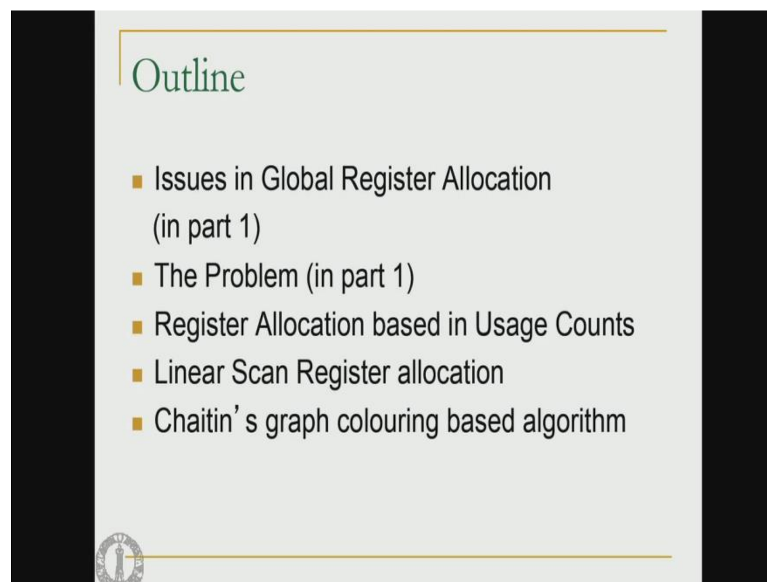
**Principles of Compiler Design**
**Prof. Y. N. Srikant**
**Department of Computer Science and Automation**
**Indian Institute of Science, Bangalore**

**Lecture - 29**
**Global Register Allocation Part – 2**

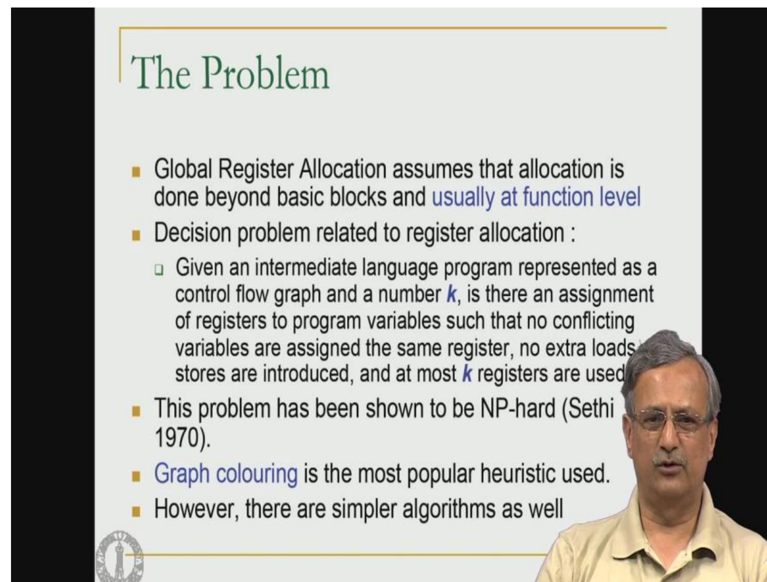
Welcome to part two of the lecture on Global Register Allocation, so in the last part, part one of the lecture I told you about the issues in global register allocation.

(Refer Slide Time: 00:36)




So, which register to use where to you know, which variables to place in the registers, etcetera, etcetera; and I also you know told about you the problem as such.

(Refer Slide Time: 00:56)



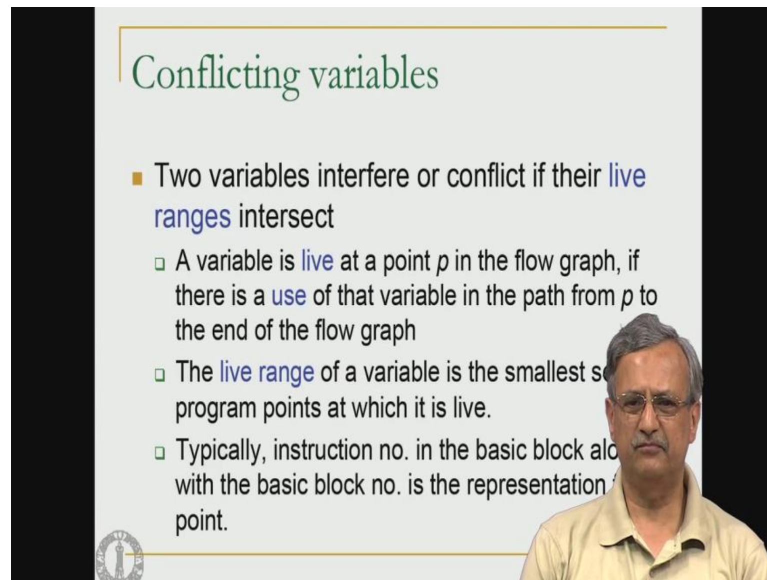

So, let us begin with the problem definition, so global register allocation assumes that allocation is done beyond basic blocks and usually at the function level. The implication is that we are not actually limited to just basic blocks in the program to do register allocation. But, we can also do at the higher level namely the function or may be a group of loops etcetera, etcetera, so these you know this sort of global register allocation actually becomes much better than a local register allocation.

Because, it saves a lot of stores and loads at the boundary of the basic blocks, there is a very important decision problem related to register allocation. So, the problem is in the slide, typically it says you know we are given a number k and we are going to represent the program in the form of a control flow graph that is the assumption. And then depending on the number of variables in the program is there an assignment of registers to program variables, such that no conflicting variables are assigned the same register.

So, the requirement is if there are two variables and if they actually carry values at the same time in the program, then they are said to be conflicting variables I will give you a proper definition of this a little later. So, if the variables conflict then they cannot be assigned a same register, the other thing is we should not introduce unnecessary loads and stores and we must choose at most k registers for the program. So, this problem has been shown to be N P complete, where back in 1970 by Ravi Sethi.

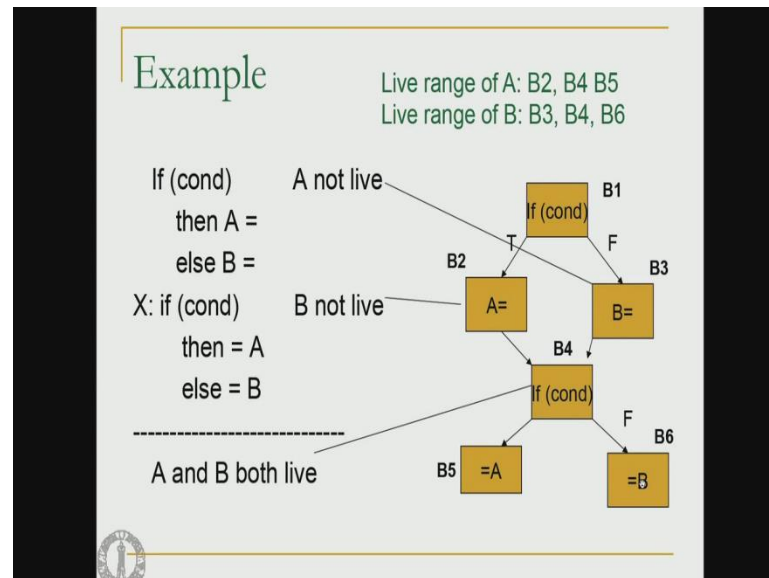
Therefore, the heuristic way of solving the problem is the only way ahead, graph coloring is one of the most important and popular heuristics that is used and that is the one we are going to discuss as well. There are simpler algorithms, so we will look at one of them possibly for the loops.

(Refer Slide Time: 03:28)




So, let us see what conflicting variables are, so two variables interfere or conflict if their live ranges intersect. So, that brings us to another terminology called the live range, so a variable is live at a point p of the flow graph, if there is a use of that variable in the path from p to the end of the flow graph that is what this definition says.

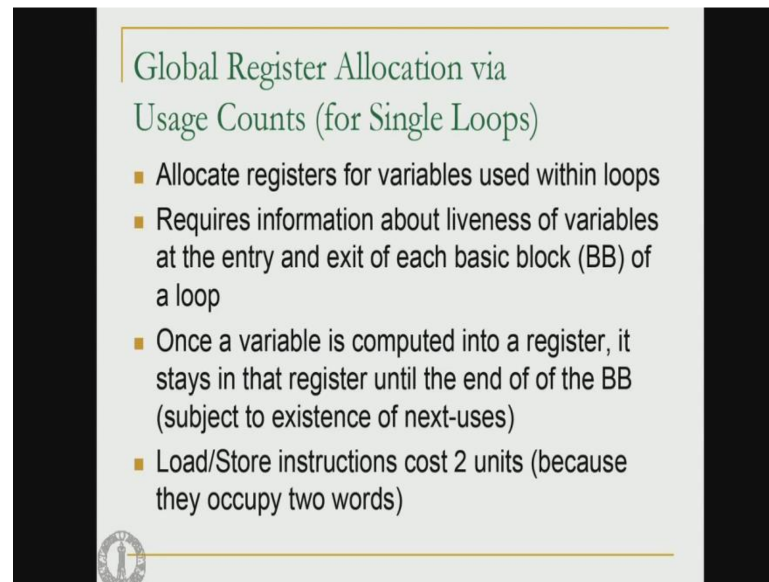
(Refer Slide Time: 03:59)




So, let us look at an example to understand it, so here is the flow graph, so we have defined the variable A here and the variable B here, there is a usage of this variable A here and the usage of this variable B here. So, at this point definitely we can say yes there is a usage of the variable A, so and similarly at this point we can say that there is a usage of the variable B as well, so this is what we mean by you know usage of a variable.

So, from here onwards the live range of A starts from its definition and it, so B 2 then at B 4 also we can answer the question of usage is there a usage of the variable A yes. So, this block, this block and the and this block constitute the live range of the variable A similarly, B 3, B 4 and B 6 constitute the live range of the variable B ((Refer Time: 05:04)). So, very strictly speaking we actually have to look at the program points, so typically instruction number in the basic block and the basic block number. So, the live range of a variable is the smallest set of program points at which it is live, so in this example here, here and here this is these are the three places where the variable A is live, so these three form the live range of A and the these three form the live range of B.

(Refer Slide Time: 05:36)



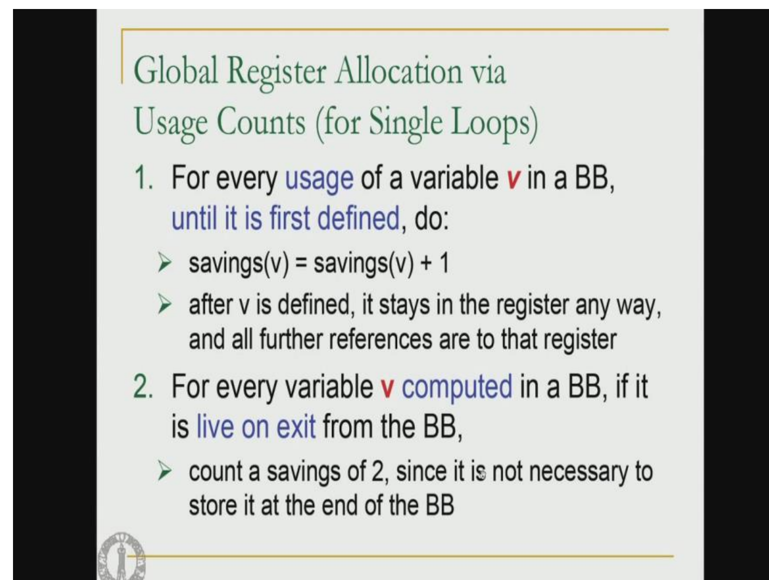

So, let us look at a simple algorithm for global register allocation, the region here is not a complete function, but we will look at loops. So, to begin with we will see how to do allocation for single loops and then apply the same algorithm for multiple loops as well. So, this algorithm can be used to allocate registers for variables used within loops and it requires information about the liveness of variables at the entry and exit of each basic block, why do we require this information, the loop is somewhere in the middle of a program.

So, if there are variables which are live on entry to the basic block, that is the variables are used within the loop and this can be brought down to the level of the basic block. So, if the variable is live at the beginning of a basic block, then you know it is used within the basic block. So, and if the variable is live at the exit of the basic block, then it is used beyond the basic block, so if this is the way it is, so if the variable is live at the entry of a basic block, then we must load that variable into a register at the entry.

And if it is live at the exit of the basic block, then we must store the variable into memory at the end of the basic block. So, to make sure that the costs of doing these operations is actually computed properly we require this live information and why are we not bothered about the variable, you know throughout the program. Once it is computed into a register then; obviously, it stays in that register till the end of the basic block, so we are only interested in the variable computation the first time, the rest of the time

anyway it stays in the register, so there is no extra saving that is possible. So, we are not interested in the usages of variables after the first computation is kind of over. So, then the load store instructions also cost two units, so because they occupy two words this is the assumption in computing the savings, when we compute what a known as a usage count.

(Refer Slide Time: 08:13)




So, let us understand how to compute the usage count, so there are two components here, so the first one says for every usage of a variable v in a basic block, until it is first defined. As I said, once it is defined we are not bothered because it stays in the register, so until it is first defined, we will say if it is assign to a register then you know we say something. So, savings v is equal to savings v plus 1 and after v is defined it stays in the registers, so we do not have to worry about the there is nothing extra saving possible.

So, this is one part of the saving that usage count, then there is another part for every variable v computed in a basic block. If it is live on a exit basic block, then count a savings of 2 since it is not necessary to store it at the end of the basic block. So, the point is if we assign a register to this variable you know which is live on exit then; that means, you know it stays in that register and we really do not have to store it that is the basic idea, otherwise we would have really stored it at the end of the basic block.

(Refer Slide Time: 09:36)

Global Register Allocation via Usage Counts (for Single Loops)

- Total savings per variable *v* are

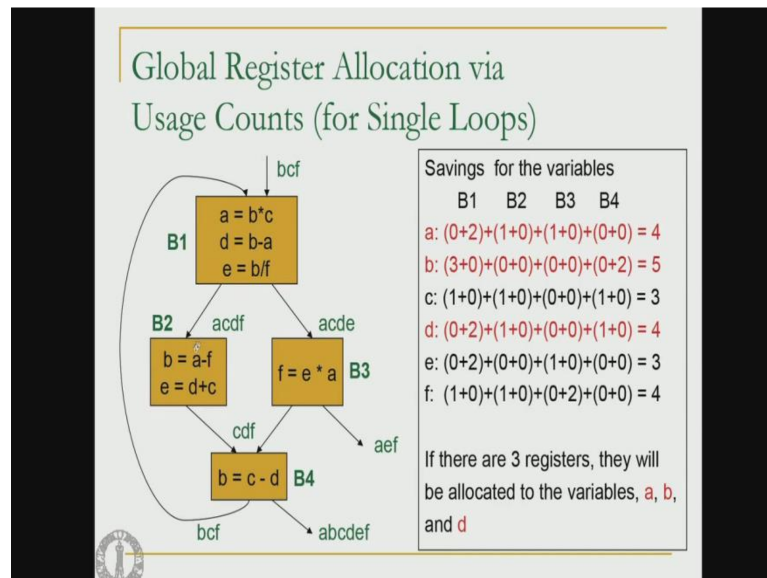
$$\sum_{B \in Loop} (savings(v, B) + 2 * liveandcomputed(v, B))$$

  - *liveandcomputed(v,B)* in the second term is 1 or 0
- On entry to (exit from) the loop, we load (store) a variable live on entry (exit), and lose 2 units for each
  - But, these are "one time" costs and are neglected
- Variables, whose savings are the highest will reside in registers

Then with these two components we are going to have the savings as sigma over all the basic blocks in the loop, savings of the variable v in the basic block plus 2 star a factor called live and computed v comma B. So, this is basically you know the variable being computed this is either 1 or 0, so this two star live and computed corresponds to this part. ((Refer Time: 10:08)) So, the variable v computed in a basic block if it is live on exit, so where as first one corresponds to this part.

So, we do that you know and there is a minor factor which we ignore, so on the entry and exit of the loop the points of entry and exit of the loop, we have to load or store a live variable. For this we require two units for the load or two units for the store at the exit, but these are one time cost because this is actually required only at the entry of the loop or exit of the loop, we are not talking about the entry and exit to the basic block. But, has been taken care of here, once we compute the total savings these variables whose savings are the highest will reside in registers.

(Refer Slide Time: 11:05)




So, let us look an example a very simple basic block the variables b c and f are live on entry to this loop and along this path a c d f A you know variables are being used. So, a c d and f are live at this point, a c d and e are live at this point, c d and f are live at this point, b c f are live here of course, existing from the loop a b c d e f are live here. So, it is easy to check why you know liveness is like this, a c d f are being used here see a c d and f all the four are being used that is why it is all these four are live.

So, let us compute the usage count, it has been listed for each of the variables here, so let us take the variable a. And again the cost has been for each of the basic blocks, for the basic block B1 the cost is 0 plus 2 why, the first component; obviously, corresponds to usage before it is defined. So, a is directly defined here in the first statement itself, so there is no usage before definition, so the cost is 0 here. And then a is indeed being computed in the basic block and then it is also live on exit from the basic block. ((Refer Time: 12:45))

So, remember this, so you know every variable v computed in a basic block, if it is live on exit from the basic block. So, it is indeed live a variable a along both the paths, so we have 2 into 1, so that is the cost here, so this is 2, in the basic block B 2 we have a usage of a before any definition of a. So, that is cost 1 and we do not have a definition of a, so the second factor become 0, similarly we have a usage of a in B3, so that cost 1 and the second part is 0 because there is no definition of a here.

In the fourth one there is a neither a usage of a nor a computation of a, So both the cost are 0 here. So, this total cost is 4, similarly for b we have three uses of the variable b you know and it is before any definition of b, in fact, there is no definition at all. So, the first cost is 3 and the second cost is 0, here we do not have any usage of b before it is definition. And b is not live on exit from the basic block, so both the costs are 0 here, similarly the same is true for this there is no usage of b and b is not live, so these are two 0's here.

And a fourth one B 4, the first component is 0 because there are no usages of b before it is definition, b is computed in the basic block and it is live on exit. So, the cost is really 2, so 2 into 1, so this cost is 5, similarly we can compute the costs of c d e and f as well. Now, 5 goes undisputed that will be one of the variables to which a register can be given, then there are three contenders a d and f two out of these can be given registers, we have arbitrarily pick a and d it is really does not matter, which is given here it could have been a and f as well.

So, once we provide registers to these variables, the code will also become different when we generate the machine code we will have to make sure that we refer to the register corresponding to a b and d. And we must make sure that of course, a is not live here, but b is, so b will have to be loaded on entry to the loop and a b and f will have to be put back into memory by using store instructions at the exit of loop. So, these are all things that will have to be done using the register corresponding to the three variables.

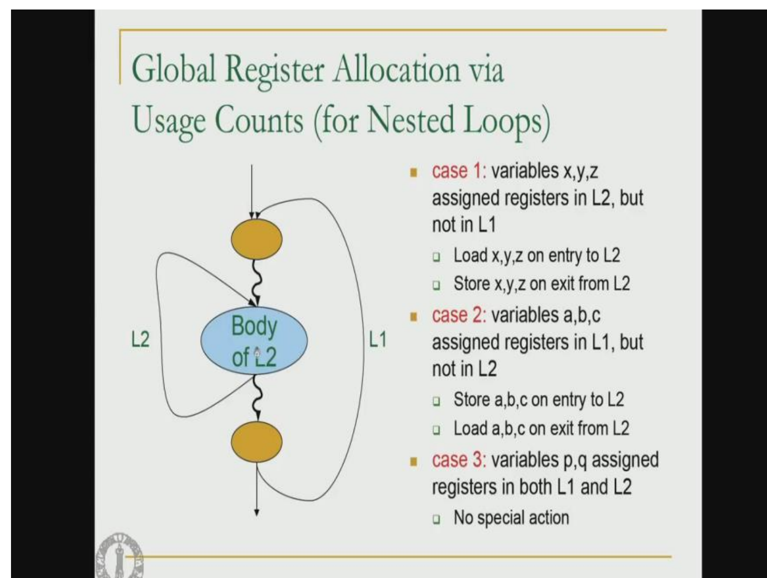
(Refer Slide Time: 15:59)

Global Register Allocation via Usage Counts (for Nested Loops)

- We first assign registers for inner loops and then consider outer loops. Let L1 nest L2
- For variables assigned registers in L2, but not in L1
  - load these variables on entry to L2 and store them on exit from L2
- For variables assigned registers in L1, but not in L2
  - store these variables on entry to L2 and load them on exit from L2
- All costs are calculated keeping the above rules

So, what happens if we have nested loops.

(Refer Slide Time: 16:06)



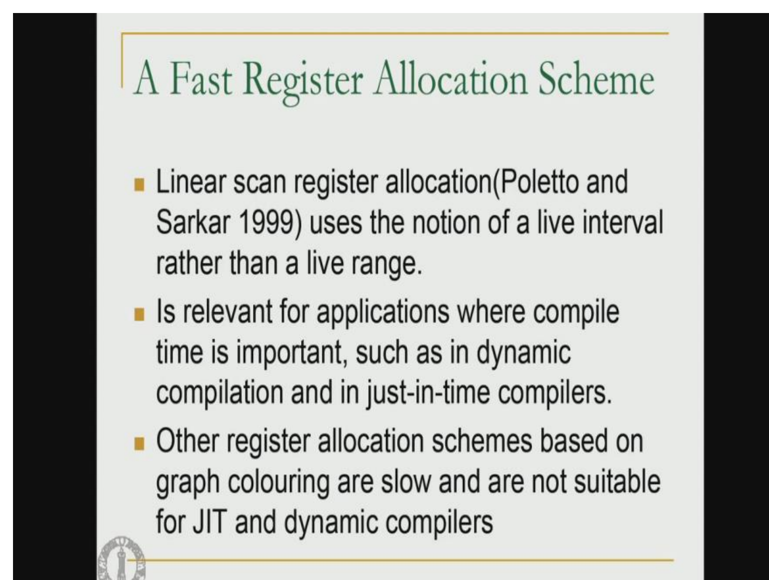

So, let me tell you using this example, so here is the a small loop L 2 which is actually, so here is the loop L 2 and it is embedded within the loop L 1. So, there are some basic blocks here and here as well, so how do we now allocate registers to the nested loops. ((Refer Time: 16:34)) So, the procedure is to assign registers for the inner loops first and then consider the outer loop. So, we have L 1 nesting within the loop you know L 2, let L 1 nest L 2, so L 2 is being nested inside the loop L 1.

So, that is what this means, the rule is for the variables assigned registers in L 2, but not in L 1, load these variables on entry to L 2 and store them on exit from L 2. So, let us see what this means here, ((Refer Time: 17:09)) so let us say variables x, y and z are assigned registers here. So, we did the register allocation, so they got the registers here, but when we did the register allocation for L 1 they did not get any registers.

So, variables should get registers only here not in these two places, so what we need to do is very obvious, we need to load x, y and z on entry to this loop L 2 and then we need to store x, y and z on exit from the loop. So, once this is accomplished you know that is it, there is no we have to take care of these costs and that particular allocation will work properly. Case 2: a, b, and c are assigned registers in L 1, but they are not contain the registers in L 2.

So, in such a case you know when we actually enter the loop L 2 because the registers corresponding to a, b and c will be given to some other variables, we need to store their values in a memory. And then when we exit the loop L 2 we need to put the values from the memory into the registers again for corresponding to the variable a, b and c, so this is done then the loop will work properly. Then the variables p, q are assigned registers in both L 1 and L 2; obviously, we do not require any special action at all you know they just continue from one loop to another. So, this is the usage count algorithm the usage count algorithm for allocating registers to you know variables.

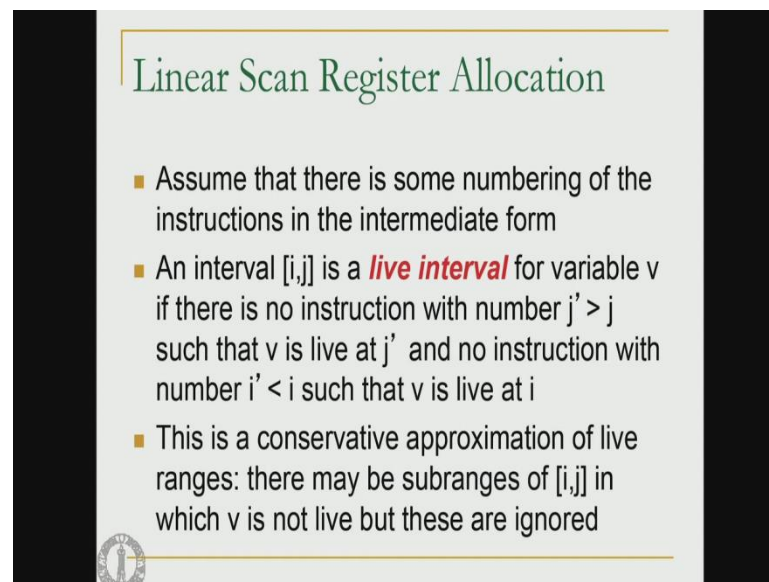
(Refer Slide Time: 19:02)



So, now let us look at a very fast register allocation scheme, this is called as linear scan register allocation. So, the reason now which is called linear scan will become clear very soon, this is due to you know Poletto and Sarkar and it was actually published in 1999, it uses the notion of what is known as a live interval, rather than a live range. So, live interval is an approximation to a live range, so in other words live ranges are subsets of live intervals, it is relevant for applications where compile time is important.

So, for example, in dynamic compilation and just in time compilation, the compilation time is also added to run time. Because, all the compilation happens on the fly, in such cases using a very expensive register allocator, you know is going to have a bad affect on the compiler. So, we must use simple register allocators, so this particular linear scan register allocator is something that can be used in dynamic and just in time compilers, register allocation schemes which are based on graph coloring are very slow, so they cannot be used in JIT and dynamic compilers.

(Refer Slide Time: 20:41)



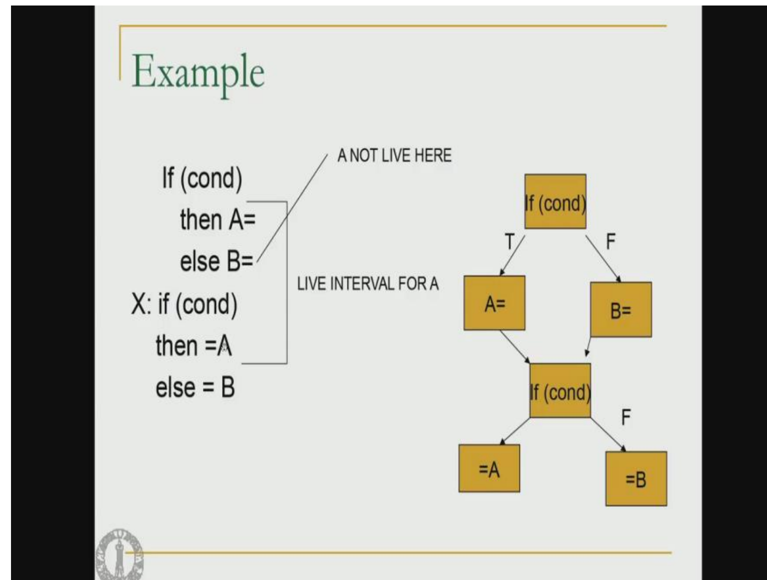

So, let us begin with a definition for the live interval, assume that there is some numbering of the instructions in the intermediate form. Now, an interval i comma j is a live interval for a variable v, if there is no instruction with number j prime greater than j such that v is live at j prime, similarly there is no instruction you know with number i prime less than i such that v is live at i.

(Refer Slide Time: 21:26)

So, let me show you an example here, so we have the i and j here, this is the you know sequence of instructions that we are considering for the variable v to v live. So, if we take another instruction i prime, you know then v any of the instructions before this will not have v as live, such a i prime where v is live does not exist. Similarly, this is the last instruction where v is live, so any other instruction j prime, where v is live does not exist. So, in if this are satisfied these two conditions are satisfied we say that i to j is the live interval for the variable v. ((Refer Time: 22:16))

So, this is a conservative approximation of live ranges, so there may be sub ranges of i comma j where we live, but these are ignored. ((Refer Time: 22:28)) So, in other words, if this, you know sequence from i to j is a long sequence, it is possible that we define you know the variable many times within this range, but all such ranges of the same variable are included in the live interval from i to j.

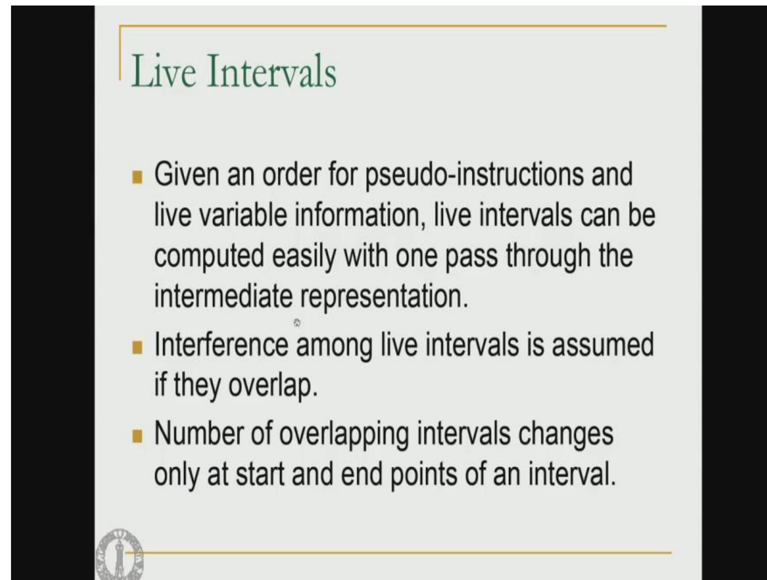
(Refer Slide Time: 22:50)




So, here is a an example to show you that, so; obviously, here is the definition of A, here is another definition of A, here is a usage of A, here is the usage of B. But, if you look at the text sequence for the instructions, the definition of A comes first then the definition of B, which is actually in the basic block this basic block. Then we have the condition corresponding to this basic block, then we have the usage of A and then the usage of B.

So, these instructions will be numbered in some order, we are going to take the instruction number from this assignment to A and the instruction number where A has been used last. So, this entire range of instructions, which includes the basic block v will be considered as the live interval for A, so even though A is not defined or A is not used here at all, so A is not live here. But, since we are going to look at the text placement, so from A to A this definition of A to usage of A is going to be considered as the live interval for A.

(Refer Slide Time: 24:11)




So, given an order for pseudo instructions and live variable information, live intervals can be very easily computed using just one pass over the intermediate representation. ((Refer Time: 24:32)) So, let me tell you how it can be done, so we start scanning the instructions one by one, so we hit a definition for a let us some variable b, let us say, then we go on looking at you know other definitions and variables usages of the variable v.

So, once we know that the last usage of v has been met, then you know the entire range i to j is taken as the live interval for variable v, interference among live intervals is assumed if they have an overlap. So, these are you know live intervals are nothing, but intervals of integer number, so if there are two intervals which overlap, then you know they have some common range between them, so that is the basic idea. So, the number of overlapping intervals changes only at the start and points of an interval, this become very clear as we take an example.

(Refer Slide Time: 25:39)

Example

Active lists (in order of increasing end pt)

Active(A)= {i1}
Active(B)={i1,i5}
Active(C)={i8,i5}
Active(D)= {i7,i4,i11}

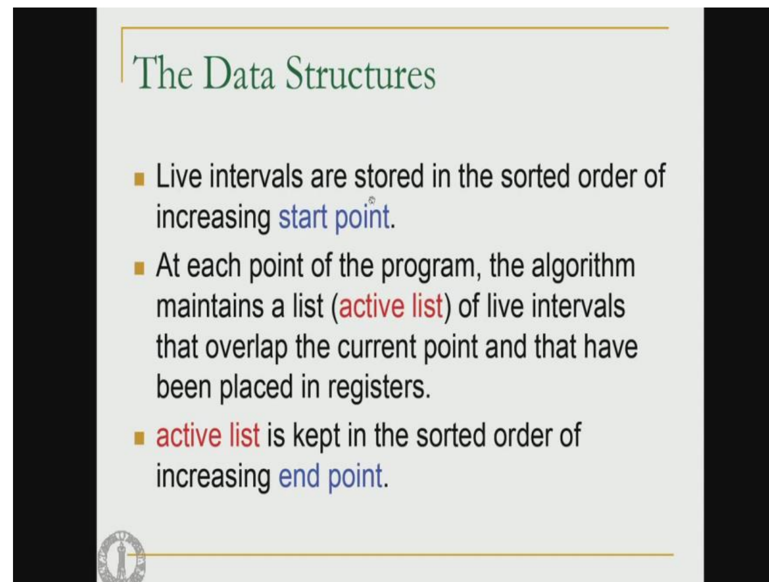
Sorted order of intervals (according to start point):
i1, i5, i8, i2, i9, i6, i3, i10, i7, i4, i11

Three registers are enough for computation without spills

So, let me show you that example here before we continue with the algorithm, so there are many live intervals here i 1, i 2, i 3, i 4, i 5, i 6, i 7, i 8, i 9, i 10, i 11. Now, we say that you know, so the some variable is actually live throughout this interval, so please observe that we are going to make a decision about the east location, at the start and end points of these intervals. So, the basically we are going to consider some information here and then the next point at which we consider it is here.

So, at these points we are going to check whether some intervals are have expired and so on and so forth. ((Refer Time: 26:33)) So, the number of overlapping intervals changes only at the start and end points of an interval, so this is how would be you know look at the intervals.

(Refer Slide Time: 26:46)




So, to do that live intervals are stored in the sorted order of increasing start point, ((Refer Time: 26:53)) so here,. So, the start point of i 1 is the first one, then we have i 5, then we have i 8, then we have i 2, then we have i 9, i 6 you know then we have i 3, then i 10, i 7, i 4 and i 1. So, this is the sorted order of the intervals, at each point in the program, the algorithm also maintains what is known as an active list, so the active list of live intervals correspond to the intervals, which are overlapping the current point and of course, very important these are the intervals which have been placed in registers.

So, the intervals which are active, but have not been placed in registers, actually are the once which have been assign to memory locations. So, they will not occur in this particular list, we will see how this happens, so this active list is kept in the sorted order of the increasing end point. Remember live interval list is sorted according to the starting point and the active list is stored, you know in the sorted order of the increasing end point. ((Refer Time: 28:12))

So, these are the active list at various points A, B, C and D, so at A i 1 you know is the only one which is active and it let us say it has been given a register. At point B we have a i 5 and i 1 both of them actually overlapping and let us assuming that they will be given registers, there are the they will both be in the active list of B, at the point C i 1 has finished. So, it will not be in the active list anymore, but i 5 and i 8 will be, so assuming that they are given registers they will be in the active list if...

So, at D here, so we have i 7 which has not finished yet, i 4 which has not finished at and i 11. So, this is kept sorted in according to the end point, so this finishes first then this finishes and then this finishes, so if for example, you know i 7 has never been given a register, then i 7 will not be in the list even though it is overlapping with i 4 and i 11, it would have been assign to memory. So, the active list at this point hypothetically will contain only i 4 and i 11, so in you know how does the algorithm work?

So, let me show you an example and then we will read through the algorithm detail. ((Refer Time: 29:55)) So, what we do is we have this sorted order of intervals, so this is according to the starting point, so we take the first item on the list, so which is i 1, we have three registers let us say that is the assumption. So, we give one register to i 1, so we are left with two more, the next list in the interval is i 5, so at this point we check whether i 1 has finished it has not finished yet.

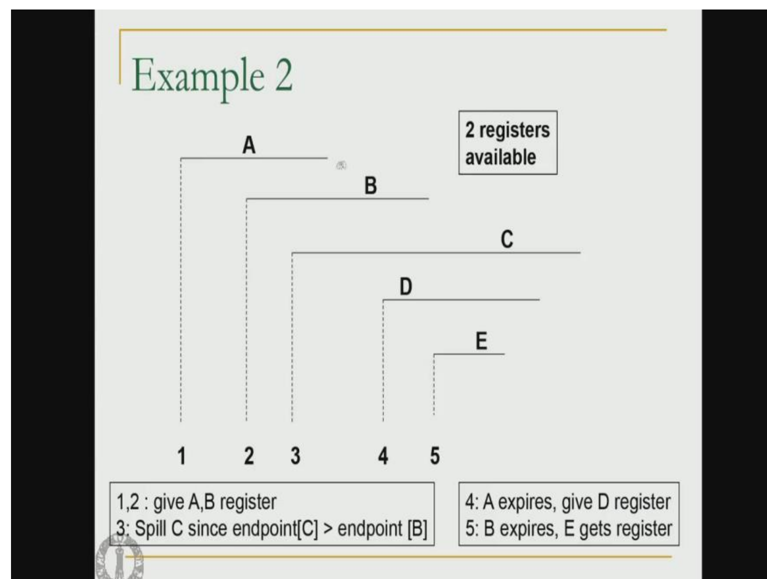
So, we add i five to you know we check the number of registers, so two more are remaining, so we can give 1 to i 5 as well. So, i 1 and i 5 are now in the active list, let us go to point C that is the next one, i 8 is the next interval in the list, so we consider i 8 and the starting point of i 8 is C. So, at this point we can check from the active lists that i 1 which is present in the active list and has been given a register has completed it is not active anymore. So, it can be removed from the active list and it is register can be returned to the free pool.

So, this was the list, so we removed i 1, so i 1 is still active now we add i 8 to this list, now we can add it because we have two registers and we can give one to i 8. Next we take up i 2 at this point neither i 5 nor i 8 have finished and we have three registers, the third one being free we can give it to i 2 and these three will actually on the active list at this point. So, after i 2 we consider i 9, so when we consider i 9 we find that both i 5 and i 8 have finished.

So, i 2 and i 9 will be on the active list and of course, they can be given registers, then after i 9 we pick up i 6. So, at this point again i 2 and i 9 are both active, i 6 can be added to the list and it can be given register, which is still available as a free register, when we go to i 3 you know. So, the i 2 has finished, but i 6 and i 9 are still active and the third register is still 3 for i 3, then we go to i 10, i 9 has finished, but i 3 and i 6 are active i 10 can be given the register which was freed by i 9.

After i 9 we go to i 7, so here you know i 6 has finished, but 3 and 10 have not finished and the register which was freed by i 6 can be given to i 7 then we go to i 4 i 3 has finished i 10 has finished only i 7 is active. So, two registers are free one can be given to i 4 and at i 11 we need all the three registers because i 7 and i 4 have not yet finished. So, this is the way in which register allocation can be done for this simple example, so now, let us look at another example, in which there is some shortage of registers.

(Refer Slide Time: 33:37)



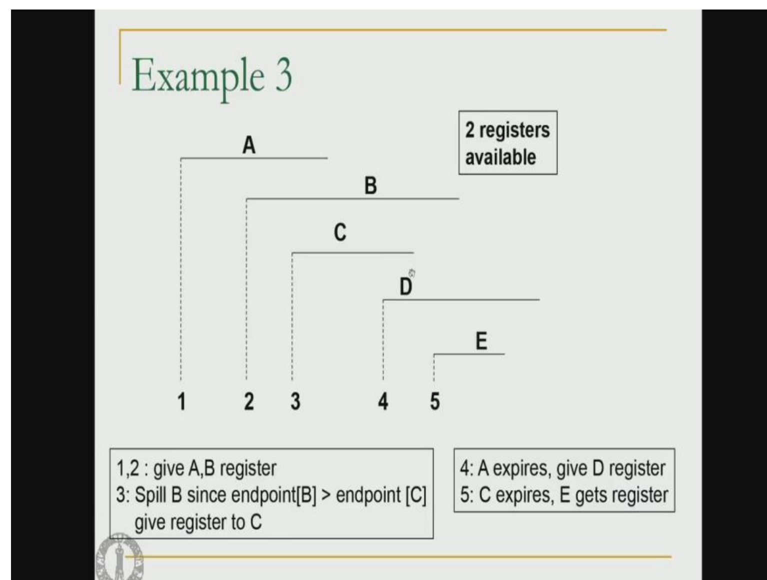

So, we have A B C D and E as a 5 live intervals and let us assume there are 2 registers available. So, what we do here is you know we begin with A, so this is the starting point of A this is point number 1, so we can give a register to A absolutely no problem with that. Then we have you know the live interval of B, this is the point and at B A is still active and we have one more register which is free, so we can give that to N, no problem, so far.

So, A and B are now on the active list, so we go to C, so A and B are still live, you know they are overlapping they have not finished. And the 2 registers are all ready have been given to A and B, now we come to C, the question to be asked is should we take away a register from either A or B and give it to C or just make C go to memory that is called as spilling. So, in this case the decision is made by looking at the end point of the 3 intervals which are active at this point.

So, C actually has an end point which is must further than that of A or B, so what we do is we the heuristic says spill C and put it in memory. The reason behind this heuristic is very simple because C takes a lot of time, you know by the time may be if we put it in memory A and B may free the registers and we will be able to assign more variables to the registers, rather we can probably give more number of variable a register. So, that is the hope and therefore, larger you know time intervals rather live interval are assign to memory by the spilling operation.

So, C is actually put into memory and then at point 4 A expires, so A has finished that can be made sure of by looking at the end point of A. So, it does not have any overlap with that of D, but B is still active C is not in the picture because it has been assign to memory. So, now, we can A, A has released A register and that can be given to D then we go to the starting point of E at this point B has finished that register can be given to E D has not at finished. So, in this example we have actually sent C to memory spilled it into memory, so what would have happened if the duration of C was much smaller.

(Refer Slide Time: 37:00)

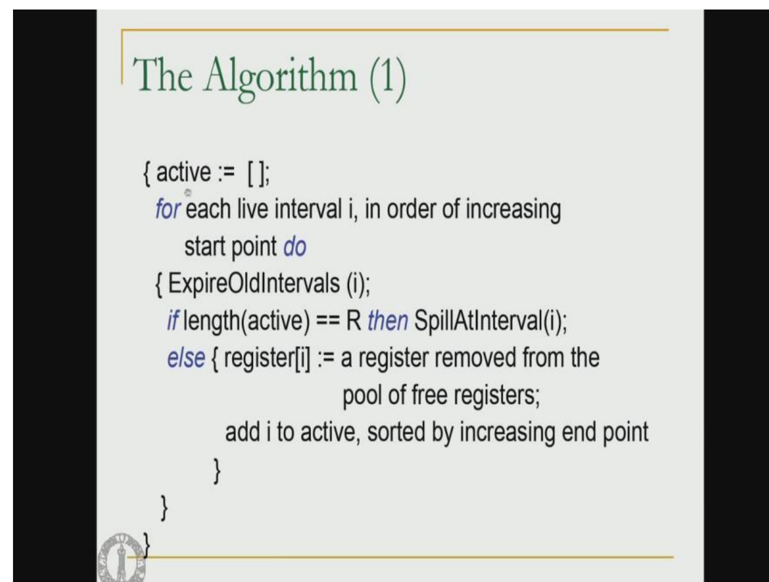



So, for example, here 1, 2 and 3 are similar rather 1 or 2 are similar, at this point of beginning live interval C we find that B's live interval, which is beyond that of either A or B. So, the candidate which is to be spilled is B and it is neither A nor C, so what we do is we take away the register, which was given to B we now give it to C, A retain it is

register because we had two register this is fine. So, spill B since end point of B was greater than endpoint of C give register to C.

Now, onwards everything is still because at the point D, A frees a register and that can be given to D, at the beginning of E, C has finished and therefore, it is registers can be given to E. So, this is the algorithm, so now let us look at the formal description of the algorithm to understand how it goes.

(Refer Slide Time: 38:07)

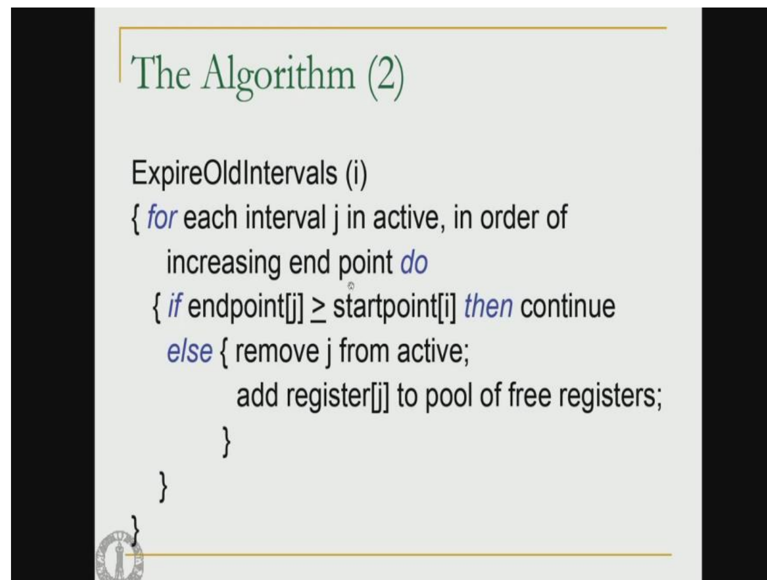



So, the active list is to begin with made empty and for each live interval i, in the order of increasing start point, we execute the following code, the first is expire old intervals. So, all the intervals which have exited, you know rather have completed their time interval are now thrown away I will give you a details of this very soon. Then whatever exists in the active list is only the list of intervals, which are still you know which have not expired and they have all been given registers.

So, if the length of the active list is R, then you know we must call spill at interval i and this may decide to give the, you know put a variable which actually has a register into memory or it can actually store the new interval itself in memory, we will see the details of this also very soon. If they register is free; that means, length of active is not equal to R, so it is much lesser than R, so there is at least one register free, so then we can assign that register to the register, you know we can actually assign the live interval i, this particular register removed from the pool of free registers. So, now, add i to the active

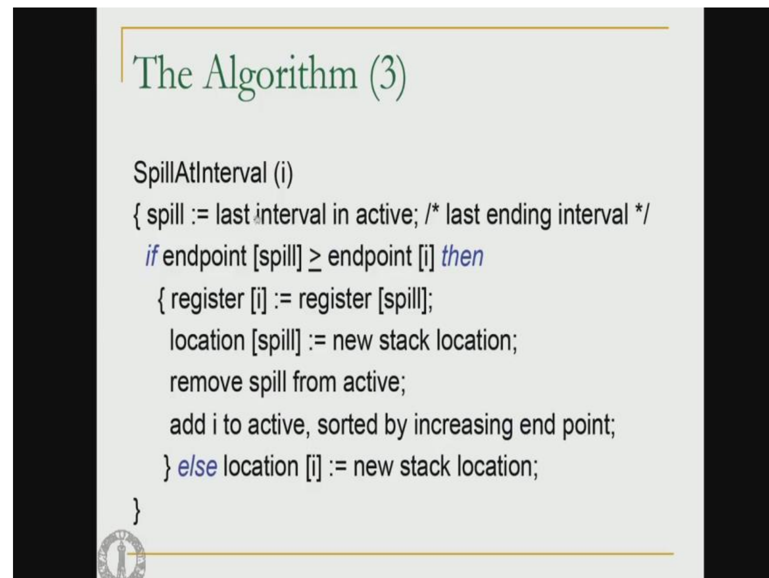
list sort it again by the increasing end point, so that you know we are ready for the next titration. So, on we still need to see the details of the two functions expire old intervals and spill that interval i.

(Refer Slide Time: 40:18)




So, how do you expire old intervals, we are going to inspect every interval j in the active list. So, we look at the increasing end points of these because it is already sorted in that order, so the remember the interval i is the new interval, j corresponds to the older once in the active list. So, if the end point of j is greater than or equal to start point of i; that means, j is still active then continue, so we do not do anything we go to the next interval. If the end point of j is less than the starting point of i; that means, j has completed, so remove j from active add register j to the pool of free registers. So, this is done for all the intervals in the active list, so whichever has retired will be you know removed from the list and those registers will be given to will be added to the pool of free registers.

(Refer Slide Time: 41:27)



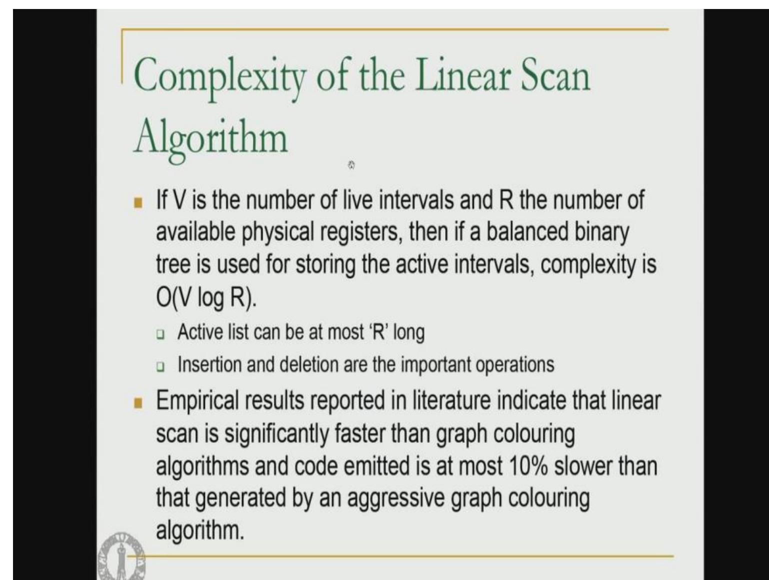

Then the function spill at interval, so again i is the new interval and we must decide whether i should be given a register or we want to put i into memory. So, let us spill be the last interval in the active list, last ending interval, so if the end point of spill is greater than or equal to the endpoint of i. So, in other word the last interval in the active list, ends much later than the new interval i, then we take away the register that was given to this interval spill and give it to the interval i.

So, register i equal to register spill this is a taking away operation and then the location of spill will be new stack location, in other words the interval spill is now banish to memory. So, this is the new location at variable location, which was created on the activation record, so that is actually now taken and given to spill, so these are the locations which are on the activation record. So, these are the off sets which will be assign to the intervals or the variables.

So, now, remove the interval spill from active, so the variable corresponding to spill is now gone, it is always going to be in memory. The new interval is added to active list and we have already given it a register and then the active list is adjusted to be in the sorted order of increasing end points. So, suppose the none of the intervals in the active list, actually have an you know ending point which is greater than that of i; that means, i itself ends much later than anyone of the intervals in the active list.

So, location of i is new stack location, so we banish i itself to the memory, so that is the way spill let interval works. So, these are the details of the algorithm that also actually done here ((Refer Time: 44:07)) you know we took away the register, which was actually given to be and we banished it to memory, where as C was given a new register. ((Refer Time: 44:18)) Where as in the this example the incoming interval C had an end point which is much greater than these two, so this was actually sent to the memory whereas, these to retain their registers.

(Refer Slide Time: 44:33)

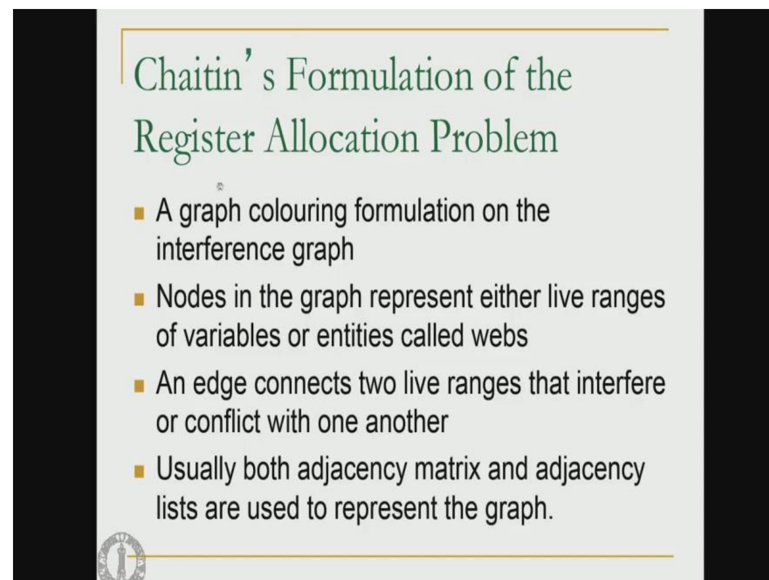



So, this is the I know linear scan register allocation algorithm, let us look at the complexity of the linear scan algorithm. So, as I said the complexity of the linear scan is very important, the reason being it is used in dynamic and just in time compilers, so suppose V is the number of live intervals and R is the number of available physical registers. Then suppose we use a balance binary tree for storing the active intervals; obviously, every one of the accesses for either insertion deletion or search can all be done in time log R, we have V number of live intervals to manage.

So, the time complexity will be V into log R remember that the active list can be at most R long. And that is why we are saying log R, insertion deletions are the important operations we remove something from the active list, we add something to the active list, so empirical results which are reported in literature, typically our you know sarkars paper indicate that linear scan is must faster than graph coloring algorithm.

And of course, there is always a price to pay for this fast algorithm, the price is that the code the machine code which is emitted is a bit slower. So, in the worst case it is about 10 percent slower than the code generated by an aggressive graph coloring based algorithm. So, graph coloring based algorithm is much better, so that you know can enable more efficient code generation, but we are paying for you know the efficiency by the speed of the allocator and slow allocators cannot be used in dynamic and just in time compilers.

(Refer Slide Time: 46:49)




So, now we move on to the next class of algorithms for register allocation, so we now look at the graph coloring based algorithm, way back in 80's Chaitin from IBM and a few others. For the first time they proposed that graph coloring can be use to solve the register allocation problem quite well, so what is the association of a graph coloring rather formulation to the program. So, nodes in the you know we have a data structure called the interference graph I am going to give you the details of interference graph very soon.

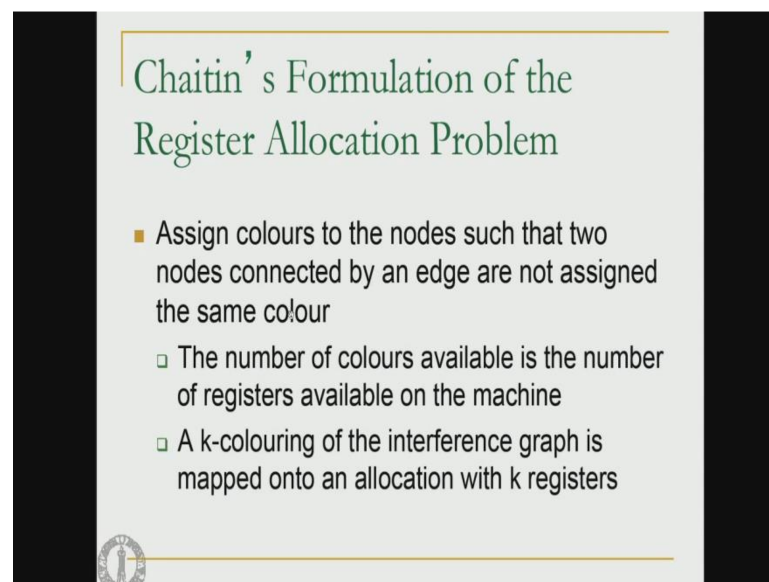
For example, to give you a you know first cut approximation, if you look at the live ranges that we actually used in you know usage based algorithm and the live intervals that we used in our linear scan algorithm. The live ranges actually are the nodes in the interference graph and there are also entities called webs which are possible, we are not getting in to details of web based register allocation in this lecture. So, now, the nodes of

the interference graph correspond to live ranges, whenever there are two live ranges which actually run through the program at the same time.

That means, they are activate the same time in the program same point in the program, then there are said to actually interfere. So, an edge connects two live ranges that interfere are conflict with one another, so the conflict is very similar to that of live intervals, you know if the range is over lap then they conflict. Usually we require two types of data structures, one the adjacency matrix, the other adjacency list to represent such an interference graph.

The reason is sometimes we want to compute the number of neighbors in the for a node, so in such a case searching the adjacency matrix for the neighbors is very inefficient. Whereas, if we have an adjacency list, then you know the searching for the neighbors is a very efficient operation, we just search the list, similarly there are other operations which may be very efficient on the adjacency matrix, where as adjacency list may be very expensive. So, both these are used by the algorithm and the over head actually is maintaining both the data structures instead of just one.

(Refer Slide Time: 50:04)

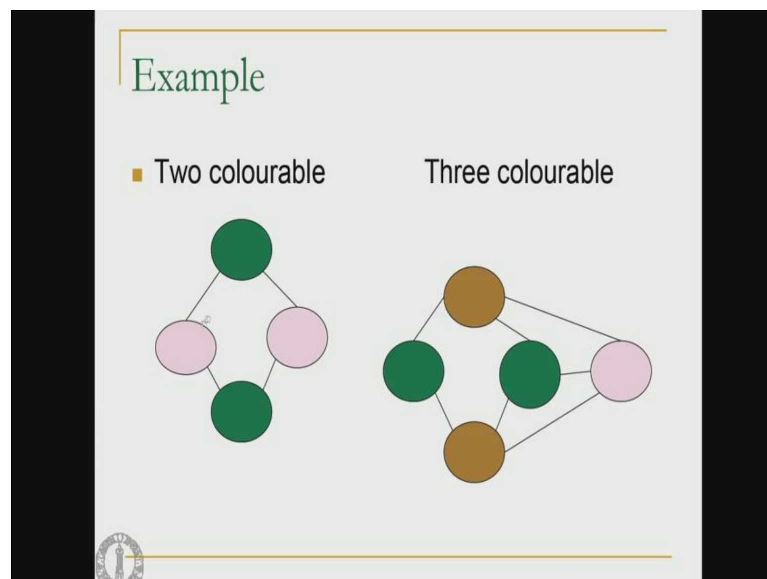



So, the basic idea is to you know take the graph and now the nodes of the graph correspond to live ranges. So, we assign colors to the nodes such that two nodes connected by an edge are not assign the same color, so this is the basic idea, so the color corresponds to a register. So, number of colors available is equal to the number registers

available on the machine and then a k coloring of interference graph is mapped on to an allocation with k registers.

Assuming that we have k registers we try to use k colors, so the difficulty is if the you know interference graph cannot be colored with k colors, then it implies that with k registers, we cannot do an optimal register allocation for the program. So, in such a case we will have to actually you know reduce the number of nodes in the graph, so we may have to remove some of the nodes by what is known as a spilling operation. So, we try to do the spilling operation on the nodes of the graph, reduce the size of the graph and then you know continue with the allocation.

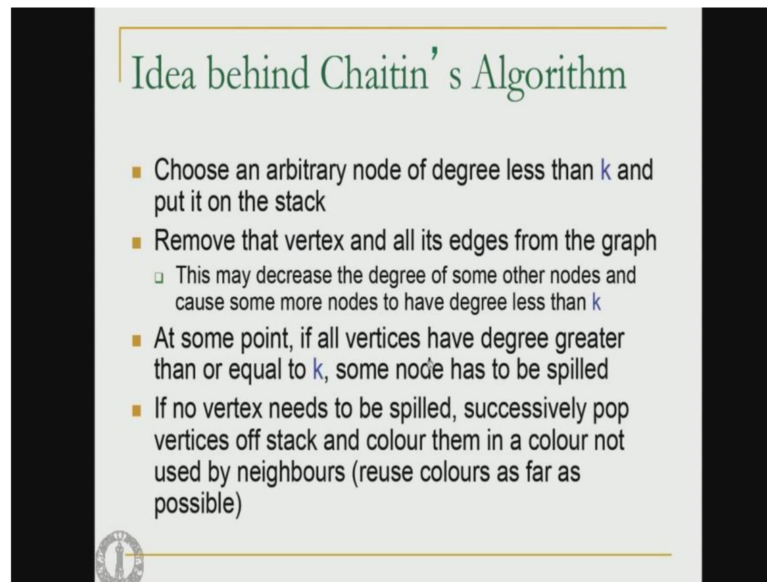
(Refer Slide Time: 51:38)




So, let me give you a very simple example here, this is an interference graph which is said to be two colorable. So, we have a green and violet two colors available here, so green corresponds to one register and this violet corresponds to another register, so in this case this is said to be three colorable. So, we have three colors corresponding to three registers of the machine. So, we have assigned the color here and the same color here, but it, so happens that the two neighbors have two different colors.

That means, the live ranges of this and this, they are connected by an edge so; that means, they are two variables are activate at the same time. But, since they are in two different registers there is no problem in the program, the same is actually valid here as well.

(Refer Slide Time: 25:31)




So, the basic idea behind Chaitin's algorithm is to choose an arbitrary node of degree less than k and then put it on a stack. So, remove that vertex and all it is edges from the graph, so you know; that means, this is the reduction of the graph, this may decrease the degree of some other nodes and cause more nodes to have degree less than k. ((Refer Time: 53:03)) So, if you look at this, suppose we you know have not assigned any colors here, we take this graph take this node we remove this, then we remove the two edges also connected to it.

The graph which is left out is only this part and we continue that operation on this part of the graph. Whereas, if we try doing it here, if we remove this node then these two edges go away and what remains is this graph, so at some point if all the vertices have degree greater than or equal to k some node has to be spilled. Because, we cannot continue this coloring operation, which we described here you know rather we cannot continue this reduction operation which we described here.

If no vertex needs to be spilled that is the best part of it, then successively pop vertices of the stack and color them, in a color not used by the neighbors reuse of colors is definitely possible. ((Refer Time: 54:01)) So, for example, here you know we let us say there are two registers, we remove this and then we are left with this graph, then we can remove this node we are left with this graph, we remove this node we are left with just one node graph, we remove this as well then the graph is empty.

So, in the reverse order we can assign colors which are available we give a color to this; obviously, then this gets a different color. Then this is added next that is get a different color and finally, this is added and that is gets a completely different color, so this is the way Chaitin's algorithm would proceed. So, I will stop here now and consider the details of the Chaitin's algorithm in the next part of the lecture.

Thank you.