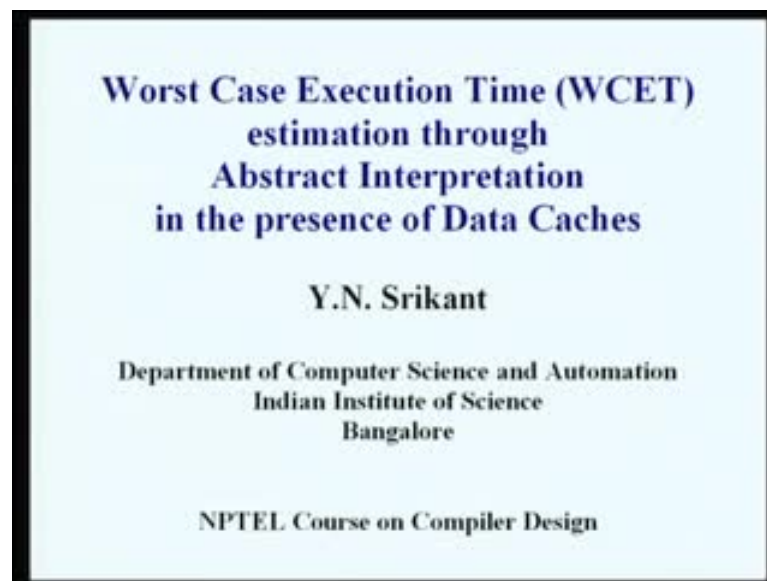


**Compiler Design**  
**Prof. Y. N. Srikant**  
**Department of Computer Science and Automation**  
**Indian Institute of Science, Bangalore**

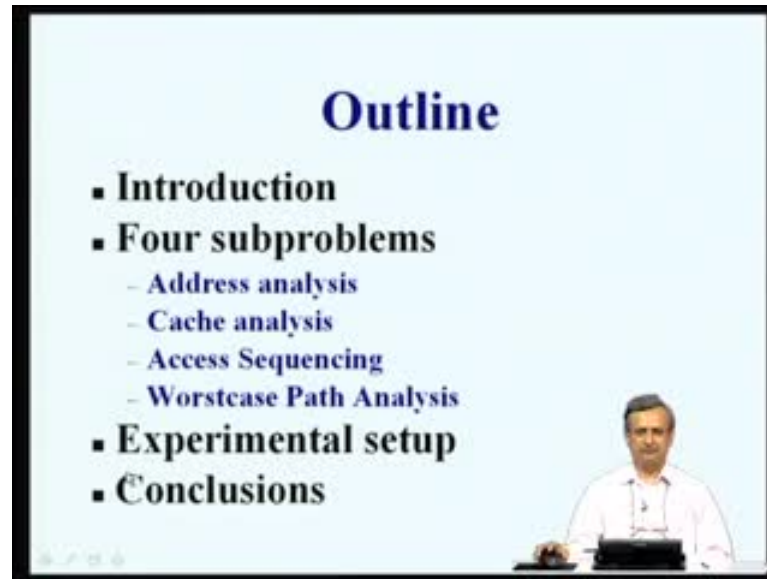
**Module No. # 21**  
**Lecture No. # 39**  
**Worst Case Execution Time**

(Refer Slide Time: 00:19)



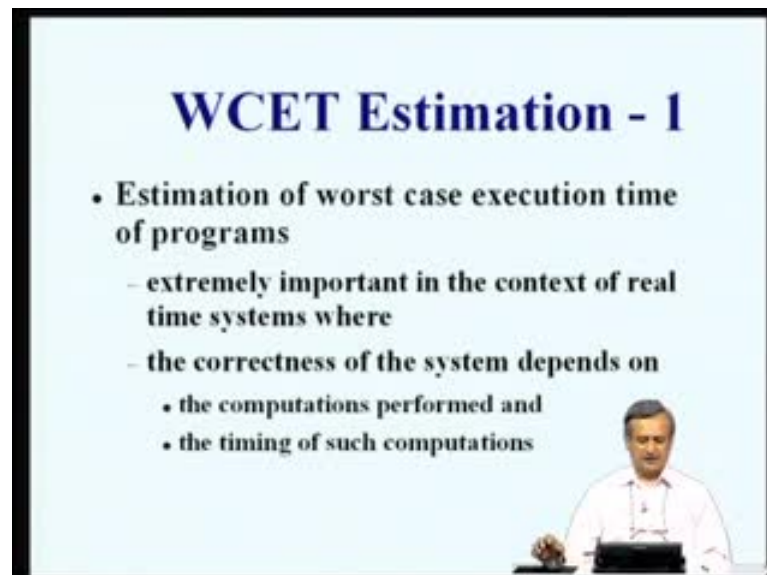
Welcome to the lecture on worst case execution time estimation. The technique we are going to use is called as abstract interpretation; so in some sense it is a very great generalization of the data flow analysis. So, you can say that this is an application of compiler techniques to execution time estimation. The difficulty is multiplied many times when there is a data cache present. People have **actually run a lot when about a** worst case execution time estimation when there is no data cache, but only instruction cache, but in the presence of data cache the problem becomes more difficult.

(Refer Slide Time: 01:13)



So, let us look at some motivation for this study. We are going to consider 4 sub problems, apart from the introduction - address analysis, cache analysis, access sequencing and worstcase path analysis. Then, we will look at some experimental setup and conclusions.

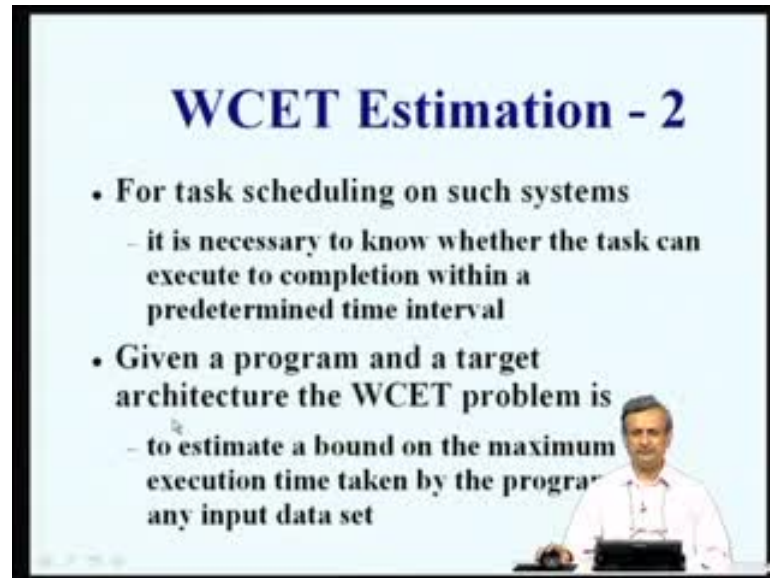
(Refer Slide Time: 01:36)



Why is the estimation of worst case execution time of programs very important? Basically, it is important in the context of real time systems; because in the real time systems scenario, the correctness of the entire system depends on the computations

performed and the timing of such computations; so, computations will have to be performed within a certain period of time.

(Refer Slide Time: 02:08)



**WCET Estimation - 2**

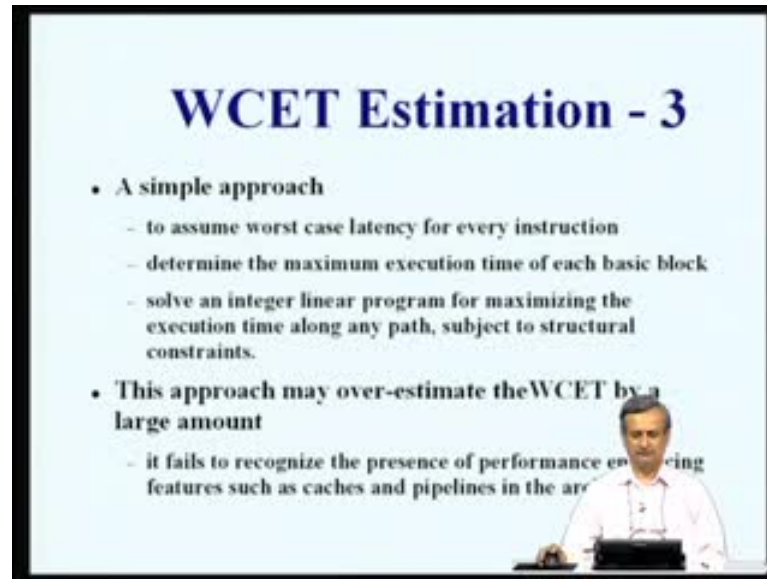
- For task scheduling on such systems
  - it is necessary to know whether the task can execute to completion within a predetermined time interval
- Given a program and a target architecture the WCET problem is
  - to estimate a bound on the maximum execution time taken by the program for any input data set

© 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100

On such systems scheduling of task is also very important. Thus it is necessary to know, whether the task can execute to completion within a predetermined time interval. So, because of these reasons, in real time systems **otherwise,** havoc may be created if we do not switch off a fan or heater etcetera, at a predetermined time and that may be based on some computation which is given a dead line.

So, the problem of WCET estimation - we can define it as given a program and target architecture. The WCET problem is to estimate a bound on the maximum execution time, taken by the program for any input data set. So, it is not the same as running the program on a given input; here we are really trying to find the maximum execution time and a bound on it and this should hold for any dataset. That is the maximum we are looking at.

(Refer Slide Time: 03:24)



**WCET Estimation - 3**

- A simple approach
  - to assume worst case latency for every instruction
  - determine the maximum execution time of each basic block
  - solve an integer linear program for maximizing the execution time along any path, subject to structural constraints.
- This approach may over-estimate the WCET by a large amount
  - it fails to recognize the presence of performance enhancing features such as caches and pipelines in the architecture

Speaker: A man in a white shirt is visible in a small video inset at the bottom right of the slide.

So, what is a very simple approach to such a WCET? Remember, we always want to consider the binary form of the program rather than the source form of the program, because the idiosyncrasies of the machine - that is the specialties of the machine cannot be captured at the high level. It has to be captured only by the instructions at the lower level.

A very simple approach would be to assume worst case latency for every instruction. For example, you assume that every add instruction perhaps, **when it is pipelined floating point add may actually give you one instruction throughput every cycle** when the pipeline is full, but if the pipeline is not full then depending on the number of phases in the pipeline say stages in the pipeline say 3 or 4 as many cycles will be required for the instruction to complete.

Similarly, in that case, we assume the worst case latency of 4 cycles for every floating point instructions. Similarly, an add instruction which fetches an operand from memory, may actually take say 2 or 3 cycles; if the operand is not available in the cache, but may take only 1 cycle if the operand is in the cache, so in such a case, we assume that the operand is really in memory and assume, that it takes 3 cycle. So, this is the worst case scenario that we are really interested in.

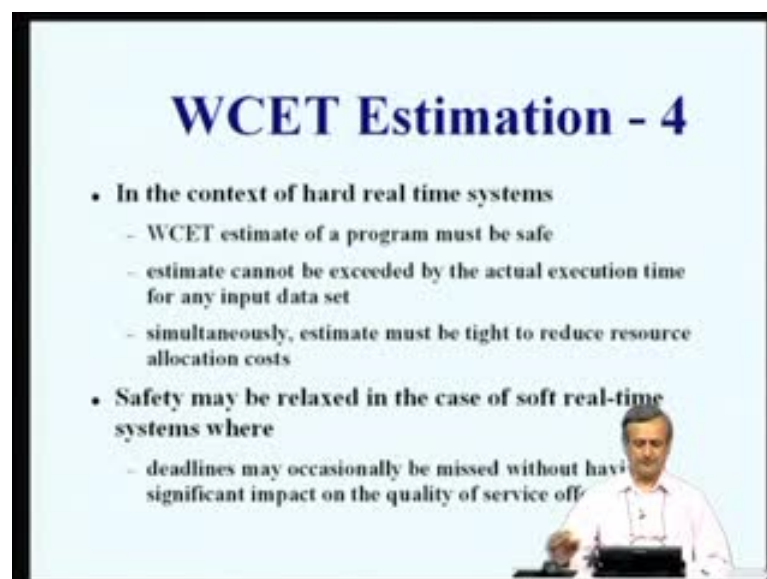
So, you determine the maximum execution time for each basic block. Then, solve an integer linear program for maximizing the execution time along any path, subject to the structural constraints of the control flow graph.

This is fairly straight forward. Once, the branch probabilities of the execution the control flow graph we can say, what is the maximum along any path and so on and so forth. Otherwise, we can always if you do not want to take the branch probabilities, which may actually not give you WCET exactly. You can take the time along every path and then take the maximum. So, that is where the integer linear program comes into picture.

So, this approach may over-estimate the WCET by a large amount. Why? We ignore the caches as I told you. We may ignore even the pipelining and thereby, we assume that each instruction takes the maximum amount of time. But this is not so in practice most of the time 97, 98, 99 percent of the time. The cache is a hit and once there are many instructions to go through the pipeline, then the throughput through the pipeline is also going to be very good.

So, in such a case, the estimate is going to be much more than what it should be. It fails to recognize the presence of performance enhancing features such as, caches and pipelines in the architecture.

(Refer Slide Time: 07:10)



**WCET Estimation - 4**

- In the context of hard real time systems
  - WCET estimate of a program must be safe
  - estimate cannot be exceeded by the actual execution time for any input data set
  - simultaneously, estimate must be tight to reduce resource allocation costs
- Safety may be relaxed in the case of soft real-time systems where
  - deadlines may occasionally be missed without having significant impact on the quality of service offered

Video inset: A man in a white shirt speaking at a podium.

Now, in the context of hard real time systems in other words, there is no way you could miss a dead line it must be satisfied in all respects. WCET estimate of programs must be safe. In other words, a safe estimate is one in which estimate cannot be exceeded by the actual execution time for any input data set. So, it is similar to the safety in data flow analysis as well.

Simultaneously, the estimate must be very tight to reduce the resource allocation costs. Otherwise, you would probably be allocating the resource for a much larger duration we do not want to do that given the WCET estimate which is very tight, we can use the resources very efficiently.

So, safety may be relaxed in the case of soft real time systems because in such a case in soft real time systems, deadlines maybe occasionally missed without having a significant impact on the quality of service offered. So, in such a case the WCET analysis can be need not consider safety thereby it may be violated sometimes, but most of the time it will be correct.

(Refer Slide Time: 08:33)

### Data cache effect on WCET

Program Name	WCET (cycles)	
	Simulation	All-Miss
hanoi100	81091 (x 2)	315097
cut	4410	9095
edu_fo	52990	103133
edu_fo_no_red_bf	40328	73114
edu_je	2274	5032
edu_latcyath	3281	5875
edu_max	3139	6104
ifdntst	2275	3481
matrix	147413	290022

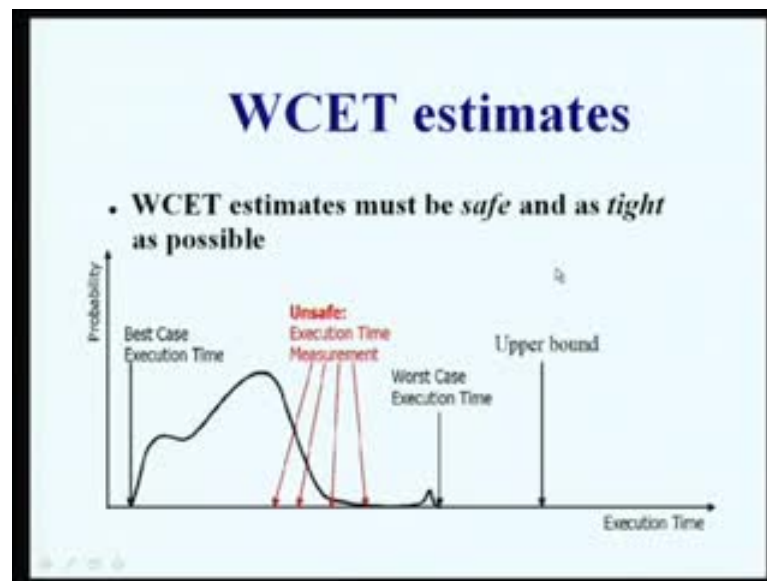
- Configuration → 4 way, 32 byte blocks, 256 sets
- Latency (cycles) → hit: 1, rd miss: 6, write miss: 4

What is the effect of data cache on WCET? So, here is a set of programs from benchmark suites, so by actual running on a very large number of datasets, we have computed the worst case execution time. This is the maximum among the data set runs that we have conducted and assuming that all cache accesses are misses, here is the time that we have computed from the program. So, the configuration is with cache is 4 way 32

byte blocks with 256 sets and the Latencies of the cache are hit requires 1 cycle, read misses 6 cycles whereas, write misses 4 cycles. So, read miss is 6 because you have to fetch from memory and also write into the block and all that whereas, write miss directly writes into the memory.

Now, you can easily see that the all miss value is almost twice the value of the actual simulation time. This is too bad, 100 percent excess is not acceptable.

(Refer Slide Time: 10:00)



So, here is a picture a graph which shows, what is safety? And what are unsafe behavior and all that. Here, is the probability of a particular time execution time taken by the program, so here is the execution time itself.

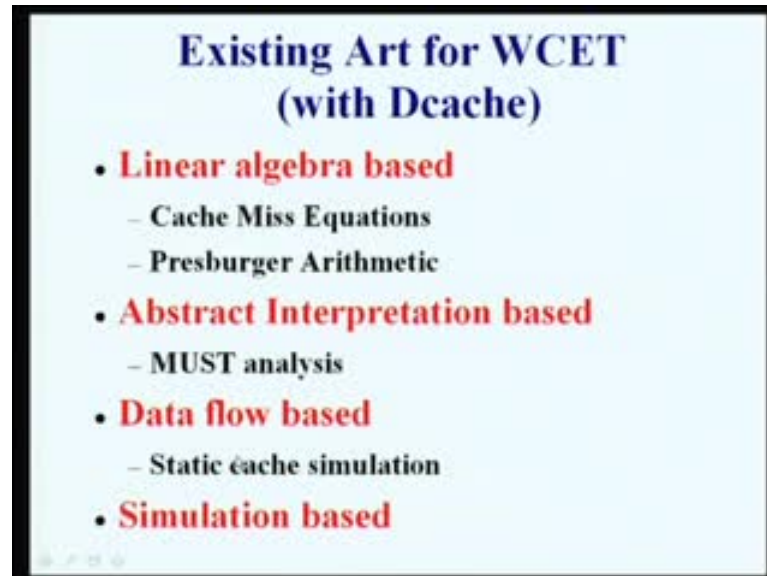
This is the best case execution time and as we go along the upper bound that is the maximum, but too far away from the actual worst case execution time is here. All these points are unsafe because we get these by measurement. So, this is the reason why you cannot use measurement based execution time, you will actually be somewhere here and you will never reach this point.

So, what this graph shows is some inputs actually take so much time. The maximum numbers of inputs probably are here, which require as much time and there are few inputs which really stretch the program to its worst case execution time and that is the



reason why, we cannot get it easily through experimentation. Most of our inputs may be somewhere in this region, so we will never reach this without too much effort.

(Refer Slide Time: 11:26)



What is the existing art for WCET? So, with Data cache of course, with instruction cache the analysis is simple, not too difficult. But with data cache the analysis is very hard. There are very complicated linear algebra based approaches. So, one of them uses cache miss equations, which are well-known safe, but they do not they really give you the behavior as tightly as desired.

Presburger arithmetic is another method. That is used again, it has similar difficulties. Then, Abstract Interpretation based methods use what is known as MUST analysis. So, MUST analysis says what accesses must be hits, what accesses must be misses and so on and so forth.

So, if this is similar to the MUST and May analysis in data flow analysis as well for example, in reaching definitions, when you compute the reaching definitions it is enough if at least one of them reaches the particular point. So, it is actually it may reach that point, when it is the point racks us and so on and so forth.

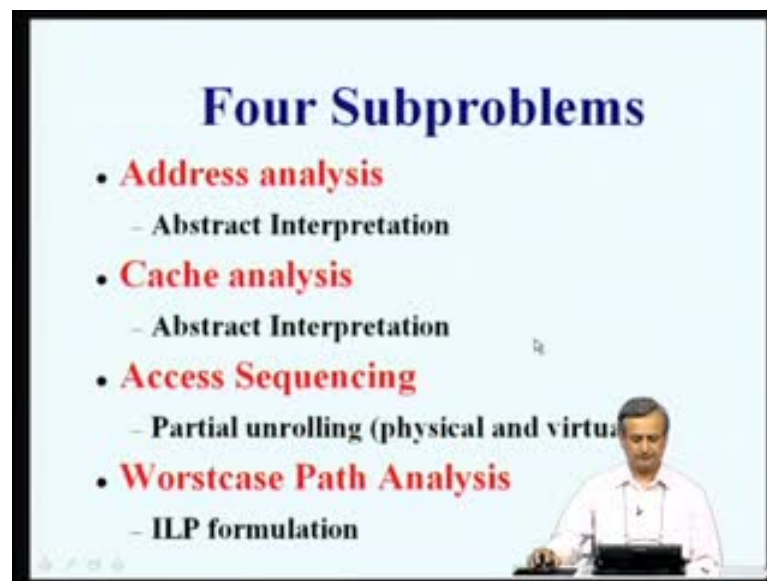
Even in actually execution if the path taken is a different one then it may not reach. So it is vice versa called a May analysis, where as in available expressions we must make sure



that the expression is available through every path at a particular point so it is an example of the MUST analysis.

Then, there are data flow based approaches, which use static cache simulation. So, we are not going to elaborate and then there are simulation based approaches which are measurement based. Our approach that actually will be Abstraction Interpretation based.

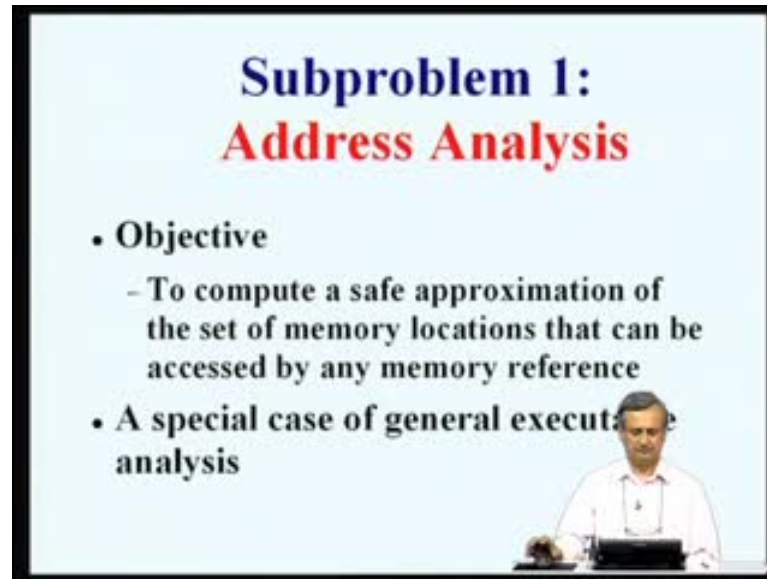
(Refer Slide Time: 13:44)



So, there are four sub problems in the worstcase execution time estimation. The first one is the address analysis - which uses the abstract interpretation technique. I am going to tell you a little bit about the abstract interpretation technique as well. Then, there is cache analysis - which is also abstracted interpretation based. We are not going to use any cache simulators here.

The third sub problem is the access sequencing problem. So, in which order should be consider the instructions and loop etcetera. Here, we use what is known as partial unrolling, physical and virtual this will become only later. Physical unrolling is something we all understand unroll the loop one way etcetera, but virtual unrolling implies we do not really physically unrolling the loop, but we go through the analysis goes through the loop many times that is what we call as virtual unrolling. The last one is the worst case path analysis - which uses the integer linear program formulation.

(Refer Slide Time: 15:00)



**Subproblem 1:**  
**Address Analysis**

- **Objective**
  - To compute a safe approximation of the set of memory locations that can be accessed by any memory reference
- **A special case of general executable analysis**

Image of a person at a laptop in the bottom right corner of the slide.

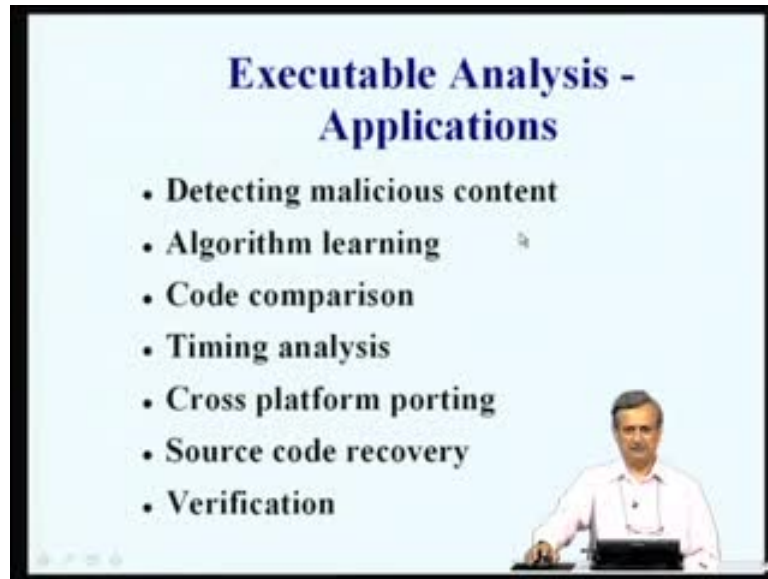
The first sub problem is the address analysis problem, so its objective is - to compute a safe approximation of the set of memory locations that can be accessed by any memory reference. Why is this essential? We are given a binary and each one of the instructions in that binary program, may access register, it may access a stack, it may access the main memory itself, it may access heap or whatever.

So, there is going to be some address, for each one of these locations and we want to actually find out or other makes a safe approximation of the set of memory locations that can be accessed by any memory reference. So, if we know that it is a particular register then we can make it concrete there is only reference at that point, but if we are actually accessing or if you are accessing a particular scalar variable directly, then we would be accessing particular memory location and that would be concrete.

But if you are accessing let us say some area in memory some memory in array, we may not to be able to say at compile time which particular element of that array is being accessed that may not be possible. But we may be able to partition the array into a few parts and then say this particular reference falls in to this partition a, the second reference falls into partition b, etcetera.

So, the objective of address analysis is basically find out carve the entire memory space into regions, such that we know at a point in time if we are making a memory reference which region we are accessing.

(Refer Slide Time: 17:35)



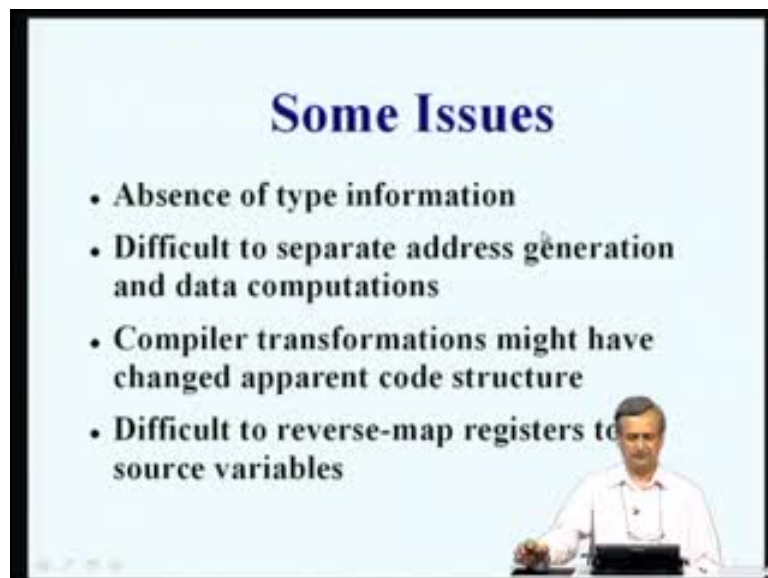
**Executable Analysis - Applications**

- Detecting malicious content
- Algorithm learning
- Code comparison
- Timing analysis
- Cross platform porting
- Source code recovery
- Verification

The slide features a light blue background with a black border. In the bottom right corner, there is a small inset image of a man in a white shirt sitting at a desk with a laptop. The text is centered and uses a serif font for the title and a sans-serif font for the list items.

So, this is going to be very helpful for us in cache analysis as well. This address analysis is in general a special case of general executable analysis. There are many applications for the executable code analysis such as - detecting malicious content in code, learning about the algorithm, comparing different code versions, timing analysis that we are interested in, the cross platform porting, source code recovery and verification. So, there are many applications it is not as if executable analysis is a speciality of our particular approach.

(Refer Slide Time: 18:08)



**Some Issues**

- Absence of type information
- Difficult to separate address generation and data computations
- Compiler transformations might have changed apparent code structure
- Difficult to reverse-map registers to source variables

The slide features a light blue background with a black border. In the bottom right corner, there is a small inset image of a man in a white shirt sitting at a desk with a laptop. The text is centered and uses a serif font for the title and a sans-serif font for the list items.

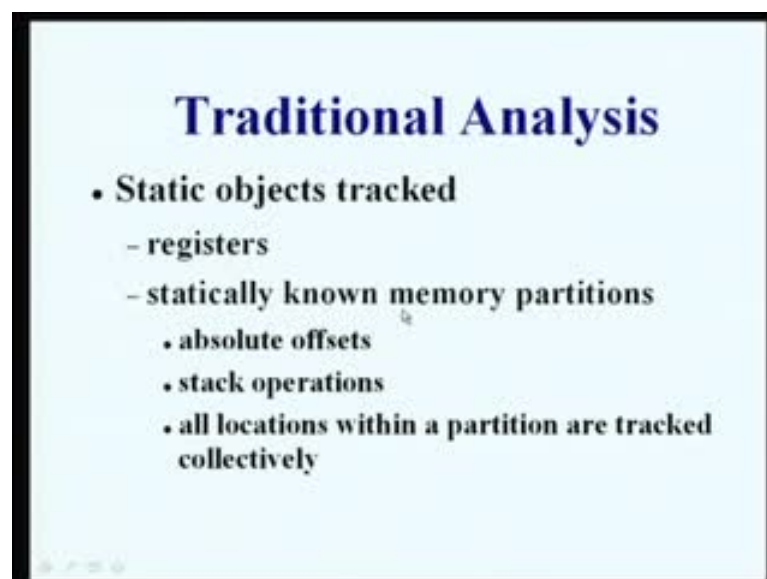
There are many issues in address analysis. So, let us look at them in some detail binary code does not have type information so at the register level it may be possible to say this register contains integer information, this register contains floating point information and so on. Because the register set may be different, but we have no information about which is a character which is a integer and whether this is user define data type or whatever.

So, we have no type information and it is difficult to separate address generation and address computation. Whether, the arithmetic that we are performing is on the address or it is some data is very difficult to actually separate, so this is another issue that we how to worry about, so compiler transformation might have changed apparent code structure remember we are using binary code.

This is already optimize code, the complier has change it and there is very little that may not be too much mapping rather, it may not be possible to retrieve much information from the source code and use it in the binary analysis, so because the transformations might have change the code structure.

It is difficult to reverse-map registers source variables we cannot say which variable we will be in register all the time by doing some reverse mapping we have been no information about that sort of anything to here.

(Refer Slide Time: 19:58)



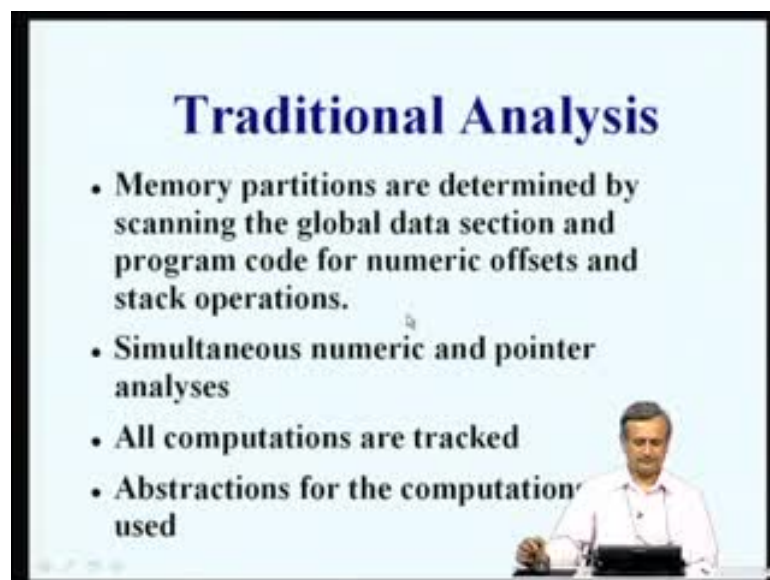
**Traditional Analysis**

- **Static objects tracked**
  - registers
  - statically known memory partitions
    - absolute offsets
    - stack operations
    - all locations within a partition are tracked collectively

What does traditional analysis do? So, we have use traditional analysis with some minor changes, so basically our changes are very are very minor in the analysis part itself, but are major when we comes to storing the various partitions. So, there are static objects which are tracked such as, registers then statically known memory partitions such as, absolute offsets stack operations all locations within a particular are tracked collectively.

So, when we go through the program line by line or instruction by instruction, we also actually update the data structures related to the various partitions. So, there is going to be some data structure for register and for each partition is as well so these are all going to be tracked and modified as and when it is necessary.

(Refer Slide Time: 21:06)



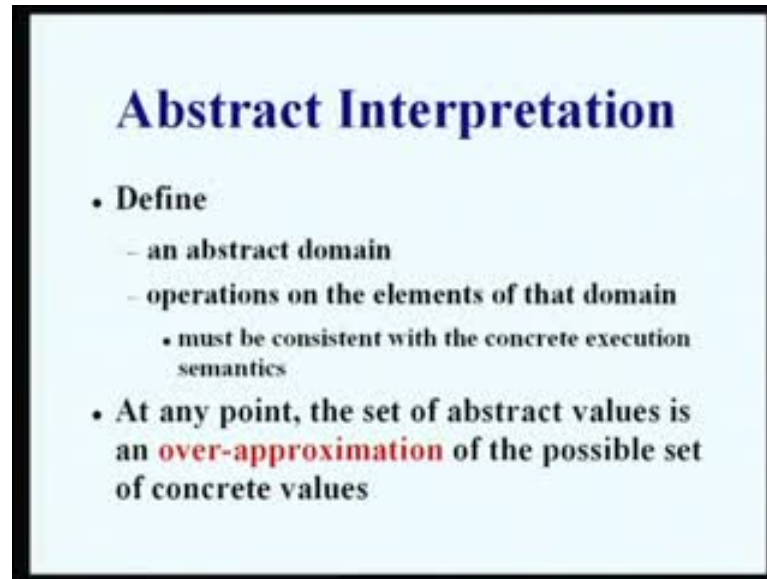
**Traditional Analysis**

- Memory partitions are determined by scanning the global data section and program code for numeric offsets and stack operations.
- Simultaneous numeric and pointer analyses
- All computations are tracked
- Abstractions for the computations used

The slide features a light blue background with a black border. In the bottom right corner, there is a small inset image of a man with short grey hair, wearing a light-colored shirt, sitting at a desk with a laptop. The text on the slide is in a dark blue, serif font.

Memory partitions are determined by scanning the global data section and program code for numeric offsets and stack operations, so this analysis gives enough information to us. We do a simultaneous numeric analysis and also pointer analyses. All computations of course, tracked and we use actually abstractions for the computations, we will talk about these abstractions in a short file from now.

(Refer Slide Time: 21:35)



**Abstract Interpretation**

- **Define**
  - an abstract domain
  - operations on the elements of that domain
    - must be consistent with the concrete execution semantics
- **At any point, the set of abstract values is an over-approximation of the possible set of concrete values**

As I said, the partitions that we get for the addresses must be stored and in fact when we look at the contents of these memory locations and registers, we go through the computations that are performed in the data. We actually have to perform the same computation during our analysis of the code on the partition as well for example, if we say we are going to store a value in a particular register and then add something to it. We will have to actually perform this addition and then store the result again in that particular directed data structure corresponding to the partition.

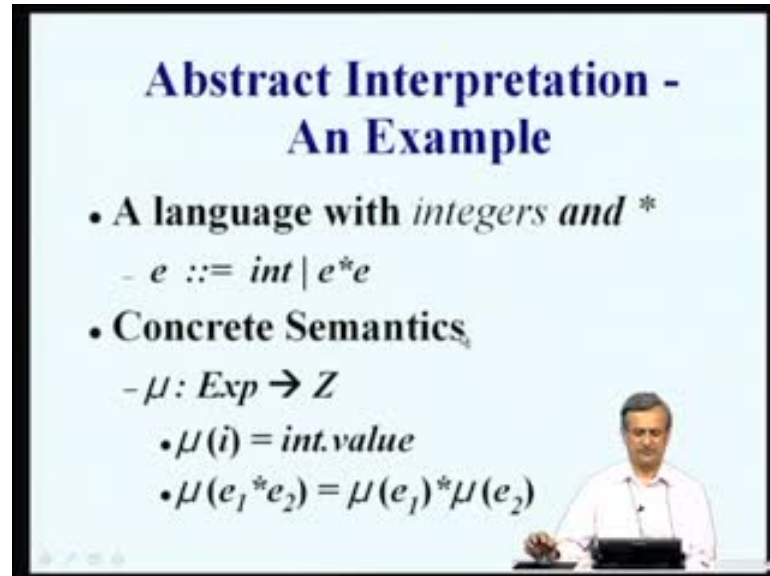
So, if we do not know the exact value, it may be a set of values and so we will have to learn how exactly a set of values can be represented and manipulated efficiently in our mechanism. We define an abstract domain, so we are going to discuss abstract interpretation some detail now and we define operations on the elements of that domain, so these must be consistency with the concrete execution semantics. So, concrete execution semantics implies the actual when we run the program whatever is the semantics of that program in an instruction by instruction manner is called as the concrete execution semantics.

Whatever we do in abstract interpretation, will be at a slightly higher or abstract level but these must be consistency with the concrete execution semantics. In other words, you cannot do addition in concrete execution semantics and multiplication in the abstract interpretation. It has to be addition all the time, but only thing is on what values you do

the addition is it on a set of elements or is it on a particular element etcetera, have to be worked out properly.

At any point in time, the set of abstract values is an over approximation of the possible set of concrete values, so this will become clear as we go long now.

(Refer Slide Time: 24:09)



**Abstract Interpretation -  
An Example**

- A language with *integers and* \*
  - $e ::= int \mid e * e$
- Concrete Semantics
  - $\mu: Exp \rightarrow Z$ 
    - $\mu(i) = int.value$
    - $\mu(e_1 * e_2) = \mu(e_1) * \mu(e_2)$

The slide also features a small inset image of a man in a white shirt sitting at a desk with a laptop, appearing to be presenting the slide.

Let us take an example for abstract interpretation. So, let us consider in extremely simple language with integers and one just one operations star multiplication. So, the grammar is here, an expression can be either in integer constant or of the form e star e. You can generate constants in this language or you can generate expression which star in this language. The expression again will have integer constraints in them.

What is the concrete semantics of this small language? So, let us say the operator is rather the mapping function is mu, which takes an expression and produces a number. When we run this program, this is what is going to happen. If there is an integer say, mu of i, i is this stand for this int. So, it is going to be int dot value the value of that particular of that integer nothing more than that.

Whereas, if it is expressions even star e 2, then the value of that expression is mu of e 1 star mu of e 2. In other words, you evaluate mu the e 1 with mu function, evaluate e 2 with the mu function and then multiply the 2, in the integer domain. For example, if you have 15 star 6 the answer should be 90 here. So, mu of 15 star mu of 6 so mu of 15 from




here would be the value of 15 which is 15 and the mu of 6 would be 6 so the multiplication in integer domain would be 60, 90.

(Refer Slide Time: 25:59)

## An Abstract Semantics

- **Compute only sign of the result**
  - $\sigma : Exp \rightarrow \{+, -, 0\}$
  - $\sigma(i) = \begin{cases} +, & \text{if } i > 0 \\ 0, & \text{if } i = 0 \\ -, & \text{if } i < 0 \end{cases}$
  - $\sigma(e_1 * e_2) = \sigma(e_1) \square \sigma(e_2)$


$\square$	+	0	-
+	+	0	-
0	0	0	0
-	-	0	+



(Refer Slide Time: 26:10)

## Abstract Interpretation - An Example

- **A language with integers and \***
  - $e ::= int \mid e * e$
- **Concrete Semantics**
  - $\mu : Exp \rightarrow Z^c$ 
    - $\mu(i) = int.value$
    - $\mu(e_1 * e_2) = \mu(e_1) * \mu(e_2)$



So, that is as for as the concrete domain in is concerned so, this is the concrete semantics and how exactly it works now let takes an abstraction of this. What is the abstraction that we want to consider?

Let us say given an expression, the previous example could take either negative numbers or positive numbers all integers of course, but either negative 0 or positive right. Let us say, we are interested in only the sign of that expression. Here is the expression, this function sigma or the mapping function sigma will take an expression and then it has to tell us, whether it is a positive number the result finally is positive negative or 0.

So, if you are given just an integer, sigma takes that integer if the integer is greater than 0 then it written, plus if the number is 0, then it gives a 0 and if the number is less than 0, it returns minus sign, so for as a single integer is concerned this is fairly straightly forward. Whereas, if it is an expression  $e_1 * e_2$  then again this is syntax directed, so we apply sigma  $e_1$ , we apply sigma  $e_2$  and in the abstract domain we apply this square operator which actually hope here is the square operator. It tells you, what is the sign of the 2 in combination will be.

So, let us take this square operator and its table. On this side are the 3 possibilities of plus, 0 and minus and on this side we have plus, 0 and minus. If sigma  $e_1$  and sigma  $e_2$  are both plus obviously, the result will be plus and the square operator also says it is plus so far so good.

If one of them is plus and other is 0. So, plus and 0 or plus and 0. The result is 0. Plus and 0 is 0, plus and 0 is 0. So, that is obvious because whatever we do in any multiplication, if one of the operands is 0, then the result is going to be 0 so that is why this the result here will also be just 0.

And similarly, sigma  $e_1$  is plus and sigma  $e_2$  is minus then the result is said to have minus sign this is also well-known. If one of them is a negative operand then multiplication not both one of them is negative the result is negative. But if both of them negative then the result is positive. So, this is also well-known. Similarly, if one of them is 0, then again it is 0 and so on.

In other words, sigma uses this square operator in its abstract domain that is plus 0 and minus in this domain, combines these two results and tells us the result of the sign of the result. This is the abstraction that we are looking at so we have actually taken integers but then a rather integer expressions, but then converted that entire thing into only we want to only the sign so that is the abstraction that we are looking at so the entire. For a single plus there could be many integers and many expressions possible right so all those

are actually map to that plus, so in that sense every element of this abstract domain maps to a large number of expressions in the x domain.

(Refer Slide Time: 30:14)

### Abstract and Concrete Values

- Associate each abstract value with the set of concrete values it represents
- We need to add  $\top$  (top) and  $\perp$  (bottom) elements to the set of abstract values

$$\gamma : \{+, -, 0\} \rightarrow 2^{\mathbb{Z}}$$
$$\gamma (+) = \{i \mid i > 0\}$$
$$\gamma (0) = \{0\}$$
$$\gamma (-) = \{i \mid i < 0\}$$

- Our abstract domain is now a *lattice*
- We can now map other operations such as +, -, and / to suitable operations on the abstract domain

So, let us elaborate a bit, what are abstract and concrete values? We know what concrete values are, so those are our integer's right, we have to look at abstract values. Let us associate each abstract value with the set of concrete values it represents. So, gamma is this function: it takes plus, minus and zero and yields an element in the power set of z which is the set of integers.

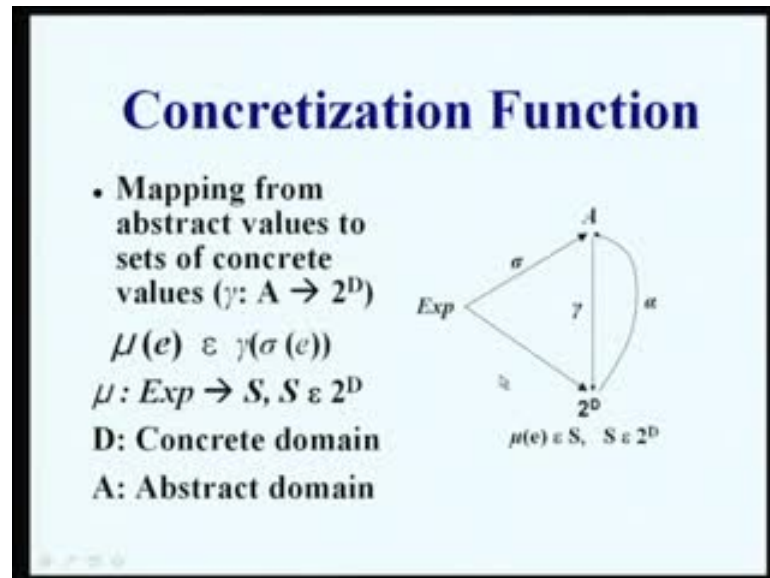
So, in other words as I was just now telling you, there are many integers corresponding to this plus and this minus. There is possibly 1, 0 with this, nothing more than that. So, that is what is indicated here. Gamma of plus is all those integers which are greater than 0, gamma of 0 just one element 0 whereas, gamma of minus is all those integers which are less than 0.

So, each of these is an infinite set in this case so very large. Suppose we add plus this top and bottom elements to the set of abstract values, then our abstract domain becomes a lattice. Why should we do this? Lattices have nice properties and there is easy to deal with their well-known structures and that is the reason we make these into a lattice.

We can now map other operations such as plus minus and slash, to suitable operations on the abstract to domain as well. So, we have considered only the simple abstraction and

we consider only the star. We could similarly consider plus operation, minus operation, unary operation etcetera and map them appropriately to these top, bottom or plus, minus and 0 values. So, without top and bottom, we will not be able to handle addition and subtraction, forget division that is not possible at all otherwise.

(Refer Slide Time: 32:33)



What is a concretization function? So, we have already seen that gamma is a mapping from abstract domain A to the power set of the concrete domain D, so mapping from abstract values to sets of concrete values is done by gamma so that is here this is the gamma part.

And mu is our concrete concretization function. We really have not mu is the operator which takes the expressions to our concrete domain, so mu of e is in gamma of sigma of e. Sigma of e really takes an expression to its abstract domain and then applying gamma on it brings it back to the concrete domain.

If you take an expression apply mu on it. Then, it will be in the set of elements produced by gamma of sigma of e, remember gamma produces sets of elements rather than a single element. We saw is there are many integers corresponding to a single A value that is either plus or minus are 0. So, mu really is from Exp to S whereas, S is actually z, but here we will says S, S is in 2 to the power D as a generalization.

(Refer Slide Time: 34:26)

## Abstraction Function

- Mapping from concrete values to abstract values
  - The dual of concretization
  - The smallest value of  $A$  that is the abstraction of a set of concrete values

$\alpha: 2^S \rightarrow A$

$\alpha(S) = \text{lub}(\{ a \mid i < 0 \wedge i \in S \},$   
 $\{ 0 \mid 0 \in S \},$   
 $\{ + \mid i > 0 \wedge i \in S \})$

$\alpha(\{24, 45, 3\}) =$   
 $\alpha(\{-2, -87, -123\}) =$   
 $\alpha(\{0\}) = 0$   
 $\alpha(\{-5, 2\}) =$

(Refer Slide Time: 34:38)

## Concretization Function

- Mapping from abstract values to sets of concrete values ( $\gamma: A \rightarrow 2^D$ )

$\mu(e) \in \gamma(\sigma(e))$

$\mu: Exp \rightarrow S, S \in 2^D$

**D: Concrete domain**

**A: Abstract domain**

$\mu(e) \in S, S \in 2^D$

So, D is a concrete domain, A is the abstract domain. Now, there are other mappings also possible from 2 to the power D that is the power set of the domain D to A, but before that the abstraction function itself, so here is the abstract function which take 2 to the power D to A. So, it takes concrete values from this domain and then produces this abstraction.

(Refer Slide Time: 34:47)

(Refer Slide Time: 35:05)

Mapping from concrete values to abstract values so this is the dual of concretization, so the smallest value of A that is the abstraction of a set of concrete values. So, remember what Exp does is to take this expression and produce a concrete value or sigma does is it takes expression and produce an abstract value, but in 2 to the power D we have sets of concrete values, so what happens what alpha does is take this set and produce an abstract value.

(Refer Slide Time: 35:23)

**Abstraction Function**

- Mapping from concrete values to abstract values
  - The dual of concretization
  - The smallest value of A that is the abstraction of a set of concrete values

$\alpha : 2^z \rightarrow A$

$\alpha(S) = \text{lub}(\{ - \mid i < 0 \wedge i \in S \},$   
 $\{ 0 \mid 0 \in S \},$   
 $\{ + \mid i > 0 \wedge i \in S \})$

$\alpha(\{24, 45, 3\}) = +$   
 $\alpha(\{-2, -87, -123\}) = -$   
 $\alpha(\{0\}) = 0$   
 $\alpha(\{-5, 2\}) = \top$

The diagram shows a lattice with four nodes:  $\top$  at the top,  $+$  on the left,  $-$  on the right, and  $\perp$  at the bottom. The node  $0$  is in the center, connected to  $+$ ,  $-$ , and  $\perp$ . The nodes  $+$  and  $-$  are also connected to  $\top$ .

So, that is what is indicated here, 2 to the power z to A or 2 to the power D to A. When we map is we will take this smallest value of A that is the abstraction of a set of concrete values so it becomes clear here. So, alpha apply to this S set of values which is an element of 2 to the power D, it is the least upper bound lowest upper bound of minus such that the integer i is less than 0 and integer belongs to S.

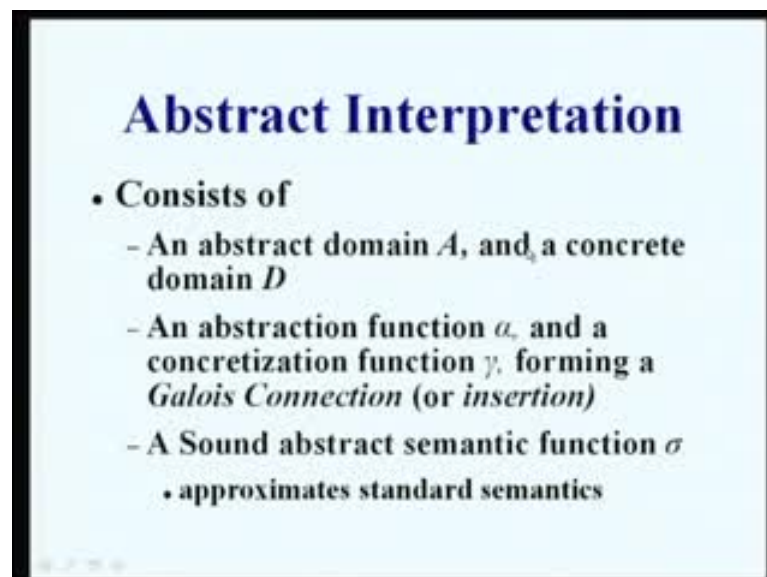
Take all those elements in S, in this S which are negative and for that minus is the representative, 0 is the representative of 0 and plus is the representative of all the positive numbers.

We at any point in time, some may actually belong to some in a set (36:24) some of them belong to this set, some of them belong to this set, some may belong to this set. We have to take the least upper bound of these so taking least upper bound implies going up.

Here are some examples, if all the numbers are positive 24, 45 and 3. There all positive so the lowest smallest value of A is really plus. All of them belong to this and there is only plus.

Similarly, if all the numbers are negative then map the least upper bound is minus. If it is just 0 then it is a 0 in all other cases for example, if we take a negative number and a positive number this will map to minus and this maps to plus, so the least upper bound minus and plus is really top. So, this is the abstraction function that we are talking about.

(Refer Slide Time: 37:24)



**Abstract Interpretation**

- Consists of
  - An abstract domain  $A$ , and a concrete domain  $D$
  - An abstraction function  $\alpha$ , and a concretization function  $\gamma$ , forming a *Galois Connection (or insertion)*
  - A Sound abstract semantic function  $\sigma$ 
    - approximates standard semantics

What exactly is abstract interpretation? It consists of an abstract domain  $A$ , a concrete domain  $D$ . An abstraction function  $\alpha$  and a concretization function  $\gamma$ , forming what is known as a Galois connection. These are the elements of abstract interpretation. So again, we must have an abstract domain and a concrete domain. We already saw examples of this. We had integer as forming a concrete domain and this plus 0 and minus forming an abstract domain.

And we were mapping elements from the sets of elements from concrete domain to abstract domain. An abstraction function  $\alpha$  and a concretization function  $\gamma$  so

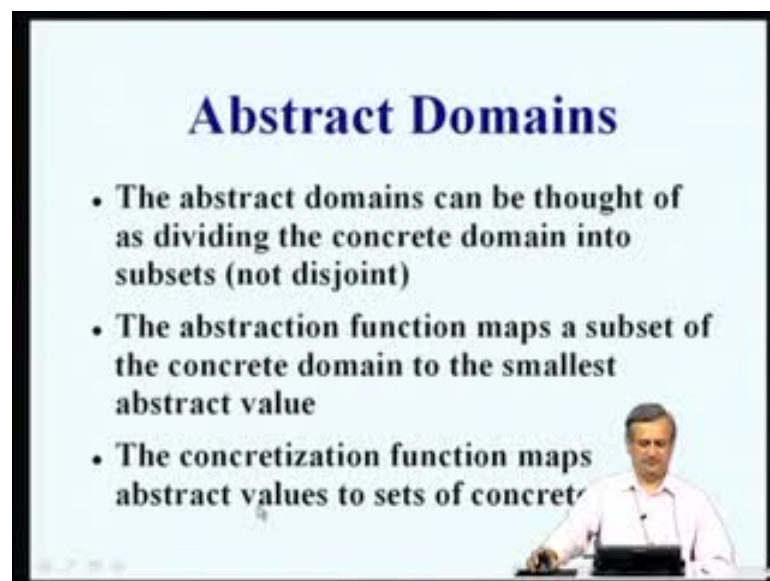


abstraction function gives takes sets of concrete elements and then gives you abstract domain element, a concretization function does the other way. It takes an element from the abstract domain and gives a set of values in the concrete domain. So, these form what is known as a Galois connection. I will elaborate a little more on this Galois connection very soon.

A sound abstract semantics function  $\sigma$  is also necessary and this approximates the standard semantics. So, this soundness is needed to make sure that a everything is correct so in other words, whatever we do here should be correct so as I said you cannot really take a set of negative and positive numbers together and then say this abstract function gives me negative minus or something like that that would be unintuitive as well.

Soundness implies correctness and this must approximate the standard semantics or the way we really want to view the entire computation. Now, we are let us look at Galois connection which is also called as insertion.

(Refer Slide Time: 39:58)

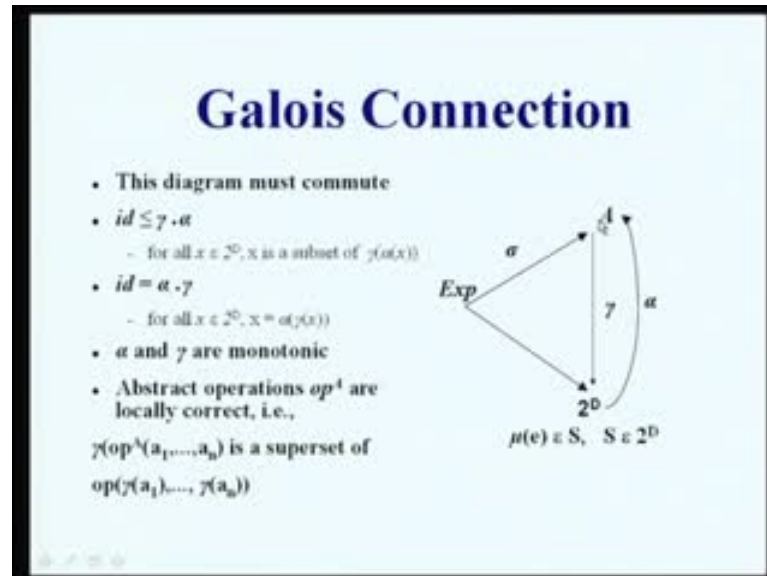


So, just to because abstract domains are very important here is summary, the abstract domains can be thought of as dividing the concrete domain into subsets, but these subsets are not necessarily disjoint.

In other words each of these the subsets map to a single abstract value. The abstract function maps a subset of the concrete domain to the smallest abstract value which is

already mention. The concretization function maps abstract values into sets of concrete values, so that is what these three arcs, it is a summary.

(Refer Slide Time: 40:40)



Let us see, what a Galois connection? So, here is the diagram which we say a must commutes so in other words we go like this and then take this path, so in other words we apply the sigma function, then the gamma function. That result must be a same or consistence with the result we get when we map Exp to this 2 to the D directly or we go like this and then come back and then go back, we must get the same value as we had started with or we must at least get the value that this consistence with our intuition, or we come this way and then go back and then come down, you must still get a consistent and intuitive result. So, that is what we mean by commuting.

So, this id is the identity element, so id less than is equal to gamma dot alpha, so gamma dot alpha. In other words, we do this once and then go back we must actually get something consistent. So, what is that really? For all x in 2 to the D that is in this power set of D, x is a subset of gamma of alpha of x. So, you apply, you take a subset the element of 2 to the D, apply alpha on it. That is the abstract abstraction function then you apply gamma on it, so we have applied alpha got an abstract value then you applied gamma you would again get some concrete values.

So, whatever you get must be a subset of  $x$  must be a subset of whatever you have got now. It cannot be missing some elements, this  $x$  must be embedded in that said you get here. When you go up and then come down again.

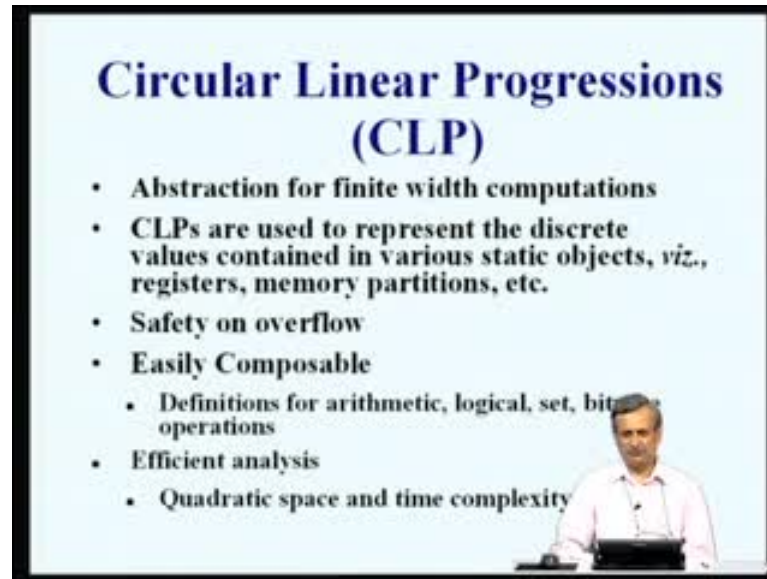
So, that is why,  $\text{id} \leq \gamma \circ \alpha$ . Whereas the other way is  $\text{id} = \alpha \circ \gamma$  so in other words you apply  $\gamma$  first, then you get a set of concrete values then you apply  $\alpha$  again you must get exactly the same the abstraction abstract value that you started with.

So, that is says for all  $x$  in  $D$ ,  $x = \alpha \circ \gamma(x)$ . So, you apply  $\gamma$  then  $\alpha$  you must get the exactly the same element. Whereas if we had apply  $\alpha$  and then  $\gamma$  you would actually get a subset, what you started with.

$\alpha$  and  $\gamma$  are monotonic functions, so this goes back to what we had studied in data flow analysis. Abstract operations  $op$  in the abstract domain  $A$  or locally correct. In other words, you apply  $op$  on  $a_1$  to  $a_n$  the abstract values here. Then you apply  $\gamma$  this is the superset of the values you get when once you apply  $\gamma$  on each of these  $a_1$  to  $a_n$  and then apply  $op$  on in this domain. So, that is what at the correctness of abstract operation in the abstract domain is.

So, in other words the abstract domain cannot be something adhoc and it cannot drop elements all these elements must be present and you can only get a subset of those you go down but you cannot really remove everything or you cannot add anything extra.

(Refer Slide Time: 44:32)



**Circular Linear Progressions  
(CLP)**

- Abstraction for finite width computations
- CLPs are used to represent the discrete values contained in various static objects, viz., registers, memory partitions, etc.
- Safety on overflow
- Easily Composable
  - Definitions for arithmetic, logical, set, bit operations
- Efficient analysis
  - Quadratic space and time complexity

Man at podium

Why did we do all these? As I told you, we are going to represent the various partitions as abstractions so because instead of a single memory location now we have a set of memory locations, so that set is an abstraction but it is not a simple set that we are looking at for example, you may be looking at elements separated by a value of 4 for example, you may say minus 8, minus 4, 0, 4, 8, 12 etcetera.

And all the elements in between such as minus 3 or 2 etcetera are all missing. So, we want to represent such intervals, may be from minus 4 to 12 with the value separated by 4, distance of 4.

Such abstractions are not easy to represent so we have our own method of representing them and that is called as a circular linear progression CLP for short. So, this is abstraction for finite width computation, so we have finite width in the representation of the numbers as well, number of bits is finite. In other words, if there may be an over flow or there may be an under flow, CLP are used to represent the discrete values contained in various static objects namely registers memory partitions etcetera, this is what I was just now explaining.

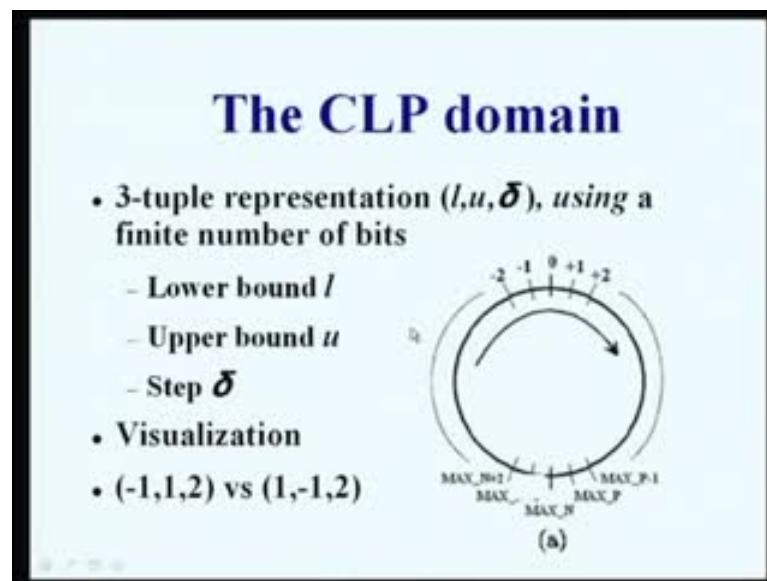
We provide safety on overflow, it is very handled very nicely, as we will see very soon and the CLPs are composable so in other words, why is this composition important? So, we are now not adding a concrete number like 4 to another concrete number like 5. We are really now taking 2 abstractions corresponding to 2 values and then an instruction

may actually be adding them. Let us say, a register is supposed to contain values which are separated by 4 in the range 4 to 12.

Another register may contain values let us say from 3 to 15 separated by 3 and we want to add these two, the instruction maybe addition of these two values.

So, these are two abstractions now we must know how to compose these two abstractions appropriately by the addition in this abstract domain so these must be defined as well. In other words this abstract interpretation frame work that we are going to present provides for definitions of arithmetic logical set bitwise operations on these abstractions and they are very efficient analyze because they require only quadratic space and time for the analysis.

(Refer Slide Time: 47:54)



This is the briefly what CLPs are about? So, what exactly is a CLP? A CLP is a 3-tuple representation there is  $l$ , there is  $u$  and there is  $\delta$  and we use a finite number of bits for example, if each of these  $l$   $u$  and  $\delta$  require  $n$  bits, we requires  $3 n$  bits for the entire representation.

$l$  is the lower bound,  $u$  is the upper bound and the step is  $\delta$ . In other words, if we let us say let us take one of these minus 1, 1 and 2. So, minus 1 is a lower bound, 1 is the upper bound, so we can visualize the entire CLP as a circle in on which the value is marked. We have minus 1 here and plus 1 here, these are the two lower and upper

bounds and we say that 2 is the step. In other words this CLP minus 1, 1 and 2 actually, is corresponds to the set minus 1, plus 1 that is it. It does not corresponding to anything else because these are the only 2 values generated by this particular abstraction of  $l, u$  delta that is minus 1, 1, 2.

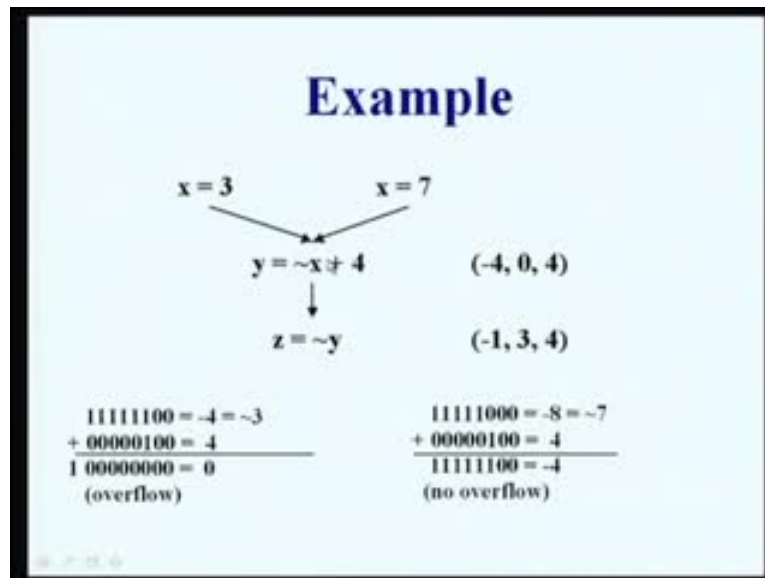
So, the concrete values that this particular CLP maps 2 or minus 1, 1 that set. whereas, if we consider the CLP 1, minus 1 and 2, so we have 1 here and then minus 1 here and the step is 2, so in other words we always go in clockwise direction, we would go to 3, 5, 7 etcetera, go round that circle and then come back to this minus 1.

So, there going to be a large number of values in this particular CLP. There is a max value max n, so that is the maximum that you can represent using whatever number of bits that you are using represent this abstraction.

Beyond that again the numbers are all negative. So, in this half the numbers are all positive and in half the numbers are all negative. You really cannot go beyond Max p for positive numbers ad max n for negative numbers if it is a tools complement representation. Because we can always represent 1 number more in the tools compliments representation.

There is a nice visualization possible here see, so you always have the circle, marked the values on the circle, travel around in the circle clockwise direction using the step. So, used go from the lower bound l, to the upper bound l, using delta as this step and once you add the values accumulate in this traversal is the concrete set that you want to represents, so that is the CLP domain and its meaning.

(Refer Slide Time: 51:00)



Let us see, why this is useful? We have  $x$  equal to 3 and we also have  $x$  equal to 7. So,  $y$  equal to complement of  $x$  bit complement of course, plus 4. This is what we want to compute when you actually have  $x$  equal to 3 and you do  $y$  equal to  $\tilde{x} + 4$  there is going to be an overflow that is the way we have the number of bits.

This is 3 and  $\tilde{3}$  is minus 4, so if this was all 0s 1, 1 you complement it. You get all 1s and 2 0s, so that is nothing but minus 4 in the two's complement notation then you added to 4, you really get a 0 with this 1. So, there is an overflow.

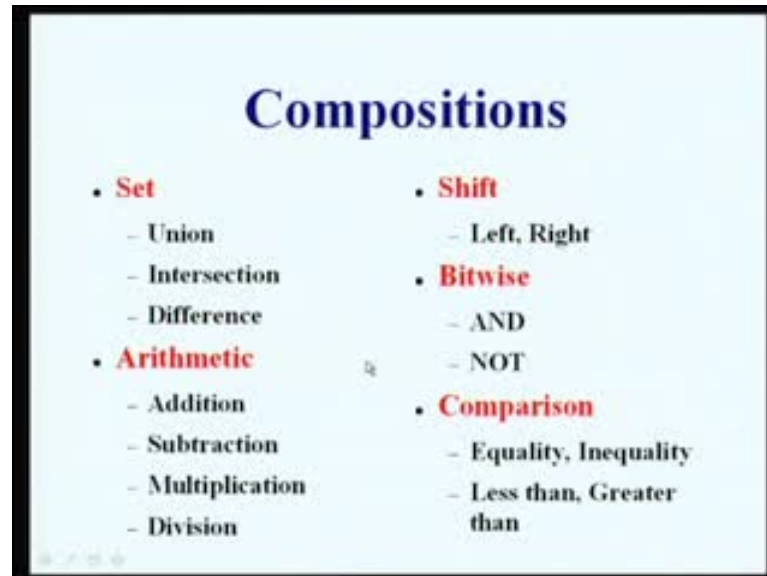
Whereas, if you take 7 and do  $\tilde{x} + 4$ , you have  $\tilde{7}$ , which is minus 8 add it to 4 and then you get minus 4 there is no overflow in this case. But the nice thing is the representation for  $y$  which is minus 4, 0, 4 captures both these factors even though there is an overflow and the value is 0, we really can capture it minus 4 and till 0. These are the lower bound and upper bound and then the step is 4, so minus 4, plus 4 is 0. You are capturing with 2 values the set contains minus 4 and 0 it captures both these.

Similarly,  $z$  is  $\tilde{y}$  so when you take 0 and complement it you get all 1s that is nothing but minus 1, so you have minus 1 here, you have minus 4 here you take a  $\tilde{ }$  on that. That is the complemented, then you get all 0s and 1, 1 that is nothing but 3. So, minus 1 to 3 with a step of 4 so again you get two values minus 1 and 3. This CLP accurately represents this computation all these computations with both the values.



This is not unrealistic situation, it is possible that x comes along this path and y x. This x comes along this path and the two values are different and now we need to actually represent the value computed by y that is can be done using our CLP very efficiently. Otherwise, we would have been stuck, if you had used a different configuration in this case.

(Refer Slide Time: 53:45)



So, we will stop this lecture at this point and in the next lecture we will continue with other operations on the CLPs and how CLPs are used in the WCET estimation. Thank you