This also makes sense. The copies can be generated in the block and not killed of course; otherwise, they could be coming from outside to the beginning of the block and not killed in the basic block. So, that is what is c out.

(Refer Slide Time: 20:00)



Now, let me show you an example first before we look at the algorithm for using this data-flow analysis result. There are several basic blocks B1, B2, B3, B4, and B5 here. B1 has s1 and s2; both are copy statements. c in of B1 is phi because nothing is coming in; this is the initial block. c out of B1 is (s1, s2) because neither of these is killed within the basic block; they are generated. c out happens to be that itself.

There is no need to loop through the statements here because this is an acyclic graph; directed acyclic graph. So, one pass over the blocks is sufficient. c in of B2 will be whatever comes from here (Refer Slide Time: 20:48); no need to take any intersection. So, s1, s2, which is the output of basic block B1. What is c out of B2? It is s2 and s4. Why? Here, we have a statement, which is not a copy and s4 redefines x. So, s1 defines x and s4 redefines x. So, the copy s1 is last; it is killed by this (Refer Slide Time: 21:17) statement, s4. Only s4 is visible at this point. So, s2 of course, is not modified and not killed. So, s2 and s4 are available, rather reach this particular point after B2.

In B3, in is the same as out of B1. So, s1 and s2 (Refer Slide Time: 21:35) and x is again killing… This statement s5 is a copy killing the statement s1, which is also a copy. As in this (Refer Slide Time: 21:45), c out of B3 is (s2,s5).

Now, take B4. In of B4 is this (Refer Slide Time: 21:55); out of B3. So, that is (s2,s5) and out of B4 is also (s2,s5) because it is not assigning anything to either x or z. The important thing here is – this particular x (Refer Slide Time: 22:11) can be replaced by z;

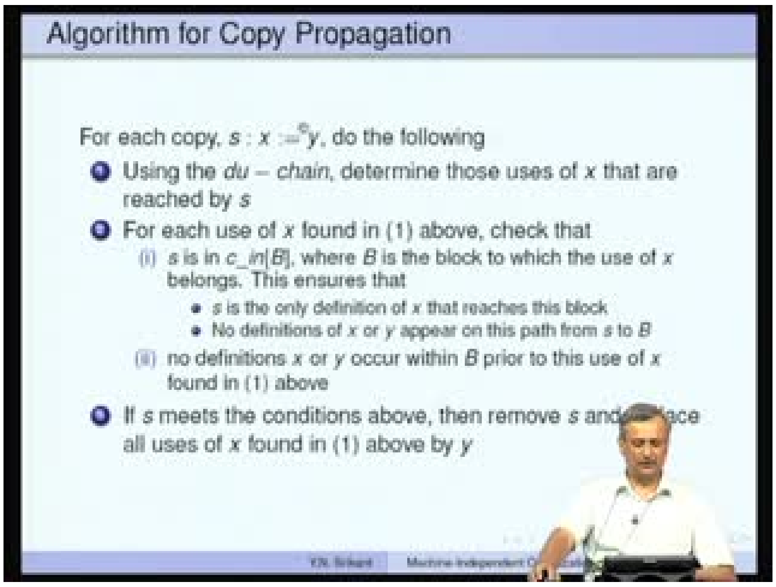
a copy propagation can be performed here for this particular x. This is the only copy, which is reaching here; x equal to s5; s2 and s5. s2 is p equal to q and s5 is x equal to z. We are using x here; so, this is the only copy, which is relevant and z can replace x.

Let us look at this (Refer Slide Time: 22:42); m equal to x plus 9 and n equal to p. c in of B5 is what comes out of this. So, that is… There is one more arc, which is coming in here. So, c out of B4 is (s2,s5) and c out of B2 is (s2,s4). So, the intersection is only s2.

Please observe that the x equal to z (Refer Slide Time: 23:11) here and the x equal to z here; even though they are syntactically the same statement, the same copy, but they are two different copies; syntactically they are similar, but s4 and s5 are two different statement numbers and they are two different copies.

Since s4 reaches here (Refer Slide Time: 23:35) and s5 reaches along this point, these two are not the same and neither this nor that actually are in c in of B5, whereas the copy s2 reaches along this path (Refer Slide Time: 23:50) and it also reaches along this path. So, it is in the set c in B5. That is how we are taking the intersection. This is a very important point; syntactically, the copies may look alike, but the statement numbers will be different because they are in different basic blocks and their statement numbers are all different. Therefore, we cannot say x equal to z reaches this point (Refer Slide Time: 24:17). So, that is very important. Then, this p can be replaced by q because s2 is reaching this particular basic block. This is the…; x here cannot be replaced by either this or (Refer Slide Time: 24:35) this because no copy corresponding to x equal to something reaches the in point of this particular basic block.

(Refer Slide Time: 24:45)



Algorithm for Copy Propagation

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

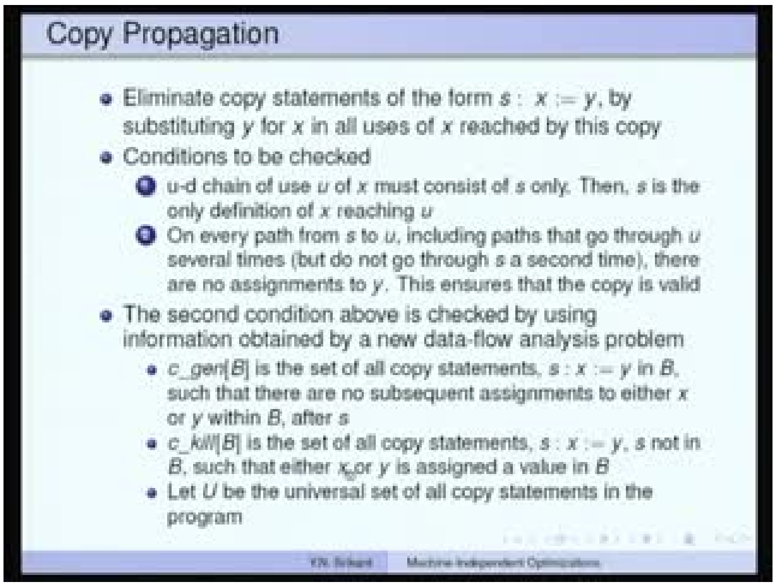
1. Using the *du – chain*, determine those uses of x that are reached by s
2. For each use of x found in (1) above, check that
   (i) s is in $c\_in[B]$, where B is the block to which the use of x belongs. This ensures that
      - s is the only definition of x that reaches this block
      - No definitions of x or y appear on this path from s to B

   (ii) no definitions x or y occur within B prior to this use of x found in (1) above
3. If s meets the conditions above, then remove s and replace all uses of x found in (1) above by y

Y.N. Srikant Machine-Independent Optimizations

Now, let us look at the algorithm for copy propagation, which uses the copy data-flow analysis problem results.

(Refer Slide Time: 24:58)



Copy Propagation

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

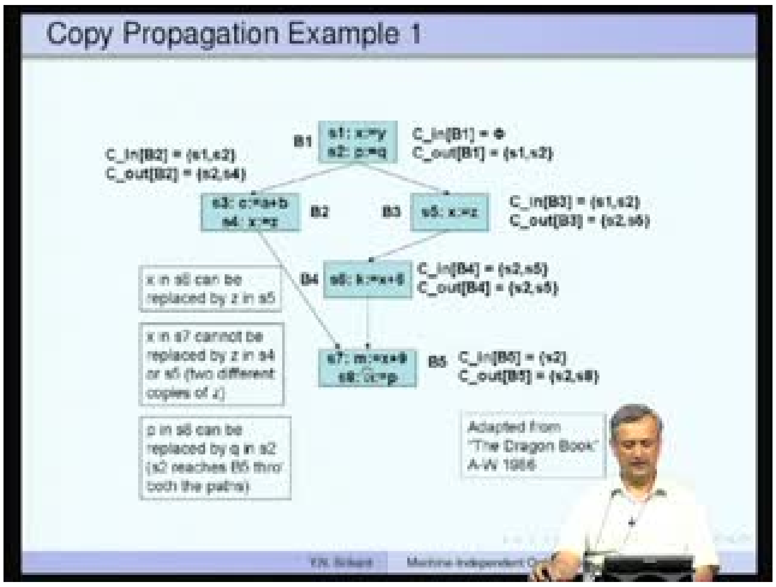
Y.N. Srikant Machine-Independent Optimizations

Remember – we had to check two conditions and I said that this condition number 1 will be subsumed in number 2 during the solution of the data-flow analysis problem. We already made sure of that because whenever there is more than one definition; (Refer Slide Time: 25:16) either x or y is assigned a value, then the previous copy is not taken at all; it is killed. That is how it is actually taking condition number 1 (Refer Slide Time:

25:27) also in it; embedding condition number 1 in condition number 2. For each copy, x equal to y, we use the d-u chain and find out all uses of x. So, there may be many uses of that. For each of these, we need to check the conditions. How do we do that? We just check whether s is in in of basic block. If it is already there, B is a basic block, which uses to which the use of x belongs. So, we found a use; at the input point of that basic block, this copy is reaching; it is there. What does this ensure us? This ensures us that this is the only definition of x that reaches this block; that is, condition number 1 and no definitions of x or y appear on this path from s to B (Refer Slide Time: 26:25). Because it is in s in of B, c in of B, this particular condition 2 is also satisfied; the copy is valid.

No definitions of x or y occur (Refer Slide Time: 26:36) within B prior to this use of x found in (1) above. In other words, we should make sure that within the basic blocks also, the 2 conditions hold. That is what we are checking - whether x or y is assigned; if it is, then I cannot do anything about this use. The copy even though it reaches the beginning of the basic block, will be modified and hence will not be valid.

(Refer Slide Time: 27:02)



Copy Propagation Example 1

x in s6 can be replaced by z in s5

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

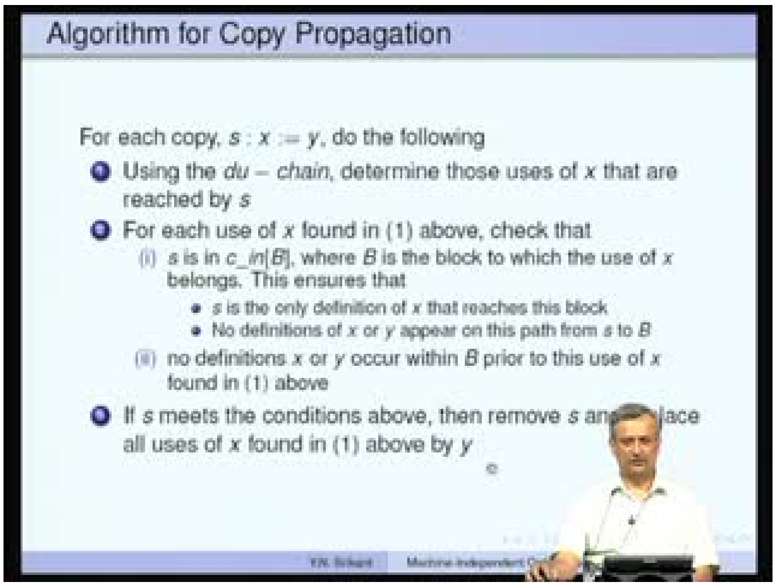
p in s8 can be replaced by q in s2 (s2 reaches B5 thro' both the paths)

Adapted from "The Dragon Book" A-W 1986

Y.N. Srikant Machine-Independent O

The point is – suppose here is n equal to p, we had p equal to something here. Even though c in the contains s2, p value would be modified by p equal to something. Therefore, this n equal to p cannot be replaced by s2. It can possibly be replaced by the previous value if it was a copy within the basic block, but not definitely in order by s2.

(Refer Slide Time: 27:28)



Algorithm for Copy Propagation

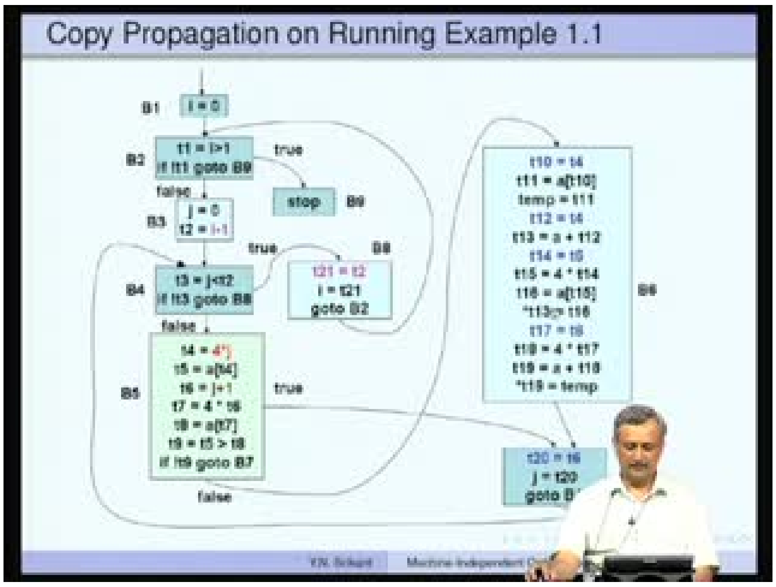
For each copy, $s : x := y$, do the following

1. Using the $du - chain$, determine those uses of x that are reached by s
2. For each use of x found in (1) above, check that
   (i) s is in $c\_in[B]$, where B is the block to which the use of x belongs. This ensures that
   - s is the only definition of x that reaches this block
   - No definitions of x or y appear on this path from s to B

   (ii) no definitions x or y occur within B prior to this use of x found in (1) above
3. If s meets the conditions above, then remove s an[illegible]lace all uses of x found in (1) above by y

If s meets the conditions above, then remove s and replace all uses of x found in (1) by y. So, this is the copy propagation. Here, is the first example (Refer Slide Time: 27:41).

(Refer Slide Time: 27:42)

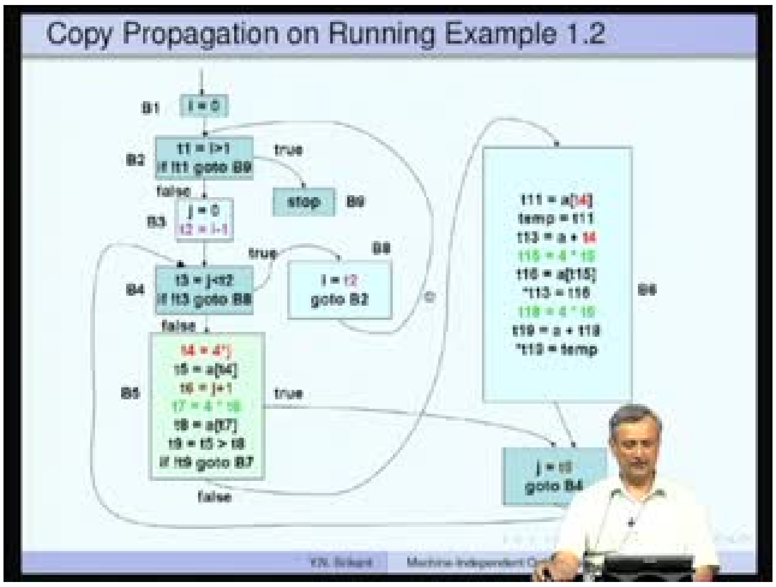


Copy Propagation on Running Example 1.1

```
B1: i = 0

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 = t4
    t11 = a[t10]
    temp = t11
    t12 = t4
    t13 = a + t12
    t14 = t6
    t15 = 4 * t14
    t16 = a[t15]
    *t13 = t16
    t17 = t6
    t18 = 4 * t17
    t19 = a + t18
    *t19 = temp

B7: t20 = t6
    j = t20
    goto B

B8: t21 = t2
    i = t21
    goto B2

B9: stop
```

Let us look at another example. In our running example, we saw so many copies; this t21 is here; it is a copy. t10 is another copy, t12 is third copy, t4 is one more copy, and so on and so forth.

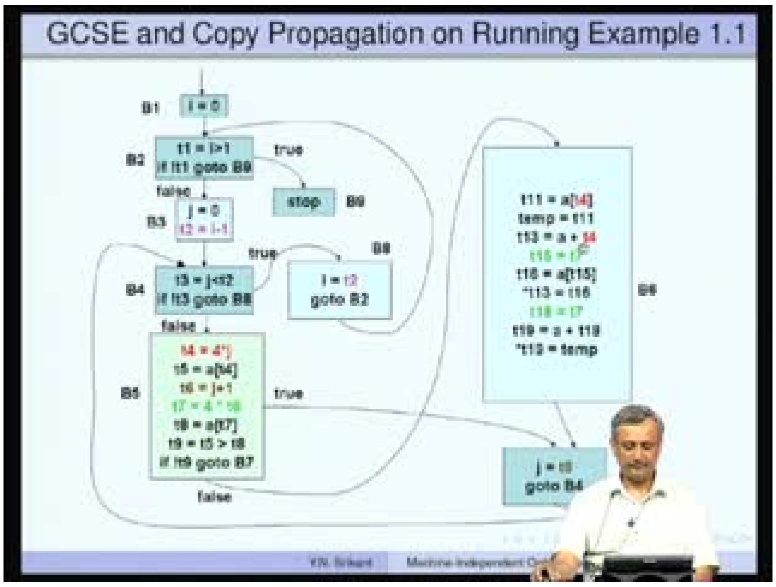
(Refer Slide Time: 28:04)



Copy Propagation on Running Example 1.2

Let us see how many of these copies can be eliminated. We replaced one of the copies by t4, then another copy by t4, then third one by t6, fourth one by t6, and so on and so forth. We have eliminated quite a few of these, but once we did copy propagation, there is chance for some more common sub-expression elimination. So, 4 star t6 is here and 4 star t6 is here. So, we can eliminate those also.

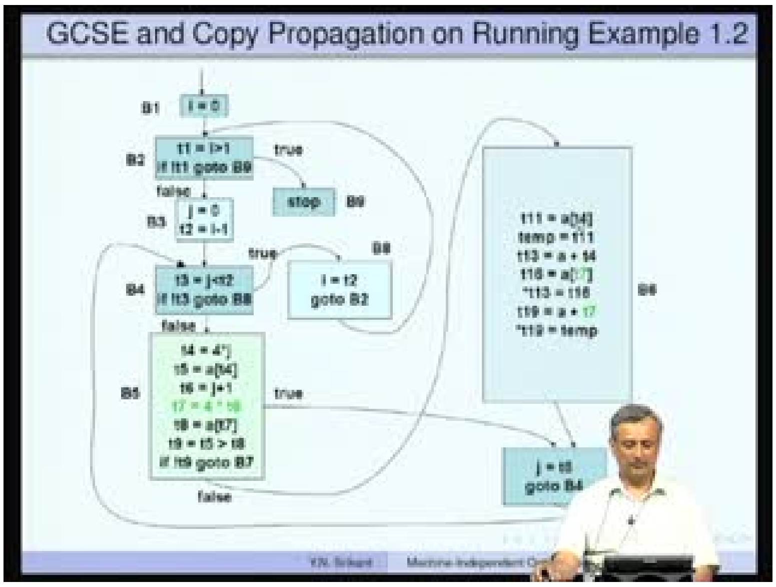
(Refer Slide Time: 28:37)



GCSE and Copy Propagation on Running Example 1.1

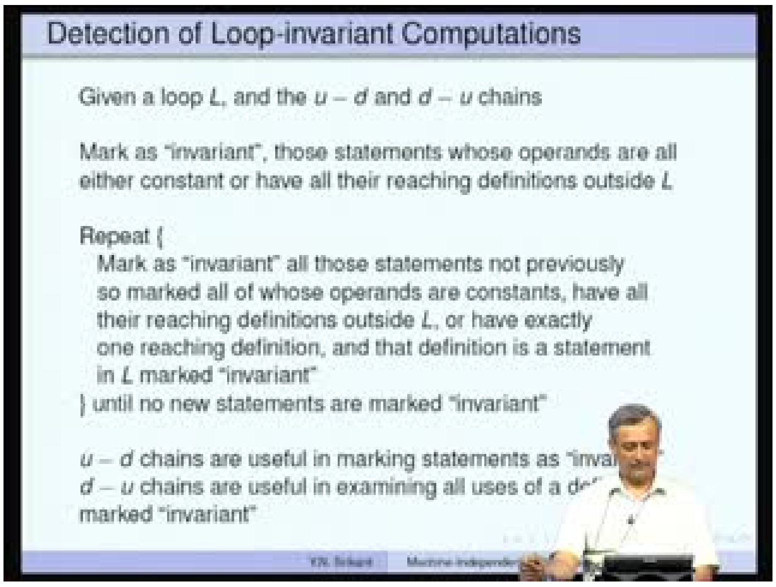
Let us assume that we do copy propagation and GCSE together. Now, we get this t15 equal to t7, t18 equal to t7, and all the others; we did some copy propagation.

(Refer Slide Time: 28:51)



Now, another round of copy propagation, will give us this particular compact code. See that (Refer Slide Time: 28:58) t15 is replaced by t7 and this again, t18 equal to t7. So, this t7 is again a copy. t18 equal to t7; so, t7 can replace t18. That is what happens here; a plus t7, a t7, a plus t4, etcetera. So, we are not computing any common sub-expression and we are not making unnecessary copies. So, this is a very compact code after common sub-expression elimination and copy propagation.

(Refer Slide Time: 29:37)



Detection of Loop-invariant Computations

Given a loop *L*, and the *u* – *d* and *d* – *u* chains

Mark as "invariant", those statements whose operands are all either constant or have all their reaching definitions outside *L*

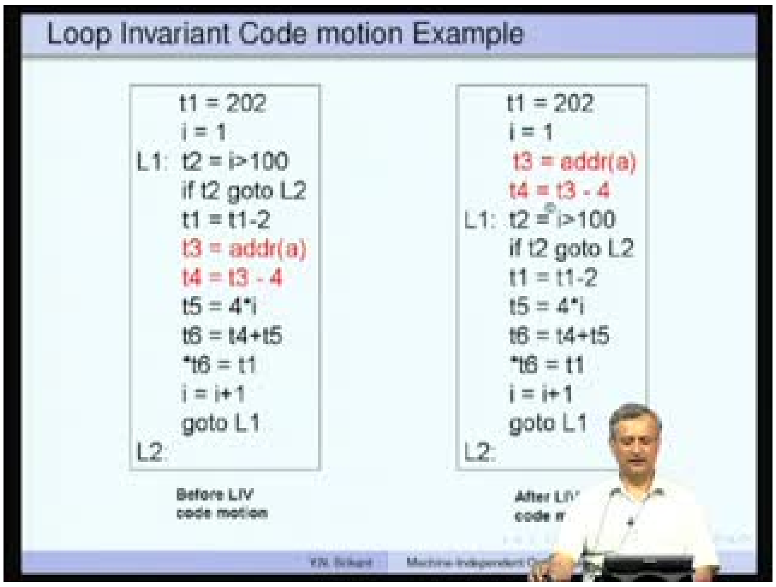
```
Repeat {
  Mark as "invariant" all those statements not previously
  so marked all of whose operands are constants, have all
  their reaching definitions outside L, or have exactly
  one reaching definition, and that definition is a statement
  in L marked "invariant"
} until no new statements are marked "invariant"
```

u – *d* chains are useful in marking statements as "inva
d – *u* chains are useful in examining all uses of a d
marked "invariant"

Y.N. Srikant Machine-Independe

What we have seen for can be done using just the data-flow analysis information. Now, we move on to loop-invariant computation and elimination of loop-invariant computation, rather movement of loop-invariant computations. To do this, it is not enough if we do data-flow analysis, we also have to do control-flow analysis, discover the loop structure, which intern implies finding dominators, back edges, natural loops, and so on. Let us assume that all that is done; that is what we mean by given a loop L. We also need the u-d chain and the d-u chain.

(Refer Slide Time: 30:33)



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

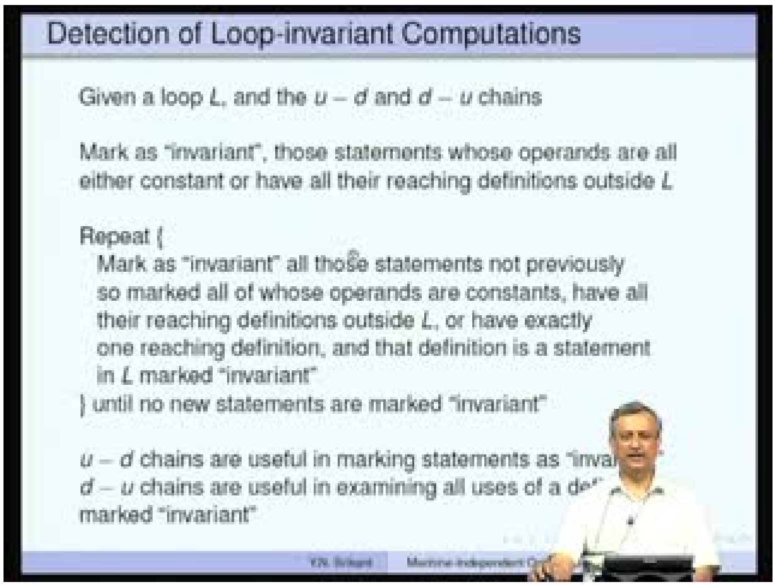
```
     t1 = 202
     i = 1
     t3 = addr(a)
     t4 = t3 - 4
L1:  t2 = i>100
     if t2 goto L2
     t1 = t1-2
     t5 = 4*i
     t6 = t4+t5
     *t6 = t1
     i = i+1
     goto L1
L2:
```

After LIV code motion

Loop-invariant computation; let me give you an example just to refresh your memory. t3 equal to address a; address a is the base address of the array a, which is a constant. This base address is within the activation record; the offset and it is a value, which is a constant assigned by the compiler.

t3 equal to address a – is an assignment of a constant to a temporary variable and there is no other assignment to t3 anywhere. So, whether we keep this assignment within the loop or we move it outside the loop, the value of t3 will remain the same. Similarly, look at t4; t4 equal to t3 minus 4. t3 itself carries a constant value and therefore, t3 minus 4 will also be a constant value. So, we can move t3 equal to address a and t4 equal to t3 minus 4 together as loop-invariant computations outside the loop. So, they do not vary within the loop. That is why they are called loop-invariant computations. They can be removed and put outside the loop.

(Refer Slide Time: 31:43)



Detection of Loop-invariant Computations

Given a loop *L*, and the *u* – *d* and *d* – *u* chains

Mark as "invariant", those statements whose operands are all either constant or have all their reaching definitions outside *L*

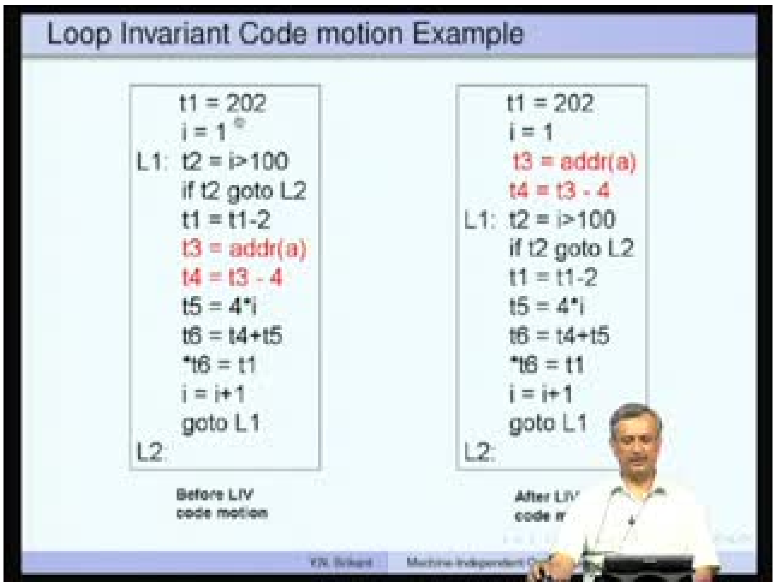
```
Repeat {
  Mark as "invariant" all those statements not previously
  so marked all of whose operands are constants, have all
  their reaching definitions outside L, or have exactly
  one reaching definition, and that definition is a statement
  in L marked "invariant"
} until no new statements are marked "invariant"
```

u – *d* chains are useful in marking statements as "inva
d – *u* chains are useful in examining all uses of a de
marked "invariant"

Y.N. Srikant   Machine-Independent O

How do we find such loop-invariant computations. The first round – mark as invariant, those statements whose operands are all either constant or have their reaching definitions outside L.

(Refer Slide Time: 32:07)



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   Machine-Independent O

Let us state this one at a time – Statements whose operands are all constants. This is a constant. So, t3 equal to address a is a loop-invariant computation, or (Refer Slide Time: 32:17) have all the reaching definitions outside L. In other words, suppose this address a is a constant and it was computed outside the loop; it is not computed within the loop, even then… Actually this is a loop-invariant computation. So, address a would not have been written as address a, it should have been written as some t prime equal address a, and we would have had t3 equal to t prime. t prime is defined outside the loop and that definition reaches that particular use here. So, that is what we mean by the definition of the variable being outside the loop and therefore, being loop-invariant. So, such computations are marked as invariant; that is the first round.

What about the second round? See this (Refer Slide Time: 33:09) was marked as invariant. This t3 is an operand in the next one; t4 equal to t3 minus 4 and the second operand of this assignment is also a constant. So, obviously, this is also a loop-invariant computation. Because t3 equal to address a, it is a loop-invariant computation.

We have to repeat this loop (Refer Slide Time: 33:35) many times. Mark as invariant all those statements not previously so marked all of whose operands are constants, obviously possible have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is a statement in L marked invariant. This is what I showed you; definition is a statement in L marked invariant; that is the example which is given here (Refer Slide Time: 34:01).

The others; the operands are all constants, have their reaching definitions outside L and this (Refer Slide Time: 34:10). This particular loop is repeated until no new statements are marked invariant. In fact, this particular last one is the one which discovers more and more new invariant computations. In doing so, u-d and d-u chains are useful. How?

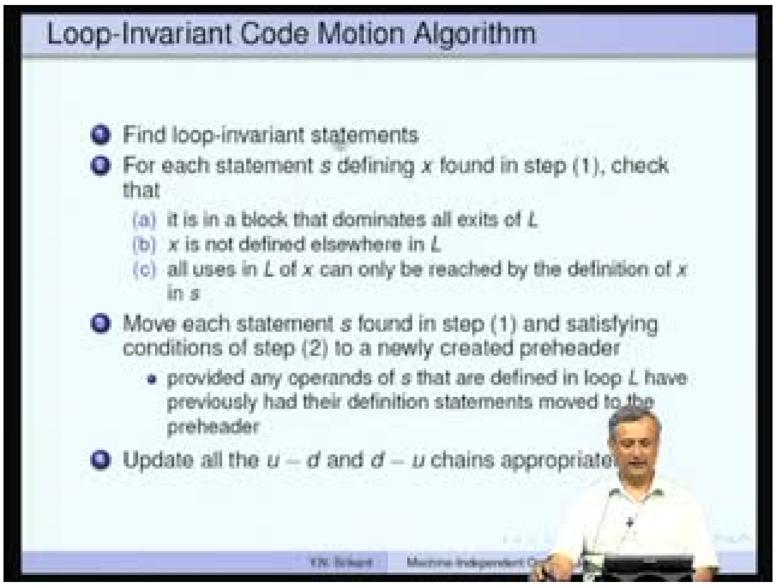
Here (Refer Slide Time: 34:38), what happens is – may be one of the operands is a constant and the other one is defined in the loop. So, like this (Refer Slide Time: 34:46); that is possible or it is also possible that both are defined in the loop, may be one of them come from outside the loop. All variations are possible. That is why, this loop is necessary. In the first round, we would not have marked anything like this (Refer Slide Time: 35:08). That is why, we need to go on and on until no more statements can be marked as invariant.

Why are u-d chains useful? (Refer Slide Time: 35:17) They are useful in marking the statements as invariant. So, you are going to look at the definitions for the particular use inside the loop and see whether those definitions come from within the loop, entirely outside the loop, etcetera. Whether they are loop-invariant computations; or otherwise, can be checked after we discover these definitions.

Why are d-u chains useful? (Refer Slide Time: 35:44) They are useful in examining all uses of a definition marked invariant. So, these are basically tricks in making the algorithm run faster. Once you mark something as invariant, for example, (Refer Slide Time: 35:55) t3, then you can look at all the uses of t3 and see whether they can be marked as invariant, which they satisfied the conditions that we have shown. So, this is one simple example of how to detect loop-invariants and move them from inside to outside.

Typically, here is the basic block, which is just outside this (Refer Slide Time: 36:19). So, we have moved these two assignment statements in the same order to the basic block, which is preceding. If actually we had a junction point; this particular basic block was a junction point with two incoming arcs, we would had to introduced a new preheader and move these computations into that particular preheader.

(Refer Slide Time: 37:03)



Loop-Invariant Code Motion Algorithm

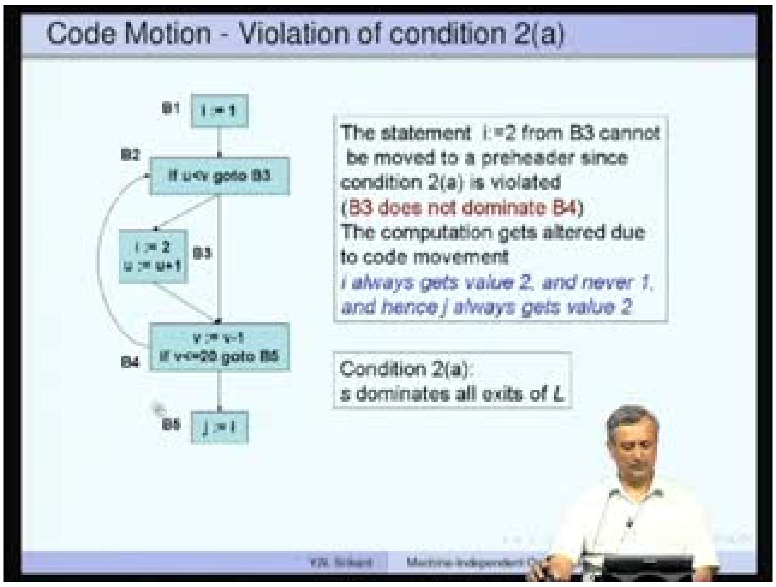
1. Find loop-invariant statements
2. For each statement *s* defining *x* found in step (1), check that
   (a) it is in a block that dominates all exits of *L*
   (b) *x* is not defined elsewhere in *L*
   (c) all uses in *L* of *x* can only be reached by the definition of *x* in *s*
3. Move each statement *s* found in step (1) and satisfying conditions of step (2) to a newly created preheader
   - provided any operands of *s* that are defined in loop *L* have previously had their definition statements moved to the preheader
4. Update all the *u* – *d* and *d* – *u* chains appropriate

Y.N. Srikant Machine-Independent O

How do you move loop-invariant code to outside the loop? That is the question. This is a very tricky situation; the example that I showed you here (Refer Slide Time: 36:57) was a very simple example and we could move very simply, but there are many conditions to be checked.

Find the loop invariant computations for each statement s defining x found in (1), check. So, there are three conditions. Let us look at each of these conditions and see what they mean. It is in a block; this statement s is in a block that dominates all exits of L.

(Refer Slide Time: 37:31)



Code Motion - Violation of condition 2(a)

The statement i:=2 from B3 cannot be moved to a preheader since condition 2(a) is violated (B3 does not dominate B4)
The computation gets altered due to code movement
i always gets value 2, and never 1, and hence j always gets value 2

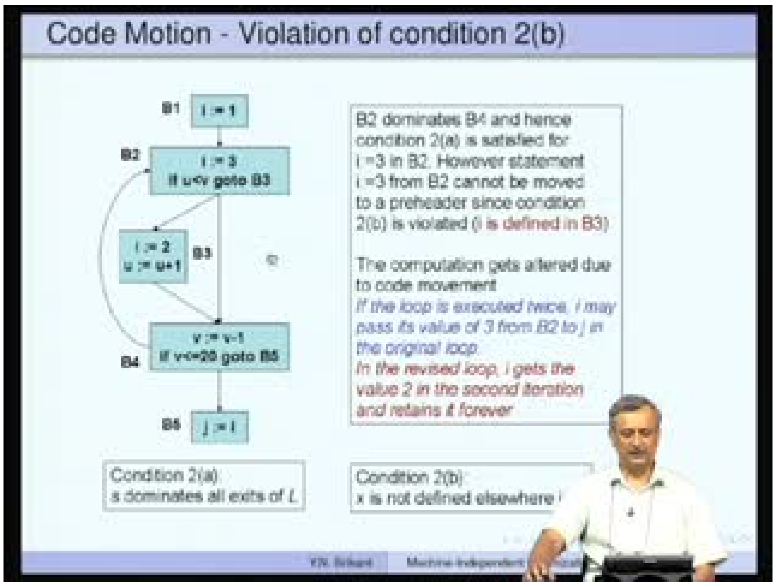
Condition 2(a):
s dominates all exits of *L*

Let us understand what happens if there is a violation. Here is a small flow graph; condition (( )); the basic block B1, B2, B3, B4, and B5. Now, the statement in basic block B3; that is, i equal 2; is what we want to consider. This is a statement, which is loop invariant. i is assigned value 2, which does not get changed within the loop. So, this is loop-invariant computation.

The question is – Can we move this to (Refer Slide Time: 38:04) B1? Now, the condition number 2(a) said s dominates all exits of L. In other words, this particular basic block B3 must dominate all exits of L. In this case, B4 is the only exit of L. Does B3 dominate B4? No because all paths from B1 to B4 do not go through B3; they can go to B2 and directly to B4. So, B3 does not dominate B4. Because this is not true, moving i equal to 2 to B1 will have some side effects.

What is that side effect? If we actually take this original flow graph; look at the value of i taken by j in B5. Value i equal to 1 could have travelled all the way from B1 to B2 to B4 and then to B5. So, j equal to 1 was possible; otherwise, we could have travelled with i equal to 1, then go to B2, go to B3; now i gets changed to 2, go to B4, and then to B5. Now, j gets the value 2. Suppose we move this i equal 2 (Refer Slide Time: 39:17) to this basic block B1 immediately after equal i to 1, then i equal to 1 is actually overridden by i equal to 2. So, the value of i permanently changes to 2. Even if we take the old path in which j would have got the value 1, now j gets the value 2 because i has changed its value permanently to 2. So, this is the change in the semantics of computation of this particular program. Therefore, any change in semantics is not permitted; this is illegal. So, just because it is loop invariant, we cannot move i equal to 2 to B1.

(Refer Slide Time: 40:03)



Code Motion - Violation of condition 2(b)

B2 dominates B4 and hence condition 2(a) is satisfied for i =3 in B2. However statement i =3 from B2 cannot be moved to a preheader since condition 2(b) is violated (i is defined in B3)

The computation gets altered due to code movement

If the loop is executed twice, i may pass its value of 3 from B2 to j in the original loop

In the revised loop, i gets the value 2 in the second iteration and retains it forever

Condition 2(a): s dominates all exits of L

Condition 2(b): x is not defined elsewhere

Y.N. Srikant | Machine-independent

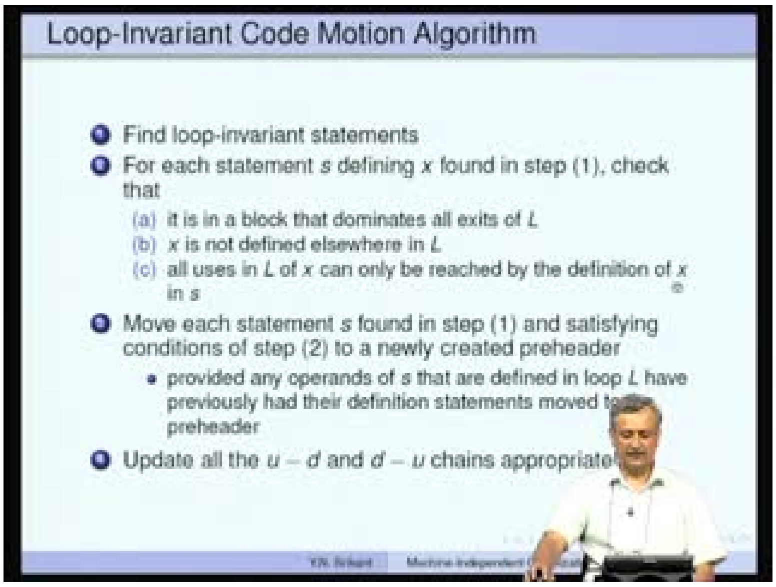
x is not defined elsewhere in the loop L. So, let us see what happens here. This flow graph now satisfies condition 2(a). What was 2(a)? s dominates all exits of L. We are looking at B2; not B3 this time. We cannot move i equal to 2; so, we have given up. Now, we consider i equal to 3 in the basic block B2. B2 definitely dominates B4, which is the exit of the flow graph, rather the loop because all paths from B1 to B4 have to go through B2. So, whether I take this path or this path, I would have crossed B2. Condition 2(a) is satisfied.

What is condition 2(b)? x is not defined elsewhere in L. i is defined here and i is defined again here (Refer Slide Time: 40:56). So, the condition is violated. Now, what happens? I want to move i equal to 3 to outside the loop. Suppose I move this to outside the loop, now, if the loop is executed twice, i may pass its value of 3 from B2 to j in the original loop. So, if we go through this, i equal to 1, then i equal to 2, i equal to 3, then possibly i can become 2 again. We come here, we go back, i gets the value 3, and then i passes out of the loop and j gets value 3 corresponding to i equal 3.

In other words, i is getting reset to 3 every time that i go back to the header, but suppose i equal to 3 goes out and goes in to the basic block B1, first time i gets the value 3. Now, if I do not execute iterate through the loop at all, j can still get the value 3. However, suppose I get back i equal to 3, come here (Refer Slide Time: 42:01), then pass through this particular block, I get the value i equal to 2.

Now, let us say I go back any number of times, i still retains the value 2; it never changes, whereas in the original loop as it is written here (Refer Slide Time: 42:14), i would have been reset to 3. Whereas with i equal to 3 moved into B1, once i travels this particular path, i gets stuck with the value 2 and never changes irrespective of the number of times i execute this loop. So, j always gets a value 2 once i travels this particular path (Refer Slide Time: 42:34). So, there is a change in the semantics of the loop and therefore, this transformation is illegal.

(Refer Slide Time: 42:41)

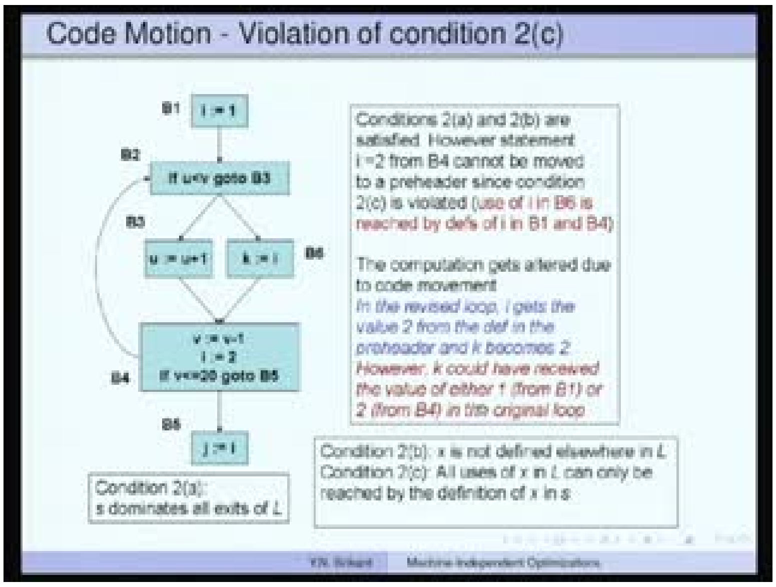


Loop-Invariant Code Motion Algorithm

- Find loop-invariant statements
- For each statement *s* defining *x* found in step (1), check that
  - (a) it is in a block that dominates all exits of *L*
  - (b) *x* is not defined elsewhere in *L*
  - (c) all uses in *L* of *x* can only be reached by the definition of *x* in *s*
- Move each statement *s* found in step (1) and satisfying conditions of step (2) to a newly created preheader
  - provided any operands of *s* that are defined in loop *L* have previously had their definition statements moved to the preheader
- Update all the $u - d$ and $d - u$ chains appropriately

The third one – all uses in L of the variable x can only be reached by the definition of x in s.

(Refer Slide Time: 42:58)



Code Motion - Violation of condition 2(c)

B1
i := 1
B2
if u<v goto B3
B3
u := u+1
k := i
B6
v := v-1
i := 2
if v<=20 goto B5
B4
B5
j := i

Conditions 2(a) and 2(b) are satisfied. However statement i =2 from B4 cannot be moved to a preheader since condition 2(c) is violated (use of i in B6 is reached by defs of i in B1 and B4)

The computation gets altered due to code movement

In the revised loop, i gets the value 2 from the def in the preheader and k becomes 2

However, k could have received the value of either 1 (from B1) or 2 (from B4) in the original loop

Condition 2(a):
s dominates all exits of *L*

Condition 2(b): *x* is not defined elsewhere in *L*
Condition 2(c): All uses of *x* in *L* can only be reached by the definition of *x* in *s*

Y.N. Srikant   Machine-Independent Optimizations

Let us see what happens. This flow graph satisfies condition 2(a), which says s dominates all exits of L. So, we are now looking at i equal to 2 in the basic block B4. B4 dominates itself and it is the exit. So 2(a) is trivially satisfied. What is 2(b)? x is not defined elsewhere in L. So, we are considering i equal 2 again in this block. i is not defined anywhere in the loop. So 2(b) also satisfied.

What is 2(c)? All uses of x in L can only be reached by the definition of x in s. So, here is i equal to 2. This is the definition of i. Here is (Refer Slide Time: 43:45) a use of i; k equal to i, but this i equal to 1 in the basic block B1 can reach this k equal to i, i equal to 2 in the basic block B4 can reach k equal to i after one iteration. So, both these definitions – i equal to 1 and i equal to 2 reach this basic block B6 and therefore, it is a violation of this condition 2(c).

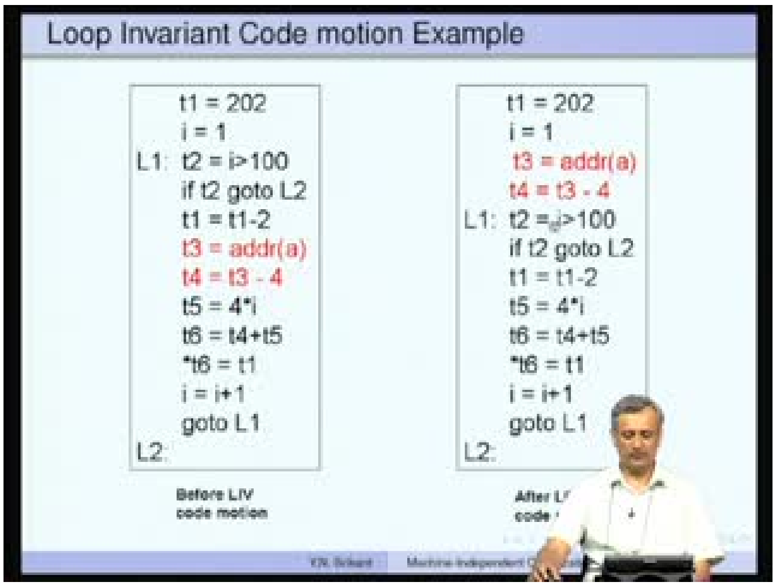
Can we now move i equal 2 into the basic block B1 because it is a loop-invariant computation? The answer is no because there is going to be a change in the semantics of this computation. k could have received the value of either 1 or 2 in the original loop; that is very clear. i equal to 1 could reach this (Refer Slide Time: 44:35), i equal 2 could reach this after one iteration. So, both the values are possible for this particular k. Whereas, once we move i equal to 2 outside, i always gets stuck with the value 2. Because i equal to 1 is overridden by i equal 2, this k will always get the value of 2 and will never get the value of 1. So, there is a change in this semantics of the computation and therefore, this is also an illegal transformation.

Assuming that the transformations are legal, we check these (Refer Slide Time: 45:09). How do we know that if we check these, they are legal? The proofs of these statements are all provided in the early papers written by Ullman and his students. So, they are all there. If you look up Sethi-Ullman’s book, at the end of it there are a large number of references, where the papers containing proofs of these statements are all provided.

Move each statement of statement s found in step (1) (Refer Slide Time: 45:44) and satisfying these conditions of step (2) to a newly created preheader. Preheader may become necessary if the basic block at the entry of the loop actually does not have a predecessor, which is a simple predecessor. In such a case, we need a preheader.

Provided any operands of s that are defined in loop L have previously had their definitions statements moved to the preheader.

(Refer Slide Time: 46:22)



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

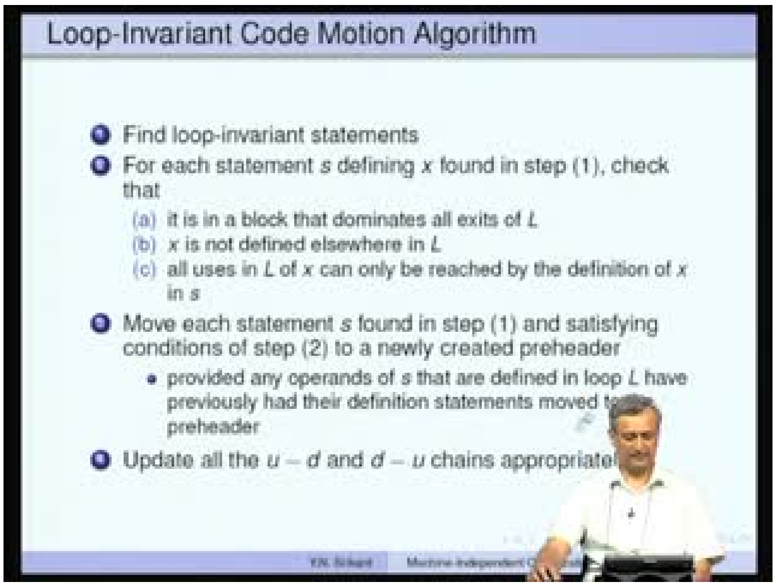
```
    t1 = 202
    i = 1
    t3 = addr(a)
    t4 = t3 - 4
L1: t2 = i>100
    if t2 goto L2
    t1 = t1-2
    t5 = 4*i
    t6 = t4+t5
    *t6 = t1
    i = i+1
    goto L1
L2:
```

After L
code

Y.N. Srikant Machine-Independent O

In other words, you cannot move k equal to i unless the statement i equal 2 is also moved. That is very clear in our simple example itself. I cannot move t4 equal to t3 minus 4 ahead of t3 equal to address a and I cannot move it in any other order except the same; that is real.

(Refer Slide Time: 46:39)


Loop-Invariant Code Motion Algorithm

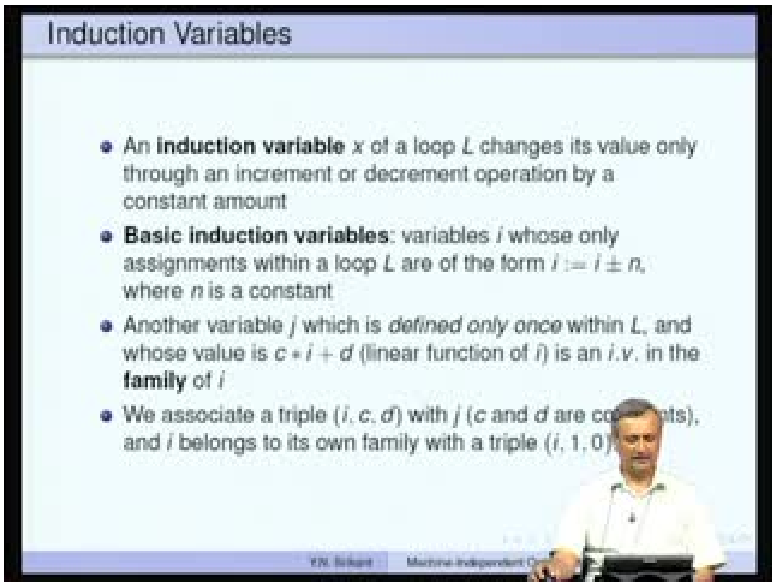
1. Find loop-invariant statements
2. For each statement s defining x found in step (1), check that
   (a) it is in a block that dominates all exits of L
   (b) x is not defined elsewhere in L
   (c) all uses in L of x can only be reached by the definition of x in s
3. Move each statement s found in step (1) and satisfying conditions of step (2) to a newly created preheader
   - provided any operands of s that are defined in loop L have previously had their definition statements moved to the preheader
4. Update all the $u-d$ and $d-u$ chains appropriately

Y.N. Srikant Machine-Independent O

So, I must move t3 equal address a first and then move t4 equal to t3 minus 4. That is what this really says. You must move the definition statements first and then move the used statements later.

Now, we need to update u-d and d u-chains also because the information has really changed. So, this is the loop-invariant code motion algorithm.

(Refer Slide Time: 46:59)


Induction Variables

- An **induction variable** x of a loop L changes its value only through an increment or decrement operation by a constant amount
- **Basic induction variables**: variables i whose only assignments within a loop L are of the form $i := i \pm n$, where n is a constant
- Another variable j which is *defined only once* within L, and whose value is $c * i + d$ (linear function of i) is an $i.v.$ in the **family** of i
- We associate a triple (i, c, d) with j (c and d are co[illegible]ts), and i belongs to its own family with a triple $(i, 1, 0)$

Y.N. Srikant Machine-Independent O

Now, we come to the next optimization, which is induction variable elimination and strength reduction.

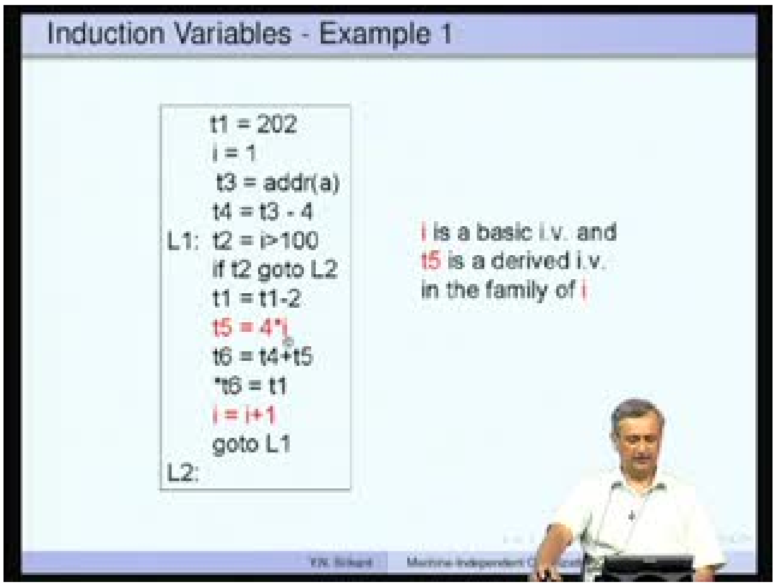
Let us see what is an induction variable is. There is a variable x in a loop L and if that variable changes its value only through an increment or decrement operation by a constant amount; so, i equal to plus or minus k, where k is a constant. Then, it is not necessary to have just one statement, which changes the value of the variable; you could have i equal to i plus 2 in the loop in one place and i equal to i plus 3 in another place in the same loop. So, there could be more than one such statement. Such a variable is called as an induction variable. Why? If we are actually either adding or subtracting a constant to or from a variable, then the value of that variable changes in arithmetic progression. That is the reason why it is called as an induction variable.

What is the basic induction variable? Variables , whose only assignments in the loop are of the form i plus minus n, where n is a constant. The above one (Refer Slide Time: 48:30) was a very general statement and it did not say that n must be a constant; it could have been i plus minus k, where k actually is incremented or decremented by a constant amount. So, indirectly i also gets incremented or decremented by a constant amount. That was possible here; we did not say it is plus minus; it could have been star also. So, a change is a constant amount; that is all. Whereas, in the basic induction variable, the specific form is given – i equal to i plus minus n.

Another variable j which is defined only once within L, and whose value is c star i plus d; linear function of d, is an induction variable in the family of the variable i. So, this is also an induction variable; it is just that j depends on i. So, if say, i is i equal to 1, so 1, 2, 3, 4, etcetera are the values of i, and then j is 4 star i; then j takes value 4, 8, 12, etcetera. Again it is an arithmetic progression, but j is defined in terms of i. Therefore, this is called as a derived induction variable, which is in the family of i.

We associate a triple – i, c, d with a derived induction variable; c and d are constants and i belongs to its own family with a triple – i, 1, 0. This is a definition; i, 1, 0, where as i, c, d is for any other variable, which is derived from i. In the previous example, with 4 star i, we would have had i, 4, and 0 as the triple.

(Refer Slide Time: 50:30)



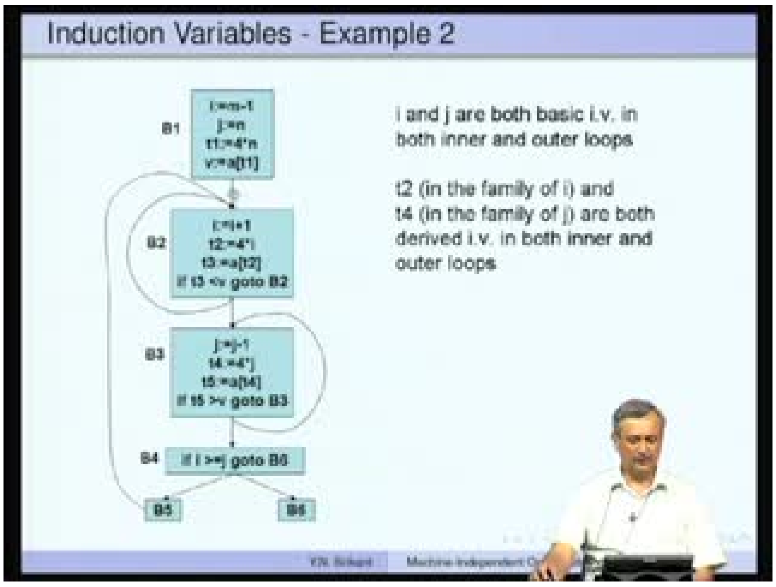
Induction Variables - Example 1

```
      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:
```

i is a basic i.v. and t5 is a derived i.v. in the family of i

Here, we have a simple example. If you take i equal to i plus 1, this is a basic induction variable. It is incremented by 1 in this particular loop. t5 equal to 4 star i is a derived induction variable, which is in the family of i. So, this is the example I have been giving. So, i has its triple as i, 1, 0 (Refer Slide Time: 50:57). Now, t5 has its triple as i, 4, 0. So, that is the triple for t5.

(Refer Slide Time: 51:09)



Induction Variables - Example 2

i and j are both basic i.v. in both inner and outer loops

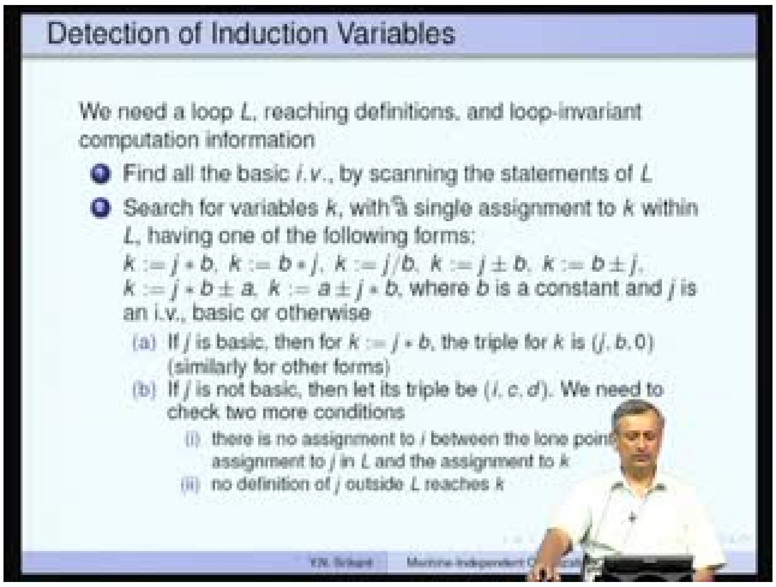
t2 (in the family of i) and t4 (in the family of j) are both derived i.v. in both inner and outer loops

In this example, there are many loops – there is a loop here, there is a another loop here, and the third loop here. When we consider induction variables, we need to look at the variable and every loop that is possible. For example, i equal i plus1 and this loop (Refer Slide Time: 51:32), it is incremented once and it is not modified in any other way. So, i is a basic induction variable in this particular loop.

If you consider the outer loop (Refer Slide Time: 51:43) and same i, it is still an induction variable in the outer loop also; that too a basic induction variable because i is not modified anywhere else and it is of the form i equal i plus minus n. The same is true for j as well; j equal j minus 1 is a basic induction variable in this loop and it is also a basic induction variable in the outer loop because j is not modified by any other quantity in the outer loop as well.

t2 equal to 4 star i is based on i. So, it is in the family of i and is a derived induction variable in this loop (Refer Slide Time: 52:23) and also in the outer loop. Similarly, t4 equal to 4 star j is an induction variable, which is derived from j; it is in the family of j both in this particular loop and also in the outer loop.

(Refer Slide Time: 52:40)



Detection of Induction Variables

We need a loop L, reaching definitions, and loop-invariant computation information

- Find all the basic *i.v.*, by scanning the statements of L
- Search for variables k, with a single assignment to k within L, having one of the following forms: $k := j * b$, $k := b * j$, $k := j/b$, $k := j \pm b$, $k := b \pm j$, $k := j * b \pm a$, $k := a \pm j * b$, where b is a constant and j is an i.v., basic or otherwise
  - (a) If j is basic, then for $k := j * b$, the triple for k is $(j, b, 0)$ (similarly for other forms)
  - (b) If j is not basic, then let its triple be (i, c, d). We need to check two more conditions
    - (i) there is no assignment to i between the lone point assignment to j in L and the assignment to k
    - (ii) no definition of j outside L reaches k

Y.N. Srikant | Machine-Independent Optimization

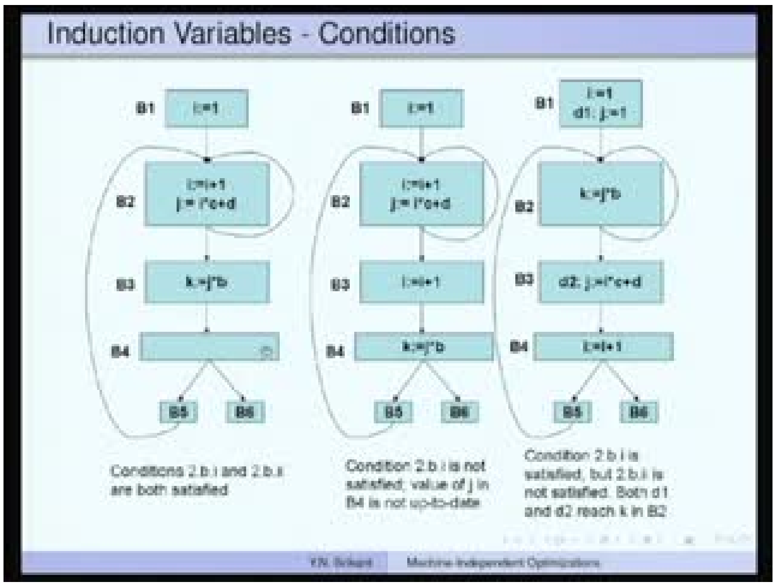
How do you detect induction variables? Easier said than done; let us see how to do that. We need a loop L, reaching definitions, and loop-invariant computation information. So, all these are necessary. Find all the basic induction variables by scanning the statements of L. This is fairly easy because the structure of the basic induction variables is i equal to i plus minus n. Then, the derived once are not so trivial.

Search for variables k, with a single assignment to k. So, k should not have been assigned more than once. Remember – Even in the definition of (Refer Slide Time: 53:28) variable in the family of I, we said j must defined only once. So, that is what we are sticking to in this particular algorithm. So, with a single assignment to k within L, having one of the following forms: k equal j star b, k equal to b star j, k equal j slash b, k equal j plus minus b; b plus minus j, and then j star b plus minus a, or a plus j star a plus minus j star b, where b is a constant and j is an induction variable; either basic or otherwise; you should have already detected it. That is what it really says. If it is basic, we should have (( )). If it is not a basic induction variable, we should have already said in which family this j lies. That information should be already available to us. Please observe that we do not have b slash j; that is not a linear form; j cannot be in the denominator.

If j is a basic induction variable, then for k equal to j star b, the triple for k is very simple, j, b, 0; this we already saw – j star b plus d; so, d is 0. Similarly, if you let us take k equal to j slash b, 1 by b is a constant, we might as well evaluate that constant at compile time and then say j comma 1 by b comma 0. That value can be substituted here (Refer Slide Time: 55:03). Similarly, for others also; j star b plus a, it is j comma b comma a or minus a and so on and so forth.

If j is not a basic induction variable, then there are many complications. Let the triple be i, c, d and we need to check two more conditions: there is no assignment to i between the lone point of assignment to j in L and the assignment to k, and no definition of j outside L reaches L.

(Refer Slide Time: 55:42)



Induction Variables - Conditions

Conditions 2.b.i and 2.b.ii are both satisfied

Condition 2.b.i is not satisfied; value of j in B4 is not up-to-date

Condition 2.b.i is satisfied, but 2.b.ii is not satisfied. Both d1 and d2 reach k in B2

Y.N. Srikant Machine-Independent Optimizations

I am going to demonstrate these conditions and explain them in the next class. So, this is the end of today’s class.

Thank you.