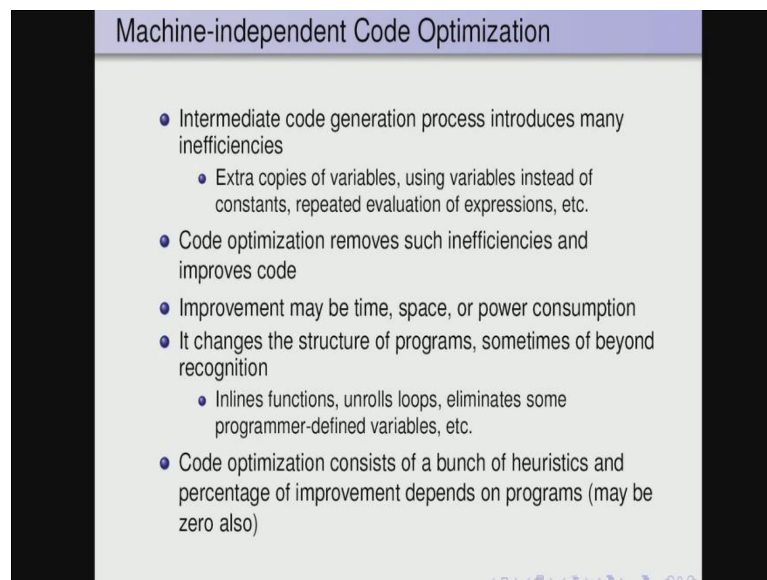
Principles of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 31
Introduction to machine independent optimizations Part -1

Welcome to part one of a lecture series on Machine Independent Optimization, so in these lectures we are going to discuss the various optimizations on intermediate code. So, we will begin with an introduction of what is code optimizations? Some illustrations of the different optimizations that are carried out by most compilers. And then we will have to consider a technique called data flow analysis, which is necessary to perform code optimizations, so we are going to look at those as well.

And then the fundamentals of control flow analysis are essential for everybody to know, because they help us in defining what exactly is a loop structure in control flow graphs. And then we will apply these principles to two of the important machine independent optimizations to understand how they are carried out and what the algorithms are; so finally we will discuss in detail the static single assignment form and the various optimizations on these static single assignment forms.

(Refer Slide Time 01:50)



So, when we consider machine independent code optimization, first of all we should understand why exactly this optimization becomes necessary. The most important reason

for this is the inefficiency which is introduced by the intermediate code generation process. So, as you would have learnt in the lectures on machine independent, rather the machine independent code generation or intermediate code generation as it is called, you would have observed that every time we want to make an assignment, we invariably end up generating a copy of the variable involved.

So, because the compiler takes an easy way out, it simply generates a new copy of a variable whenever necessary and it knows that the optimization phase actually is going to get rid of it. So, extra copies of variables, and then we store constants in variables and then use the variables instead of using the constants over and over again, because that is easier for us. Then there are many expressions which actually will be evaluated again and again, either because the programmer has not observed them which is actually not the major reason.

But, mostly because the compiler has introduced extra intermediate evaluations as will become clear very soon, these code optimization as I said removes such inefficiencies and improves code. So, whenever there are extra copies it gets rid of those, whenever there is repeated evaluation of expression it get rids of these things repeated evaluations and whenever it is possible to use a constant instead of a variable it does so. So, whenever the code optimization is applied, the optimization can be in time, space or power domain.

So far whatever I mentioned the removing extra copies etcetera, etcetera, they basically improve time and space, but a new dimension to the problem would be added if we want to save power, which is very important in embedded systems. So, reducing power consumption in code is not so trivial, we need models of the power consumption of the device and so on, and so forth, so they are not really topics for discussion in this lecture.

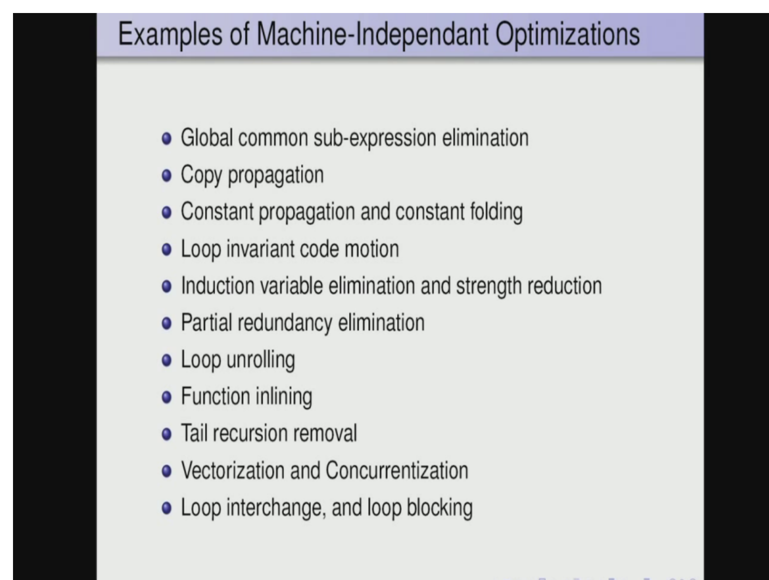
Code optimization algorithms often change the structure of the programs, sometimes beyond recognition as well for example, they may inline functions, so inlining of functions. That means, the function call will be replaced by the body of the function with appropriate replacements to the parameters, the temporary variables etcetera, etcetera. So, if this happens then the original call to the function gets deleted, it is replaced by the body of the function itself and then, it is possible to apply what is known as loop unrolling.

So, when a loop is unrolled obviously, say twice, thrice, four times etcetera, the number of iterations of the unrolled loop will be smaller, lesser than the original, so again the loop will not be iterating for the same number of times as the original. Then the induction variable elimination, it actually removes some of the programmer defined variables. So, for example, if there is a loop which is controlled by a variable i, it is possible that the induction variable elimination process removes the variable i and uses a different variable already present in the program for controlling the loop.

So, again this the such transformations make it very difficult for the debugger to be used along with optimized programs, so if we want to insert a breakpoint at a function call there is no function. If we want to look at the value of a variable which has already been eliminated, then it does not work out at all. So, therefore, the usually compilers you know stop most of the optimizations, if the user requests that the debugger be turned on.

So, when the debugger is on the program that we are debugging is usually unoptimized program, so code optimization really consists of a bunch of heuristics and the percentage of improvement depends on the programs, sometimes it may be zero as well. So, for example, if there is just a couple of there are a few assignment statements, and there is no way you can change any of that by any optimization, then the improvement would be zero. So, in such a case it does not mean that the optimization phase has been is generally ineffective, but it is just that for that program the improvement cannot be made.

(Refer Slide Time 08:15)



Here are some of the common machine independent optimizations that are used in compilers, so I am going to give you an example of each of these, there is what is known as global common sub-expression elimination. So, repeated evaluation of expressions is removed by this process, then there is the process of copy propagation, so if we have many copies of the same variable, then we can retain just one of them and eliminate them.

Constant propagation and constant folding it tries to promote the use of the constant, instead of the variable and it also tries to simply expressions involving only constants, and thereby promote the use of constants instead of variables. Loop invariant code motion it removes code which is inside a loop and is not going to change, because of the iterations of the loop, so such code is called loop invariant code and sometimes such code can be removed from the loop and it can be placed outside the loop.

Induction variable elimination and strength reduction this typically involves removing one or two variables and which are involved in iteration and then, replace these and try to control the loop using the rest of the variables in the program. Strength reduction tries to replace expensive operations such as multiplication and division by addition or shift and so on, and so forth. Partial redundancy elimination is a bit difficult to explain without an example, so I will defer the explanation to the time at which we discuss the example.

Loop unrolling is something I already mentioned we unroll the loop many times, function inlining also has been mention, so we replace the function call by it is body. Tail recursion removal implies that a recursive function call at the end of a function, a loop can be rather not a loop, a recursive call at the end of a function can be possibly replaced by a loop. Vectorization and concurrentization or transformations, which are useful to make the program work on vector computers or multiprocessors, and so on. So, loop interchange and loop blocking operations help in the process of vectorization and concurrentization.

(Refer Slide Time 11:18)

Bubble Sort Running Example

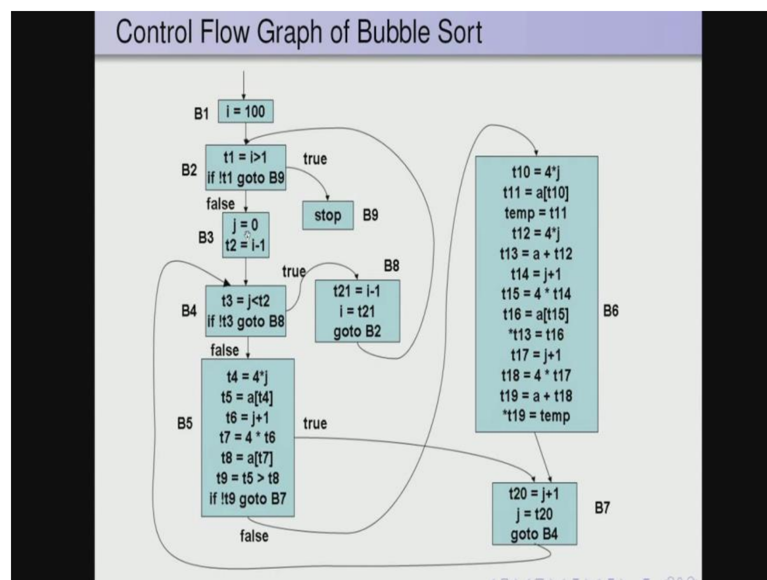
Bubble Sort

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

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

So, we are going to use this bubble sort program which is quite simple as a running example, so this is the standard bubble sort program, it sorts the array a with 100 elements. So, a like in c runs from 0 to 99 and we assume that there is no special jump out of it, if the array is already sorted, so we definitely go through all the iterations even if it is not exactly necessary. So, it is a standard program with i equal to 100 and then, there is a j loop, then there is a comparison, there is a swap and so on, so let us look at the intermediate code for this particular program.

(Refer Slide Time 12:07)



So, here is the condition for the i loop and then, here is the condition for the j loop, so if the i loop has to terminate it comes out here and if the j loop has to terminate it comes out here and then, goes back to increment the i. If the i loop does not terminate it goes into the j loop and then, the j loop actually works in this sort of a thing, so in all these. So, here we have actually the comparison and the rather the swap operation, this is just the comparison is right here.

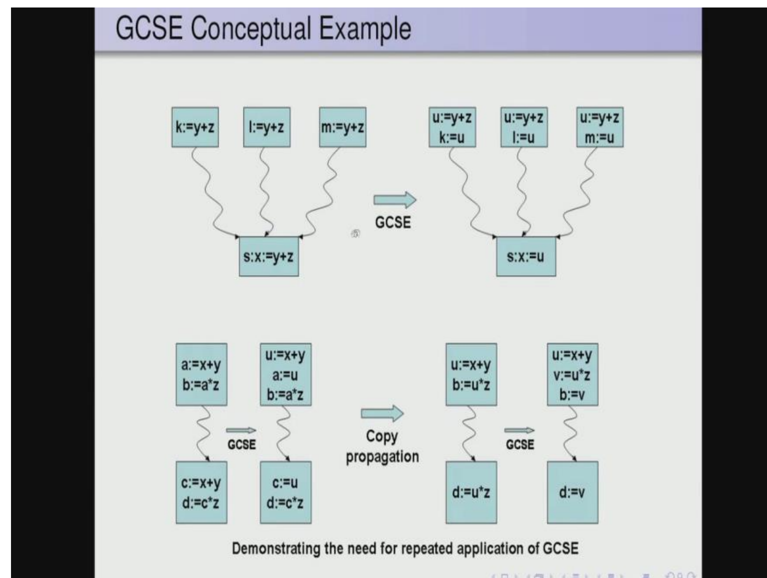
So, you can see that we have one element of the a, another element of a and then there is a comparison, so if we need to swap we come to this block, here is the swap operation, so even though there were only three statements in the swap block, the code generated is quite long, so let me explain why this happens. So, when we want to do a swap, the first thing is we want to do temp equal to a j, then a j plus 1 equal to a j and then a j equal to temp.

So, this is the sequence of operations, three operations which are required for the first assignment statement temp equal to a of j, so assuming that each integer requires 4 bytes the increment on the array which is in terms of bytes is going to be by 4, so we need to multiply the j index by 4, then we get the element from the array and we assign it to temp. So, this is the sequence of three operations for just that temp equal to a of j, then we have the other one a j plus one equal to a j, so here again we compute 4 star j.

So, you can now observe the repetition of the computation 4 star j here and 4 star j here as well, then we take the address of the j plus j'th element, rather j plus first element, and then this is a j'th element. So, t 14 is J plus 1, t 15 is 4 star t 14 and then t 16 takes the element of a, so now star of t 13 equal to t 16, this is the assignment of a j plus 1 equal to a j. So, then the last one is the a j equal to temp operation here, so if you observe all this block and this block together, we have a 4 star j here and then, we have 4 star j here and here as well.

And then, we have j plus 1 computed here and then, we have j plus 1 computed here ((Refer Time: 15:36)) and here and here as well, so in other words we have i minus 1 here and i minus 1 here; so there are many places where the same computation is being repeated. So, in such a case we will be able to perform many of the optimizations, so we are going to look at those.

(Refer Slide Time 16:00)



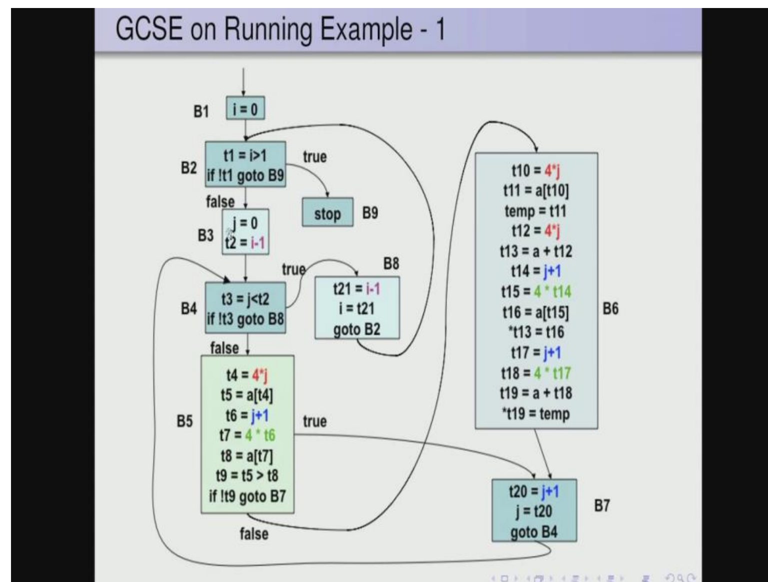
So, let me explain the first optimization global common sub-expression elimination, so this says consider a situation where there is a computation of some expression y plus z in this block, and every path preceding that block has a computation of y plus z. Of course, it is obvious that y and z should not changed along these paths, so this y plus z, this y plus z and this y plus z will have identical values and there is no need to compute this y plus z all over again. You might as well say put that y plus z computation in a temporary, the same temporary along all paths u, u, u and then, just say x equal to u.

So, in this fashion whenever we it does not matter which path we take, we are going to compute the y plus z expression only once, it will not be computed twice as in this particular case, it will be computed only once. So, this is the other part of the example which shows the need for repeated application of GCSE, so here we have x plus y and x plus y, so we get rid of the repeated computation by introducing the temporary u and assigning c equal to u here.

But, once we perform what is known as a copy operation, copy propagation, so for example, observe here that this is a equal to u and it is really a copy of u, so we can replace this a by this u directly and get rid of this particular assignments or copy operation. The same is true here, could actually replace this c in d equal to c star z by u and get rid of this statement, if we do that we have u equal x plus y and b equal to u star z and here, we have d equal to u star z.

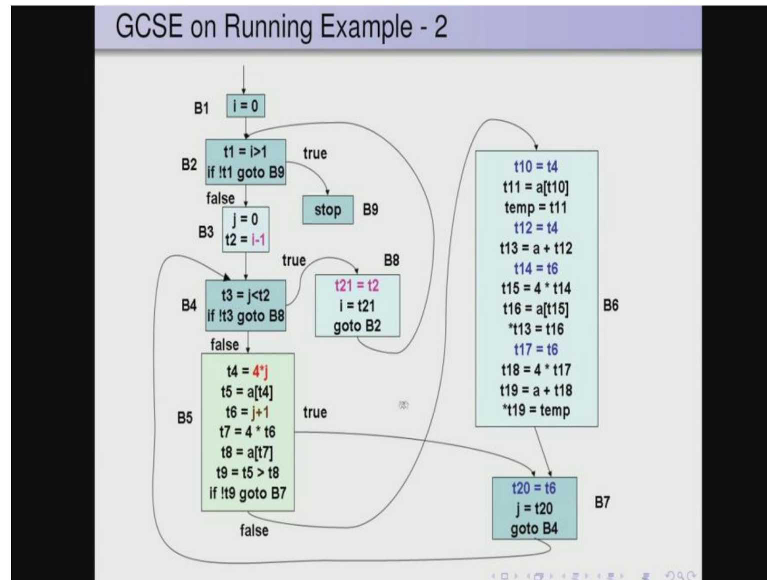
So, because of the copy propagation, we have discovered another instance of common sub expression here, one here and one here two instances and we can apply GCSE again to get rid of this repeated computation, so we have v equal to u star z and d equal to v. So, the moral of this example is that you require many applications of common sub-expression elimination and copy propagation, in order to eliminate most of the common sub expressions which would otherwise be hidden. So, in general optimizers supply, the optimizations many times in an iterative mode until not much improvement is possible or a definite number of times have actually taken place.

(Refer Slide Time 19:03)



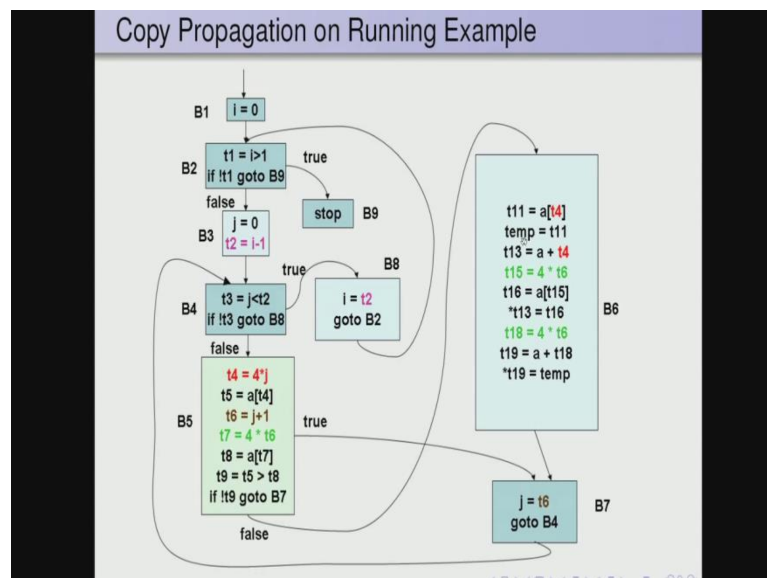
So, let us see how it works on our running example, so I have marked in color the various common sub expressions I minus 1, then here is 4 star j, then we have j plus 1 in many places, so if we eliminate these we get this program.

(Refer Slide Time 19:25)



So, we have t 2 equal to i minus 1, so here instead of t 21 equal to i minus 1, we have t 21 equal to this t 2, we have t 4 equal to 4 star j, so wherever there was 4 star j we replaced it by t 4 see and then, wherever we had j plus 1, we replaced it by t 6, so here is t 6 and here is t 6 and so on. Now, this has given rise to many copies, so here is t 21 equal to t 2 and i equal to t 21, obviously we can make this i equal to t 2 and so on.

(Refer Slide Time 20:08)

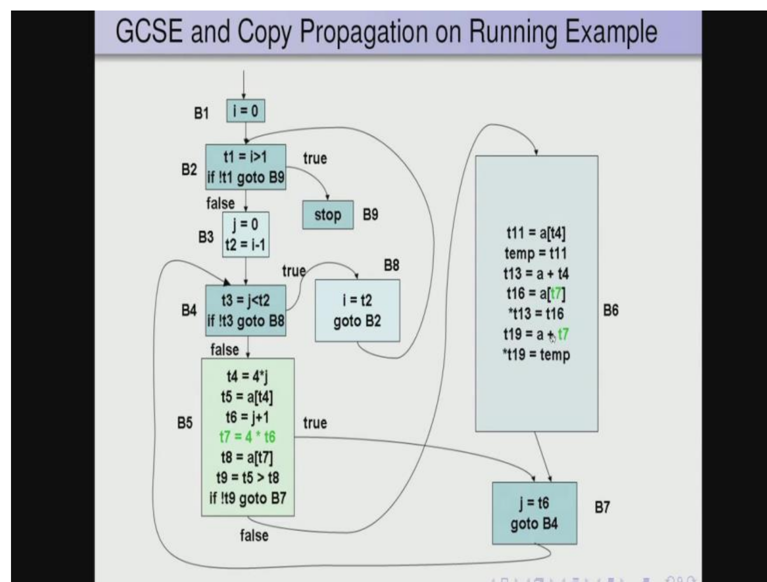


So, we do that here is i equal to t 2, then in this case we, ((Refer Time: 20:14)) in the previous case for example, t 11 equal to a of t 10 could have been made t 11 equal to a of

t 4. So, similarly this t 12 equal to t 4 could have been eliminated and we could have made this t 13 equal to a plus t 4. So, if we do such optimizations the copy propagation, we get this code, so we have a t 4, then a plus t 4 and similarly, this j equal to t 6, so the copy propagation example or rather the optimization when applied removes many of these copies.

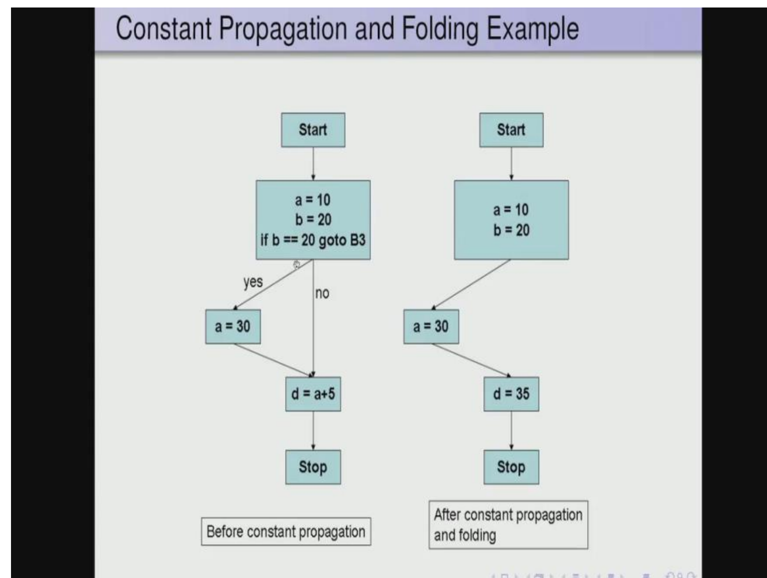
But then, there is a an opportunity for further optimization now, see for example, the expression four star t 6, now becomes a common sub-expression, so here is 4 star t 6 and again 4 star t 6, so we can eliminate that as well and then, perform another round of copy propagation. So, here if we had set t 5 equal to t 7, the t 15 equal to t 7, then we could have replaced this t 15 by t 7 and similarly, this t 18 could have been replaced by t 7, so that is what is done here.

(Refer Slide Time 21:40)



So, after the round of GCSE and copy propagation we get this short piece of code, in which there are many instructions which have been eliminated. The point is even in such a simple program such as bubble sort, there seems to be an opportunity to perform GCSE and copy propagation several times, so if this is so in a simple program, there is certainly there are many chances to perform these optimizations in larger programs.

(Refer Slide Time 22:17)



So, let us consider an example to understand how constant propagation and folding take place, so there is a program we have a equal to 10, b equal to 20, if b is 20 go to b 3 you know yes, no, so if it is yes then we assign a equal to 30, then d equal to a plus 5 and stop. So, it is very clear that since b is a constant 20, this evaluation of 20 equal to 20 can be carried out by the compiler itself, so that is basically propagating this constant value of b to this particular use and then, evaluating this equality amounts to constant folding.

So, if this becomes true and therefore, the code for the rather the edge for the no part can be removed from the control flow graph, so we will have only one edge here, so this becomes a equal to 30. And here we have only these two instructions, because this has already been evaluated, here it is very clear that the value of a can only be 30, because this edge does not exist anymore. So, we can evaluate the constant 30 plus 5 as 35, so this is also another example for constant folding, so the program because of constant propagation folding has become quite simple.

(Refer Slide Time 23:50)

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

So, let us move on to the next example of loop invariant code motion, so here is a very simple loop, so we look at the two statements in red, one says t 3 equal to address of a, the other says t 4 equal to t 3 minus 4. So, consider just one statement t 3 equal to address of a, address of a is a constant it is nothing but the offset of the array a inside the activation record. So, this would be this a statement which does not change it is value during the iterations of this particular loop, so it is obvious that this statement can be moved outside the loop like this.

But, then the next statement t 4 equal to t 3 minus 4 depends only on this statement which is loop invariant, so therefore in turn this statement also becomes loop invariant, it does not change its value during the iterations of the loop and even that can be moved outside the loop. But, remember you must move these statements in the same order as they are present in this loop they cannot be swapped, otherwise the program might be incorrect.

So, this is what is known as loop invariant code motion, in this particular example there is only one basic block, one thread of control, so moving code outside was a very simple operation. But, as you will see in the later parts of the lecture, there are many conditions that need to be satisfied in order to move the loop invariant code to outside the loop.

(Refer Slide Time 25:37)

Strength Reduction

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

Before strength reduction for t5

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

After strength reduction for t5 and copy propagation

The next example is strength reduction, so here is a multiplication 4 star i, suppose the processor is a very simple processor, say in the embedded system domain and it does not even support a multiplication of integers forget floating point. In such a case, usually the software implements multiplication if it is essential by a subroutine, which is very expensive to be called. So, in such cases we may want to replace this 4 star i by repeated addition process.

So, 4 star i t 5 equal to 4 star i as i increments will take the value 4, 8, 12 etcetera, so we might as well add 4 to it and get the new value of t 5, so that is precisely what we intend to do here, but we replaced t 5 by a new variable called t 7. So, t 7 and then, t 6 equal to t four plus t 7 and we have t 7 equal to t 7 plus 4, which is placed immediately after i equal to i plus 1. So, that we do not forget to compute the value of t 7, which is required for this iteration, so now t 7 increments in force and it is supplied to t 6, exactly the way t 5 was being supplied to this particular assignment. So, the semantics of the program does not change and we have actually, there are two steps in this replacement, we would have first set t 5 equal to t 7 and then, done a copy propagation to remove t 5 in this example and make it t 7, so that is a two step process.

(Refer Slide Time 27:50)

Induction Variable Elimination

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

Before induction variable elimination (i)

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

After eliminating i and replacing it with t7

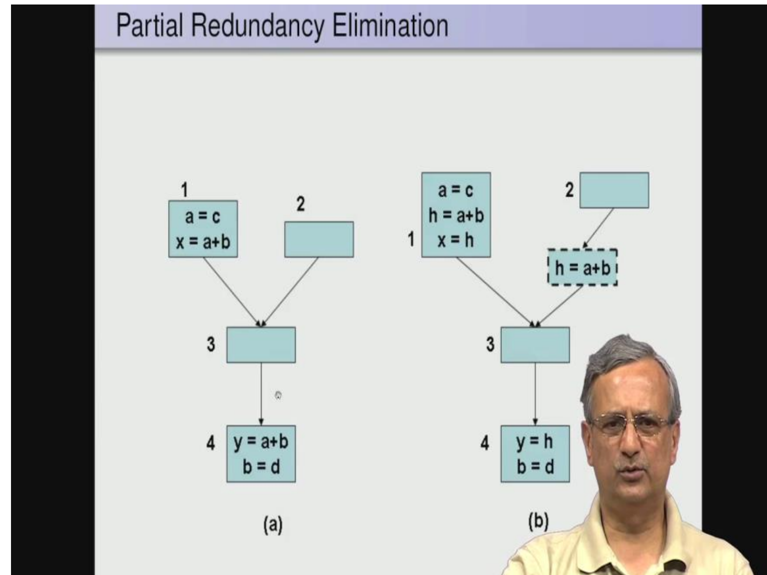
We will move on to the next optimization called as the induction variable elimination, so usually the induction variables are variables, which are used to control a loop. So, even here for example, we say check i greater than 100 and then, increment i if it is not, so this is the loop control and i is used for that purpose. So, suppose you look at the program a little more closely, you find that there is another variable t 7, which is also being incremented in tandem with i. So, this is the variable we introduced in the previous example with lower strength reduction process.

So, as we increment i t 7 also monotonically increases by and the increment is 4, so if we actually want to get rid of a variable, it is possible to get rid of i and then, use t 7 in it is place with the appropriate change in the operands of the expression. So, we have used i here and we have actually computed i here, so we replace this by t 7 greater than 400, because as i increases from one onwards 1, 2, 3, 4 etcetera, because the increment is by 1, t 7 starts with the value 4 and increments by 4, so 8, 12 and so on, and so forth.

So, whenever we compare i greater than 100, we need to compare t 7 with 400 and now once we replace that the variable i with t 7 and the operands are changed appropriately, there is no need to retain this variable and it is associated statements, so we remove it and now the program becomes smaller. So, this is what is known as induction variable elimination, so we could remove i and replace it with t 7 of course, if you observe carefully it is also possible to remove t 7 itself and replace it with appropriate values of i.

But, that would defeat the purpose of the strength reduction that we perform, so we will be undoing strength reduction if we replace t 7 with a usage of i, so that is not intended.

(Refer Slide Time 31:03)



Now, we move on to partial redundancy elimination, global common sub-expression elimination GCSE as it is called is actually, a can be termed as a total redundancy elimination transformation. So, if you recall the example, we must have in order to remove this a plus b, we must have a computation of a plus b along every path that reaches this basic block. So, here is a path and here is a computation of a plus b that reaches this path, this block, but unfortunately along this path there is no computation of a plus b that reaches the basic block number 4.

So, we cannot apply the global common sub-expression elimination process here, because the expression a plus b is not available along this path, it is available only along this path. So, this particular example a plus b is said to be partially redundant, it is not totally redundant, it is actually available along this path, but it is not available along this path, in such a case sometimes it is possibly cheaper to insert a computation of a plus b in this edge.

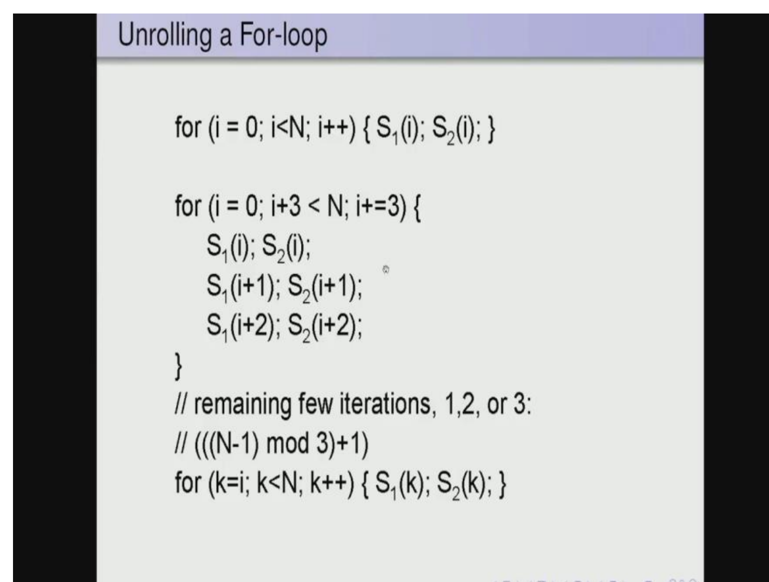
So, we have dissected this edge introduced a new block and put a computation h equal to a plus b here, and for this x equal to a plus b we have replaced it with h equal to a plus b and x equal to h, so the semantics of the program remains the same. So, now, consider the expression a plus b, so a plus b is now available along this path, at the entry of this

block, a plus b is available along this path at the entry of this particular block. So, we can perform ordinary common sub-expression elimination, and instead of this y equal to a plus b we can replace it by y equal to h.

So, this is the essence of partial redundancy elimination, there are many difficulties here, first of all we need to make sure that this a plus b is partially redundant, and there are a couple of conditions to be checked there. And then, we need to find the arc which is the best for the introduction of the extra computation that we have shown here. So, you know if we introduce it here it is worse, we would have computed it twice rand this is the computation we are going to remove, but you have not gained anything.

So, we are introducing it here which is the best, but as you easily can imagine there is this program may actually grow in this direction and there may be many way paths. So, which path of this program should be taken, in which arc of the program should be cut in order to introduce the computation of a plus b that is a difficult question. So, that is another there are a couple of conditions that we need to check, in order to make sure that we introduce a plus b in the best possible place.

(Refer Slide Time 34:58)



Then unrolling a for loop, so here is the for loop for i equal to 0 i less than N i plus plus, then some statement S 1 which for the indication here is that S 1 for the value of i, so this is the instance which is relevant for the iteration i and this is the instance of S 2 which is relevant for the iteration of i. So, the reason why we mention it like this is, if the code

has used the value of i, then when we perform loop unrolling, we may have to replace it with the appropriate value of i plus 1 or i plus 2 etcetera, etcetera.

So, what we have shown here is an instance of S 1 with the value of i, so when we write S 1 of i plus 1, we imply that it is an instance of S 1 with the i being replace by i plus 1 and here the i has been replaced by i plus 2. So, we can unroll this loop, so S 1 of i and S 2 of i correspond to loop iteration i S 1 of i plus 1 and S 2 of i plus 1, they correspond to the iteration i plus 1. So, S 1 of i plus 2 and S 2 of i plus 2, they correspond to the iteration number i plus 2, so there are three instances of the body for loop here.

So, it is very obvious that this loop must operate only 1 3rd the number of times that the original loop operates, so if the original loop operate at some N number of times, then we need to perform the once we perform the unrolling, we make sure that the check i less than N is change to i plus 3 less than N. And we also make sure that the increment by one is now increased to three and there are three instances of the body of the loop, but it is also possible that the number of iterations is not exactly divisible by 3.

So, in such a case there would be a few iterations which remain, so for that there is a small sequential loop which would operate once or twice or maximum of three times and so for example, here if we have i equal to if this is suppose to run only three times, so i less than 3, then this you cannot even unroll it. So, we will have to execute it in a sequential mode, so this test would fail 0 plus 3 less than 3 would fail, so this part will not even be executed, so we will have to execute it here.

The same is true if the number is 4 or 5, so in that case we will be left with one or two iterations which need to be executed. So, this is the condition k less than N k plus plus and we start with the iteration number i with which we end at this particular loop. So, why should we do loop unrolling, there are many reasons for this, the first one is in instruction scheduling we actually need a large basic block. So, that the parallelism in the basic block can be used by the instruction scheduler to it is advantage.

So, if we have a very small basic block with 5 or 10 instructions, instruction schedulers do not work very well, so they work very well if there are at least 50 or 100 instructions. So, in such a case unrolling a large loop by 10 or 20 times yields large basic blocks and therefore, instructions scheduling becomes a very efficient process. The second one is the decreasing the number of iterations the overheads of the jump they actually reduce,

so as you realize every jump instruction kind of creates a problem for the pipelines, so we must reduce the number of jump instructions as far as possible. So, if we reduce the number of iterations of the loop, the number of jump instructions will automatically execute, it will automatically come down, so that is another reason why we may want to unroll a loop.

(Refer Slide Time 40:11)

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;
```

So, here are two examples of unrolling a while loop and unrolling a repeat until loop, so while C S 1 S 2 can be unrolled as while C, then S 1 S 2 and once we have executed we need to check whether a condition holds or not. So, if not C break again S 1 S 2 if not C break again S 1 S 2, so this is the unrolling pattern for the while loop the repeat until is very similar. So, we do repeat S 1 S 2 if C then break, because we need to iterate until C is true, then S 1 S 2, if C then break S 1 S 2 until C.

So, again the number of times we have unrolled twice here, so there are 3 instances of the loop body the number of times the loop would iterate will be 1 3rd approximately to compare to the original ((Refer Time 41:12)). The next optimization is function inlining, so take a simple function definition int find greater which tries to find the largest number in the array a. So, here is a parameter a size ten and then the number n, so here is a loop which goes on from 0 to 9 and if this particular array contains an element which is greater than n, then it returns the index of that element, otherwise it increments the loop and it not the greatest of the array, but an element greater than this element n.

And then, it iterates until it finds it, otherwise the loop terminates and comes out, so if there is a call x equal to find greater y comma 250 by inlining these particular function we need to introduce new variables for the local variables of this particular function. So, we let us call them new i and new of a 10, so this is the parameter, so the new a now is assigned the value y which is the formal parameter, because this is a call by value we have to make a copy.

And in the loop we use the instead of i which is supposed to be new i here, we just use it with the same conditions new i equal to 0 new i less than 10 new i plus plus; and we compare a new a of new i instead of comparing a of i, because now this is a copy greater than 250, then x equal to new i and go to exit. So, return here actually is replaced by x equal to new i, where x is the variable on this left hand side, this accumulates the return value and then, we could have add up break as well, so we exit the loop.

So, this is the inlining of functions what do we gain by inlining functions, so when we inline a function the most important thing is there is no subroutine call instruction necessary. Subroutine jump instructions or subroutine call instructions are very expensive, because they imply creation of a an activation record, then pushing parameters into that activation record and then, getting the result from the activation record and finally, destroying the activation record.

And whenever we want to accesses a variable on the activation record, there is a bit of cost attach to it, so if you inline the function creation of the activation record etcetera, destruction, pushing parameters, they are all not there at all. So, it is much cheaper and the code runs in a much faster way compared to the un inlined call, so inlining introduces efficiency into the program by eliminating a number of subroutine calls.

(Refer Slide Time 44:52)

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; continue;}
   }
}
```

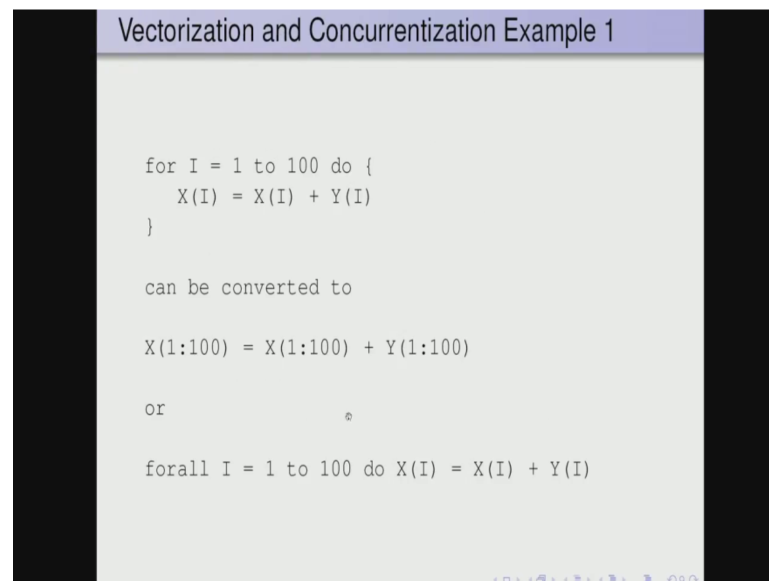
The next optimization is the tail recursion removal, so let us understand what exactly is tail recursion, here is a simple function called sum, which takes an integer array as one of the parameters a number n as the second parameter and a pointer to an integer x as the third parameter. So, this says if n equal to 0, then add the 0th element of a star x, so star x equal to star x plus a of 0, so the sum is being accumulated in star x, otherwise add the n'th element and call sum recursively with n reduced, but x remains the same.

So, this recursive call is the last statement in this particular function, so in other words it is at the tail of the function that is why this is called as a tail recursive call. So, in such cases with appropriate checks it is possible to remove this tail recursion and replace it with a while loop. So, the same function declaration remains the same and instead of recursion we have a while true loop which runs forever, but there is a break inside which make sure that we get out of the loop.

If n equal to 0 then star x plus a of 0 which remains as it is and then, returning from the function we have a break which goes out and then of the loop and then, terminates the function. Otherwise, if n is not 0 if we had a recursive call here we have the same sum and then, we reduce the value on n by 1 and then, continue with the loop, so this loop executes as many times as the number of values as the variable n, so once n reaches 0 it breaks the loop.

So, we have successfully replaced this tail recursion by a while loop and the number of times this while loop operates is the same as the number of times this recursion happens, whereas this while loop is very efficient compared to this particular recursive call. So, recursion as I said is much more expensive, because we need to set up an activation record, push parameters, extract the result and finally, destroy the activation record, so all that gets eliminated in this process.

(Refer Slide Time 47:53)

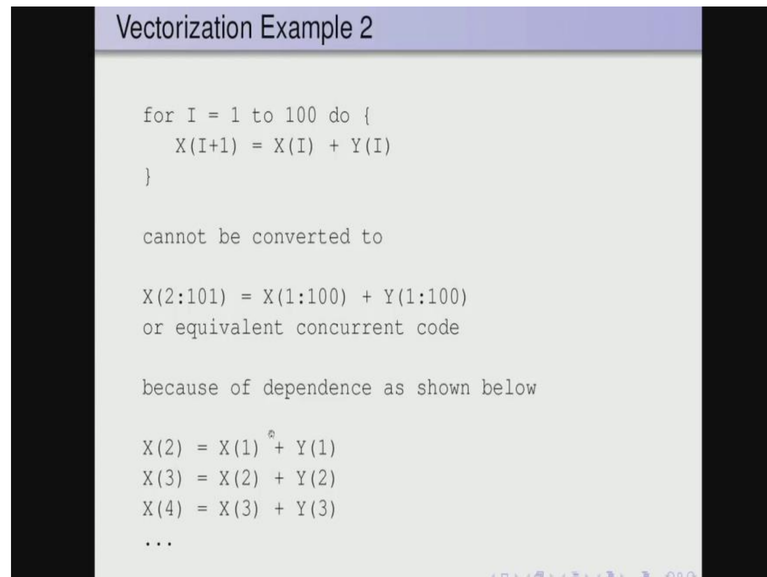


We move on to vectorization and concurrentization, so take a simple loop X I equal to X I plus Y I, where the loop iterates from 1 to 100, so because we are only extracting the old values of X and Y. And then, summing it up and putting it into the array again, we can actually execute all these statements with the help of a vector processor. So, here is a vector 1 to 100 which whose value is extracted first, another vector 1 to 100 whose value is extracted next, they are added the corresponding elements are added and assign to the vector X again.

So, this is very easy to see, because there are no usages of the value, which is computed within the loop, I will give you an example where such vectorization is not possible. If the processor were to be multi core processor, we could actually start a thread for each one of the iterations of this loop and each thread would do this summation. So, that is indicated by for I equal to 1 to 100 X I equal to X I plus Y I, so we start 100 threads each thread doing X I equal to X I plus Y I for a particular value of I. So, that is the way it

would operate on a multi core processor and the these are called vectorization and concurrentization.

(Refer Slide Time 49:29)



So, here if you look at this example the statement says X I plus 1 equal to X I plus Y I, look at the expanded version X 2 equal to X 1 plus Y 1 X 3 equal to X 2 the Y 2 X 4 equal to X 3 plus Y 3 and so on. So, X 2 is computed here used in the next iteration, X 3 is computed here and used in the next iteration and so on, and so forth. So, because of this dependence of the value on the previous iteration the code cannot be either vectorized or concurrentized. So, emitting such code even though syntactically it looks correct would be wrong, we are not trying to reuse the rather use the computed value of X, but we are just using the old values of X here, so this is incorrect.

(Refer Slide Time 50:23)

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)
   }
}
```

In certain cases it is possible that the outer loop cannot be converted to a parallel loop and the inner loop can be converted to a parallel loop, the problem is inner loop iterates certain number of times, which may be too small for giving sufficient work to if you parallelize the inner loop, then it the work involved just one statement may be too small for each thread. So, in such cases sometimes we are allowed to do what is known as a loop interchange, so the J loop goes outside and I loop comes inside.

If we are allowed to do this then the J loop can be operated in parallel and the I loop with its assignment statement operates in a sequential node. So, for each thread which is created for J, there is a whole loop which executes which is sufficient work for the thread. So, the code generated would be something like this, so this is the loop interchange for parallelizability which is beneficial in certain cases.

(Refer Slide Time 51:37)

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];
}
```

When we have a fixed amount of cache and we know the cache size, it is also possible to actually do what is known as loop blocking assuming that the cache size is 64, the block size is 64, we actually break the I loop into iterations with an increment of 64. The same is do done for the J loop we break it with an increment of 64 and inside we actually iterate from one to 64. If we do this every time we finish one iteration of this, we are going to get a block of 64 into the cache memory and then, these two loops actually work only on the elements of the cache. So, if this is the case then the program becomes much faster, so loop blocking also is very beneficial when we have cache memory.

(Refer Slide Time 52:42)

Data-flow analysis

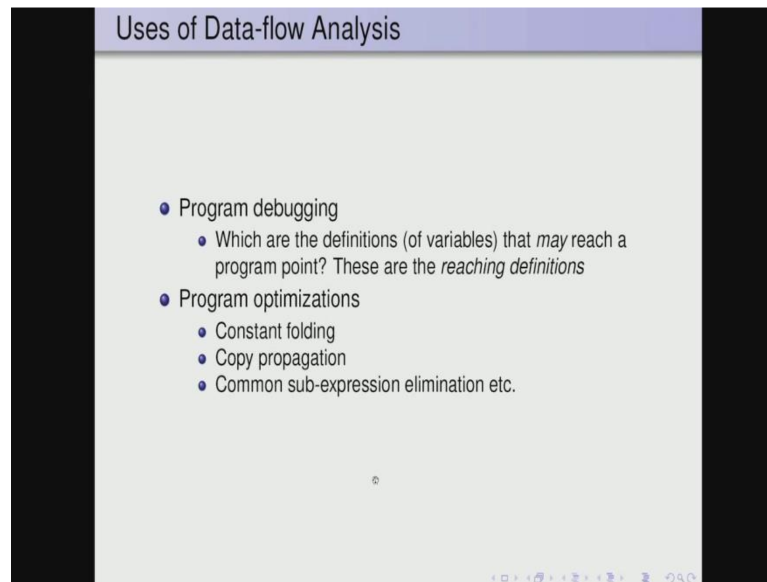
- These are techniques that derive information about the flow of data along program execution paths
- An *execution path* (or *path*) from point p_1 to point p_n is a sequence of points $p_1, p_2, ..., p_n$ such that for each $i = 1, 2, ..., n-1$, either
  1. p_i is the point immediately preceding a 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 program states that can occur at a point in the program with a finite set of facts
- No analysis is necessarily a perfect representation of the state

Now, we move on to the fundamentals of data flow analysis, so data flow analysis basically is a bunch of technique that derive information about the flow of data along the program execution paths. So, essentially we look at the execution path in a program from point p 1 to p n, so we consider the points just before the statement and just after the statement. So, p if you look at p 2, p 1 is a point just before the statement, and then p 3 would be a point just after the statement.

So, this is how p i is a point immediately preceding a statement and p i plus 1 is the point immediately following the statement and of course, we could reach the end of the block, so in such a case p i is the end of some block and p i plus 1 is the beginning of the successor block. So, we are essentially looking at the paths in the control flow graph, so and in general there is an infinite number of paths through a program and there is no on the length of a path either.

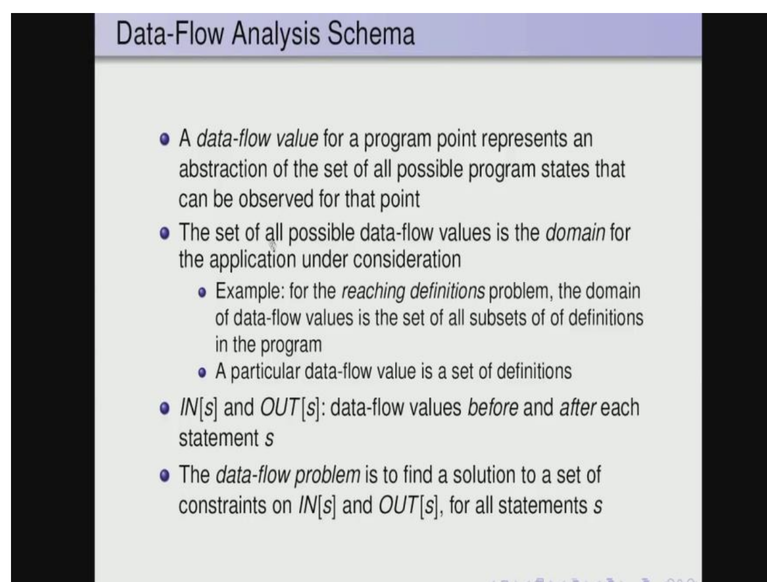
So, this is true, because we if we have a loop then again we do not know the number of times the it iterates, so there could be very large number of paths. Basically data flow analysis or program analysis summaries the program you know states that can occur at a point with a finite set of facts. So, even though there are a huge number of paths to a particular program, we want the summary of the information coming along all these paths and put that into you know that particular state. So, the analysis is certainly not a perfect representation, because we are summarizing the effect of many paths into that particular point.

(Refer Slide Time 54:43)



So, data flow analysis aims to produce such summaries of information, so what these are will become clear later, but the applications of such program analysis techniques are in program debugging. Where we want to ask questions such as which are the definitions that reach a point and these are the all useful in optimization such as constant folding, copy propagation, CSE and so on.

(Refer Slide Time 55:12)



So, a data flow value for a program represents an abstraction of the set of all possible program states that can be observed for that point. So, for reaching definitions this could

be the set of all definitions that reach a point, you know the set of all data flow values is the domain of that application. So, we are going to look at the reaching definitions problem, where the domain is the set of all subsets of definitions of the program for available expressions we would consider expressions and a set of all subsets of expressions as the domain and so on and so forth. A particular data flow value is a set of definitions, in general all the data flow problems involve equations with in and out and where s is a statement.

So, we want to find the find a solution to the set of constraints that are imposed on in and out and then say with given these constraints these are the values of in and out. We want to do this for all the statements and that would be a considered as a solution to the data flow analysis problem. So, we will stop here and then continue with this lecture in the next part.

Thank you.