Compiler Design

Prof. Y. N. Srikant

Department of computer Science and Automation

Indian Institute of Science, Bangalore

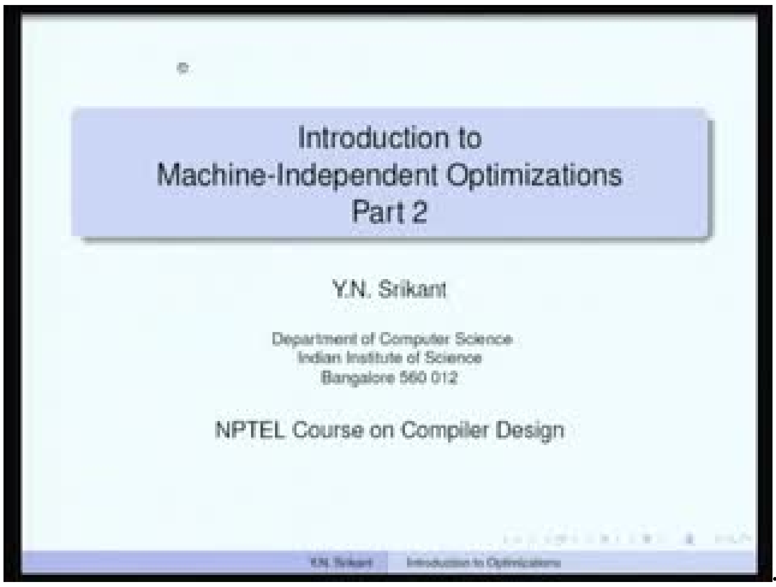
Module No. # 07

Lecture No. # 18

Introduction to Machine-Independent Optimizations-Part 2

Data-flow Analysis

(Refer Slide Time: 00:21)



Introduction to
Machine-Independent Optimizations
Part 2

Y.N. Srikant

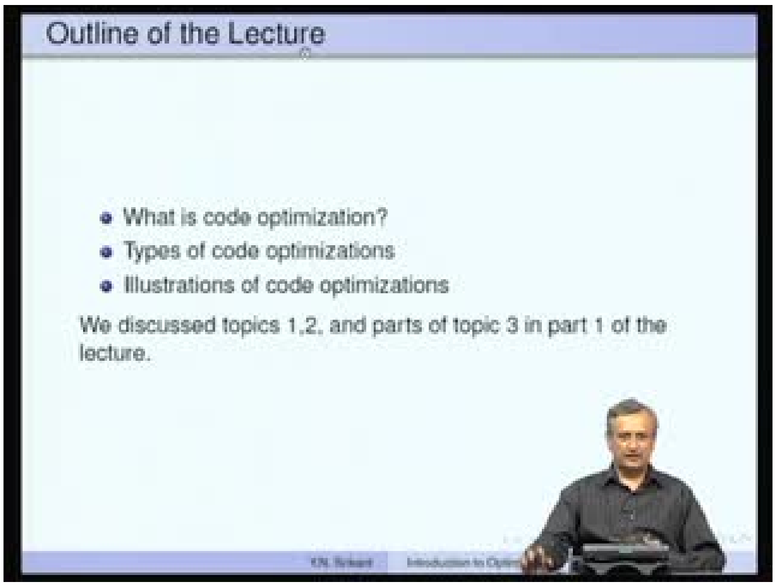
Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Y.N. Srikant Introduction to Optimizations

.

(Refer Slide Time: 00:24)


Outline of the Lecture

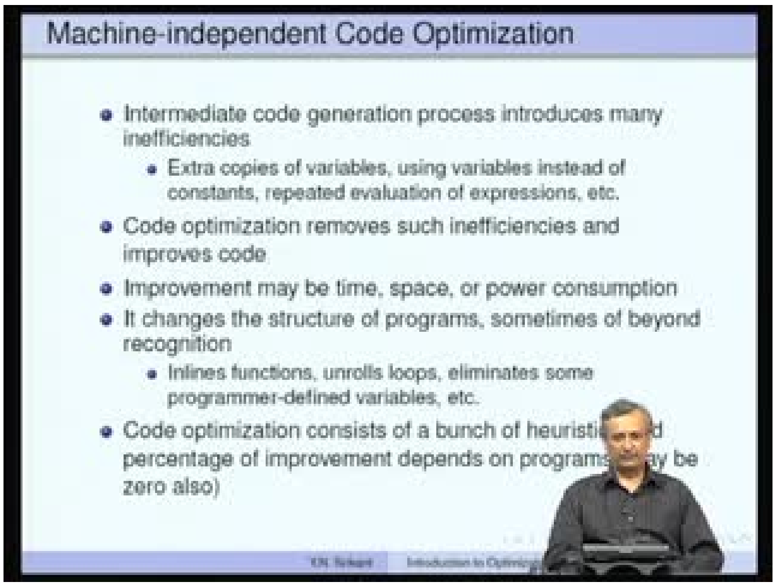
- What is code optimization?
- Types of code optimizations
- Illustrations of code optimizations

We discussed topics 1,2, and parts of topic 3 in part 1 of the lecture.

Y.N. Srikant Introduction to Optim

Welcome to part 2 of the lecture on machine independent optimizations. In the last lecture, we discussed the purpose of optimization, a couple of illustrations for optimizations. Today we will continue with the illustrations, look at a few more optimizations.

(Refer Slide Time: 00:43)


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 heuristi[illegible] percentage of improvement depends on program[illegible] be zero also)

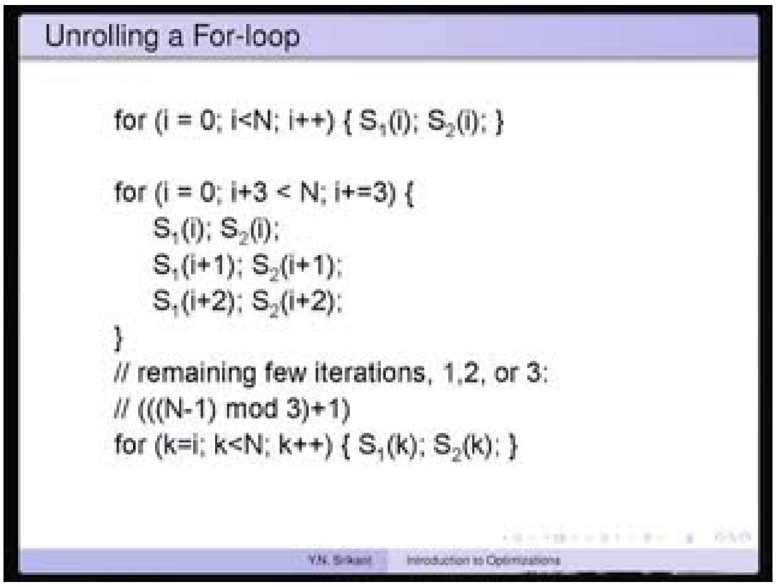
Y.N. Srikant Introduction to Optimiz

A quick recap. The purpose of machine independent code optimization is to remove the inefficiencies which are introduced by intermediate code generation. Improvement could be in the time domains, space domain, or power consumption domain. It changes the structure of the programs, sometimes beyond recognition, and it is a bunch of heuristics.

So, there is no guarantee that there will be improvement, but there will be improvement whenever there is a possibility of doing so.

The other important point is all the code optimizations actually are safe. In other words, they do not change the character of the program; whatever the program was doing, it will continue to do. So, the functionality of program is definitely not changed.

(Refer Slide Time: 01:41)



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, 1,2, or 3:
// (((N-1) mod 3)+1)
for (k=i; k<N; k++) { S1(k); S2(k); }
```

Y.N. Srikant   Introduction to Optimizations

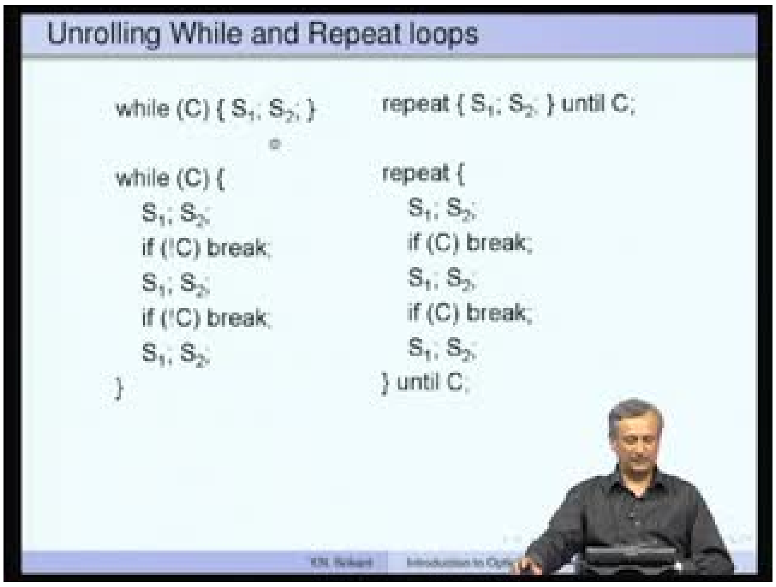
We looked at an optimization called unrolling a for loop last time. Let us quickly go through it once more, because that is needed for the other types of unrolling as well. Here is a for loop, i equal to 0, i less than N, and i plus plus, with two statements - S_1 and $S_{2,}$ the i inside parenthesis indicates that this is the i th instance of the loop iteration. So, in each iteration, it is a different statement as such instantiation of the statement.

When we want to unroll the loop, let us say three times, S 1 and S 2 will be repeated three times; it is indicated as S 1 i ,S 1 i plus 1, and S 1 i plus 2; - similarly, S 2 I, S 2 i plus 1, and S_2 i plus 2. Why do we want to unroll a loop? The primary purpose of unrolling a loop is to make the body of the loop much larger. Perhaps, it is needed to reduce the overheads of parallel execution as we will see very soon; so, perhaps for instructions scheduling, the body of the loop must be much larger, and so on and so forth.

So, in this case, the loop will again start from 0, but it will not run as many times as the old one that is the N part; so, here we always check i plus 3 less than N and we increment i by 3; so, it is really approximately N by 3 number of times that the loop really runs.

Then at the end of it, suppose N minus 1 is not a multiple of 3, suppose N is 6, then N minus 1 is 5 and it is not a multiple of 3. So, to begin with i equal to 0, the unroll loop runs once, then i is incremented by 3, i becomes 3, and now 3, 4 and 5 are actually few more than what is needed, this part of the loop comes into play.

(Refer Slide Time: 04:45)



Unrolling While and Repeat loops

```
while (C) { S1; S2; }          repeat { S1; S2; } until C;

while (C) {                    repeat {
    S1; S2;                        S1; S2;
    if (!C) break;                 if (C) break;
    S1; S2;                        S1; S2;
    if (!C) break;                 if (C) break;
    S1; S2;                        S1; S2;
}                              } until C;
```

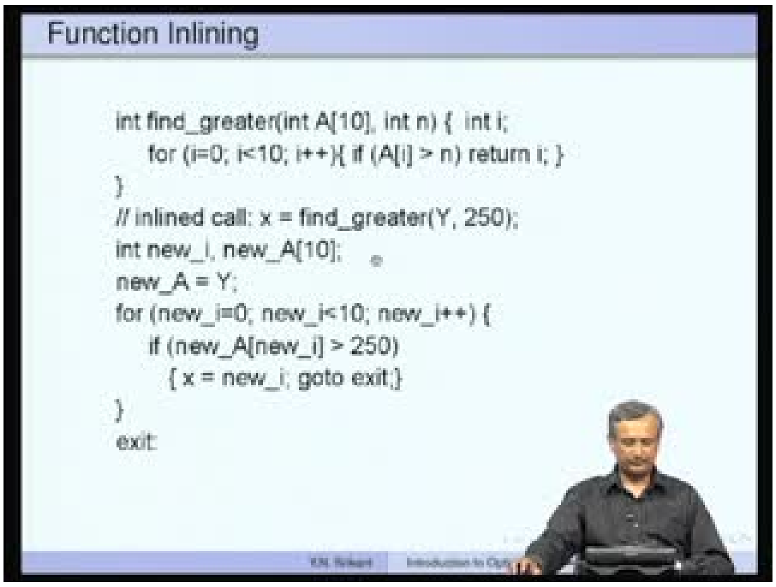
Here, we begin with k equal to i that is 3, and then we are going to run 3, 4 etcetera in this part of the loop and complete the rest of it; otherwise, some of the iterations may be left out, that is the major problem here. Unrolling while and repeat loops is similar, for example, we have while C S_1 $S_{2;}$ so, two statements in a while loop and repeat S_1 S_2 until C is a repeat loop; so, if the while loop is unrolled then again let us say 3 times we have three instance of S_1 and S_2 here, the only problem here is we do not know whether that condition C will hold true after one iteration, two or three iterations.

After every set of statement S_1 and S_2 we need to introduce a break, conditional break; so, if the condition C is false, then you really get out of the loop and stop the loop. You should not be executing these two.

So, if it is true then we come here, S_1 and S_2 executed again, and you check again. So, these two checks are necessary to make sure that these two instances are not executed unnecessarily. It is exactly the other way here, repeat until loop executes at least once; so, we execute here at least once, does not hold; C is checked here the first time and if it is false, we do not execute the loop at all. That is already taken care of by the while condition here.

Here, S_1 and S_2 will be executed at least once and then you check it. $S_{1,}$ S_2 goes on executing until this condition C is true, and once it is true, it stops execution of the loop. Instead of checking not C as in this previous case, we check whether C is true. If C, then break; similarly, S_1 S_2 and then if C then break, S_1 S_2 and until C. So, this is rolling the while and repeat until loops three times very similar to the for loop, but there are conditional breaks in between.

(Refer Slide Time: 06:47)



Function Inlining

```
int find_greater(int A[10], int n) {  int i;
    for (i=0; i<10; i++){ if (A[i] > n) return i; }
}
// inlined call: x = find_greater(Y, 250);
int new_i, new_A[10];
new_A = Y;
for (new_i=0; new_i<10; new_i++) {
    if (new_A[new_i] > 250)
        { x = new_i; goto exit;}
}
exit:
```

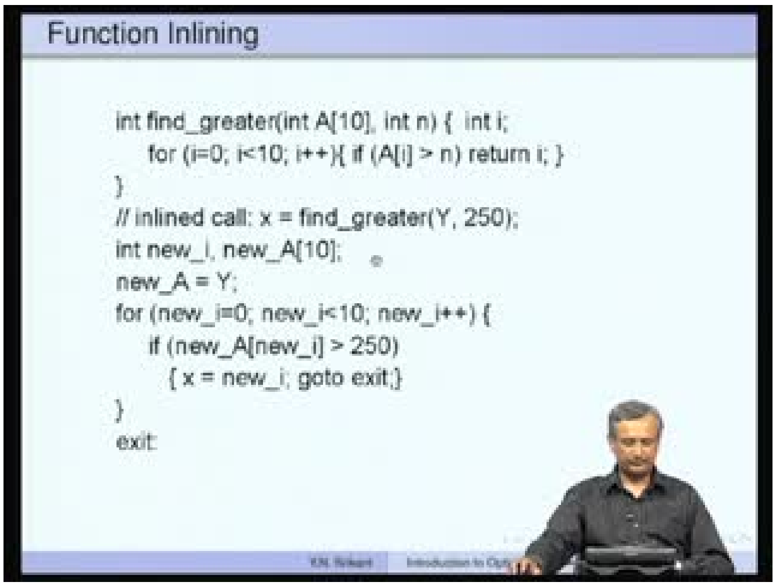
The next optimization that we are going to consider is the function inlining. Let us take a very simple function, find_greater int A 10 comma int n. This is supposed to traverse the entire array a, check whether there is an element A i greater than n, and if so, return that index. A very simple function.

What does inlining this function imply? Inlining function implies that whenever there is a call, x equal to find_greater y comma 250, there will be no sub routine jump executed, but this particular statement, find_greater function call will be replaced by the body of this particular function with some renaming of variables.

Why should we do this? This is really going to make the execution much faster because the overheads of sub routine jump and sub routine return or creation of activation record etcetera are all eliminated.

Let us see how it works. Let us say, the call is x is equal to find_greater (y, 250) array is y and n is 250, in the place of i that we have declared a new I, and in the place of this array A, we have a new array called new A. We begin with new A equal to y, then you iterate - new i equal to 0, new i less than 10, and new i plus plus - this is the replacement for the i variable that is here and the check - Y A i greater than n - will be new A with new i greater than this n, which is 250 and if so we just assign x equal to new I - that is we are executing the return statement now, and then go to exit, jump out of the body.

(Refer Slide Time: 06:47)



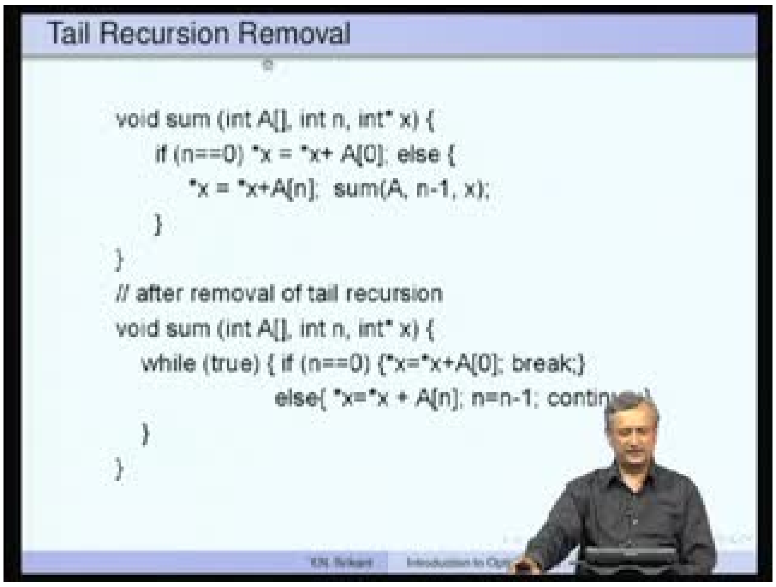
Function Inlining

```
int find_greater(int A[10], int n) {  int i;
    for (i=0; i<10; i++){ if (A[i] > n) return i; }
}
// inlined call: x = find_greater(Y, 250);
int new_i, new_A[10];
new_A = Y;
for (new_i=0; new_i<10; new_i++) {
    if (new_A[new_i] > 250)
        { x = new_i; goto exit;}
}
exit:
```

So, there is no question of sub routine jump and sub routine return, but the value that is necessary will indeed be returned by this particular statement. The advantage here is, as I said, the overheads are much lesser and the disadvantage is in the place of one sub routine call, we now have a complete body of the sub routine call that is the code itself. So, if there are ten places where the sub routine is the function or procedure is called, you would have ten pieces of this code instantiated in those places; so, the code has really blown up and if you inline every function in the program, the code size could become impossible to manage - unmanageable.

Therefore, a compilers provide directives to say whether some code should be inlined or it should not be inlined, and based on that inlining of the functions and procedures happens. The other point is, it is not possible to inline recursive functions and recursive procedures. We really do not know how long that procedure will run and that may depend on some value which is not known at compile time. So, inlining of recursive function is not possible.

(Refer Slide Time: 10:41)



Tail Recursion Removal

```
void sum (int A[], int n, int* x) {
      if (n==0) *x = *x+ A[0]; else {
            *x = *x+A[n];  sum(A, n-1, x);
      }
}
// after removal of tail recursion
void sum (int A[], int n, int* x) {
   while (true) { if (n==0) {*x=*x+A[0]; break;}
                  else{ *x=*x + A[n]; n=n-1; contin
   }
}
```

Now, we come to next optimization, called Tail Recursion Removal. Just now I said, recursion cannot be removed, so, I am not contradicting myself when we say it is possible to remove tail recursion. Let us understand what is tail recursion first.

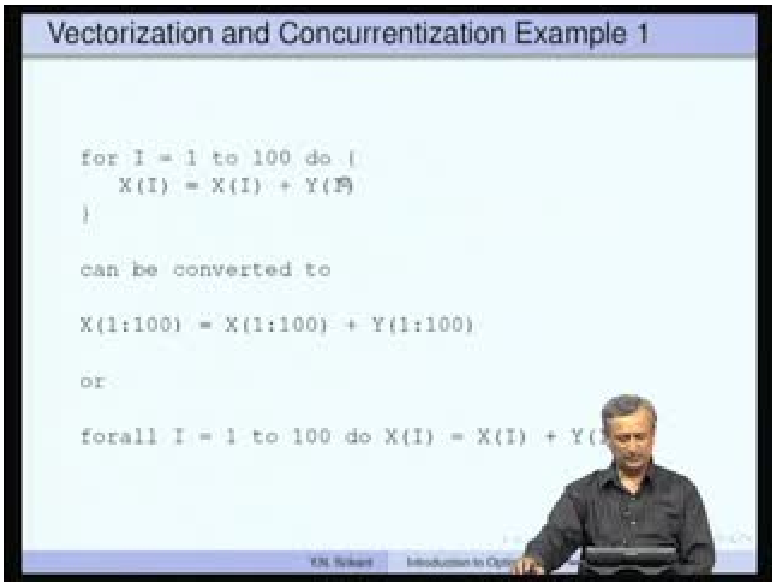
Here, you have a function called sum, a very simple one, which takes an array A as a parameter, an integer called n and a pointer to integer called x in *x, the body is, if n equal to 0, then *x equal to *x plus A 0, it is actually assuming that *x has been initialized to some particular value, now this function is supposed to sum up all the elements of the array A, so, if n is 0, the zeroth element is put in to *x, otherwise *x is equal to *x plus A n, and then sum is called again with n minus 1 as the parameter and the same x.

As we go on with small and smaller value of n, this will be n minus 1, n minus 2 etcetera. Eventually it becomes a zero and recursion terminates here. This is called a tail recursive function simply, because you know the recursive function call is last executable statement in the body of the function. Please see here, then part has no recursion at all and in the else part, there is an assigment statement and then there is a sum call. Such a recursive statement, a procedure is called a tail recursive procedure and it is possible to convert such a tail recursive function or procedure to a while loop and that is called removal of tail recursion.

Let us see what happens, you have the same function declaration as before, int A, int n, and int *x and instead of recursion, we simply have while true and then the first part is the same, if n equal to 0 *x equal to *x plus A 0 and then get out of the while loop with a break, because this is where the recursion terminates, that is we are doing here. Otherwise, we are now looking at this body, *x equal to *x plus A, n remains the same instead of calling of function sum with A n minus 1 A x, we do n equal to n minus 1 and then say continue, the while loop continues and the execution of the body of the while loop also continues, it terminates when n is equal to zero.

So, this is the translation of the tail recursive procedure to a while loop and this will work only if the function or procedure is tail recursive and it will not work in the case of general recursion.

(Refer Slide Time: 13:55)



Vectorization and Concurrentization Example 1

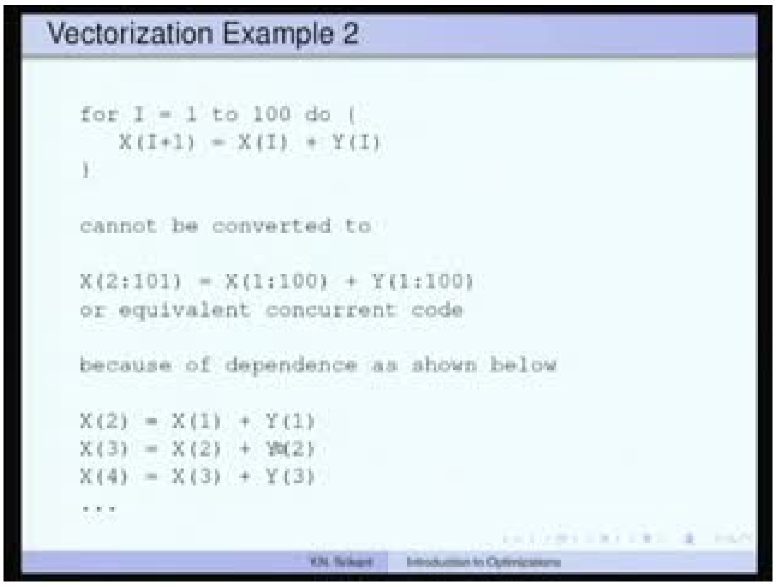
```
for I = 1 to 100 do {
   X(I) = X(I) + Y(I)
}

can be converted to

X(1:100) = X(1:100) + Y(1:100)

or

forall I = 1 to 100 do X(I) = X(I) + Y(
```

Vectorization and Concurrentization. This is another optimization which is possible. We are going to study this in some detail later on. Let us say, we have a simple for loop, I equal 1 to 100, do X I equal to X I plus Y I. It is easy to see that for every value of I, the array element which is X X that of X and Y are different, in I equal to 1 it is X I equal to X I plus Y I, it is I is equal to 2 it is X 2 plus y 2 and so on and so forth.

So, it is possible to run every iteration of this particular loop in parallel. In a vector computer, it is possible to execute all these statements, let us say hundred of them in a This entire array X is accessed once into a vector register. The entire array Y is accessed once into the vector register of twice hundred. So, there will be hundred additions, which happens at the same time with corresponding elements of X into Y taken, and this assignment is also a vector assignment which assigns it back to the variable X, which would be in a vector register.

If it is a parallel multi core type of machine, then every iteration will run in parallel mode. That is why, the indication is for all I equal to 1 to 100, so, iteration 1 2 3 etcetera will run on let us say, one hundred cores of the multi core machine, each one of them executing single assignment, but with a different I. That completes the purpose of for all loop, you can execute them in parallel.

(Refer Slide Time: 15:56)



Vectorization Example 2

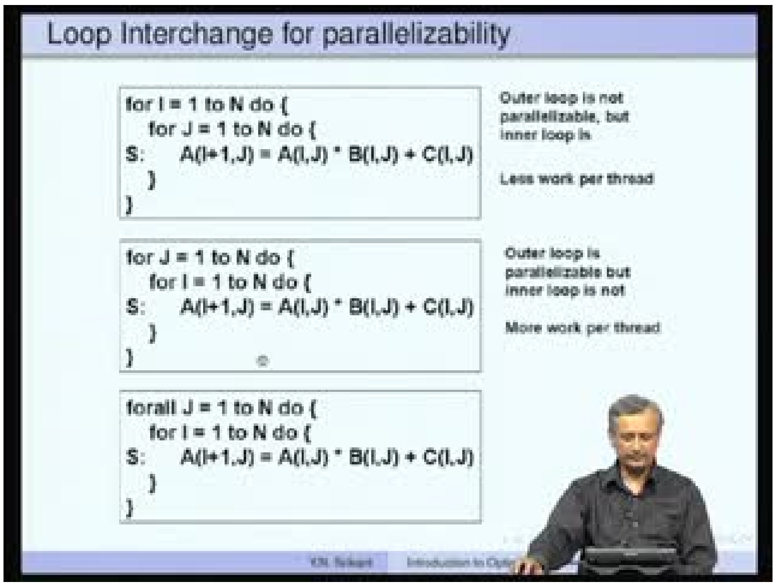
```
for I = 1 to 100 do {
   X(I+1) = X(I) + Y(I)
}

cannot be converted to

X(2:101) = X(1:100) + Y(1:100)
or equivalent concurrent code

because of dependence as shown below

X(2) = X(1) + Y(1)
X(3) = X(2) + Y(2)
X(4) = X(3) + Y(3)
...
```

Y.N. Srikant Introduction to Optimizations

Let us look at another example of vectorization to see what is not possible. We have again a very simple loop, but with a small change, instead of X I equal to X I plus Y I, we have X I plus 1 equal to X I plus Y I, this is not a parallelizable loop, the reason is, let us expand the loop, take I equal 1 so we get X is equal to X 1 plus Y 1, take Y X I equal 2 we get X 3 equal 2 X 2 plus Y 2 then X 4 equal to X 3 plus Y 3 etcetera.

The X 2 which was computed with I equal to 1 is used in I equal to 2, X 3 which was computed with I equal to 2 is used in I equal 3, so, every value which is computed in iteration I is used in the iteration I plus 1. Because of this, it is not possible to run the iterations in vector or parallel mode independently. They have to run in a particular sequential order as indicated by the for loop. So, this loop is not parallelizable and it is not vectorizable. We are going to study later conditions for vectorization, conditions for parallelization, how to actually modify the loops so that these conditions are modified and so on and so forth.

(Refer Slide Time: 17:19)



Loop Interchange for parallelizability

```
for I = 1 to N do {
   for J = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Outer loop is not parallelizable, but inner loop is

Less work per thread

```
for J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Outer loop is parallelizable but inner loop is not

More work per thread

```
forall J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

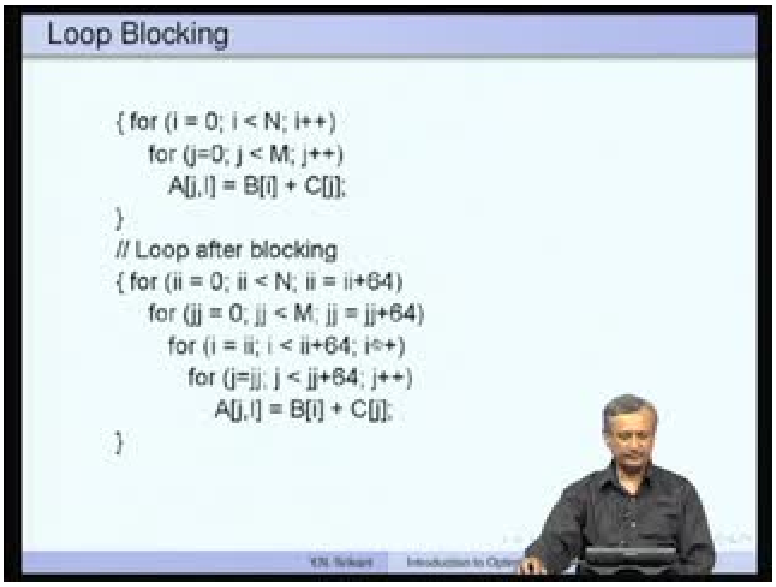
Then, there is a transformation called loop interchange. Let us look at this particular loop, I equal to 1 to N, J equal to 1 to N, A I plus 1 comma J is A I J * B I J plus C L J. In this case, the outer loop is not parallelizable. You will have to take my word for this particular thing, because it is not possible to demonstrate how it is so. The technical machinery that we already know, we need to learn about dependences which we are going to do a little later.

Let us assume that the outer loop is not parallelizable, but the inner loop is definitely parallelizable. If you actually parallelize just the inner loop and run the outer loop in sequential mode, then each one of the inner loop statements will run in parallel mode, so, the work is actually very small for each iteration, so, in a parallelizable loop, the course must get as much work as possible for each thread that runs on the court.

So, what we really do is interchange these two loops, the J loop go outside and I loop comes inside. Now, we have J equal to 1 to N and I equal to 1 to N, now the outer loop is parallelizable and the inner loop is not. So, we are going to run the inner loop in sequential mode and run the outer loop in parallel mode, so, for each iteration, we create a thread and that thread actually execute this entire for loop.

There is a lot more work in this case per thread, because this entire loop is executed for each iteration of J, so, this is a much more beneficial case for us. The overheads of parallel execution are going to be small, because the work inside the thread is quite large. So, we are going to employ such loop interchange whenever necessary during parallelization.

(Refer Slide Time: 19:31)

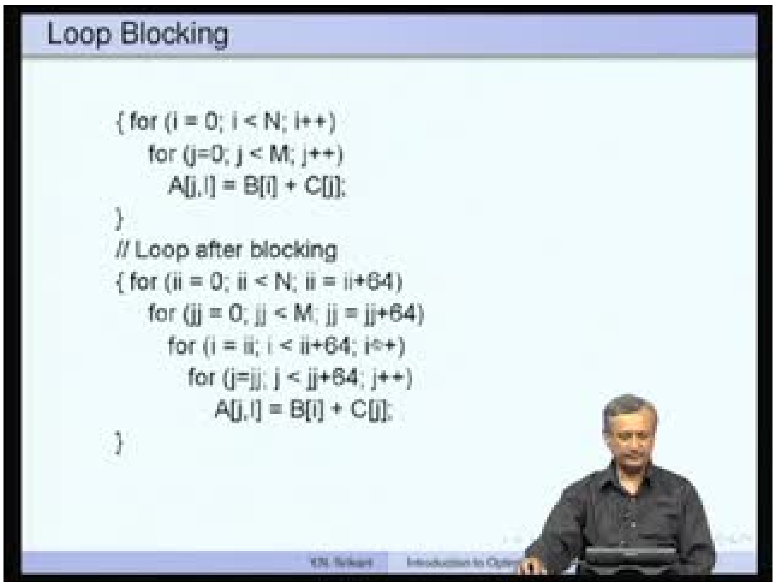


Loop Blocking

```
{ for (i = 0; i < N; i++)
    for (j=0; j < M; j++)
      A[j,i] = B[i] + C[j];
}
// Loop after blocking
{ for (ii = 0; ii < N; ii = ii+64)
    for (jj = 0; jj < M; jj = jj+64)
      for (i = ii; i < ii+64; i++)
        for (j=jj; j < jj+64; j++)
          A[j,i] = B[i] + C[j];
}
```

The last optimization we are going to see, an illustration of which is here, is called loop blocking. Here is a small loop i equal to 0, i less than N, i plus plus, and j equal to 0, j less than M, and j plus plus, so, the loop is A j i equal to B i plus C i, if this loop is run as it is, it may so happen that is because of the large sizes of N and M, the cache misses the N and M, may be much larger than the cache size of the machine, so every time we execute this, there may be a number of cache misses, so the speed of the loop will become very less.

(Refer Slide Time: 19:31)



Loop Blocking

```
{ for (i = 0; i < N; i++)
    for (j=0; j < M; j++)
      A[j,i] = B[i] + C[j];
}
// Loop after blocking
{ for (ii = 0; ii < N; ii = ii+64)
    for (jj = 0; jj < M; jj = jj+64)
      for (i = ii; i < ii+64; i++)
        for (j=jj; j < jj+64; j++)
          A[j,i] = B[i] + C[j];
}
```

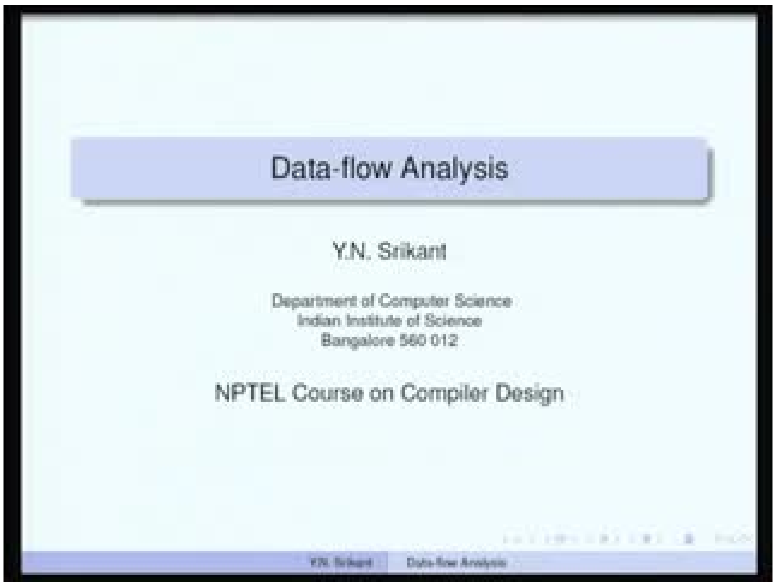
What we can try to do is, divide these two loops into smaller loops, so instead of two loops, now we have four loops, the outer two loops run with an increment of 64 each, ii equal to 0, ii less than N, ii is equal to ii plus 64, and similarly, jj is equal to 0, jj less than M, and jj equal to jj plus 64.

But the original i and j loops now run from 0 to 63, rather ii to 63, so first time it is 0 to 63, second time it will be 64 to 127 and so on and so forth. Similarly, jj will also run from jj to jj plus 64. Now, what happens is, because of the small size 64, this is not a magic number, it is just that the 64 will be the size of the cache line in a particular processor, it could be vary from processor to processor, so if that happens, then this entire inner loop now iterates on j, so we are going to modify C j very quickly, but B i will not be modified so quickly.

So, this entire C j M i fit into a cache line. There will be only hits from then onwards similarly, the entire B into a cache line. So, we are going actually keep the B part remain in a cache for a very long time, j at this C part every 64 numbers will be effected and in the next iteration 64 new numbers will be placed in the cache and so on. So, every time we get the 64 numbers, they will be used immediately. Number of caches will be very large and therefore, it is a much more efficiency scheme compare to the previous one.

This is the end of this particular lecture on introduction to optimizations. Now, we are going to continue with the next one called data flow analysis.

(Refer Slide Time: 22:37)



Data-flow Analysis

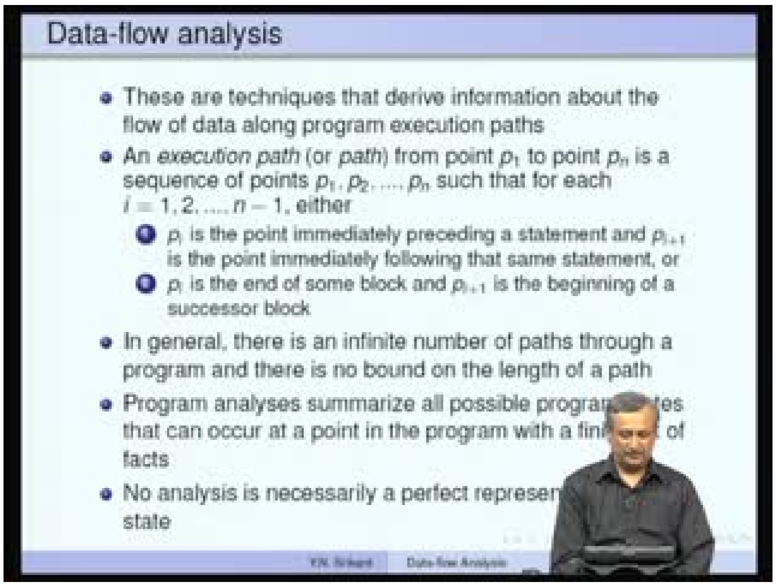
Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Y.N. Srikant | Data-flow Analysis

(Refer Slide Time: 22:38)



Data-flow analysis

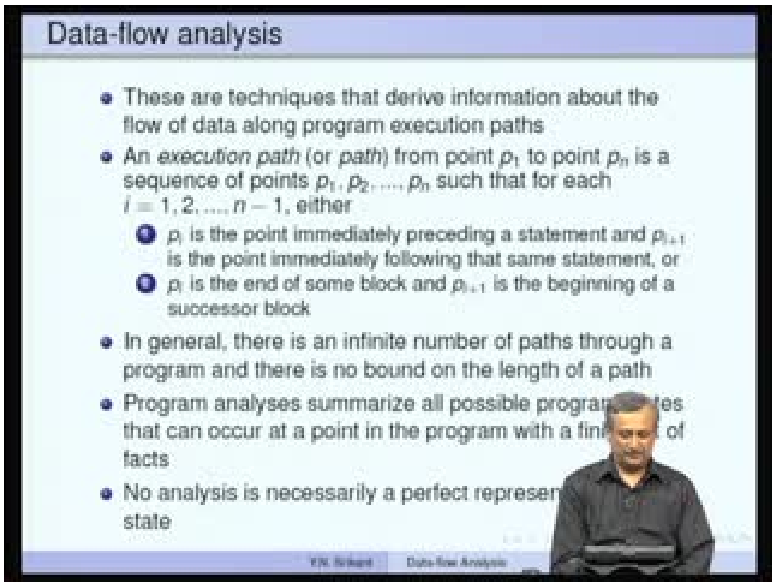
- These are techniques that derive information about the flow of data along program execution paths
- An *execution path* (or *path*) from point p_1 to point p_n is a sequence of points $p_1, p_2, \ldots, p_n$ such that for each $i = 1, 2, \ldots, n-1$, either
  1. p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
  2. p_i is the end of some block and p_{i+1} is the beginning of a successor block
- In general, there is an infinite number of paths through a program and there is no bound on the length of a path
- Program analyses summarize all possible progra[illegible]es that can occur at a point in the program with a fin[illegible] of facts
- No analysis is necessarily a perfect represen[illegible] state

Y.N. Srikant | Data-flow Analysis

Welcome to a lecture on data flow analysis. What is data flow analysis? We saw many examples of optimizations so for, and we are yet to see how exactly these optimizations can be performed inside a compiler. We need to actually compute a lot of information that is available in the program, take it inside the compiler, and then the information will be used by the compiler in order to perform the optimizations. That is how it goes.

Collection of information by the compiler by analyzing the program, traversing the program is called data flow analysis. These are techniques that derive information about the flow of data along program execution paths. What is a path in a program? You have a beginning for the program and end for the program, let us assume one function or one main program. You start from a point $p_{1,}$ an execution path from point p_1 to p_n is a sequence of points p_1 e to $p_{2,}$ such that for each i, p_i is the point immediately preceding a statement and p_i plus 1 is the point immediately following the same statement or p_i is the end of some block and p_i plus 1 is the beginning a successor block. In other words, the program begins execution, it follows particular sequence, executes particular sequence of instructions. Between each of these instructions is a point and if you actually connect all these points, we will be able to collect information by traversing these paths p_1 to and p_n.

(Refer Slide Time: 22:38)



Data-flow analysis

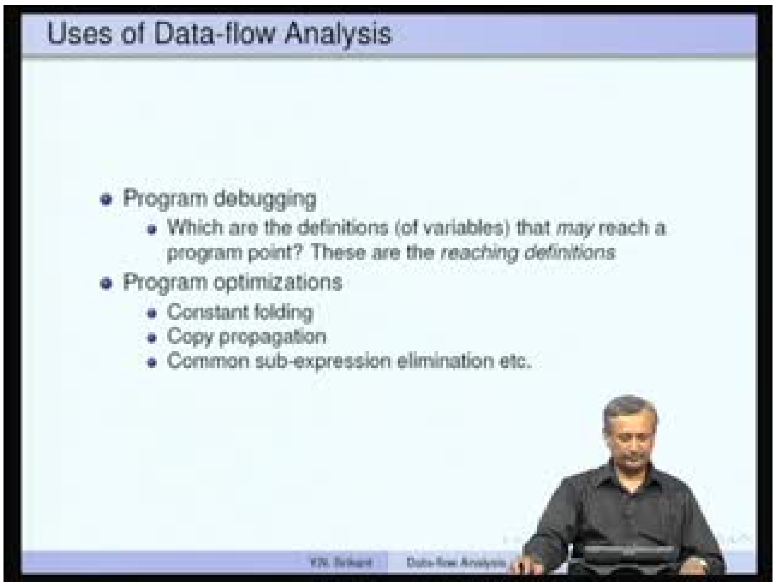
- These are techniques that derive information about the flow of data along program execution paths
- An *execution path* (or *path*) from point p_1 to point p_n is a sequence of points $p_1, p_2, \ldots, p_n$ such that for each $i = 1, 2, \ldots, n-1$, either
  1. p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
  2. p_i is the end of some block and p_{i+1} is the beginning of a successor block
- In general, there is an infinite number of paths through a program and there is no bound on the length of a path
- Program analyses summarize all possible progra es that can occur at a point in the program with a fin of facts
- No analysis is necessarily a perfect represen state

Y.N. Srikant Data-flow Analysis

In general, there is an infinite number of paths through a program and there is no bound on the length of a path. So, it is not possible to actually execute all the paths in the program and then collect information. It is just not possible to do that, whereas, program analyses summarize all possible program states that can occur at a point in the program with a finite set of facts. This is a very important point. At any point in the program, we are actually going to only store summary information, so there will be large number of programs states that are possible at a point, perhaps if the program can run continuously without termination there will be in infinite number of states.

If the program is going to terminate in a certain amount of time, then the number of possible states is not infinite, it is finite, but it is still extraordinarily large. Therefore, this summarization is a very important point. We will see how it is done. No analysis is necessarily a perfect representation of the state, so, it does not matter which analysis we are actually doing. Every one of these is in some sense going to create a loss of information. Nothing is a perfect representation and this is the conservative nature of the analysis.

(Refer Slide Time: 27:10)



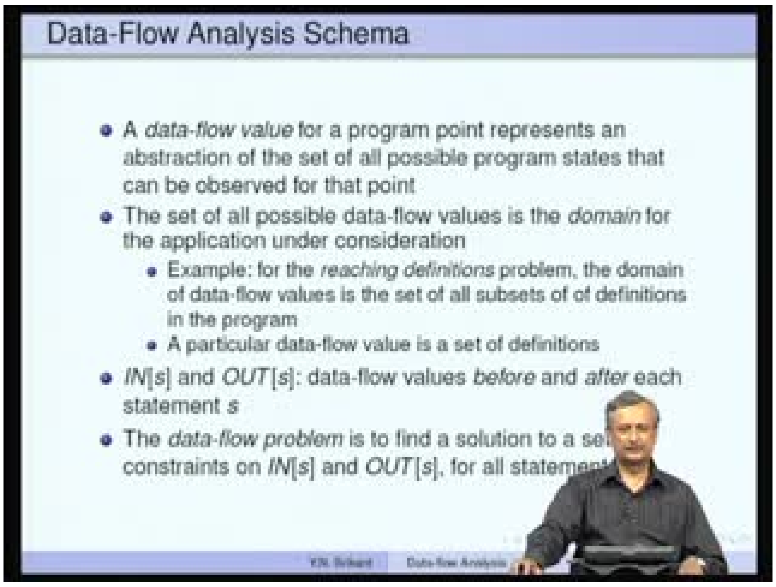
Uses of Data-flow Analysis

- Program debugging
  - Which are the definitions (of variables) that *may* reach a program point? These are the *reaching definitions*
- Program optimizations
  - Constant folding
  - Copy propagation
  - Common sub-expression elimination etc.

Y.N. Srikant
Data-flow Analysis

What are the uses of the data flow analysis? Data flow analysis information is used for program optimization, of course, I have given you plenty of examples, for constant propagation, copy propagation, etcetera and it is also useful for program debugging purposes, for example, which are the definitions of variables that may reach a program point, there are what are known as reaching definitions, so we may want to compute the reaching definitions and then use it in program debugging.

(Refer Slide Time: 27:49)



Data-Flow Analysis Schema

- A *data-flow value* for a program point represents an abstraction of the set of all possible program states that can be observed for that point
- The set of all possible data-flow values is the *domain* for the application under consideration
    - Example: for the *reaching definitions* problem, the domain of data-flow values is the set of all subsets of of definitions in the program
    - A particular data-flow value is a set of definitions
- *IN*[*s*] and *OUT*[*s*]: data-flow values *before* and *after* each statement *s*
- The *data-flow problem* is to find a solution to a se constraints on *IN*[*s*] and *OUT*[*s*], for all statem

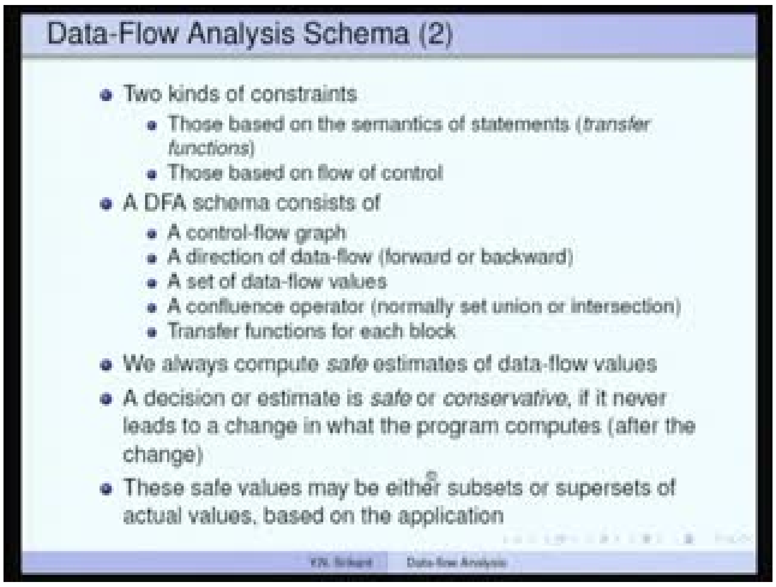
Y.N. Srikant Data-flow Analysis

Let us see what exactly is a data flow analysis schema. Before that, we need to understand, what is a data flow value? Data flow value for a program represents an abstraction of the sets of all program states that can be observed for that particular point. The set of all possible data flow values is the domain for the application under consideration. So, the point is, the data flow value itself is an abstraction and the set of all data flow values is the domain.

Let us take a simple example for the reaching definitions problem. We are going to define this as the first data flow analysis problem very soon. The domain of data flow values happens to be the set of all subsets of definitions in the program. In other words, if there are ten definitions we are going to consider the subsets of all the ten definitions in the program there will be 2 to the power 10 subsets possible and each one of these is a possible data flow value. So, this is the abstraction that we are looking at.

Why is this important? We will see very soon. Each data flow value will possibly be an estimate of the information that reaches a particular point. Then, we should also know what is IN s and OUT s as notation? These are data flow values just before and just after the statement s. If we are looking at basic blocks, this becomes IN b and OUT b. So, IN b is the data flow value before the basic block is entered and OUT b is the data flow value just after the basic block is completed.

(Refer Slide Time: 30:44)



Data-Flow Analysis Schema (2)

- Two kinds of constraints
  - Those based on the semantics of statements (*transfer functions*)
  - Those based on flow of control
- A DFA schema consists of
  - A control-flow graph
  - A direction of data-flow (forward or backward)
  - A set of data-flow values
  - A confluence operator (normally set union or intersection)
  - Transfer functions for each block
- We always compute *safe* estimates of data-flow values
- A decision or estimate is *safe* or *conservative*, if it never leads to a change in what the program computes (after the change)
- These safe values may be either subsets or supersets of actual values, based on the application

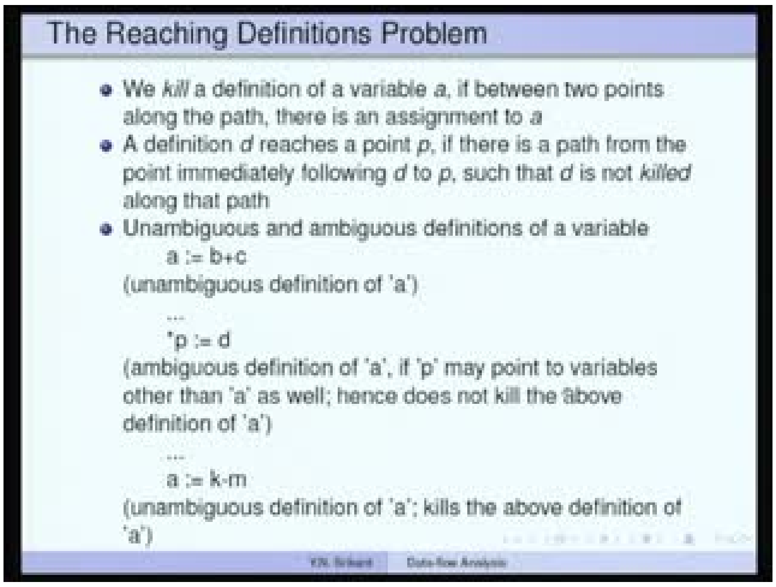
Y.N. Srikant Data-flow Analysis

The data flow problem itself is to find a solution to a set of constraints on IN s and OUT s for all the statements s. So, for us it may become the basic block IN b, so, we will be looking at constraints on IN b and OUT b. Continuing with the data flow analysis schema, we are yet to see what the constraints are? There are two types of constraints possible - one those based on the semantics of the statements, how exactly statements in the program modified the data flow values, so, these are transfer functions and those based on the control flow.

The control flow may change because of the conditions becoming true or false, thereby the data flow will also change that set of constraints. Now, we are ready to define informally what a data flow analysis schema is. It consists of a control flow graph, a direction of data flow either forward or backward. This will be clear only when we see an example, a set of a data flow values is a domain, a confluence operator, normally set union or set intersection operation, and then we have transfer function for each of the blocks. So, these are the five entities in our data flow analysis schema.

Another important point is, we always computed what are known as safe estimates of data flow values. What is safe about? A decision or estimate is safe or conservative, if it never leads to a change in what the program computes after the change. So, these safe values may be either subsets or supersets of actual values based on the application. It is difficult to say or define the safe estimate without looking at a particular application. The only thing we say is, it never leads to a change in what the program computes. Therefore, we will see what safety means as we take up each particular example of data flow analysis.

(Refer Slide Time: 33:22)



The Reaching Definitions Problem

- We *kill* a definition of a variable *a*, if between two points along the path, there is an assignment to *a*
- A definition *d* reaches a point *p*, if there is a path from the point immediately following *d* to *p*, such that *d* is not *killed* along that path
- Unambiguous and ambiguous definitions of a variable

```
      a := b+c
(unambiguous definition of 'a')
      ...
      *p := d
(ambiguous definition of 'a', if 'p' may point to variables
other than 'a' as well; hence does not kill the above
definition of 'a')
      ...
      a := k-m
(unambiguous definition of 'a'; kills the above definition of
'a')
```

Y.N. Srikant Data-flow Analysis

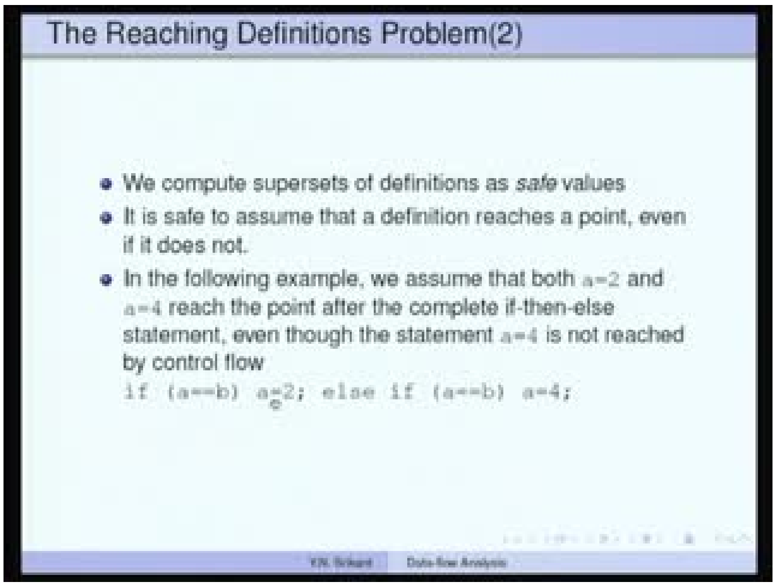
The first problem is the reaching definitions problem. Before we define what exactly we mean by a definition d reaches a point, we need to understand what we mean by killing a definition. So, will kill a definition of a variable if between two points along the path, that is we have a definition of a variable and then we have a path starting from that particular definition and it is going on and on. Along that path, pick up any two points and between these two points, there is an assignment to a, so the previous definition is not valid anymore, there is a new value assigned to a, so the new value will now be available along that path from now on.

So, this new assignment of a kills the old definitions of the variable a. A definition d reaches the point p, if there is a path from the point immediately following the definition d to p, the point p, such that d is not killed along that path. So, this is what we mean by a reaching definition. For a particular point there may be many definitions which reached that point and are not killed along which they reach the point. Let us take a simple example and understand what we mean by unambiguous and ambiguous definitions of a variable. Here is an example of an assignment called, which is a equal to b plus c, this does not involve any pointers, there all ordinary variables, so, this is called as unambiguous definition of a. There is definition attached to a.

Let us say, there is another assignment star p equal to d, this is called ambiguous definition of a, because this pointer p may point to a, it may point to other variables also. It is not certain during compilation, what exactly p points to, it may be a, it may be a few others also, in fact sometimes we may not know what p points at to all unless some kind of pointer analysis has been conducted. Assuming that we have done some pointer analysis, it is still not going to be possible for us to say precisely what p points to at this point during the execution. We can only say, p will possibly point to a b c that is all, which one of these we do not know.

Because we are not certain that this is an assignment to a, it could be assignment, it need not be also, we will say this ambiguous definition will not kill the previous unambiguous definition, whereas, sometime along the same path from this onwards, we had another assignment a equal to k minus m, this assignment has ordinary variables, so this is a new definition of a and this will definitely kill the old definition of a. This is how we understand killing of definitions and what ambiguous definitions do.

(Refer Slide Time: 37:04)



The Reaching Definitions Problem(2)

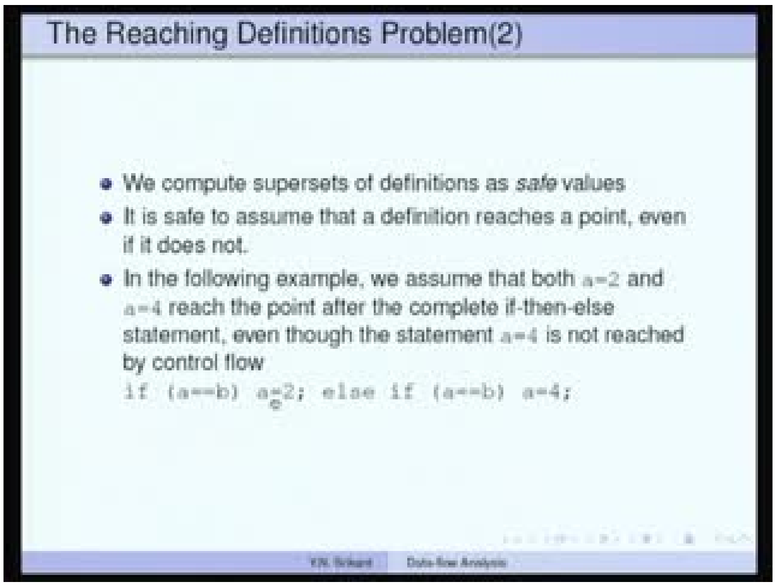
- We compute supersets of definitions as *safe* values
- It is safe to assume that a definition reaches a point, even if it does not.
- In the following example, we assume that both `a=2` and `a=4` reach the point after the complete if-then-else statement, even though the statement `a=4` is not reached by control flow

```
if (a==b) a=2; else if (a==b) a=4;
```

Y.N. Srikant Data-flow Analysis

So, for the reaching definitions problem we compute supersets of definitions as safe values. In other words, it is safe to assume that a definition reaches a point even if does not, it is not going to harm the application. If you do not know whether it reaches or not then let us not say it reaches. Suppose we said that, then we may be actually harming the application. So, that is why the conservative or safe estimate of a data flow value at any point will depend on the application.

(Refer Slide Time: 37:04)



The Reaching Definitions Problem(2)

- We compute supersets of definitions as *safe* values
- It is safe to assume that a definition reaches a point, even if it does not.
- In the following example, we assume that both a=2 and a=4 reach the point after the complete if-then-else statement, even though the statement a=4 is not reached by control flow

```
if (a==b) a=2; else if (a==b) a=4;
```

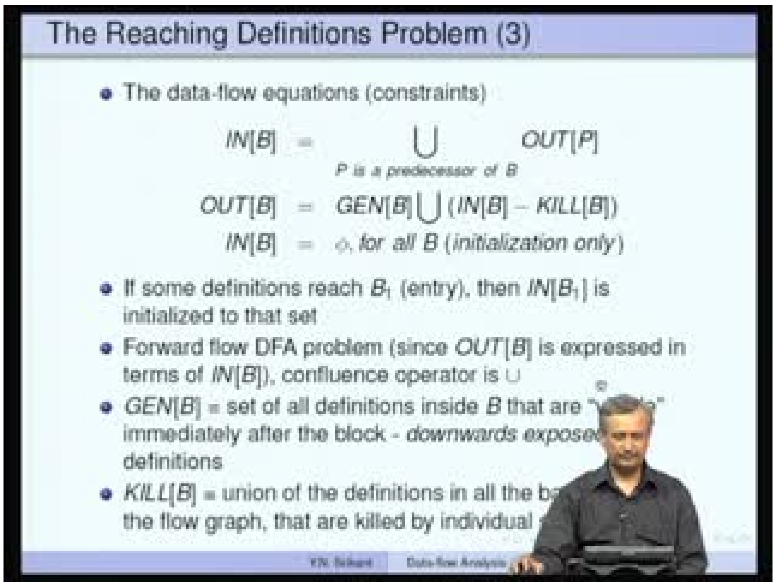
Y.N. Srikant Data-flow Analysis

Let us take the following simple example- a equal 2, and a equal to 4, we assume that both of them reach the point after the completion or complete if then else statement is over, that is you have a equal to b, a equal to 2, else a equal to b, a equal to 4, this is the completion point of the if then else statement. So, we are going to assume that a equal to 2 and a equal to 4 reach this point, both of them.

If you look at it very closely, suppose a was equal to b then a equal to 2 would have been executed and the else part would have been skipped, in that case a equal to 4 would never have been executed; suppose a equal to b was false, in that case, we are again checking whether a equal to b which is again false, so a equal to 4 is not executed in the else class also.

In other words, a equal to 4, this definition, that is the value of 4 in a will never reach this point, but we have no idea of what the values of a and b are. We have not tried to actually interpret these values as we go along. So, in case of if then else of this kind, we are going to assume a equal to 2, and a equal to 4 both of them reach here, but in reality only a equal 2 possibly reaches this point. So, this is the conservative estimate and since we do not know the values of a and b, we said both of them will reach even one of them reaches.

(Refer Slide Time: 39:30)



The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

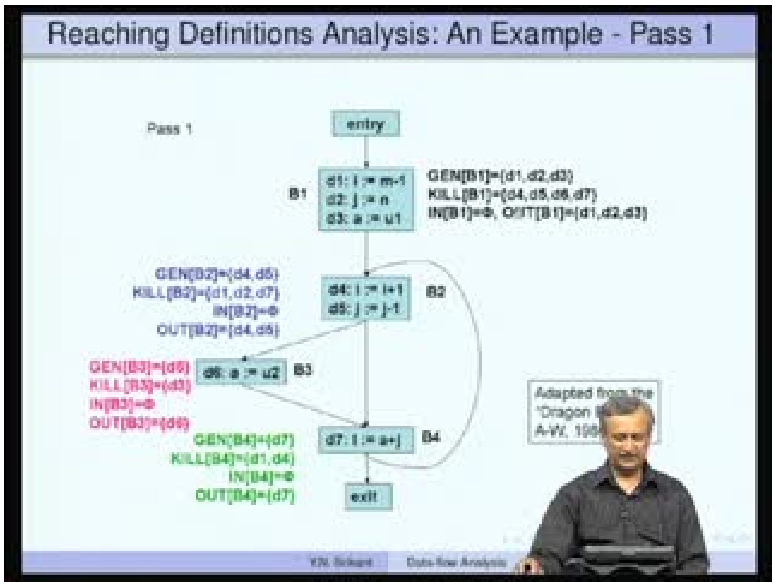
$$IN[B] = \phi, \textit{ for all } B \textit{ (initialization only)}$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
- $KILL[B]$ = union of the definitions in all the b the flow graph, that are killed by individual

Y.N. Srikant Data-flow Analysis

Now, we come to the heart of the problem formulation. All the data flow constraints are normally written as equations, here we have two equations, one for IN B, another one for OUT B and some initialization. Let us look at the syntax of the equation, it says IN of B, B is a basic block, IN of B is the set of values which reach the beginning of a basic block is the union of OUT P where P is a predecessor of the basic block B.

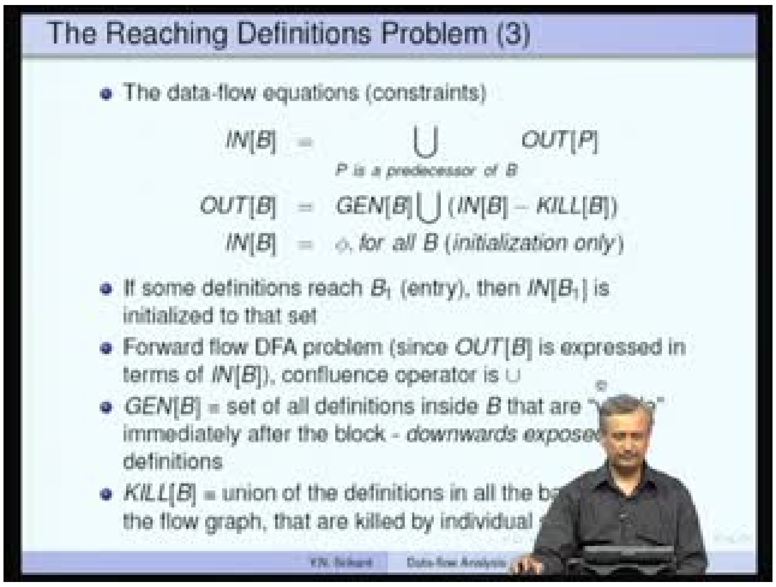
(Refer Slide Time: 40:26)



Reaching Definitions Analysis: An Example - Pass 1

Y.N. Srikant Data-flow Analysis

(Refer Slide Time: 40:45)



The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

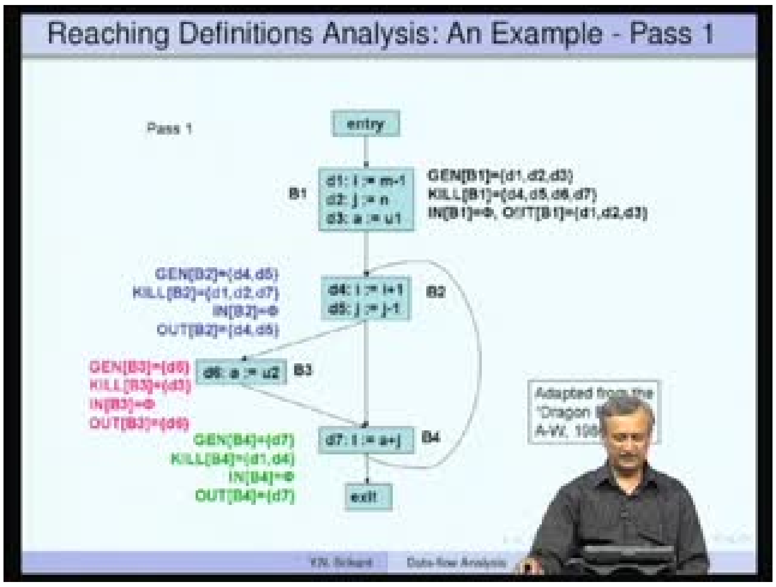
$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
- $KILL[B]$ = union of the definitions in all the basic blocks of the flow graph, that are killed by individual

Y.N. Srikant Data-flow Analysis

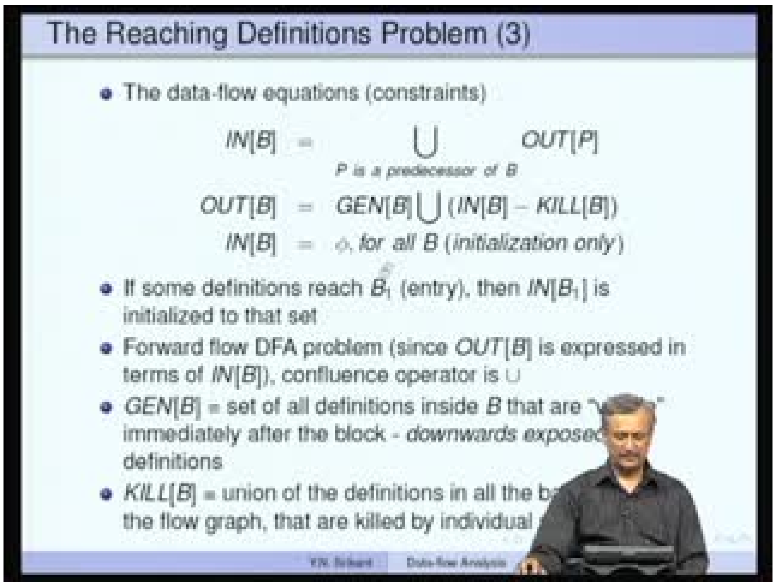
Let us take this example, here you have a basic block B1, this is the IN point of a basic block, this is the OUT point of the basic block, similarly, here is IN of B2 and OUT of B2, what it says is IN of B is a union of the outsets of all the predecessor of B.

(Refer Slide Time: 40:52)



Let us take this B2, this has B1 as one of the predecessors and B4 as the other predecessor. When we compute the inset for B2, we are going to take the outsets of these two and then make a union out of them. That is what this really says.

(Refer Slide Time: 41:15)



The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

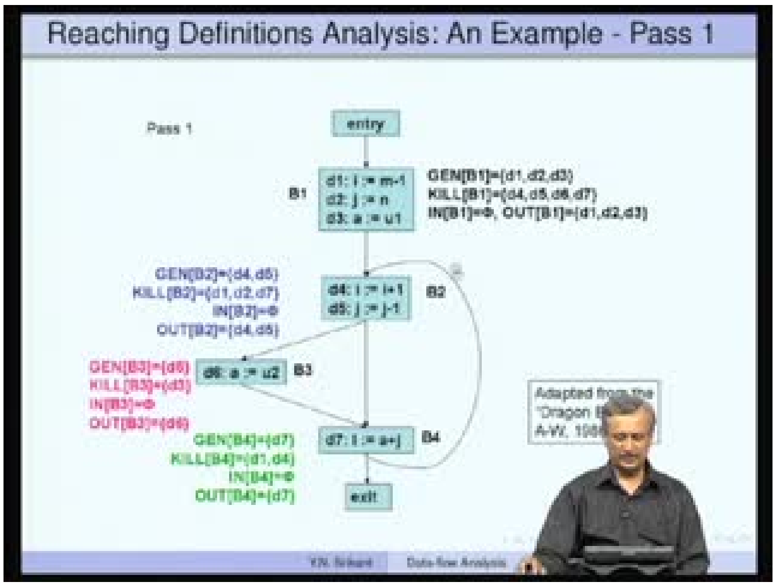
$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$
$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$
$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "v
immediately after the block - *downwards expose*
definitions
- $KILL[B]$ = union of the definitions in all the b
the flow graph, that are killed by individual

Y.N. Srikant Data-flow Analysis

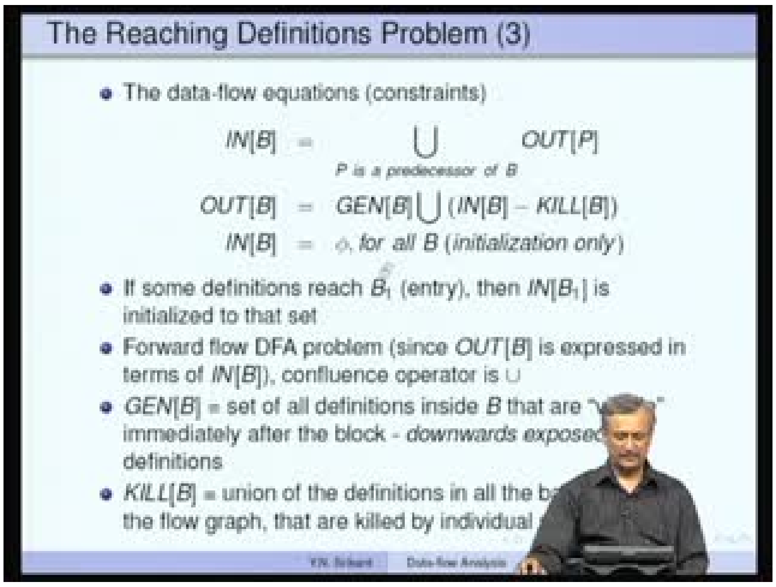
So, this union operator which is present here is called as the confluence operator and this could be intersection. It is possible that this is intersection. The set OUT B, the equation is written as GEN of B union with IN B minus KILL B, so here the equation is written as OUT B equal to, this GEN B and KILL B are constants they are computed before the data flow analysis problem is solved they are not actually computed during the execution of the problem, so, they are constants, IN B is a variable, so, OUT B is really a function of IN B.

(Refer Slide Time: 42:15)



Reaching Definitions Analysis: An Example - Pass 1

(Refer Slide Time: 42:23)



The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

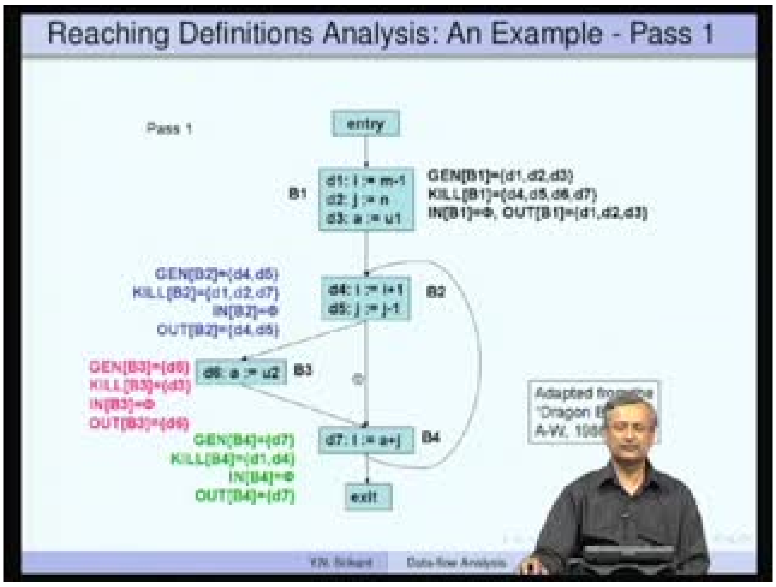
- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "v
immediately after the block - *downwards expose*
definitions
- $KILL[B]$ = union of the definitions in all the b
the flow graph, that are killed by individual

Y.N. Srikant Data-flow Analysis

In such a case, we first compute this value IN and then we compute the value OUT. So, this type of a problem where OUT is a function of IN is called as a forward data flow analysis problem. Here, we have a confluence operator union and forward flow analysis in this particular example.

Now, let us understand what GEN B and KILL B are. GEN B is the set all definitions inside the basic block B that are visible immediately after the block, that is, downwards exposed definitions. Let us understand this with the example.

(Refer Slide Time: 43:02)



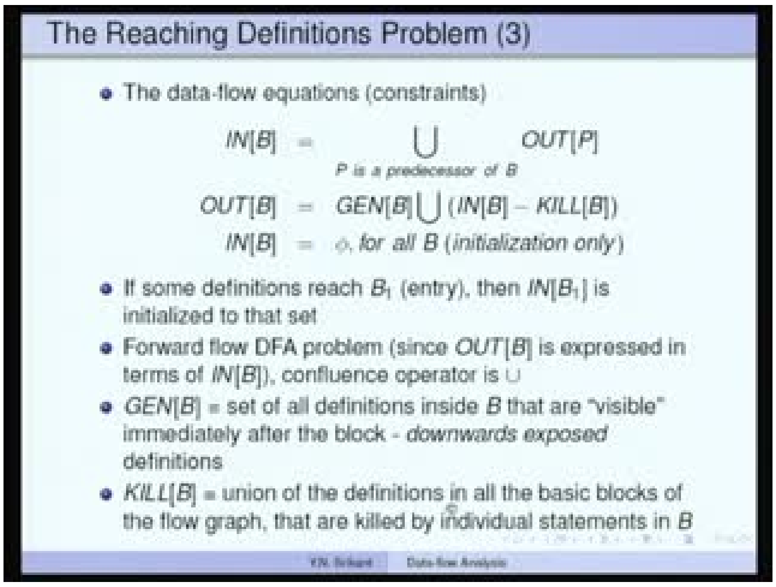
Reaching Definitions Analysis: An Example - Pass 1

Here is the basic block B1, there are three definitions with d1, d2, d3 in this particular basic block. We are looking at this particular point when we compute GEN B. d1 assigns a value to i, d2 assigns a value to j and d3 assigns a value to a. At this point, all these three definitions are valid because none of them have been super seeded any other definitions.

For example, we had in the same block, suppose we had d4, i equal to some k minus 1. In such a case, d1 would have been killed by d4, the value defined by the d1 would not have been visible after d4, so the GEN B at this point would not have contained d1, whereas in this case, this is not so, all the three definitions are visible at this point, the values they compute are all valid at this point, so GEN of B1 is d1 comma d2 comma d3.

Now, things are very similar in B2. It defines i and j, d4, and d5 and they are still valid at the exit point of B2. So, GEN of B2 is d4 and d5. GEN of B3 similarly, is d6. There is only one definition and there is no question of super seeding it and GEN of B4 is d7, then again only one definition.

(Refer Slide Time: 44:35)



The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

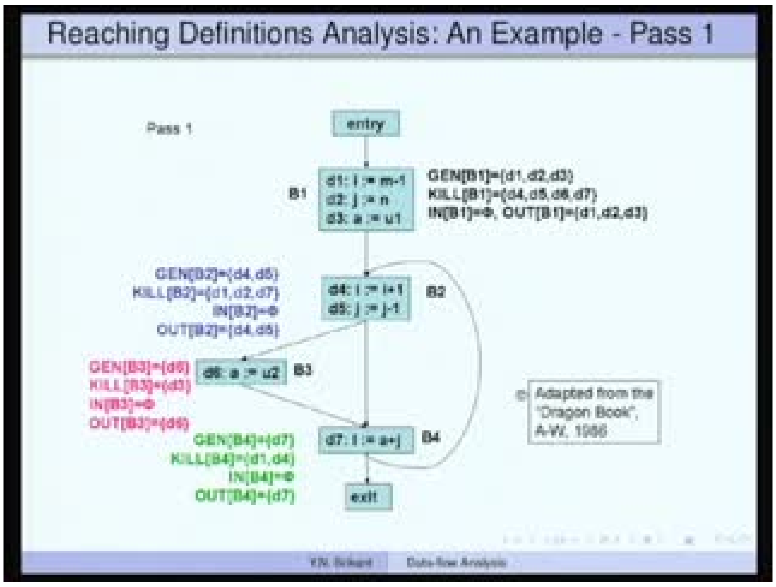
$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
- $KILL[B]$ = union of the definitions in all the basic blocks of the flow graph, that are killed by individual statements in B

Y.N. Srikant Data-flow Analysis

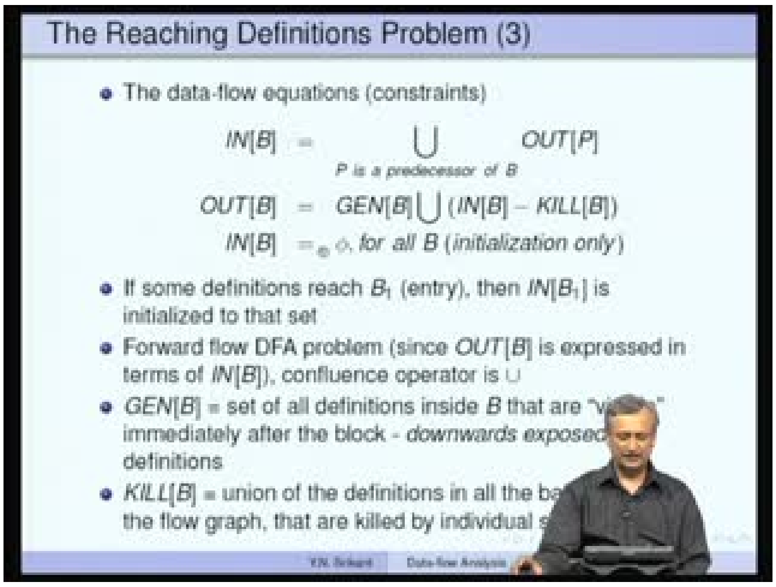
(Refer Slide Time: 44:56)



KILL B, there is a small problem which is interesting. It is the union of the definitions in all the basic blocks in the flow graphs that are killed by individual statements in B. Let us say this particular block B2 defines i and defines a. We will say that the kill sets for B2 consist of all the statements in the other blocks, B1, B3, and B4 which actually define i. So, for d5 similarly, we are going to consider all the statements in B1, B3, and B4 which define j. This i will actually kill d1; it will not allow this particular value to go through the basic blocks. That is what we mean by killing.

Similarly, the value of i defined by d7 will not be allowed to go through this particular block, because there is a new definition of i here, B4. It is irrelevant whether control can actually reach this point at all; it just does not matter to us, the reason is, we are going to compute kill set of a particular basic block as an over approximation.

(Refer Slide Time: 46:39)



The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

$$IN[B] =_{0} \phi, \textit{ for all } B \textit{ (initialization only)}$$

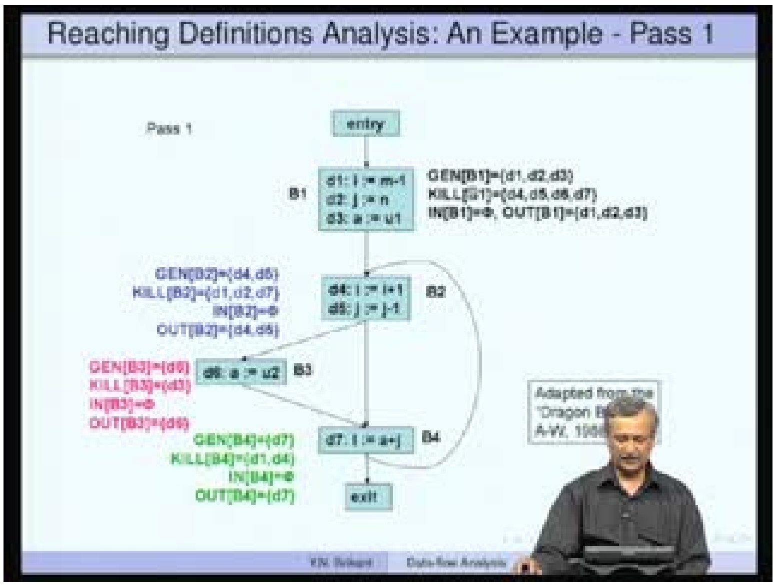
- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "vi[illegible]" immediately after the block - *downwards exposed* definitions
- $KILL[B]$ = union of the definitions in all the ba[illegible] the flow graph, that are killed by individual s[illegible]

Y.N. Srikant Data-flow Analysis

There will be some elements in it which are not relevant, but they are not harmful to us. If we want to compute the kill set of a basic block exactly the set of statements which is actually killed by that basic block, then we should actually solve the reaching definitions problem and then go back and compute the kill set. So, this is like having the chicken before the egg. We would like the KILL set to be available for computing the data flow values and we cannot say that we will have to solve the problem of reaching definitions to compute the value of KILL.

So, we actually over approximate KILL as the union of the definitions in all the basic blocks, whether there is a control flow possible from that basic blocks to B at all is irrelevant; we are not even going to check it. We just take all those which are definitions which are killed by individual statements B, put them in a set, and that is our KILL set.

(Refer Slide Time: 47:17)

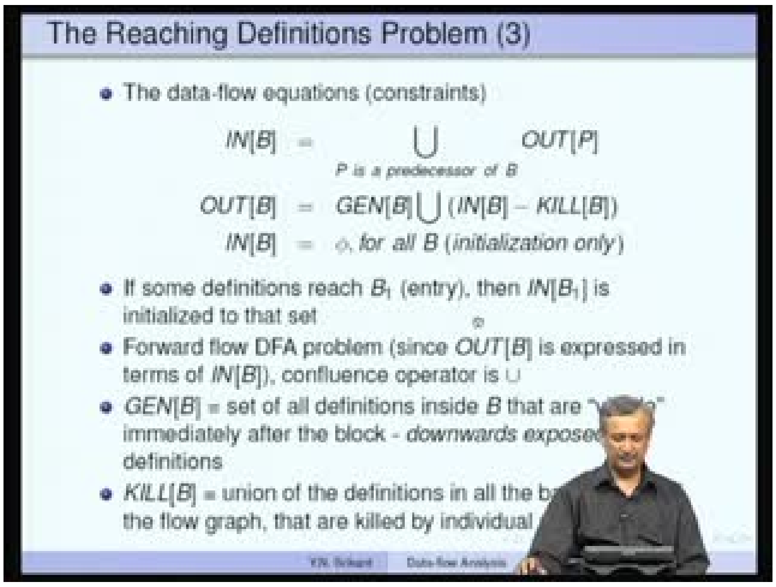


For example, here is a nice illustration of what I just now said. See d1 defines i and it is not possible for the definitions d4 and d7 to enter the basic block B1 at all, because from B1 once we go out, we will not be able to come back to B1, but still the KILL set of B1 will contain d4 and d7 and corresponding to j it will contain d5 and corresponding to a, it will contain d6.

So, all these are in the KILL set. This is the over approximation, i really fine. In reality, this would have been really five, nothing at all, these definitions cannot KILL anything.

So, now for this particular block KILL B2, d4 kills d1, d4 kills d7, and then d5 kills d2. That is why the KILL set here. For the basic block B3, the KILL set is just d3 because there is only one definition of a that is available here that is in basic block B1 and for B4, it is d1 and d4 because these are the only two definitions of i in the other two blocks.

(Refer Slide Time: 48:52)



The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

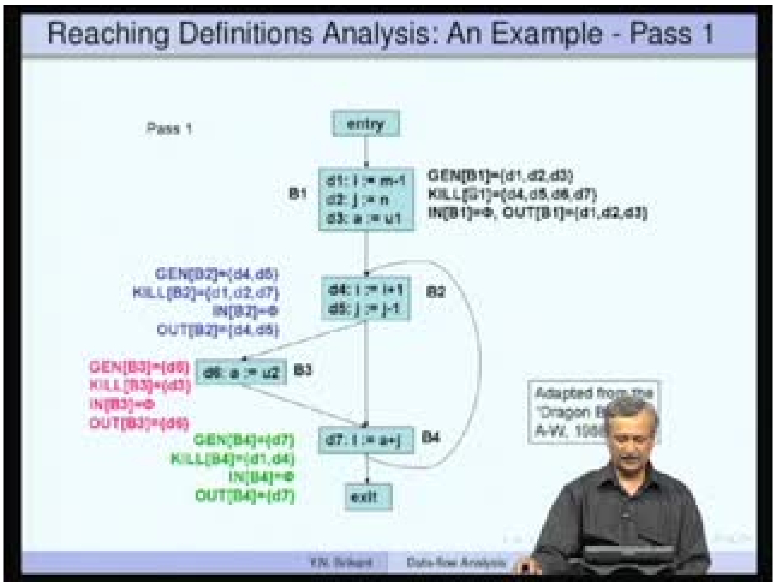
$$IN[B] = \phi, \textit{ for all } B \textit{ (initialization only)}$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
- $KILL[B]$ = union of the definitions in all the b the flow graph, that are killed by individual

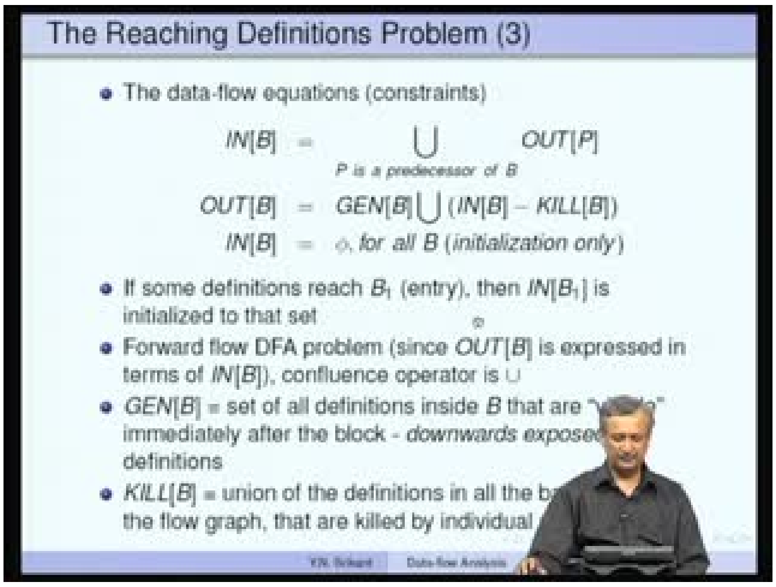
Y.N. Srikant Data-flow Analysis

Now, this is pass one. We have computed GEN, we have computed KILL, let us understand the meaning of these two equations. The outsets says, out of B says whatever is computed in the basic block that is GEN the locally generated set of definitions and then union it with whatever comes from the top as IN and whatever is killed by the basic block, that is removed minus KILL. So, if you look at this, suppose we take the basic block B2 and we are computing this outset, whatever is generated by the basic block B2 will definitely be visible at this point, these are definitely the definitions which reach this particular point, the exit point of basic block B2.

(Refer Slide Time: 49:22)



(Refer Slide Time: 49:29)



The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

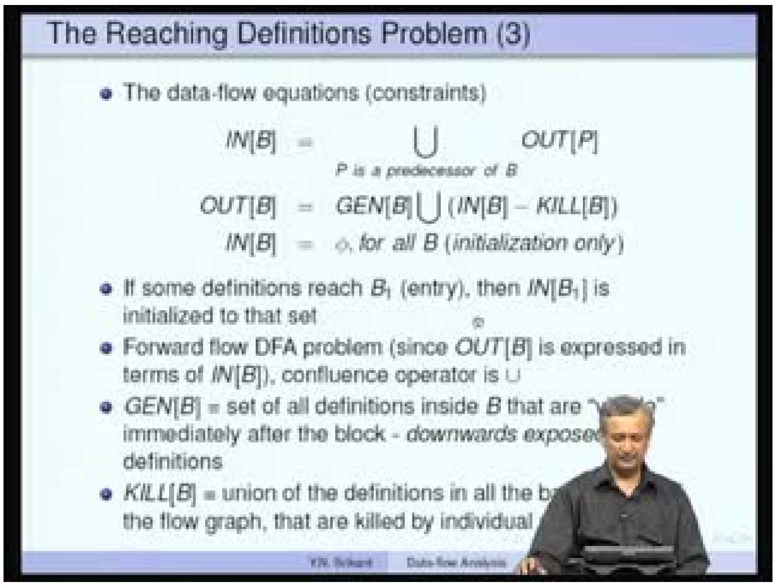
$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "[illegible]" immediately after the block - *downwards expose* definitions
- $KILL[B]$ = union of the definitions in all the b the flow graph, that are killed by individual

Y.N. Srikant Data-flow Analysis

(Refer Slide Time: 50:01)



The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

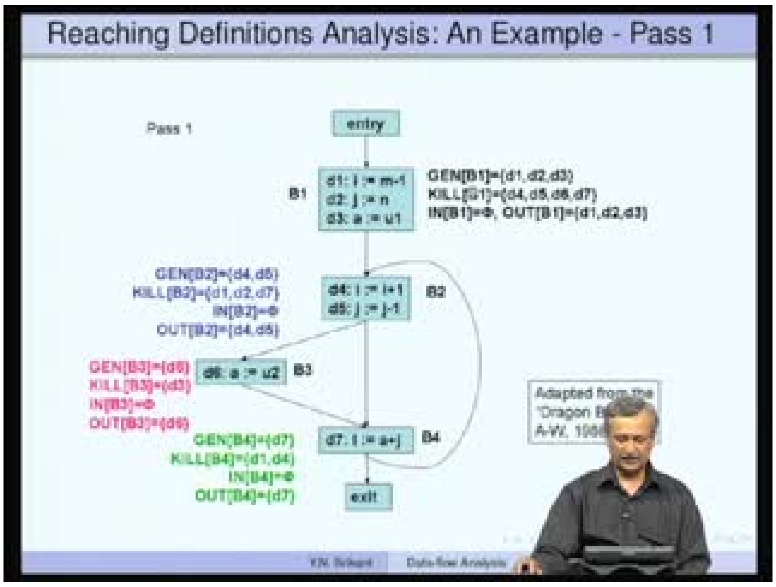
$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "[illegible]" immediately after the block - *downwards expose* definitions
- $KILL[B]$ = union of the definitions in all the b the flow graph, that are killed by individual

Y.N. Srikant Data-flow Analysis

Then, there will be some definitions which will be reaching this point, that is IN set and then these statements which are reaching here, some of them will be killed because of these statements here. So, we have to remove the killed part from that and that will give us the OUT set.

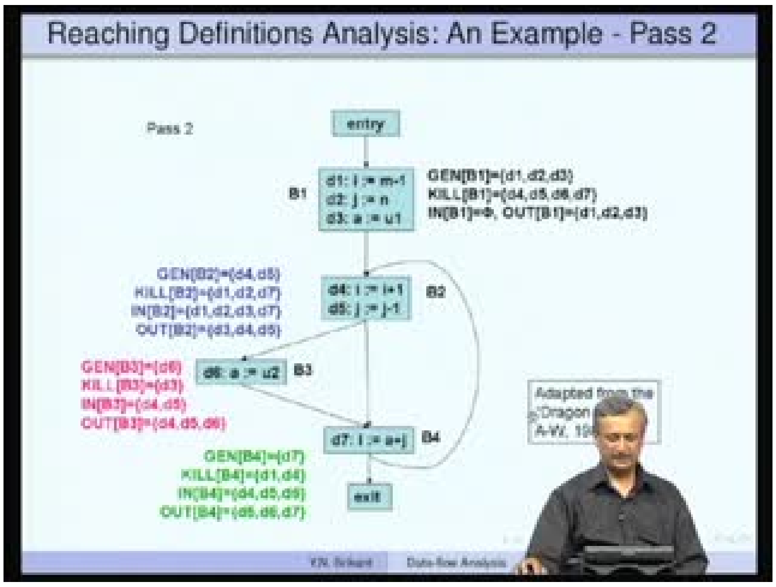
(Refer Slide Time: 50:08)



Reaching Definitions Analysis: An Example - Pass 1

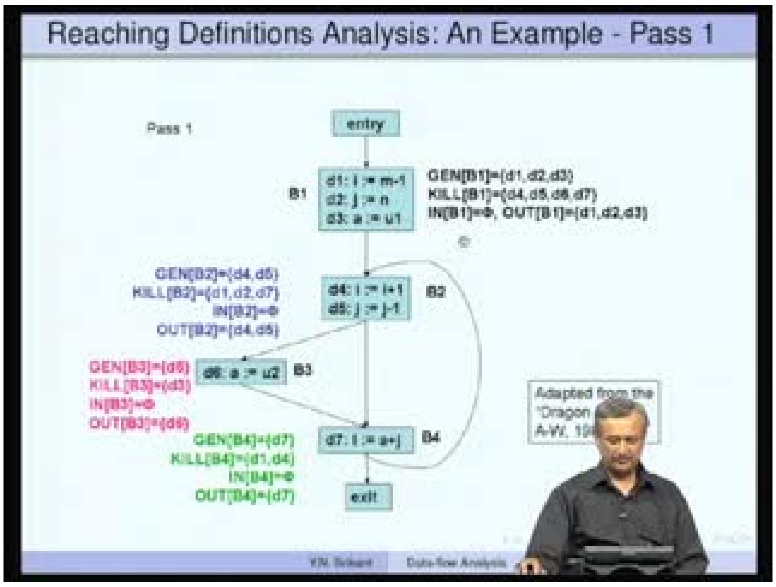
If we do that here, the initialization is IN of B2 equal to five for all basic blocks. Please see that IN of B1 is 5, B2 is 5, B3 is 5, and B4 is 5. It is very simple. OUT B1 is d1, d2, d3, this is five, so, nothing can be taken out of five, in the first pass, OUT of B2 will be just GEN of B2 and OUT of B3 is GEN of B3, OUT of B4 is GEN of B4.

(Refer Slide Time: 50:41)



Reaching Definitions Analysis: An Example - Pass 2

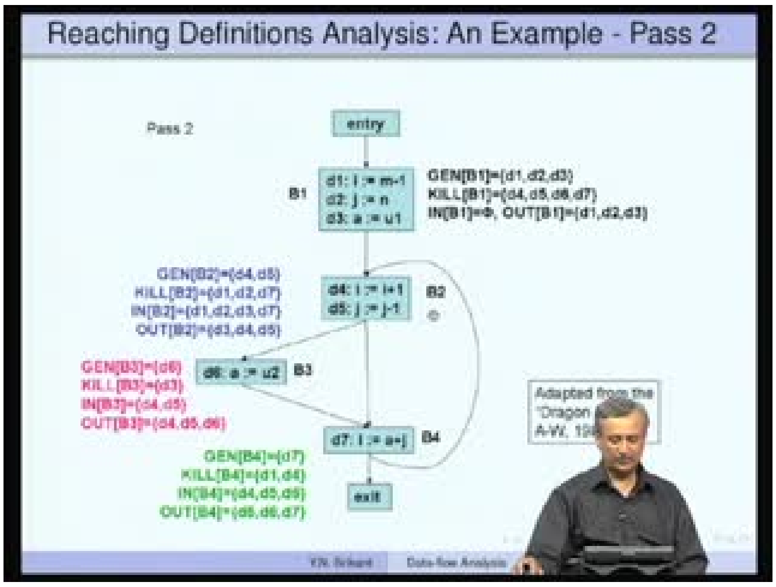
(Refer Slide Time: 50:50)



Reaching Definitions Analysis: An Example - Pass 1

Let us look at the second pass. In the first pass, the computation is over. Now, we need to compute the inset for the second pass using these values of outsets. So, for this particular thing, no special possibility, nothing gets modified. IN B1 remains 5, nothing comes in here, but for this, the outset of B1 is d1, d2, d3, outset of B4 is d7, these are the two non-empty sets which are coming along these two paths.

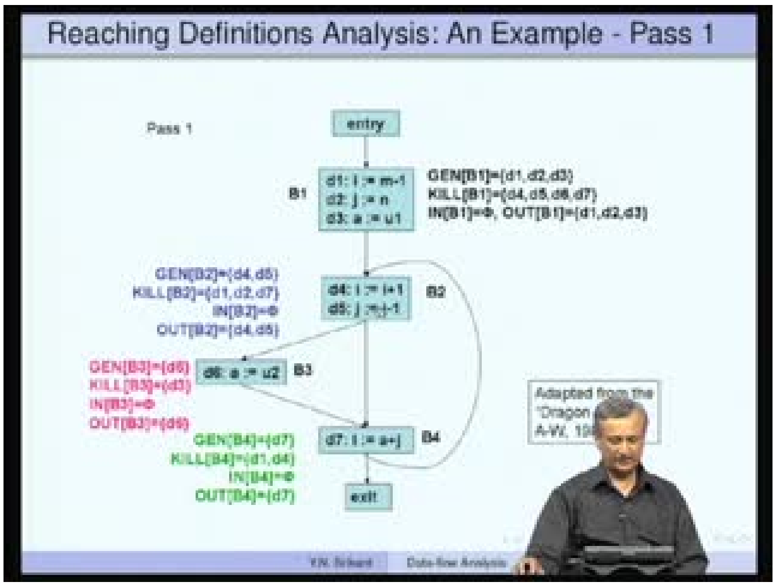
(Refer Slide Time: 51:54)



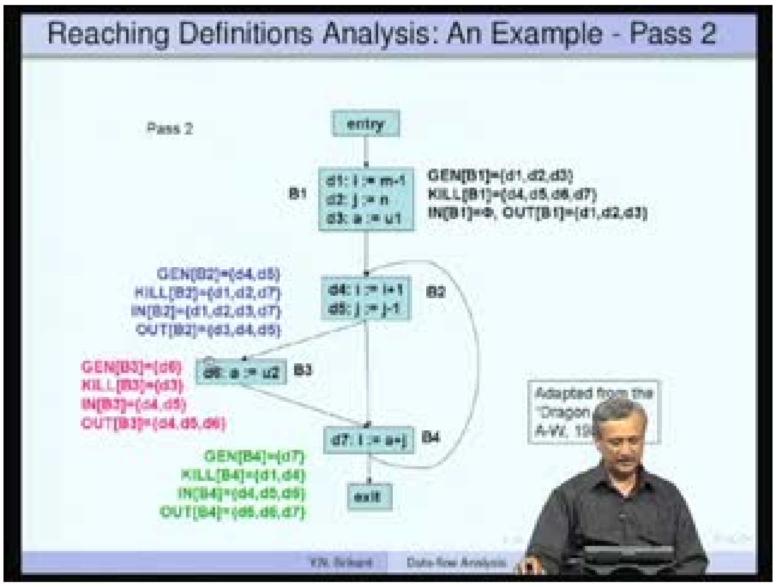
Reaching Definitions Analysis: An Example - Pass 2

So, we need to take the union of these two, because all the definitions which come here along this path also reach this point; all the definitions which come along this path also reach this particular point, so for OUT of B2, we are actually going to take this IN of B2, we are going to take OUT of B1 which is d1, d2, d3, we are going to take OUT of B4 that is d7 and put them inside IN of B2, d1 d2 d3 d7.

(Refer Slide Time: 52:01)



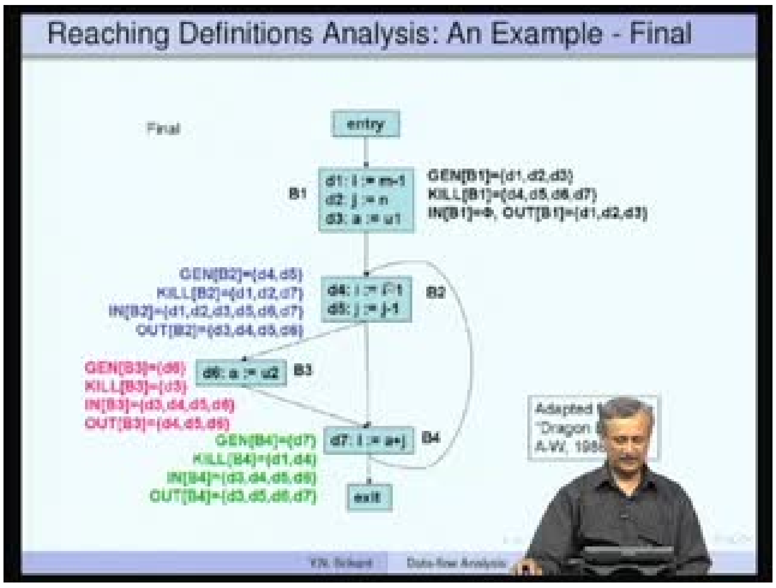
(Refer Slide Time: 52:12)



Similarly, if you look at the inset of this, there is only one arc here, so outset of B2 will become the inset of B3 in the next pass. That is here. Similarly, there are two arcs coming in here, so we are going to take the outset of B3 that is d6, and the outset of B2 that is d4, d5, take the union of these d4, d5, d6, will be the inset for B4.

So, this computation actually has to be carried out a number of times. Start with, we compute GEN and KILL once for all, we initialize IN B1 to 5, B2 of the 5 and so on. In the first iteration, we compute OUT of B1, OUT of B2 etcetera. In the second iteration, we compute again IN and OUT. Third iteration, we again compute IN and OUT based on the values which we computed before.

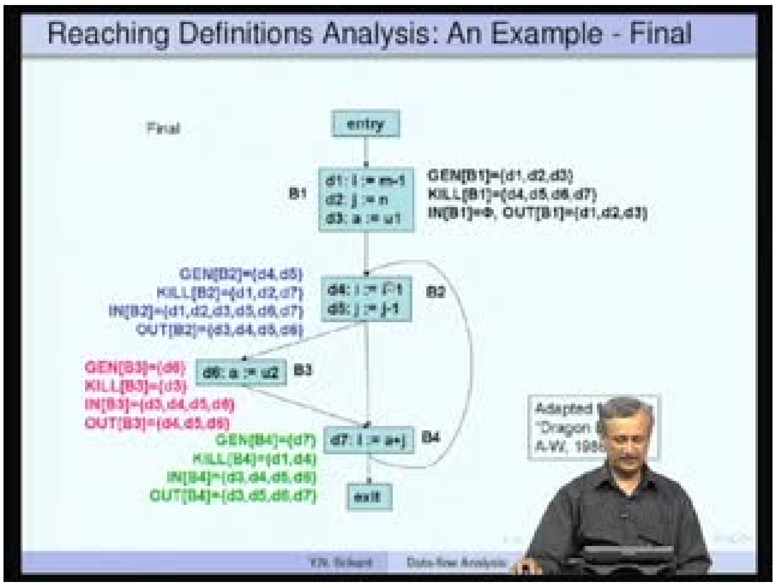
(Refer Slide Time: 53:15)



So, this set of iteration goes on until none of the values change. For example, once we reach this set of values, OUT B1 is d1, d2, d3, OUT B2 is d3, d4, d5, d6 etcetera. These are the various definitions which we can just verify once. At this point for example, OUT B2 says d4, d5, d6. d3 takes this path and it is a, so a is not modified here. So, it reaches this point. d4 and d5 of course reach this particular point. How does d6 reach? d6 is here, it goes out comes here and travels along the loop edge, enters this point and comes out again. So, that is how d4, d5, and d6 reach this particular point.

If you take this point, it is again d3, d5, d6, d7. d3 takes this part and enters this point. d6 takes this path and enters this particular block and goes out and then d5 comes here. What you should observe is d4 does not come here nor does d1, the reason is, there is a new definition of i here, this kills d4 and d1, so this d7 supersedes d4 and d1 and therefore, they can actually never come to this particular point. They can never reach this point whereas, this is different. There is only one definition, it reaches this point.

(Refer Slide Time: 55:01)



Reaching Definitions Analysis: An Example - Final

Once these values are computed, in the next pass, there is no change of values at all. This is the final iteration. This is the iterative algorithm for computing reaching definitions. This is what I was saying. You have initialization IN B equal to 5, OUT B equal to GEN B, then change is set to true and we go on until actually change remains true, so, first change, set change equal to false then compute the two equations, IN B equal to and OUT B equal to, if the old value of OUT is not the same as a new value of OUT, then change equal to true. So, we have to iterate once more.

Once OUT b equal to old OUT, change is false, so set change equal to true and then we come out. Until the blocks keep changing we need to iterate and once none of them change, we come out. This is the way the algorithm computes these values. GEN, KILL, IN, and OUT sets are all represented as bit vectors with one bit for each of the definitions in the flow graph. All these values of operations of union, minus, etcetera become bit vector operations. I will explain this again in the next lecture and this is the end of today’s lecture. Thank you.