Principles of Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

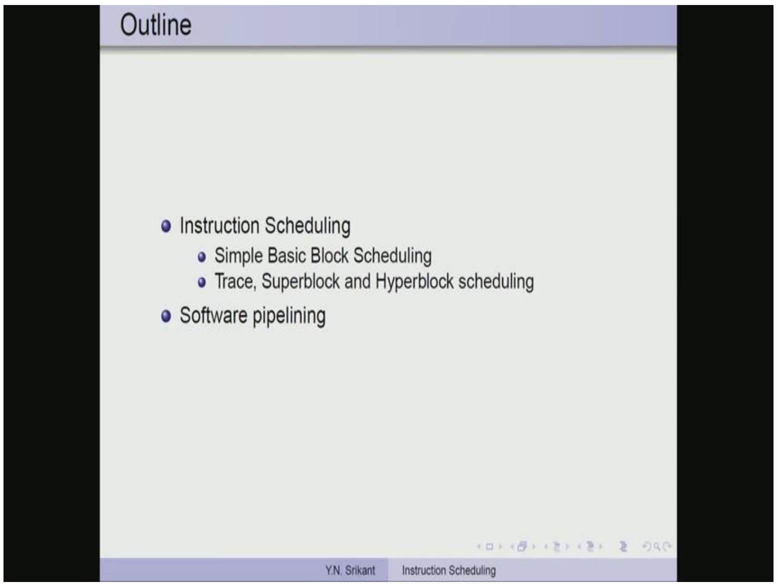
Indian Institute of Science, Bangalore

Lecture - 39

Instruction Scheduling and Software Pipelining Part-3

Automatic Parallelization Part-1

(Refer Slide Time: 00:30)



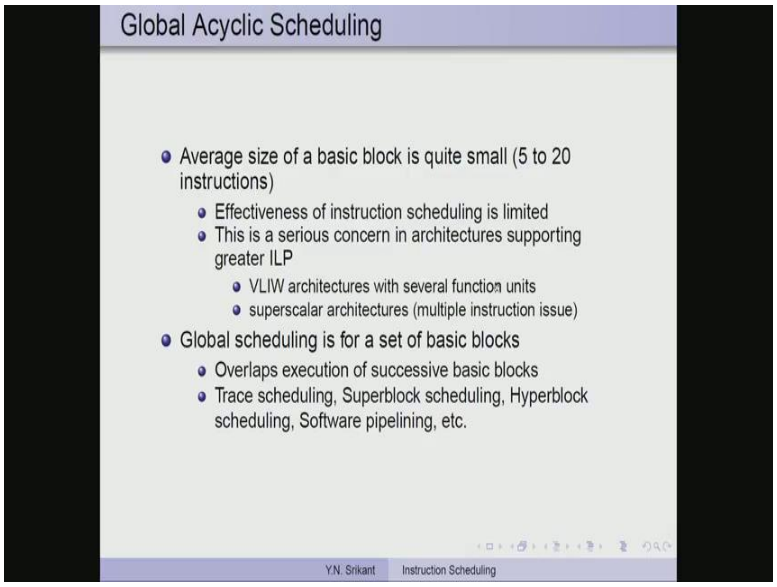
Outline

- Instruction Scheduling
  - Simple Basic Block Scheduling
  - Trace, Superblock and Hyperblock scheduling
- Software pipelining

Y.N. Srikant Instruction Scheduling

Welcome to part three of the lecture on instruction scheduling and software pipelining. Today, we will continue our discussion on instruction scheduling. More specifically the superblock and hyper block scheduling, and then we will continue with software pipelining.

(Refer Slide Time: 00:35)



Global Acyclic Scheduling

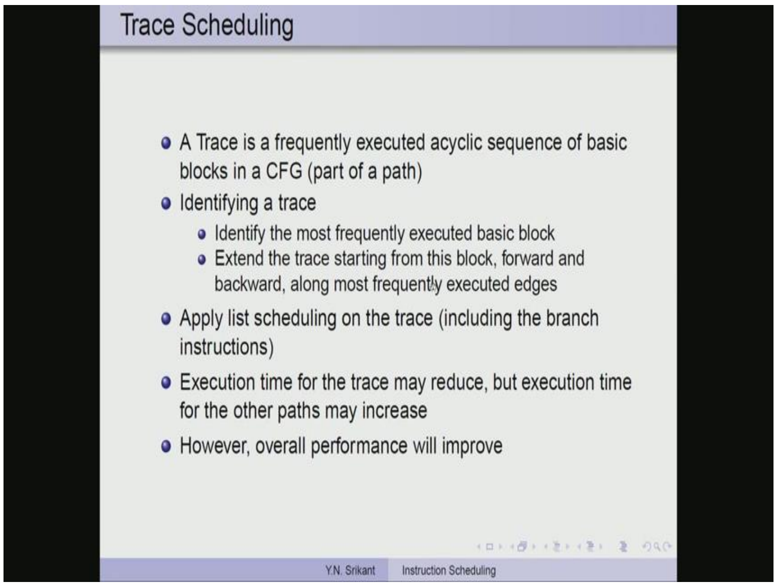
- Average size of a basic block is quite small (5 to 20 instructions)
  - Effectiveness of instruction scheduling is limited
  - This is a serious concern in architectures supporting greater ILP
    - VLIW architectures with several function units
    - superscalar architectures (multiple instruction issue)
- Global scheduling is for a set of basic blocks
  - Overlaps execution of successive basic blocks
  - Trace scheduling, Superblock scheduling, Hyperblock scheduling, Software pipelining, etc.

Y.N. Srikant Instruction Scheduling

So, to do a bit of recap global acyclic scheduling is the general name given to all the scheduling algorithms, which look at more than one basic block. So for example, you know trace scheduling, superblock scheduling, hyper block scheduling, software pipelining, all these belong to this type of you know scheduling called global acyclic scheduling.

The reason why such an algorithm is required is due to the size of the basic block being quite small on the average. So, if with very small basic blocks instructions scheduling of the basic block kind becomes a bit ineffective. And this is a serious concern in architectures such with lots of instruction level parallelism such as VLIW and superscalar.

(Refer Slide Time: 01:38)



Trace Scheduling

- A Trace is a frequently executed acyclic sequence of basic blocks in a CFG (part of a path)
- Identifying a trace
  - Identify the most frequently executed basic block
  - Extend the trace starting from this block, forward and backward, along most frequently executed edges
- Apply list scheduling on the trace (including the branch instructions)
- Execution time for the trace may reduce, but execution time for the other paths may increase
- However, overall performance will improve

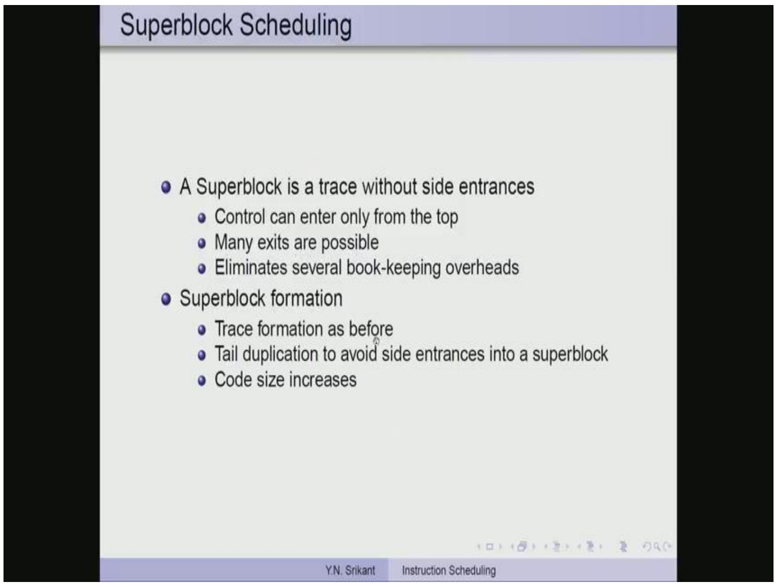
Y.N. Srikant Instruction Scheduling

So, trace scheduling is a method of scheduling and there are many types within that such as, ordinary trace scheduling, superblock scheduling and hyper block scheduling. A trace is nothing but a frequently executed acyclic sequence of basic blocks in a control flow graph. So, this is a part of a path, so there is no very rigorous definition of a trace you can make the trace as big as possible or as small as necessary, but the reasons why we may want to make it big or small should also be kept in mind.

So, identifying a trace we identify the most frequently executed block and then extend the trace starting from this block forward and backward along the most frequently executed edges. So, when we perform trace scheduling basically we combine the blocks of the trace and schedule them as if all of them together form a single basic block.

So, when we do this instruct the execution time for the trace usually reduces, but the execution time for other paths definitely will increase. However, the overall performance generally increases, but the concern is regarding compensation code which needs to be you know inserted for the off trace blocks. So, sometimes such compensation code may become quite large in size. And that may be a reason why we do not want extremely large traces, but would like to reduce the size of the traces.

(Refer Slide Time: 03:29)



Superblock Scheduling

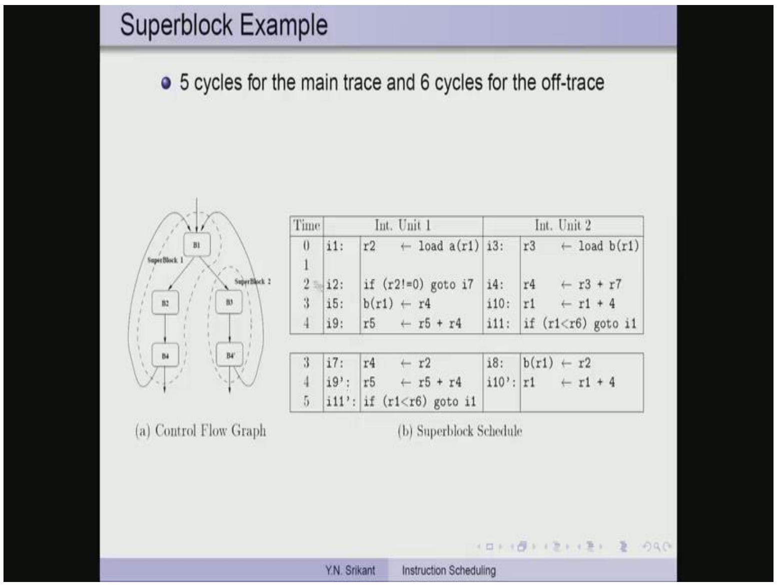
- A Superblock is a trace without side entrances
  - Control can enter only from the top
  - Many exits are possible
  - Eliminates several book-keeping overheads
- Superblock formation
  - Trace formation as before
  - Tail duplication to avoid side entrances into a superblock
  - Code size increases

Y.N. Srikant Instruction Scheduling

So, the second variety of scheduling is the superblock scheduling. A superblock is nothing but a trace again, but it does not have side entrances. The reason we the trace you know it is because of the side entrances, that we had to introduce compensation code for the you know ordinary traces.

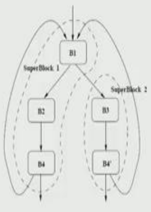
So, control can enter only from the top in a superblock, but many exits are possible. And this eliminates several book keeping overheads, specially the compensation code insertion. So, how do you form superblocks form a trace as before and then duplicate the tail to avoid side entrances into a superblock of course, this will increase the code size.

(Refer Slide Time: 04:24)



Superblock Example

- 5 cycles for the main trace and 6 cycles for the off-trace



(a) Control Flow Graph

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2:	if (r2!=0) goto i7	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4	i10:	r1 ← r1 + 4
4	i9:	r5 ← r5 + r4	i11:	if (r1<r6) goto i1
3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i9':	r5 ← r5 + r4	i10':	r1 ← r1 + 4
5	i11':	if (r1<r6) goto i1		

(b) Superblock Schedule

Y.N. Srikant Instruction Scheduling

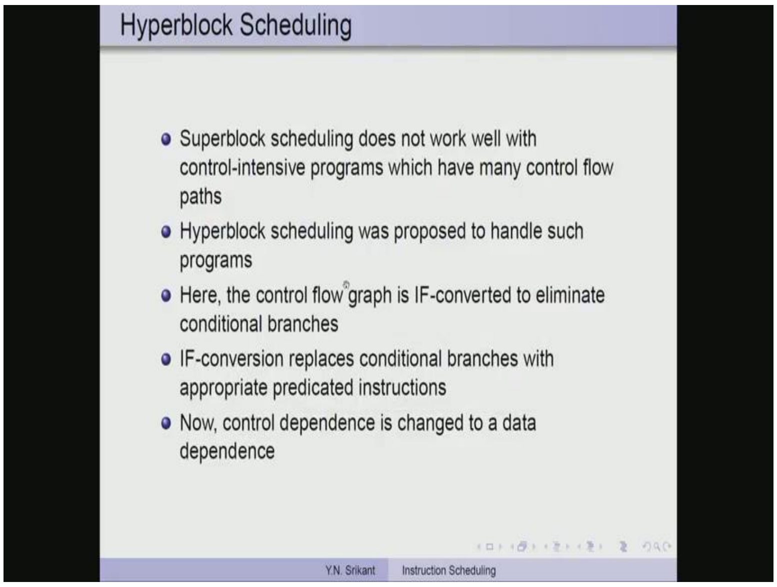
So, here is a simple example of the same set of instructions and the control flow graph. So, we had this block, this block, this block and these two were combined into a single block here, we formed this, this and this as the main trace. And now for the off trace we form a copy of this block b 4 so, that the side entrance into from the off trace into the main trace is avoided.

So, with the and then of course, the scheduling continues as before, we combine these three blocks and schedule them. And then we combine these two blocks and schedule them, but observe that there is code duplication here. And in general you know with a large control flow graph will have a lot of such duplication of code and this increases the code size. So, in the superblock we have done even better, there are only 5 cycles required for the main trace and 6 cycles for the off trace.

The reason why it has reduced further is because of this you know the duplication of code. So, we do not have to jump into the middle of the main trace and we do not have to jump out of the middle of the main trace etcetera, etcetera the only jump is here. So, the code this is the main trace so, 0 1 2 3 4 so we take 5 cycles and this is one part of the main trace. And here, it says if r 1 less than r 6 goto i 1 so, this is the loop part so here.

So, we execute these as if there is no and in the middle here there is an if-condition, if r 2 not equal to 0 goto i 7. So, that would be the off trace we jump to the off trace if the condition indicate so. Otherwise, we just continue and everything has been scheduled in a very packed fashion and for the off trace we require up to this anyway 0 1 2. And then let us say we jump here so 3 4 5 so, that would be 6 cycles whereas, for this we require 0 1 2 3 4 5 so, 5 cycles for the main trace. So, the advantage of superblock is it is even better than the ordinary trace, but it requires extra code, because of the duplication.

(Refer Slide Time: 07:05)



Hyperblock Scheduling

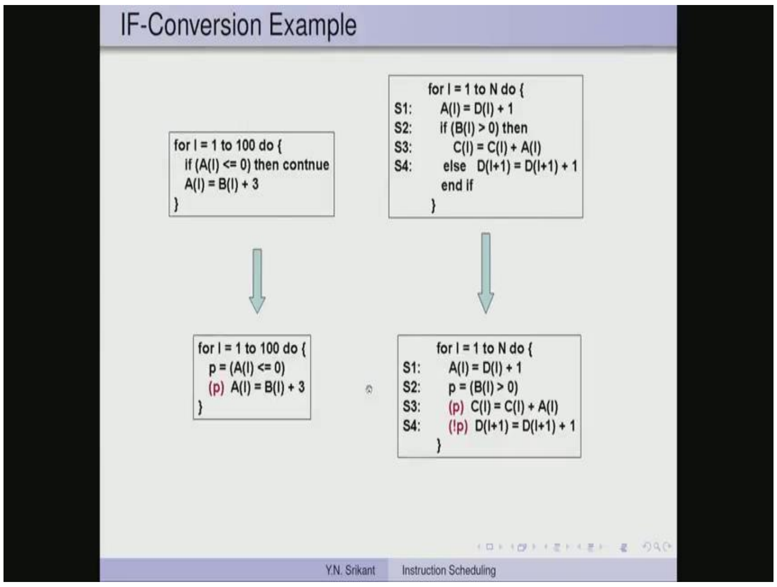
- Superblock scheduling does not work well with control-intensive programs which have many control flow paths
- Hyperblock scheduling was proposed to handle such programs
- Here, the control flow graph is IF-converted to eliminate conditional branches
- IF-conversion replaces conditional branches with appropriate predicated instructions
- Now, control dependence is changed to a data dependence

Y.N. Srikant Instruction Scheduling

Then the next technique in trace scheduling is called as hyper block scheduling. Superblock scheduling does not work well with control intensive programs, which have many control flow paths because if you have many control flow paths. Then the code duplication would be excessive so, this does not work well. Hyper block scheduling was proposed to handle such programs.

The basic idea is to introduce guarded commands so with a rather predicated commands. So, the control flow graph is actually it goes through a process called if-conversion, I am going to give you an example of this very soon. To eliminate the conditional branches and if-conversion replaces conditional branches with appropriate predicated instructions now, the control dependence gets changed to a data dependence.

(Refer Slide Time: 08:07)



IF-Conversion Example

```
for I = 1 to 100 do {
  if (A(I) <= 0) then contnue
  A(I) = B(I) + 3
}
```

```
for I = 1 to 100 do {
  p = (A(I) <= 0)
  (p) A(I) = B(I) + 3
}
```

```
      for I = 1 to N do {
S1:     A(I) = D(I) + 1
S2:     if (B(I) > 0) then
S3:        C(I) = C(I) + A(I)
S4:      else   D(I+1) = D(I+1) + 1
        end if
      }
```

```
      for I = 1 to N do {
S1:     A(I) = D(I) + 1
S2:     p = (B(I) > 0)
S3:     (p)  C(I) = C(I) + A(I)
S4:     (!p) D(I+1) = D(I+1) + 1
      }
```

Y.N. Srikant Instruction Scheduling

An example, of if-conversion we have a loop i equal to 1 to hundred do, if a i less than or equal to 0 then continue and otherwise we go down and perform a i equal to b i plus 3. So, if i a i is greater than 0 we do this otherwise we go to the next iteration of the loop Now, because of this condition we would have code duplication so, instead of that we are going to compute a i less than equal to 0 as a predicate so p.

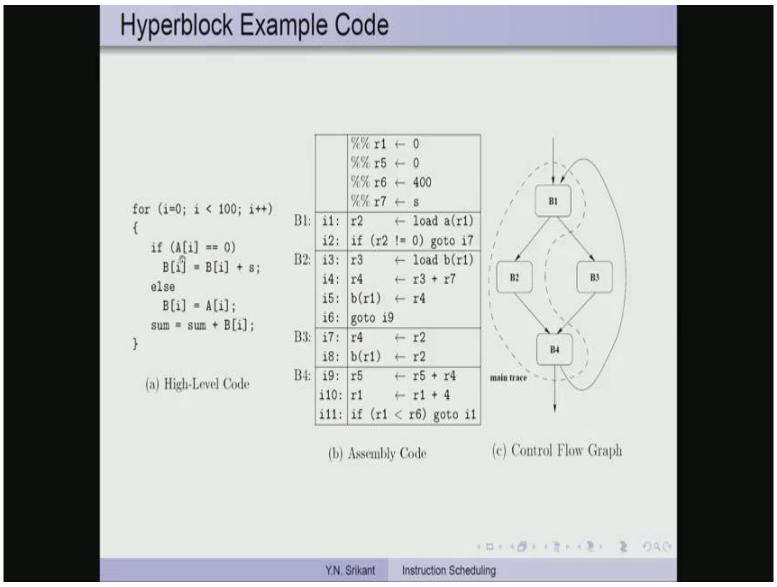
And if the hardware permits predicated instructions, then we can say a i equal to b i plus 3 will be executed as a predicated instruction. So, that is how we work and if the predicate is you know true, then this will be executed and if the predicate is false, then you know the loop just continues so, that is how it would be. So, I think there is a minor mistake in the predicate computation and the guard.

So, the predicate is a i less than or equal to 0 and if it is false then a i equal to a b i plus 3 will be executed so, it is indicated as p here this should have been not p. Now, the other example has four statements and there is a for loop, for i equal to 1 to n. And inside that we have a i equal to d i plus 1 if b i greater than 0 then c i equal to c i plus a i otherwise d i plus 1 equal to d i plus 1 plus 1.

So, there is a block of code which is executed in the then part and also another block of code, which is executed in the else part. As before, we compute the predicate p equal to b i greater than 0, if the predicate is true then we must execute this. So, what we do is the predicated instruction has c i equal to c i plus a i and if the predicate is false, then we execute the second statement.

So, the second instruction is predicated with not p so, not p has d i plus 1 equal to d i plus 1 plus 1. So, the semantics are as before if this instruction executes only when the predicate is true and this instruction is executes only when the predicate is this predicate is false. So, but this requires extra support from the hardware side.

(Refer Slide Time: 10:59)



Hyperblock Example Code

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

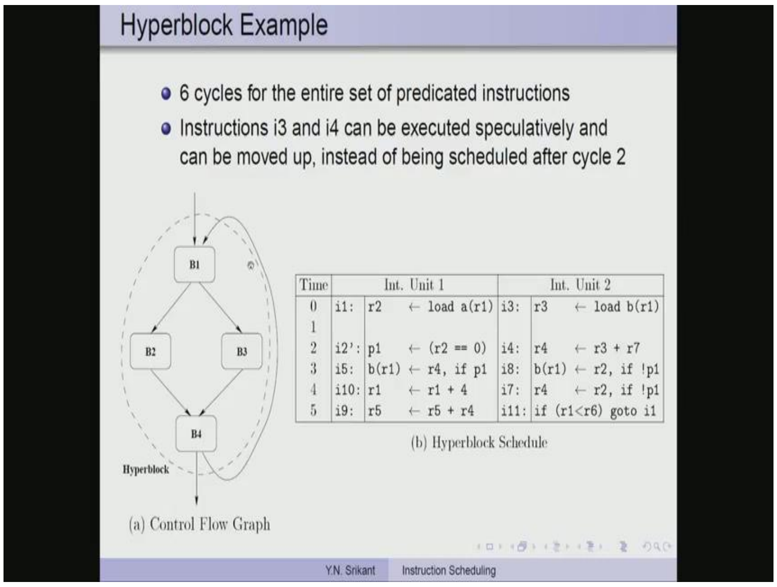
```
          %% r1  ← 0
          %% r5  ← 0
          %% r6  ← 400
          %% r7  ← s
B1:  i1:  r2     ← load a(r1)
     i2:  if (r2 != 0) goto i7
B2:  i3:  r3     ← load b(r1)
     i4:  r4     ← r3 + r7
     i5:  b(r1)  ← r4
     i6:  goto i9
B3:  i7:  r4     ← r2
     i8:  b(r1)  ← r2
B4:  i9:  r5     ← r5 + r4
     i10: r1     ← r1 + 4
     i11: if (r1 < r6) goto i1
```

(b) Assembly Code

(c) Control Flow Graph

So, coming back to our example so, we have you know one piece of code which is executed in the then part, another piece of code which is executed in the else part and then the join executes one more instruction. So, we have 4 basic blocks as before so, this was the diagram we have already seen.

(Refer Slide Time: 11:20)



Hyperblock Example

- 6 cycles for the entire set of predicated instructions
- Instructions i3 and i4 can be executed speculatively and can be moved up, instead of being scheduled after cycle 2

(a) Control Flow Graph

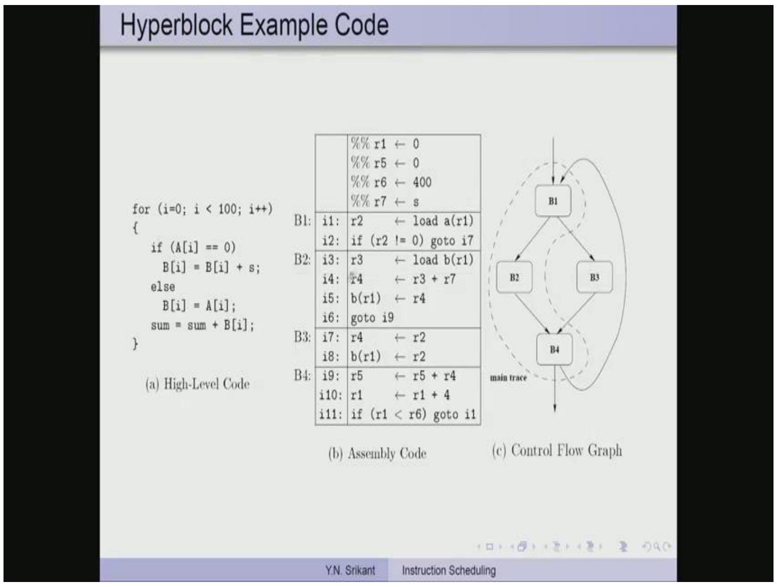
Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2':	p1 ← (r2 == 0)	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4, if p1	i8:	b(r1) ← r2, if !p1
4	i10:	r1 ← r1 + 4	i7:	r4 ← r2, if !p1
5	i9:	r5 ← r5 + r4	i11:	if (r1<r6) goto i1

(b) Hyperblock Schedule

Y.N. Srikant Instruction Scheduling

Now, the hyper block is actually the entire control flow graph in this case, except for this back arc all right. So, we require 6 cycles for the entire set of predicated instructions. Now, if you observe this code the integer unit 1 and integer unit 2, they compute instructions load a r 1 and load b r 1. And then the instruction i 2 prime computes the predicate.

(Refer Slide Time: 11:57)



Hyperblock Example Code

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

		%% r1 ← 0
		%% r5 ← 0
		%% r6 ← 400
		%% r7 ← s
B1:	i1:	r2 ← load a(r1)
	i2:	if (r2 != 0) goto i7
B2:	i3:	r3 ← load b(r1)
	i4:	r4 ← r3 + r7
	i5:	b(r1) ← r4
	i6:	goto i9
B3:	i7:	r4 ← r2
	i8:	b(r1) ← r2
B4:	i9:	r5 ← r5 + r4
	i10:	r1 ← r1 + 4
	i11:	if (r1 < r6) goto i1

(b) Assembly Code

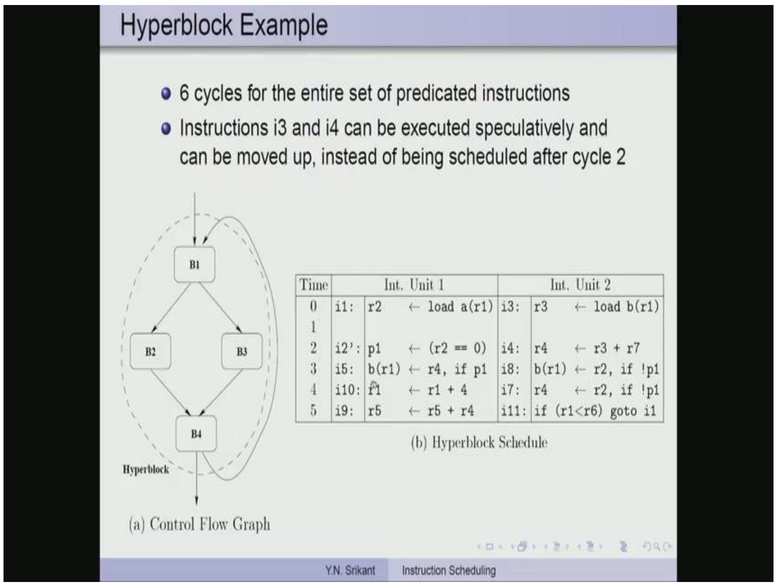
(c) Control Flow Graph

Y.N. Srikant Instruction Scheduling

B1
B2
B3
B4
main trace

So, the predicate was here right this a i equal to 0, that was the predicate here. So, that is computed here right if r 2 not equal to 0.

(Refer Slide Time: 12:08)



Hyperblock Example

- 6 cycles for the entire set of predicated instructions
- Instructions i3 and i4 can be executed speculatively and can be moved up, instead of being scheduled after cycle 2

(a) Control Flow Graph

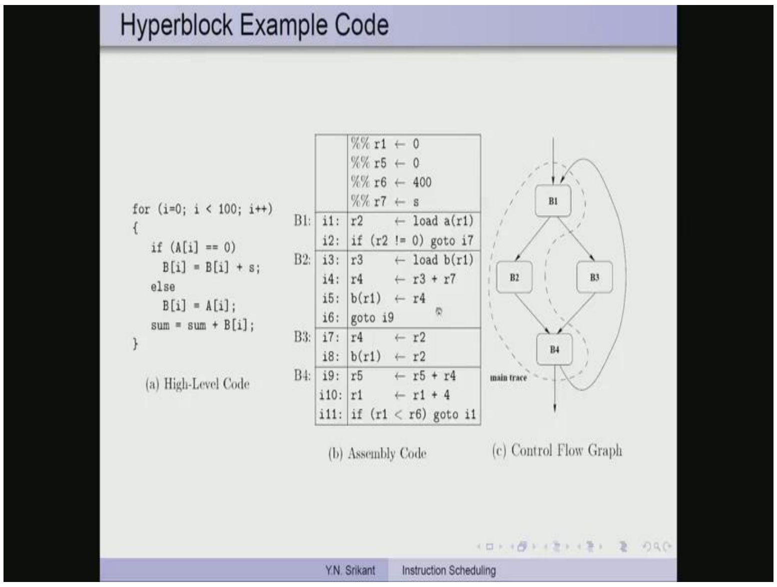
Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2':	p1 ← (r2 == 0)	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4, if p1	i8:	b(r1) ← r2, if !p1
4	i10:	r1 ← r1 + 4	i7:	r4 ← r2, if !p1
5	i9:	r5 ← r5 + r4	i11:	if (r1<r6) goto i1

(b) Hyperblock Schedule

Y.N. Srikant Instruction Scheduling

So, r 2 equal to 0 is the predicate that we compute here and then the instruction b r 1 equal to r 4. If p 1 is the predicated instruction which executes, if p 1 is true and b r 1 equal to r 2 if you know p 1 is false so, not of p 1. So, if p 1 is false this is executed if p 1 is true this is executed. Then we have the other instructions as before, but we also have the second instruction r 4 equal to r 2 if not of p 1 all right. So, we have two instructions under the not p 1 category and one instruction under the p 1 category and then of course, we have other instructions which are not predicated.

(Refer Slide Time : 12:59)



Hyperblock Example Code

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

```
            %% r1 ← 0
            %% r5 ← 0
            %% r6 ← 400
            %% r7 ← s
B1:  i1:    r2     ← load a(r1)
     i2:    if (r2 != 0) goto i7
B2:  i3:    r3     ← load b(r1)
     i4:    r4     ← r3 + r7
     i5:    b(r1)  ← r4
     i6:    goto i9
B3:  i7:    r4     ← r2
     i8:    b(r1)  ← r2
B4:  i9:    r5     ← r5 + r4
     i10:   r1     ← r1 + 4
     i11:   if (r1 < r6) goto i1
```

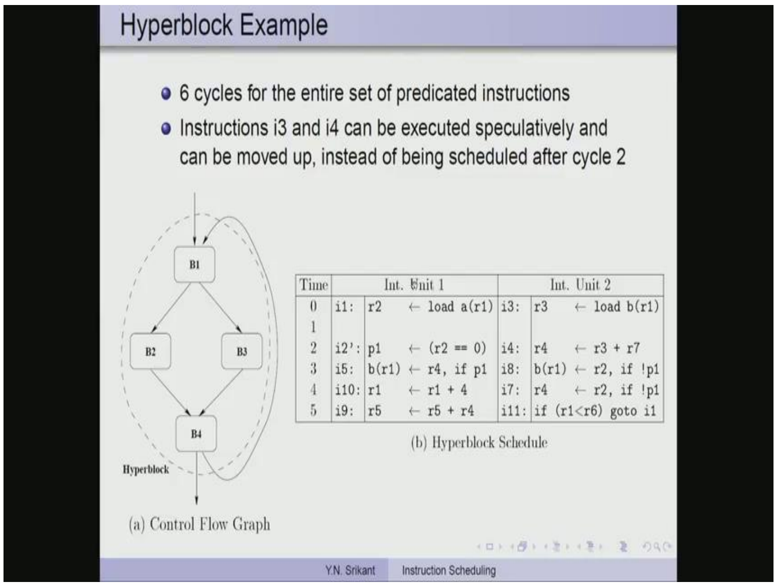
(b) Assembly Code

(c) Control Flow Graph

Y.N. Srikant Instruction Scheduling

So, this is how it is so in this case, you know we have two instructions in this basic block b 3, this is the off trace that is the reason, why we have those instructions here as well in the else part.

(Refer Slide Time: 13:16)



Hyperblock Example

- 6 cycles for the entire set of predicated instructions
- Instructions i3 and i4 can be executed speculatively and can be moved up, instead of being scheduled after cycle 2

B1
B2
B3
B4
Hyperblock

(a) Control Flow Graph

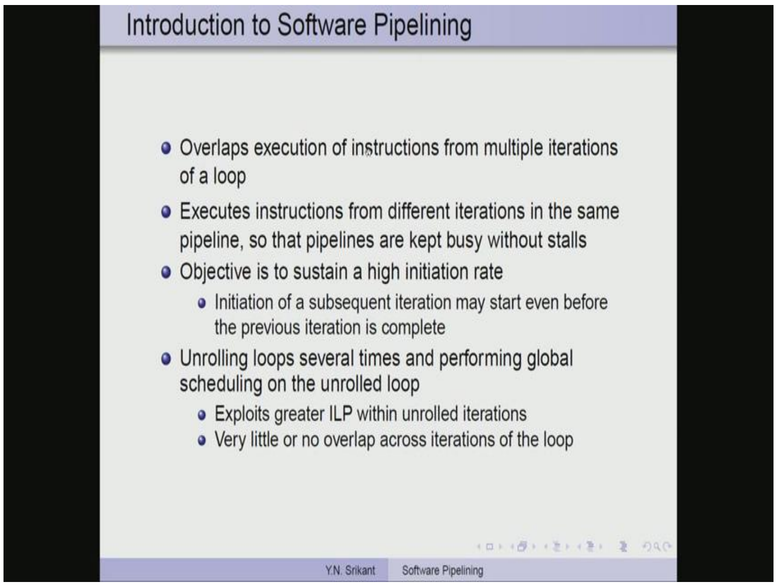
Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2':	p1 ← (r2 == 0)	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4, if p1	i8:	b(r1) ← r2, if !p1
4	i10:	r1 ← r1 + 4	i7:	r4 ← r2, if !p1
5	i9:	r5 ← r5 + r4	i11:	if (r1<r6) goto i1

(b) Hyperblock Schedule

Y.N. Srikant Instruction Scheduling

So, this is the hyper block scheduling example basically, we do an if-conversion which implies we compute the predicates and emit predicated instructions. And those predicated instructions are scheduled as if they are ordinary instructions. So, if you observe here there is absolutely no except for the last instruction here, there is no branch at all everything is a predicated instruction.

(Refer Slide Time: 13:47)



Introduction to Software Pipelining

- Overlaps execution of instructions from multiple iterations of a loop
- Executes instructions from different iterations in the same pipeline, so that pipelines are kept busy without stalls
- Objective is to sustain a high initiation rate
    - Initiation of a subsequent iteration may start even before the previous iteration is complete
- Unrolling loops several times and performing global scheduling on the unrolled loop
    - Exploits greater ILP within unrolled iterations
    - Very little or no overlap across iterations of the loop

Y.N. Srikant Software Pipelining

So, now we move on to the next technique of global scheduling so, this is called software pipelining. And here I cannot say that this is acyclic scheduling because it is essentially for loops. So, this is a cyclic scheduling strategy, but it is a global scheduling strategy because it looks at more than one basic block. So, the most important aspect of software pipelining is that it overlaps execution of instructions from multiple iterations of a loop.

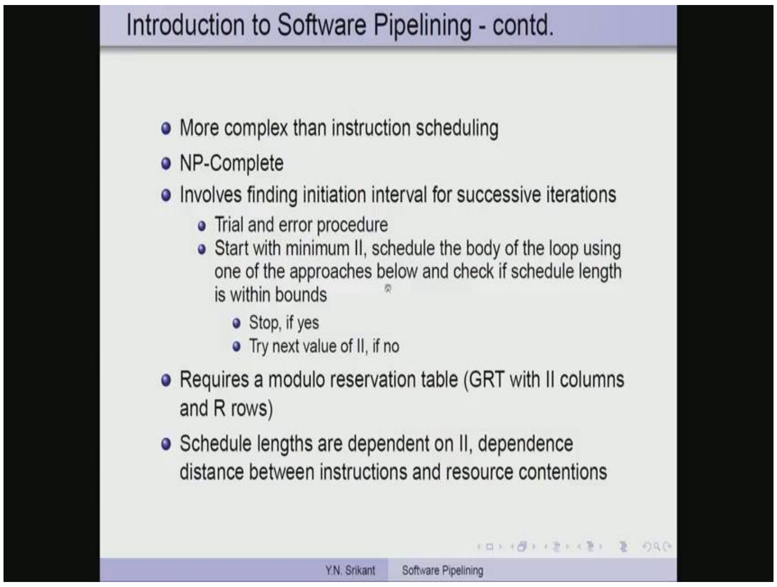
So whereas, instruction scheduling looks at exactly one iteration of a loop. Whereas, the software pipelining scheme it overlaps execution of instructions from multiple iterations. So, I am going to give you many examples of software pipelining very soon. So, it execute instructions from different iterations in the same pipeline so, that the pipelines are kept busy without stalls.

So, instruction you know scheduling actually helps in cutting down stalls, but if we take instructions from different iterations. Then there are even more possibilities for executing instructions without stalls. The objective is to sustain a high initiation rate so, initiation rate basically says how soon we can start the next iteration. So, iteration of a subsequent initiation of a subsequent iteration may start even before the previous iteration is complete.

So, one of the iterations you know which has been started may be going through many phases. So, before it is complete it has completed all the phases the second iteration may actually begin so, this is the objective of software pipelining. The other way of doing it you know appears to be unrolling loop several times and performing global scheduling on the unrolled loop.

But definitely this is much better than you know scheduling you know loops without unrolling, but there is no overlap between across the iterations of the loop. Usually, even if there is then it would be no only near the border of the loop. So, this and of course, software pipelining in general has been observed to provide much more speedup than this sort of unrolled loop scheduling.

(Refer Slide Time: 16:34)



Introduction to Software Pipelining - contd.

- More complex than instruction scheduling
- NP-Complete
- Involves finding initiation interval for successive iterations
  - Trial and error procedure
  - Start with minimum II, schedule the body of the loop using one of the approaches below and check if schedule length is within bounds
    - Stop, if yes
    - Try next value of II, if no
- Requires a modulo reservation table (GRT with II columns and R rows)
- Schedule lengths are dependent on II, dependence distance between instructions and resource contentions

Y.N. Srikant Software Pipelining

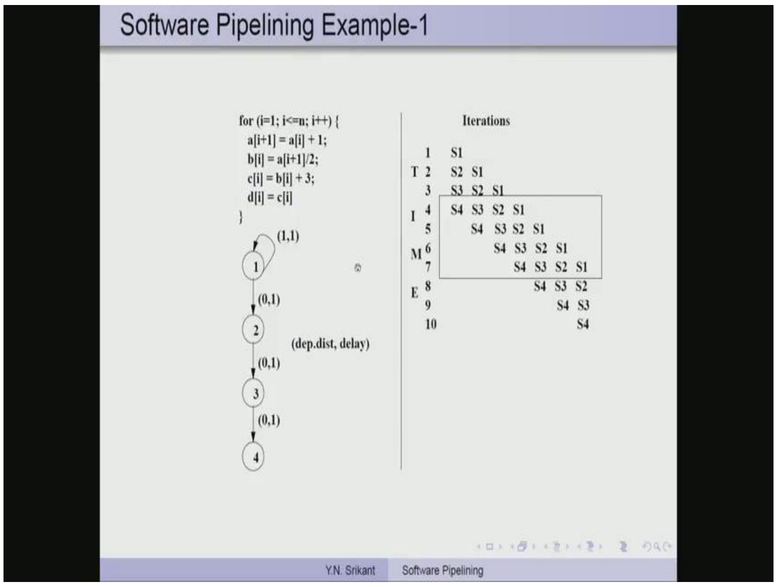
This technique is obviously more complex than instruction scheduling and just like instruction scheduling this is also an n p complete technique. So, there is no option, but to use heuristics, the basic idea involves finding what is known as an initiation interval for successive iterations. So, again initiation interval says this is the interval with which we initiate the iterations of a loop. So, if the initiation interval is 1, then we can start each iteration after 1 cycle in successive cycles we can start the successive iterations.

Whereas, if the initiation interval is say 3 then we start the iteration i and the next iteration can be started only you know in i, i is the iteration i plus 1, i plus 2 we can start it only in i plus 3. So, this is the you know significance of initiation interval, how do you find the initiation interval there is no short cut to it, it is only a trial and error procedure. So, we start with the minimum initiation interval, which can be computed using some techniques which we are not going to discuss here.

Then we schedule the body of the loop using one of the approaches below and check if the schedule length is within bounds. So, if yes stop otherwise try the next value of the initiation interval. Basically, it requires a modulo reservation table so, this is a global reservation table with i i which is the initiation interval i i number of columns and r is a number of resources r rows. So, instead of the GRT having as many rows as the rather you know as many columns as the number of the length of the schedule here, we have i i columns and r rows.

Schedule lengths are dependent on the initiation interval and dependence distance between instructions and resource contentions. So, it is not the just the precedence and resource constraints, but we have the you know schedule length being dependent on the initiation interval. The dependence distance between the instructions and the resource constraints so all these form a part of the package. So, computing the initiation interval and then checking out whether the schedule is within the bounds is the only way to find a proper initiation interval.

(Refer Slide Time: 19:27)



Software Pipelining Example-1

```
for (i=1; i<=n; i++) {
  a[i+1] = a[i] + 1;
  b[i] = a[i+1]/2;
  c[i] = b[i] + 3;
  d[i] = c[i]
}
```

$$\begin{tikzcd}
1 \arrow[loop above, "{(1,1)}"] \arrow[d, "{(0,1)}"] \\
2 \arrow[d, "{(0,1)}"] \\
3 \arrow[d, "{(0,1)}"] \\
4
\end{tikzcd}$$

(dep.dist, delay)

Iterations

TIME							
1	S1						
2	S2	S1					
3	S3	S2	S1				
4	S4	S3	S2	S1			
5		S4	S3	S2	S1		
6			S4	S3	S2	S1	
7				S4	S3	S2	S1
8					S4	S3	S2
9						S4	S3
10							S4

Y.N. Srikant Software Pipelining

So, let us take this simple example, we have a for loop i equal to 1 i less than or equal to n i plus plus. Then we have a i plus 1 equal to a i plus 1, b i equal to a i plus 1 by 2 c i equal to b i plus 3 and d i equal to c i. The dependence diagram for this the instructions in this loop is shown here, so let us say these are the 4 instructions in the loop. And the first instruction is number one and the last instruction is number 4.

So, the you know a i plus 1 is computed here and then used here so there is a dependence from 1 to 2. Then the same is true for b i and c i as well so 2 to 3 and 3 to 4 further, we have a we have a computation a i plus 1 equal to a i plus 1. So, in other words in computing i plus 1 the i plus one'th value of a we are also using the i'th value of a. So, this is the dependence on the same instruction, because this is the instruction which computes the value. So, I am computing some value and using it in the next iteration.

Now, coming to the labels on these arcs the first component of the label is the, what is known as the dependence distance and the second component is the well-known delay. So, the second one is the time required to execute the instruction whereas, the first one is the dependence distance. Dependence distance, simply says the number of iterations between the definition and the use, if you consider b i and c i, b i is computed in instruction and the value of this value which is computed here, is used in the same instruction same iteration, but in the next instruction.

So, the dependence distance between these two is really 0 because the usage is in the same iteration. Therefore, from 2 to 3 so we have a dependence distance of 0 here. Similarly, from 3 to 4 also we have a dependence distance of 0 and from 1 to 2 again both are a i plus 1. So, the dependence distance is 0 here as well, but this usage versus this computation this is from the previous iteration and this is in the current iteration. So, the dependence distance is actually 1.

So, dependence distance 1 indicates that the value computed one iteration before is being used in the current iteration. So, this is our dependence diagram and any schedule that we produce must satisfy the dependences and the delays in this diagram. So, let us see how we can schedule the instructions here, so this is the time line and these are the iterations. Now, we have the instruction s this is s 1 s 2 s 3 and s 4 so, in timeslot 1 we start s 1 so absolutely no issues there. And then in the timeslot 2 obviously s 2 executes, timeslot 3 s 3 executes and timeslot 4 s 4 executes. So, all these actually belong to the same iteration.

Now, the question is in a given enough number of resources it is possible to start the iteration number 2 in the timeslot 2 itself. That is, the first instruction of iteration 2 can it be started in timeslot 2, concurrently with the second instruction of iteration number 1, the answer is yes. Let us, look at the dependence diagram to understand why, this is s 2, this is s 1.

So, between these two the dependence distance is actually 0 and so there is absolutely no problem in executing s 2 and s 1 together all right, but now the s 2 of course, belongs to the previous iteration and s 1 belongs to the next iteration. So, otherwise we could not have, we actually could not have started s 1 of the second iteration in this timeslot 1. That is not possible, but that is because from you know from s 1 there is a self-loop on itself with a dependence distance of 1.

So, in other words the value computed in iteration number 1 and the instruction number s 1 is required for the next iteration instruction number s 1. So, and it requires one cycle to complete s 1 so, we could not have started s 1 here for the second iteration. We can start s 1 only in the second cycle so, s 1 of the second iteration we can start it only in the second timeslot. So, that should be clear.

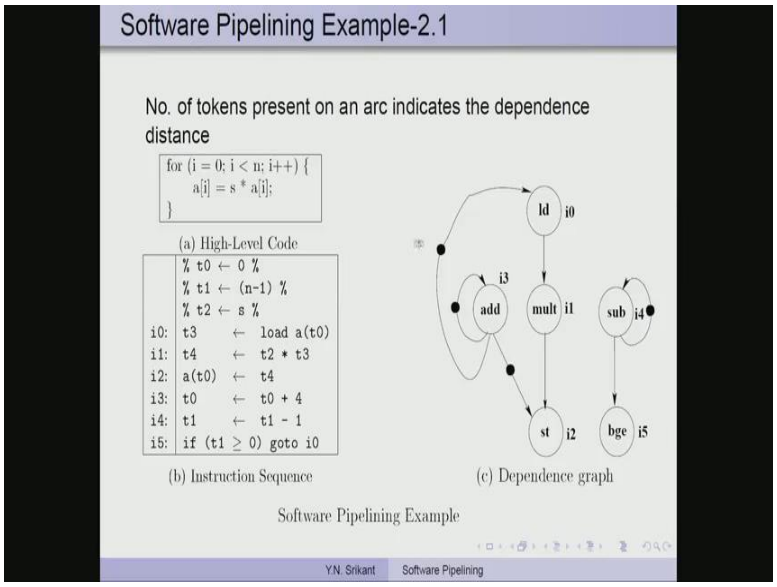
Now, there are no resource constraints so this thing completes on its own in a on the hardware. Now, this also begins its execution in time step 3 we have s 2 of iteration 2, in time step 4 s 3 and time step 5 we have s 4. Now, the same question can be asked again in time step 3 can we start another iteration concurrently with s 3 of iteration 1 and s 2 of iteration 2, the answer again is yes, if there are enough resources.

Of course, as I told you before we could not have started s 1 here because of this dependence distance being 1. So, this is the and of course, we could not have executed s 1 s 2 s 3 s 4 in parallel because of these dependences. This is strictly sequential and again we have to wait for one cycle to start another s 1, but once started we can continue. So, the same is true for time step 4 we start another s 1 here.

Then onwards you know it is a kind of a stable situation steady state, in time step 5 the iteration number 1 has completed because this is the last instruction of iteration number 1. So, this has actually completed, what we are left with is only the instruction number s 4 of iteration 2, s 3 of iteration 3, s 2 of iteration 4 and s 1 of iteration 5. So, this situation continues here, this completes so this iteration is over. So, at any point in time if you observe this steady state there are only four instructions which are executing start you know. So, this is the most recent instruction and this is the last instruction now latest you know or the last instruction.

And each of these instructions belongs to a different iteration number so and this s 4 s 3 s 2 s 1 is the software pipeline that we are trying to understand. Here, the initiation interval is 1 because we have been able to initiate a new iteration in every cycle. So, in fact the software pipeline consists of just these 4 instructions. And assuming that there are enough resources to take care of all the instructions here, this all the four can execute in parallel. So, this is the concept of software pipeline, so let us go further and take another example.

(Refer Slide Time: 28:17)



Software Pipelining Example-2.1

No. of tokens present on an arc indicates the dependence distance

```
for (i = 0; i < n; i++) {
    a[i] = s * a[i];
}
```

(a) High-Level Code

```
     % t0 ← 0 %
     % t1 ← (n-1) %
     % t2 ← s %
i0:  t3     ←  load a(t0)
i1:  t4     ←  t2 * t3
i2:  a(t0)  ←  t4
i3:  t0     ←  t0 + 4
i4:  t1     ←  t1 - 1
i5:  if (t1 ≥ 0) goto i0
```

(b) Instruction Sequence

(c) Dependence graph

Software Pipelining Example

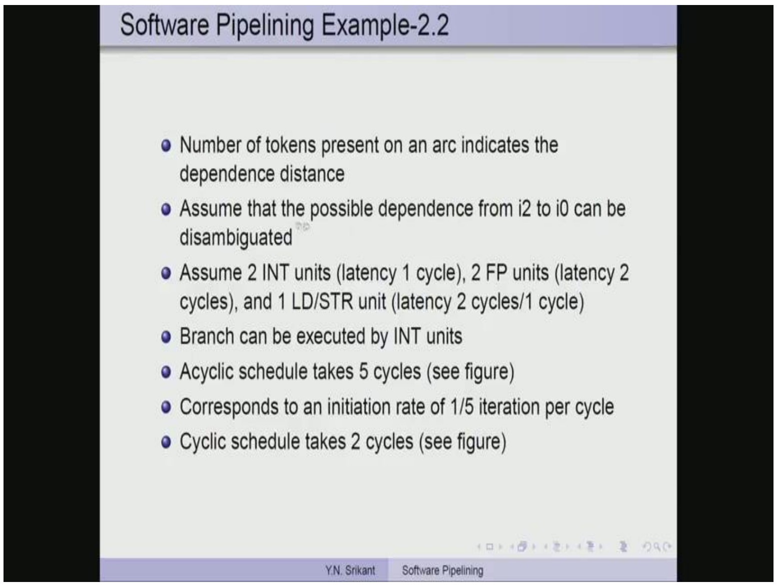
Y.N. Srikant | Software Pipelining

Here, again the example is quite simple a i equal to s star a i and here is the machine code corresponding to it. So, we have load and then we have a multiply and then we have a store instruction. Then the other instructions correspond to the loop increment value etcetera, etcetera, then we are checking the loop and going back to i 0. If the loop is not yet complete we keep iterating and then we get out. The dependence graph for this small program is shown here.

So, as usual the dependences are shown by these arcs and the dots on the you know arcs are the tokens present on the arc, indicates the dependence distance. So, here i 3 actually supplies a value to i 0 we can see that, i 3 supplies a value to i 0. And this single token indicates that the value computed in this iteration is used by i 0 in the next iteration. So, dependence distance of 1, i 3 also supplies a value to itself and with a dependence distance of 1. So, i 3 is actually t naught equal to t naught plus 4. So, this is the value from the previous iteration and this is the new computed value.

So, there is a self-loop, self-dependence and the dependence distance is also 1, so because of these two. Similarly, from i 3 to i 2 there is a dependence so, this is i 3 and this is i 2 with a dependence distance of 1 again. So, the a t 0 which is used here is you know we compute it here in iteration i and use it in the next iteration i plus 1. So, this is the way the dependences are to be understood. So, let us understand how to schedule these.

(Refer Slide Time: 30:43)



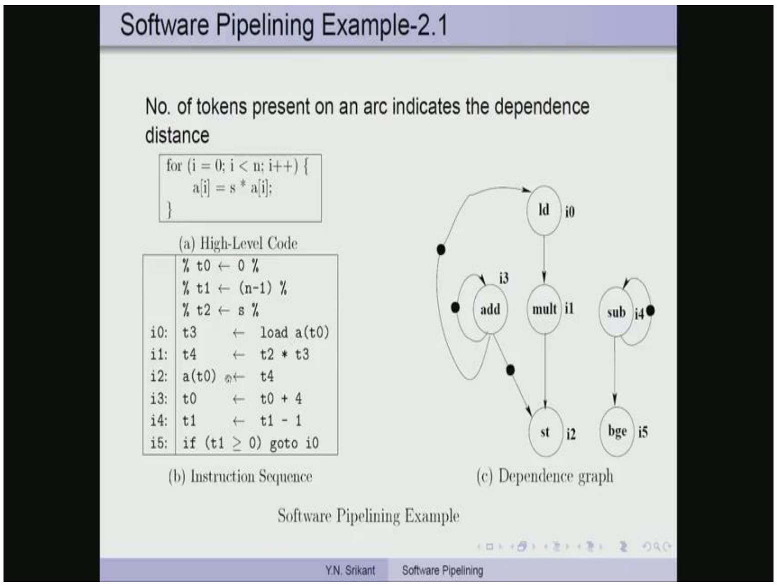
Software Pipelining Example-2.2

- Number of tokens present on an arc indicates the dependence distance
- Assume that the possible dependence from i2 to i0 can be disambiguated
- Assume 2 INT units (latency 1 cycle), 2 FP units (latency 2 cycles), and 1 LD/STR unit (latency 2 cycles/1 cycle)
- Branch can be executed by INT units
- Acyclic schedule takes 5 cycles (see figure)
- Corresponds to an initiation rate of 1/5 iteration per cycle
- Cyclic schedule takes 2 cycles (see figure)

Y.N. Srikant Software Pipelining

So, the number of tokens indicates the dependence distance this is something which I already explained. Assume, that the possible dependence from i 2 to i 0 can be disambiguated i 2 to i 0.

(Refer Slide Time: 31:00)



Software Pipelining Example-2.1

No. of tokens present on an arc indicates the dependence distance

```
for (i = 0; i < n; i++) {
    a[i] = s * a[i];
}
```

(a) High-Level Code

```
     % t0 ← 0 %
     % t1 ← (n-1) %
     % t2 ← s %
i0:  t3      ←  load a(t0)
i1:  t4      ←  t2 * t3
i2:  a(t0)   ←  t4
i3:  t0      ←  t0 + 4
i4:  t1      ←  t1 - 1
i5:  if (t1 ≥ 0) goto i0
```

(b) Instruction Sequence

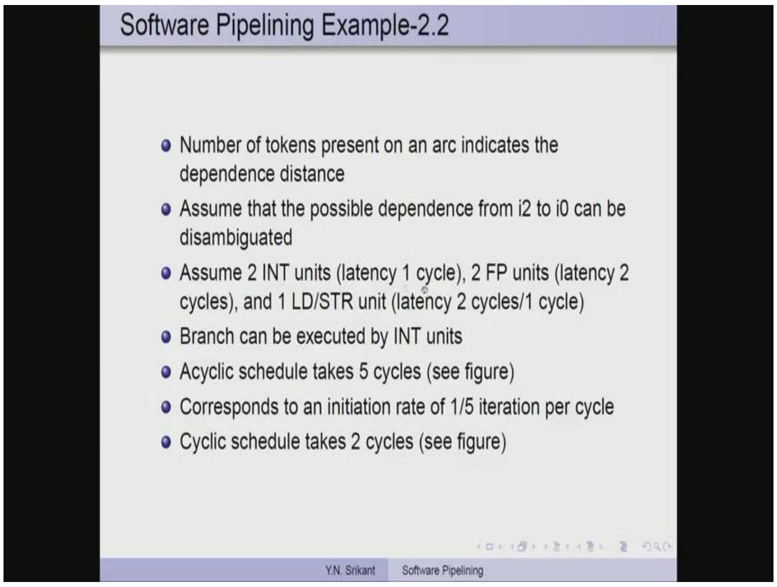
(c) Dependence graph

Software Pipelining Example

Y.N. Srikant Software Pipelining

So, here is t 4 and we are writing into the same location t 4. So, let us assume that we can disambiguate it so, just for the sake of example so i 2 to i 0. So, this is i 2 and here is i 0 so that is why there is no dependence that we have shown between the two.

(Refer Slide Time: 31:26)



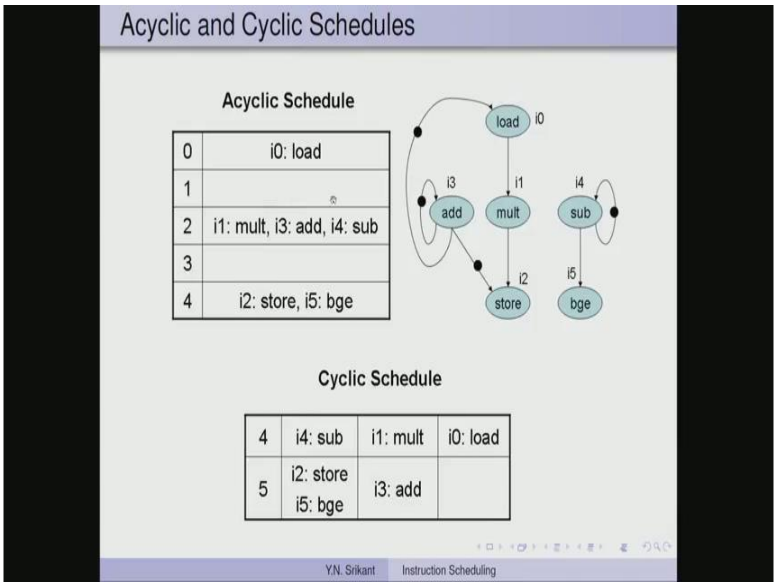
Software Pipelining Example-2.2

- Number of tokens present on an arc indicates the dependence distance
- Assume that the possible dependence from i2 to i0 can be disambiguated
- Assume 2 INT units (latency 1 cycle), 2 FP units (latency 2 cycles), and 1 LD/STR unit (latency 2 cycles/1 cycle)
- Branch can be executed by INT units
- Acyclic schedule takes 5 cycles (see figure)
- Corresponds to an initiation rate of 1/5 iteration per cycle
- Cyclic schedule takes 2 cycles (see figure)

Y.N. Srikant Software Pipelining

So, assume there are two integer units with a latency of 1 cycle, 2 floating point units latency 2 cycles and then we have one load store unit with a latency of 2 cycles and 1 cycle each. So, load has 2 cycles latency and the store unit has 1 cycle latency. The branch can be executed by integer units, so the acyclic schedule takes 5 cycles so we let me show you the picture.

(Refer Slide Time: 32:00)



Acyclic and Cyclic Schedules

Acyclic Schedule

0	i0: load
1	
2	i1: mult, i3: add, i4: sub
3	
4	i2: store, i5: bge

load
i0
i3
add
i1
mult
i4
sub
i2
store
i5
bge

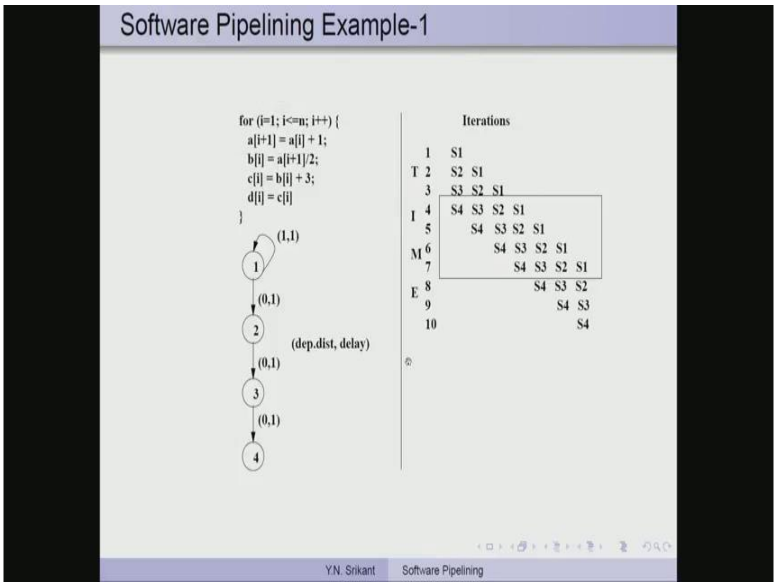
Cyclic Schedule

4	i4: sub	i1: mult	i0: load
5	i2: store i5: bge	i3: add	

Y.N. Srikant Instruction Scheduling

So, here is the acyclic schedule which has i 0 then there is a knop instruction and finally, i 2 there is another knop instruction here and then i 2 and i 5. So, in the timeslot 2 there is i 1 i 3 and i 4 whereas, in timeslot 4 there is i 2 and i 5. Compare to this if we do a software pipeline then we really require only 2 cycles. So, by the way before this we have instructions to fill up the pipeline and after this we have instructions to empty the pipeline.

(Refer Slide Time: 32:47)



Software Pipelining Example-1

```
for (i=1; i<=n; i++) {
  a[i+1] = a[i] + 1;
  b[i] = a[i+1]/2;
  c[i] = b[i] + 3;
  d[i] = c[i]
}
```

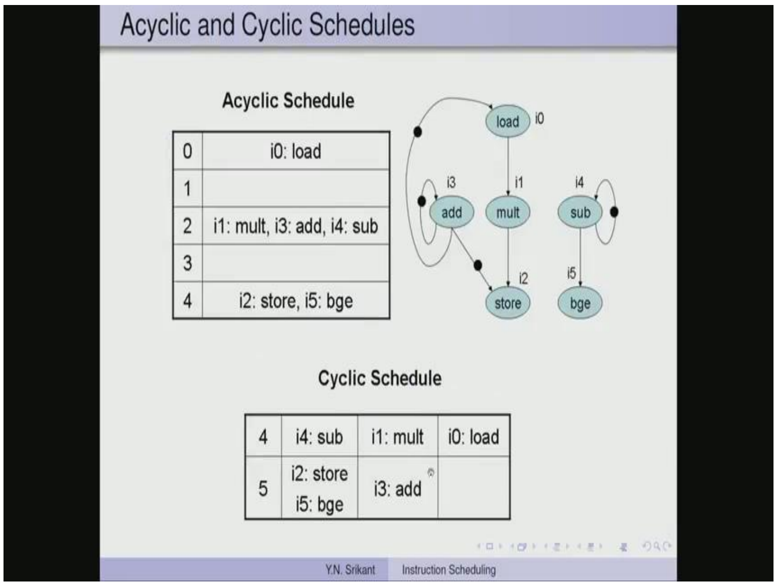
(1,1)
1
(0,1)
2
(dep.dist, delay)
(0,1)
3
(0,1)
4

```
              Iterations
     1   S1
  T  2   S2  S1
     3   S3  S2  S1
  I  4   S4  S3  S2  S1
     5       S4  S3  S2  S1
  M  6           S4  S3  S2  S1
     7               S4  S3  S2  S1
  E  8                   S4  S3  S2
     9                       S4  S3
    10                           S4
```

Y.N. Srikant Software Pipelining

So, let me show you that in the previous example so, let us assume that that entire loop completes in ten time units. So, actually this is these are all the instructions which are required to fill the pipeline. Now, the pipeline is full at this point it continues in that state until here and once the you know, the pipeline cannot be sustain and the loop starts coming to an end we execute the rest of the iterations in this epilogue. So, this is the prologue this is the epilogue and this is the kernel or the steady state of the pipeline.

(Refer Slide Time: 33:30)



Acyclic and Cyclic Schedules

Acyclic Schedule

0	i0: load
1	
2	i1: mult, i3: add, i4: sub
3	
4	i2: store, i5: bge

load
i0
i3
add
i1
mult
i4
sub
i2
store
i5
bge

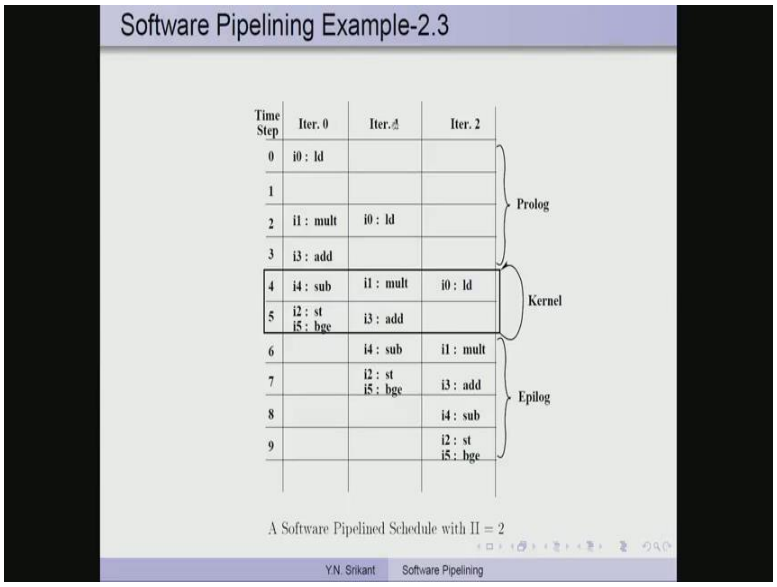
Cyclic Schedule

4	i4: sub	i1: mult	i0: load
5	i2: store i5: bge	i3: add	

Y.N. Srikant Instruction Scheduling

So, let me show you how it requires only 2 cycles instead of the acyclic normal basic blocks schedule of 5 cycles.

(Refer Slide Time: 33:40)



Software Pipelining Example-2.3

Time Step	Iter. 0	Iter. 1	Iter. 2	
0	i0 : ld			Prolog
1				Prolog
2	i1 : mult	i0 : ld		Prolog
3	i3 : add			Prolog
4	i4 : sub	i1 : mult	i0 : ld	Kernel
5	i2 : st i5 : bge	i3 : add		Kernel
6		i4 : sub	i1 : mult	Epilog
7		i2 : st i5 : bge	i3 : add	Epilog
8			i4 : sub	Epilog
9			i2 : st i5 : bge	Epilog

A Software Pipelined Schedule with II = 2

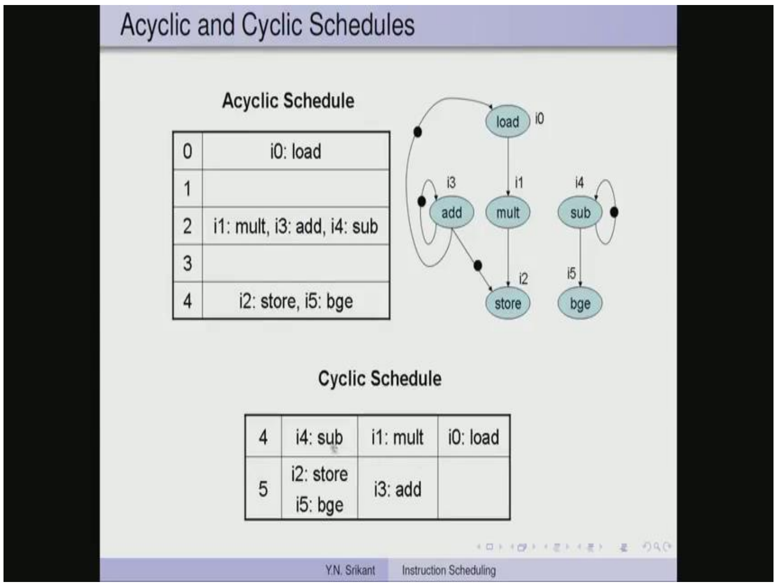
Y.N. Srikant Software Pipelining

So, this is iteration 0, iteration 1 and iteration 2 and here are the time steps required for this pipeline execution. So, iteration 0 we have this schedule so as usual this is a you know there is a load here, then the knop, multiply, add then sub store branch greater than equal to. So, this is iteration 0 and here are the instructions which have been scheduled. And now, at time step 2 we can actually initiate i 0 of the iteration 1.

So, this iteration now starts executing in the pipelines, these two execute in a parallel fashion. There is nothing to execute here in parallel and then this and this execute in parallel this and this execute in parallel. And in at time step 4 now we can actually initiate iteration 2 so, this again has these instructions following it. Now, if you observe this is our steady state right so, if we actually have another instruction, which is going to be you know for the iteration 3 that would be here.

So, again we see the same pattern so sub mult and then load and here store b g e and add. So, these two instructions actually form the kernel or the steady state of the pipeline. So, we are going to iterate over this steady state of the pipeline as long as the loop executes and finally, the epilogue part of it executes the rest of the iterations.

(Refer Slide Time: 35:34)



Acyclic and Cyclic Schedules

Acyclic Schedule

0	i0: load
1	
2	i1: mult, i3: add, i4: sub
3	
4	i2: store, i5: bge

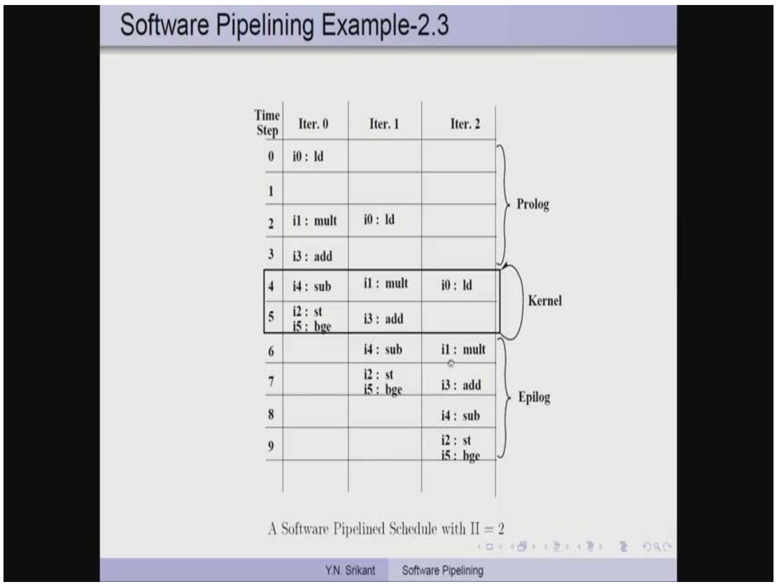
Cyclic Schedule

4	i4: sub	i1: mult	i0: load
5	i2: store i5: bge	i3: add	

Y.N. Srikant Instruction Scheduling

So, this is what was shown here it requires three cycles 0 1 2 3 4 cycles to actually fill the pipeline that is the prologue and then from 4 onwards, there is a steady state which executes many times.

(Refer Slide Time: 35:52)



Software Pipelining Example-2.3

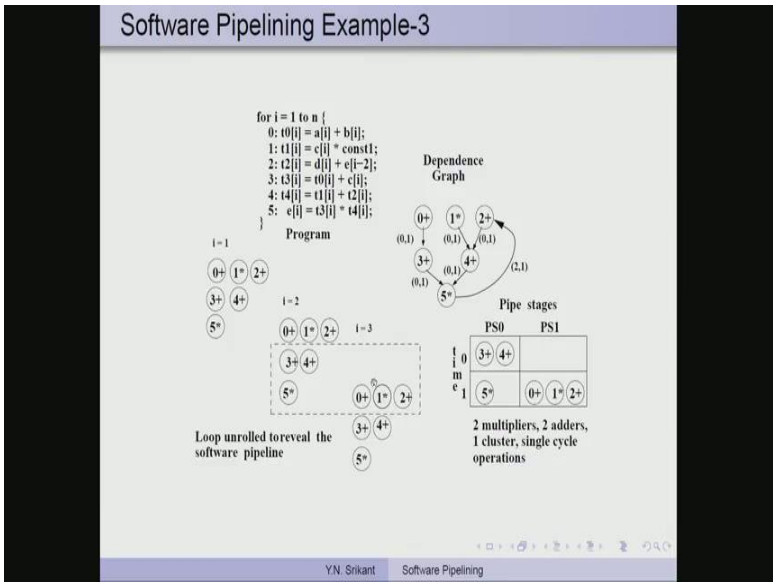
Time Step	Iter. 0	Iter. 1	Iter. 2	
0	i0 : ld			Prolog
1				
2	i1 : mult	i0 : ld		
3	i3 : add			
4	i4 : sub	i1 : mult	i0 : ld	Kernel
5	i2 : st i5 : bge	i3 : add		
6		i4 : sub	i1 : mult	Epilog
7		i2 : st i5 : bge	i3 : add	
8			i4 : sub	
9			i2 : st i5 : bge	

A Software Pipelined Schedule with II = 2

Y.N. Srikant Software Pipelining

And then an epilogue of a couple of cycles to flush the entire pipeline. So, this is the way a software pipeline loop executes, so in the steady state is executing many instructions from different iterations and the pipeline stages are all these are the various pipeline stages. This is the first stage this is the second stage and this is the third stage.

(Refer Slide Time: 36:17)



Software Pipelining Example-3

```
for i = 1 to n {
   0: t0[i] = a[i] + b[i];
   1: t1[i] = c[i] * const1;
   2: t2[i] = d[i] + e[i−2];
   3: t3[i] = t0[i] + c[i];
   4: t4[i] = t1[i] + t2[i];
   5:   e[i] = t3[i] * t4[i];
}
```

Program

Loop unrolled to reveal the software pipeline

2 multipliers, 2 adders, 1 cluster, single cycle operations

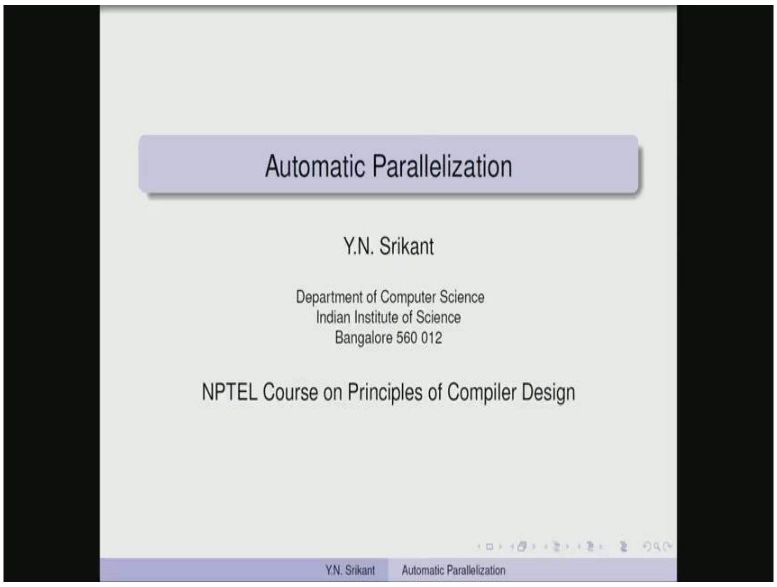
Y.N. Srikant Software Pipelining

So, let us take another example here again we have you know 0 1 2 3 4 5 6 instructions, this is the dependence diagram for these instructions so and there is a cycle as well. So of course, if we unroll you know and start scheduling the instructions here, according to the software pipeline. So, we can execute 0 1 2 in one cycle 3 4 in another cycle and 5 in the next cycle because that is indicated by the dependences here without looking at this.

And because of this we can actually begin the next iteration only concurrently with 5. So, that is what we really do here, so i equal to 2 we have again the same pattern 3 4 and 5, i equal to 3 we have again 0 1 2 3 4 and 5. So, this will turn out to be the steady state for our software pipeline in which we have 3 4 executing, you know in time step 1 and 5 0 1 2 all of them executing in the time step number 2.

So, this is actually driven by the resource constraints as well so, we have two multipliers, two adders, one cluster single cycle operation. So, this make sure that we execute the instructions in this fashion. So, you know this is the software pipeline loop which is executing in a steady state. And of course, I have shown you some prologue instructions and epilogue as well here.

(Refer Slide Time: 38:03)



Automatic Parallelization

Y.N. Srikant

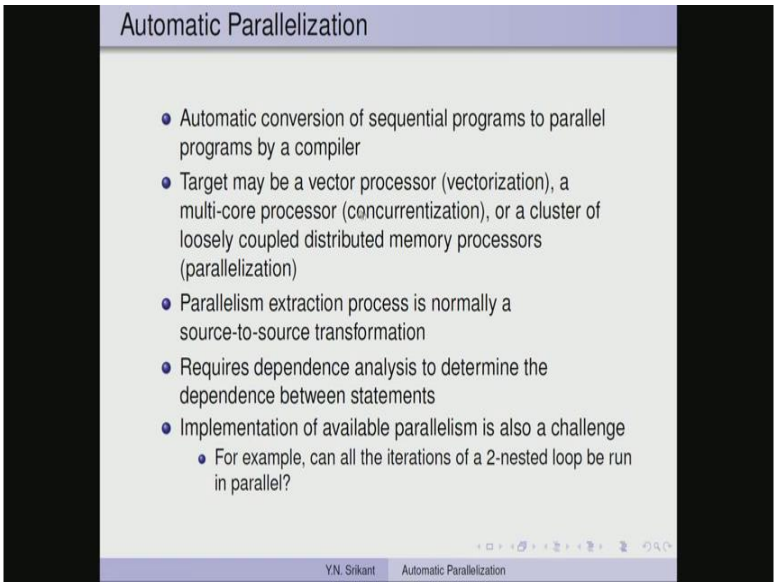
Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Y.N. Srikant Automatic Parallelization

Now, that is the end of software pipelining instruction scheduling etcetera. Now, we move on to the next topic, which is very important called as the automatic parallelization.

(Refer Slide Time: 38:16)



Automatic Parallelization

- Automatic conversion of sequential programs to parallel programs by a compiler
- Target may be a vector processor (vectorization), a multi-core processor (concurrentization), or a cluster of loosely coupled distributed memory processors (parallelization)
- Parallelism extraction process is normally a source-to-source transformation
- Requires dependence analysis to determine the dependence between statements
- Implementation of available parallelism is also a challenge
  - For example, can all the iterations of a 2-nested loop be run in parallel?

Y.N. Srikant Automatic Parallelization

So, why do we require automatic parallelization and what is the process. So, automatic parallelization is the automatic conversion of sequential programs to parallel programs by a compiler. So, in other words the programmer does not write a parallel program, the programmer writes a sequential program and an automatically parallelising a compiler converts the sequential program to parallel programs. So, this is the purpose of automatic parallelization.

The target machine may be a vector processor so, in which case it is called as a vectorization. It could be a multi core processor in which case it is called as concurrentization or a cluster of loosely coupled distributed memory processors, in which case it is called as parallelization. Of course, we use parallelization and concurrentization you know with the same meaning we do not really differentiate too much between them, but vectorization definitely is a different process.

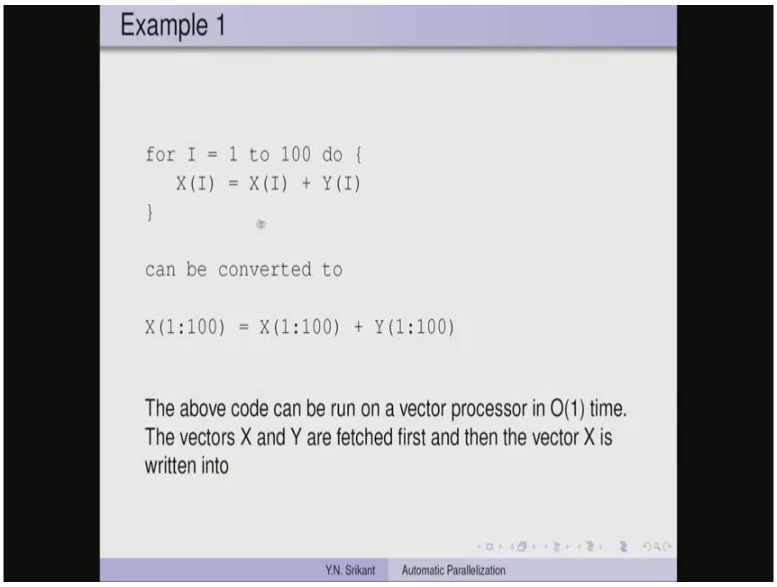
So, we are going to see examples of vectorization and also parallelization. So, why is vectorization relevant at all you know some of the single core, even single core processors of the x 86 variety? They actually have a small vector set of instructions for multimedia operations. So, even those can be used if we are able to perform some vectorization the efficiency of the program will thereby increase.

Parallelism extraction process is normally a source to source transformation. So, in other words if we take c or Fortran code the output is also c or Fortran. It is not as if we go through the entire process of till the intermediate code generation and then perform parallelism extraction. In fact some, of the parallelism may not be so easily visible at the lower levels. So, this is the reason why we want to perform parallelism extraction at the source level itself.

It requires a technique called dependence analysis to determine the dependence between the statements. Informally, I have already shown you many examples in the instruction scheduling and software pipelining and of dependence diagrams, but we have still not learnt how to determine the dependences. So, this is a fairly complicated process I am going to give you only a flavour of dependences in this lecture.

The implementation of available parallelism is also a challenge. So, you know if you have a multi core processor say with you know 8 cores, then it is easy to see that 8 iterations can run in parallel on this multi core processor, but suppose we have a 2 nested loop right. Then is it possible to really run both the loops, the outer loop and the inner loop in parallel mode well it is not so easy. In fact we know how to run single loops in parallel, but running nested loops in parallel would be a very difficult task because of resource constraints.

(Refer Slide Time: 42:01)



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)
```

The above code can be run on a vector processor in O(1) time. The vectors X and Y are fetched first and then the vector X is written into

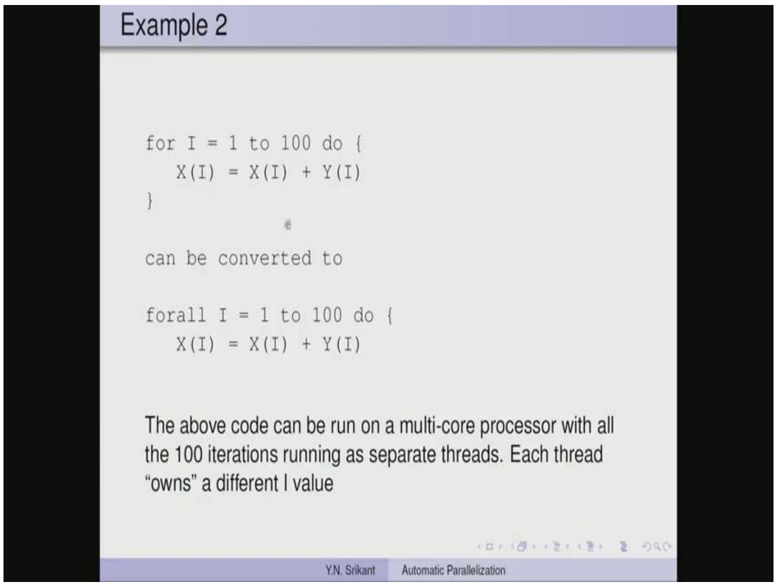
Y.N. Srikant Automatic Parallelization

So, let us look at some examples here is a very simple loop for i equal to 1 to 100 x i equal to x i plus y i. So, if we have some vector instructions on the machine then this code can be very easily converted to this vector code. So, this can be read as the vector x 1 to 100 is vector x 1 to 100 plus the vector y to 1 to 100. So, assuming that there are vectors of length 100 we basically, read x 1 to 100 into a vector set of registers. And add this you know all the vector registers are parallely added.

So, 1 to 100 of x will be added to 1 to 100 of y in a parallel manner so all this can happen in one time. And then in the next cycle we can actually store the value into x so, if the instruction permits direct addition into the same register, then there is no need to actually write it back to the memory location. So, the vectors x and y are fetched first so, that is very important.

So, usually the vector set of registers are used and then they are stored back into memory as well. So, this is very important this is read first, this is read again and then the computation happens. So, there is a overwriting all right, but the values from the previous iteration are not used in the current iteration. So, there is no dependence from one iteration to the next iteration so that is very important.

(Refer Slide Time: 44:00)



Example 2

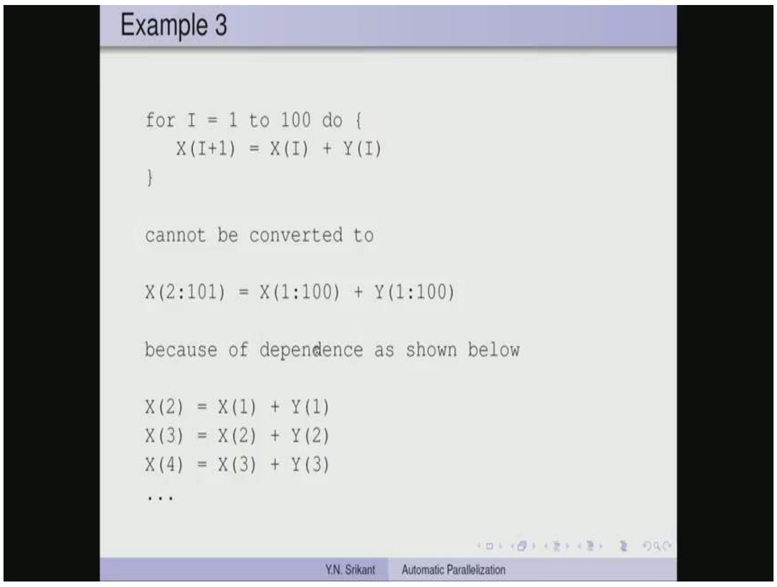
```
for I = 1 to 100 do {
   X(I) = X(I) + Y(I)
}

can be converted to

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

The above code can be run on a multi-core processor with all the 100 iterations running as separate threads. Each thread "owns" a different I value

Y.N. Srikant Automatic Parallelization

If we want to run the same code on a multi core processor we can do that. Assume, that we can start 100 threads so, each one of the iterations can actually become an independent thread. And for each value of i 1 to hundred there would be a separate thread, which does the addition x i equal to x i plus y i. None, of these interfere with each other that is very clear each iteration is different and each thread will do the work of just that iteration. So, this can be run in parallel on a multi core processor as well.

(Refer Slide Time: 44:46)



Example 3

```
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)

because of dependence as shown below

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

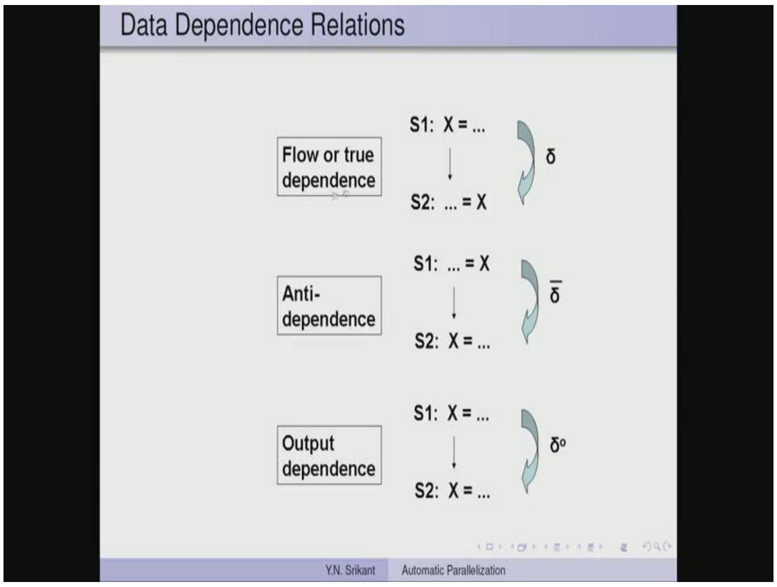
Y.N. Srikant Automatic Parallelization

Now, suppose there is a change in the code so, this is x i plus 1 becoming is being assigned value of x i plus y i. So, this cannot be converted to this vector code, that is even though it simply appears as i plus 1, it looks like we are computing x 2 here we are using x 1 here and y 1 here. So, if we simply write this as x 2 to 1 0 1 x of 1 to 100 plus y of 1 to 100 it would be incorrect. The reason is there is a dependence here that becomes very clear when we expand the loop.

So, this first iteration becomes x 2 equal to x 1 plus y 1, the second iteration becomes x 3 equal to x 2 plus y 2, the third iteration is x 4 equal to x 3 plus y 3, etcetera. So, observe that what is computed in the first iteration is being used in the second iteration, what is being computed in the second iteration is being used in the third iteration. So, there is a dependence of values from first to second from second to third etcetera, etcetera.

So whereas, this particular code does not actually respect this dependence that we have shown here. This says, read the value of x from 1 to 100 the vector, read the value of y 1 to 100 and then add them and put it into 2 to 1 0 1. So, in other words the value which is computed in a particular iteration is not being used in the next iteration, but the old values are being used. So therefore, this vector assignment is incorrect.

(Refer Slide Time: 46:40)



Data Dependence Relations

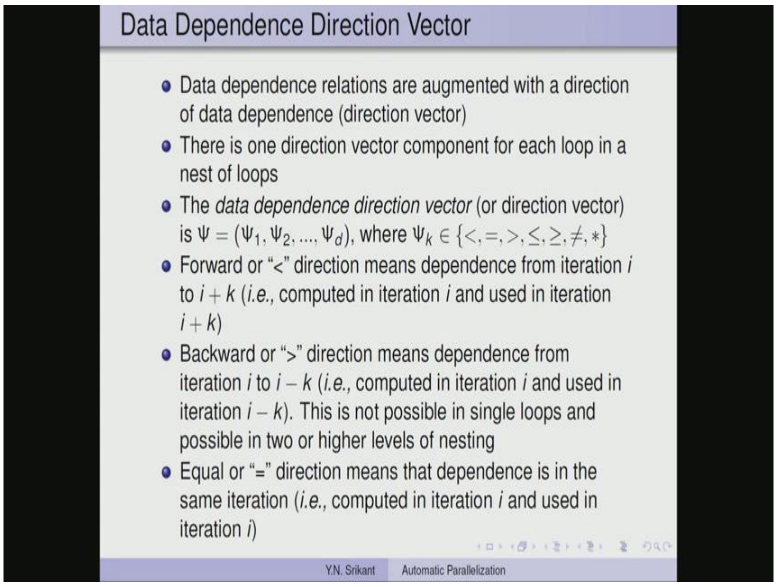
Flow or true dependence	S1: X = ... → S2: ... = X	δ
Anti-dependence	S1: ... = X → S2: X = ...	$\bar{\delta}$
Output dependence	S1: X = ... → S2: X = ...	δ^o

Y.N. Srikant Automatic Parallelization

Just to do a bit of recap on the data dependence relations, so if we have an assignment to a scalar variable x and then we have an read of x and there are no more assignments to x here. Then it is a flow dependence or true dependence. If we have a read of x here and then a write into x here and there are no other writes into x here.

Then this is known as an anti-dependence. And the output dependence is similar we have two writes nothing. In between no other writes in between, then from s 1 to s 2, we have an output dependence here. This is anti-dependence from s 1 to s 2 and this is flow dependence from s 1 to s 2.

(Refer Slide Time: 47:32)



Data Dependence Direction Vector

- Data dependence relations are augmented with a direction of data dependence (direction vector)
- There is one direction vector component for each loop in a nest of loops
- The *data dependence direction vector* (or direction vector) is $\Psi = (\Psi_1, \Psi_2, ..., \Psi_d)$, where $\Psi_k \in \{<, =, >, \leq, \geq, \neq, *\}$
- Forward or "<" direction means dependence from iteration i to $i + k$ (*i.e.*, computed in iteration i and used in iteration $i + k$)
- Backward or ">" direction means dependence from iteration i to $i - k$ (*i.e.*, computed in iteration i and used in iteration $i - k$). This is not possible in single loops and possible in two or higher levels of nesting
- Equal or "=" direction means that dependence is in the same iteration (*i.e.*, computed in iteration i and used in iteration i)

Y.N. Srikant Automatic Parallelization

Then we also have to understand, the notion of a data dependence direction vector you know, what data dependence relations are. We augment this information with a direction of data dependence, this is a vector called the direction vector. So, I will give you examples to show what this is so, there is one direction vector component for each loop in a nest of loops.

So, if it is a 3 nested loops so, in other words there is one loop outer, another loop inner and third loop which is inside the second. Then the direction vector will have three components one for each of these loops. Then the data dependence direction vector is written as psi equal to psi 1 comma psi 2 etcetera psi d, where is d is the depth of nesting. So, this makes it one component for each level of nesting.

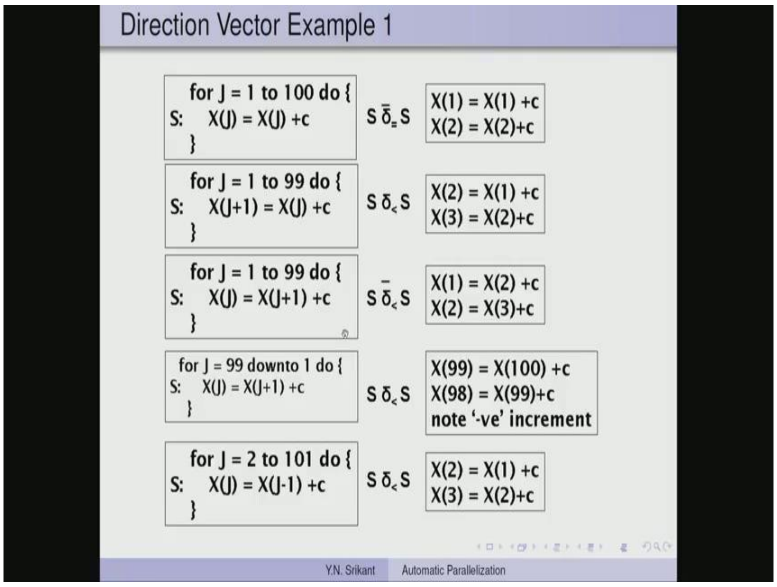
And each of these psi k's can be one of these six less than, equal to, greater than, less than or equal to, greater than or equal to, not equal to and star. Out of these the primary direction vector components are less than equal to and greater than. This less than or equal to is a combination of these two, greater than or equal to is a combination of these two, not equal to is none of these and star may be any one of these.

So, the last three are last four are basically combinations of these. So, we must basically understand less than equal to greater than in detail and the rest automatically can be understood. There are 3 types of directions possible so, that is what is shown here less than equal to and greater than. Less than is called as a forward direction, which means that the dependence from is from iteration i to i plus k. That is, we compute a value in iteration i and use it in iteration i plus k, k being positive.

So, if the loop is running backwards even then you know the iterations actually the values may run backwards, but the iterations always proceed. So, if we number the iterations as 1 2 3 4 etcetera, then we take i and then go to k, plus k so k is always positive. So, this is the forward direction.

The second is the backward direction, which means we compute in i and use it in i minus k, well looks ridiculous right. So, it is true that this is not possible in single loops, but in 2 or higher levels of nesting this is definitely possible and I am going to give you some examples later. The third type of direction is the equal to direction, which means that the dependence is in the same iteration. That is computed in iteration i and used in iteration i.

(Refer Slide Time: 51:06)



Direction Vector Example 1

```
  for J = 1 to 100 do {
S:    X(J) = X(J) +c
  }
```

$S\ \bar{\delta}_= S$

X(1) = X(1) +c
X(2) = X(2)+c

```
  for J = 1 to 99 do {
S:    X(J+1) = X(J) +c
  }
```

$S\ \delta_< S$

X(2) = X(1) +c
X(3) = X(2)+c

```
  for J = 1 to 99 do {
S:    X(J) = X(J+1) +c
  }
```

$S\ \bar{\delta}_< S$

X(1) = X(2) +c
X(2) = X(3)+c

```
  for J = 99 downto 1 do {
S:    X(J) = X(J+1) +c
  }
```

$S\ \delta_< S$

X(99) = X(100) +c
X(98) = X(99)+c
note '-ve' increment

```
  for J = 2 to 101 do {
S:    X(J) = X(J-1) +c
  }
```

$S\ \delta_< S$

X(2) = X(1) +c
X(3) = X(2)+c

Y.N. Srikant Automatic Parallelization

So, let us understand the less than equal to less than and equal to greater than can be understood only with respect to doubly nested loops. So, we have a loop j equal to 1 to 100, the statement is x j equal to x j plus c. So, let us expand the loop twice x 1 equal to x 1 plus c, x 2 equal to x 2 plus c so, here you know we have actually used x 1 and then computed into x 1.

So, the x 1 is being and we are not using x 1 again in any other iteration. So, we are using and then computing so, that means it is an anti-dependence so delta bar. And since the iteration in which we are doing it is the same iteration right. Here, in this case it is iteration number 1 here, it is iteration number 2 etcetera. So, the direction vector has the component equal to, indicating that the value is used and then computed into in the same iteration.

This is a single loop so we have only one direction vector component. Here, we have j equal to 1 to 99, x j plus 1 equal to x j plus c again when we unroll we find this as x 2 equal to x 1 plus c and this as x 3 equal to x 2 plus c. So, we have produced a value x 2 in 1 of the iterations and in a next iteration we are using it. So, there is a flow dependence between these 2 values that is easy to see.

Now, this is iteration 1 and this is iteration 2 so that means, we are producing in iteration i and using it in iteration i plus 1. That would be the direction vector would be less than and the dependence is flow dependence delta. So, we indicate it as s delta less than s saying, the value which is computed by this statement s in a particular iteration i is going to be used in a later iteration in this case of course, it is i plus 1.

The third example, j equal to 1 to 99 do, x j equal to x j plus 1 plus c again when we unroll we find that this is x 1 equal to x 2 plus c and this is x 2 equal to x 3 plus c. So, we have used x 2 here and then computed x 2, this iteration 1 and this is iteration 2. So, there is a there is an anti-dependence between these two and the iteration number is 1 here and 2 here. So, this is use first and computed later, it is an anti-dependence with a direction less than because the usage happens first and then computation in a later iteration number.

Just to give you an example, of a loop which runs backwards j equal to 99 down to 1, x j equal to x j plus 1 plus c. So, unrolling again we have x 99 equal to x 100 plus c x 98 equal to x 99 plus c etcetera. So, the loop the iterations are going forward, but the increment is negative. So, observe here that in this iteration again if we number the iterations as 1 2 3 etcetera, in this iteration we compute and in the next iteration we use.

So, even though the loop is running with a negative increment since the loop is running forward you know the iterations are increasing, we have a flow dependence with a less than from s to s. So, compute in a particular iteration use it in the in a later iteration. And this example shows j equal to 2 to 1 0 1 of x j equal to x j minus 1 plus c.

The idea of all these examples is to you know make you familiar with the usual type of subscripts. That are used in various automatically parallelising you know rather the loops which can be automatically parallelize. Again, this is x 2 equal to x 1 plus c and this is x 3 equal to x 2 plus c. So, there is a flow dependence with a forward direction so, this is s delta less than s. So, we will stop here and continue with rest of the parallelization in the next part of the lecture.

Thank you.