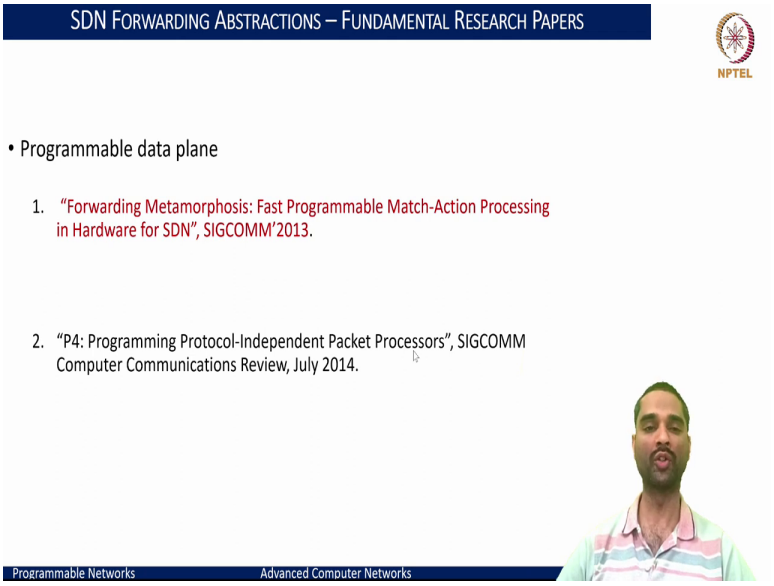
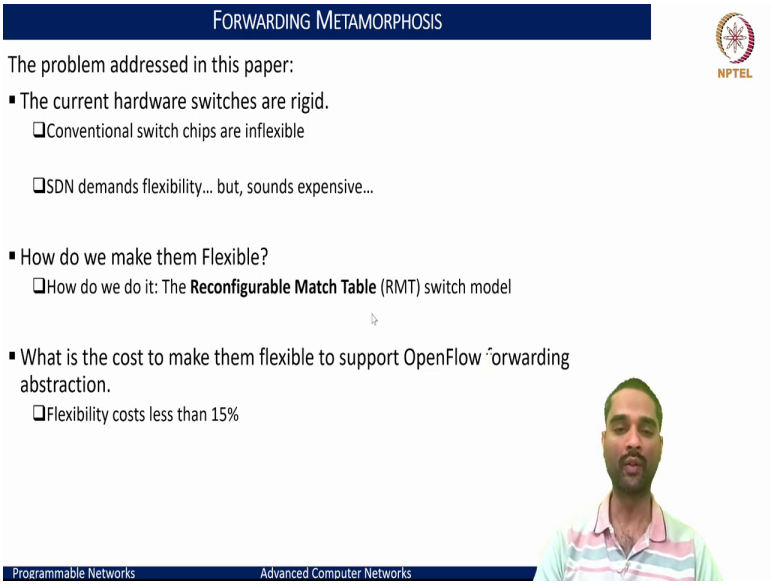
Advanced Computer Networks
Professor Dr Sameer Kulkarni
Department of Computer Science Engineering
Indian Institute of Technology, Gandhinagar
Reconfigurable Match Action Tables

(Refer Slide Time: 0:17)



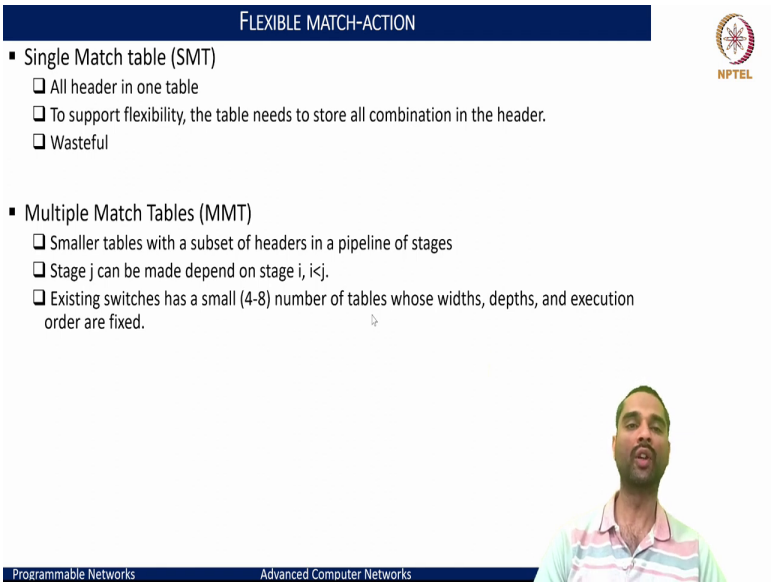


Let us now try to look into the core aspect and the components, and how forwarding metamorphosis or the fast programmable match action processing and hardware for SDN was realized. The problem that was addressed in this paper is to look at what were the current hardware switches and why they were rigid in terms of where the inflexibility arose in the

conventional switches in terms of programming the match action pipeline or the parser themselves and how to meet the demands of SDN that is to bring in the flexibility with less of expenses in terms of the hardware, costs or incurring the power costs and so on.

And the key means that was devised in this work was to build what we call as the reconfigurable match table, which later got generalized into what we learned as PISA. So let us try to understand the key aspects in terms of what optimizations, and what mechanisms were pulled in in this reconfigurable match table to make the ASICs, the hardwares to be worked in a programmable fashion.

(Refer Slide Time: 1:38)



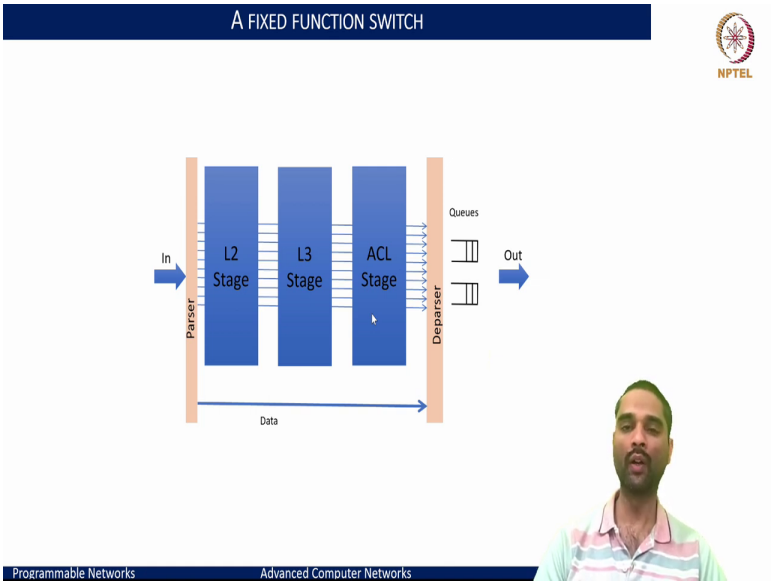
And like I mentioned earlier, the earlier switching hardware used to have like what we call as a single match action table where all the entries would be put in just one table where you want to match the bits of layer L2 or L3 all these would be populated in just a single table so that you had all the headers in one table where different combinations of the fields would be set, so you need to store all the combinations of the headers and see which entries would match and then process and of course, you can see that this is going to be wasteful in the sense that you are going to populate so many of entries like n bits would correspond to 2^n entries and all of the possibilities that you would want to put which you would likely to match would have to be put in one table.

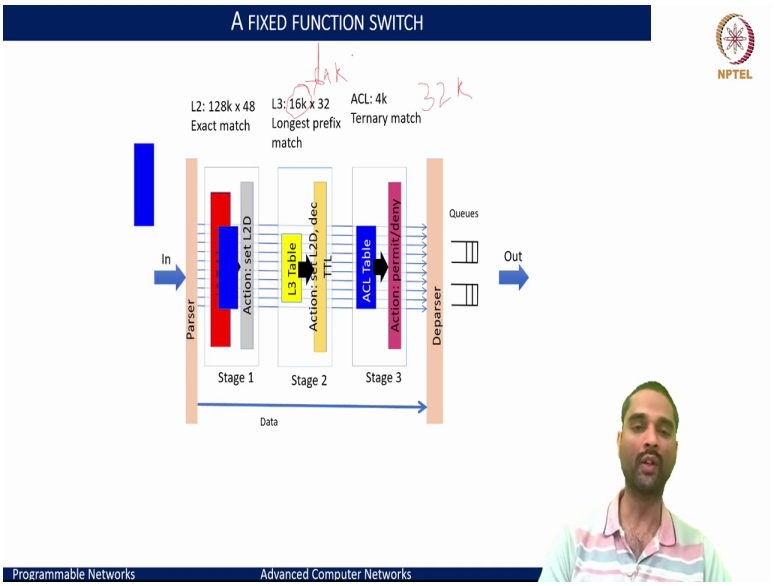
And alternatives grew as the match action mechanisms become much more efficient in terms of providing what we call the multiple match tables, where you had a smaller subset of headers that were being parsed and matched in different tables so you had a multiple number of tables or the smaller tables compared to having one big table and each table would match on to a set of header fields, this way it ensured that you are able to make the processing one in a pipelined stage wherein you can match on the header fields in a stage j which can be made depending on the stage i, where i is always before the stage j.

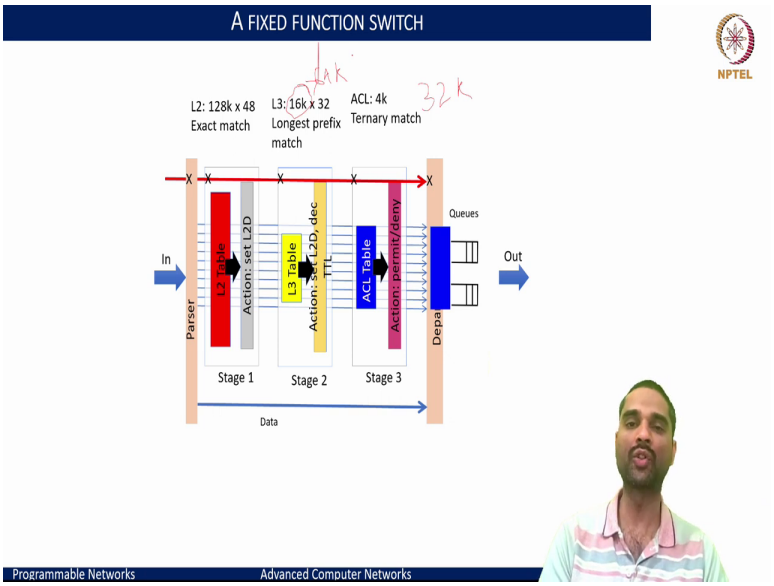
And you can process three stages in the pipeline, so it would get better optimization in terms of processing multiple of the packets as they go through and also you would save a lot on space because now different tables can be configured based on the width of the number of bits that each table needs to check.

So the multiple match action table is what is followed in most of this switching ASICs, and what you typically see is a set of four to eight tables whose widths, depths, and the execution order is somewhat fixed when they are baked-in into the ASIC but nonetheless, this is a much more better variant compared to the single match action table.

(Refer Slide Time: 4:26)







But when we look into this in a deeper view, what we can see here is a simple three-table entry that is being set up, and as the packets come you would want to do stage 1 that is maybe layer 2 table trying to check for layer 2 data, match on the layer 2 like source MAC destination MAC type and try to do specific action like set the layer 2 destination actions and as it proceeds further to stage 2 it would basically do layer 3 table where it would look up for specific IP source destination in the packet header, match on them and then take specific actions like it wants to decrement a detail before forwarding the packet out.

And stage three could be an ACL table or the access control where you would want to take specific actions on whether to permit or deny the packet from going forward. And each of these tables could be set with different configurations, so you can see at the layer 2, we could set it up as 128k entries each of 48-bit width that is exactly the ethernet MAC ID that we want to match on for the destination MAC ID that you would want to look up and take decisions, so we can make it as an exact match for the destination MAC.

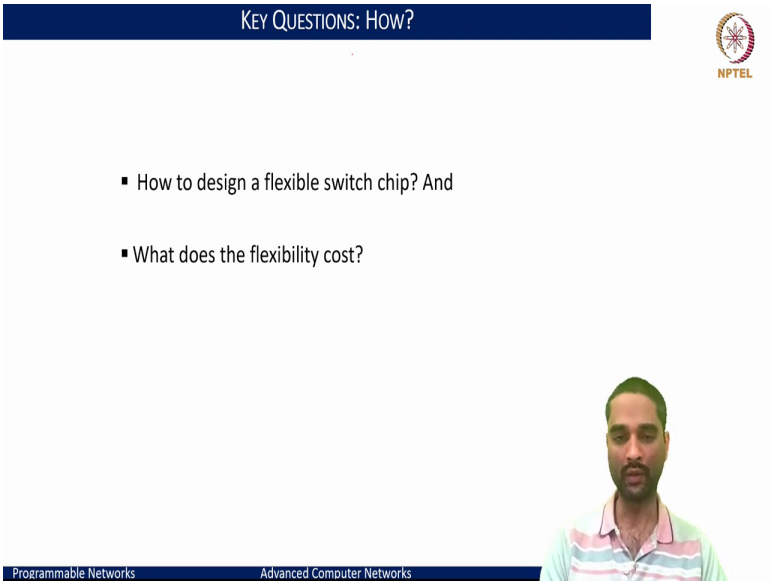
On layer 3, we could configure this layer 3 table with 16k entries a total of 16k entries each matching trying to match on a 32-bit address that is the IPv4 address space that we want to contain and see what IP addresses that they are going to take and we can even coalesce them into having a lot because of longest prefix match and put them in 16k entries rather than 2^{32} entries that would otherwise need and likewise ensure that we can match on a single entry in a table can match for multiple of the IP addresses.

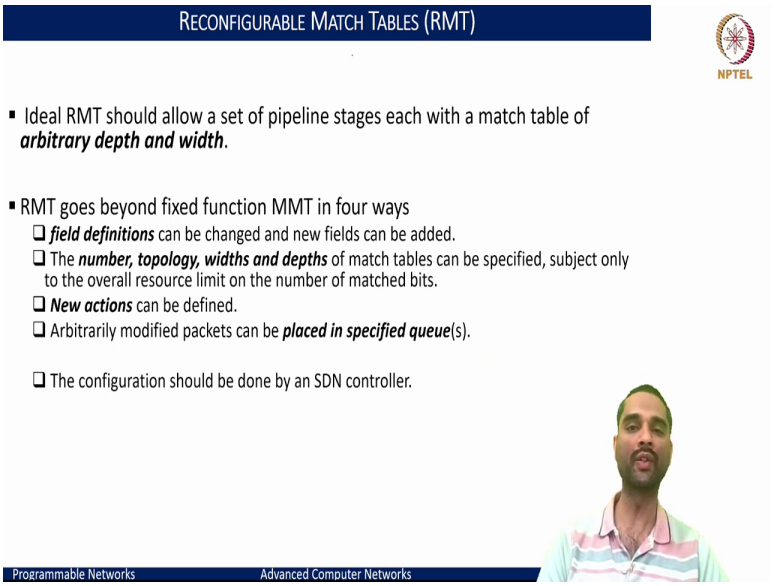
And here, we could apply the principle of longest prefix match, and likewise, for the access control, maybe we can have 4k entries trying to match on whether of many of the aspects in terms of the number of rules that you would want to say whether to take permit or deny so this is at best the flexibility that we get with the fixed function switch but once these are rolled in, there is no way that we can change any of these characteristics.

Now if I want to apply the same in an Enterprise Network, where my ACL requirements is around 32k or greater than 4k, I would not be able to do that at least, or if I want to apply this in a core network where my entries for layer 3 would be like more than 64k or greater number than 16k, I would not be able to change it either. So these were the major concerns when we looked at the specific fixed function forwarding patterns and in this, all the packets would also go through each of these tables and get processed one after the other, stage 1 and then stage 2 and stage 3 and then eventually deparse and then send it out.

And further, if we had any of these packets where it would not match on any of the entries in either layer 2 or layer 3, or layer 4 it would still do the processing at each of these stages, and you would incur saying that there were no matches and eventually it would deparse out and take specific actions so the overheads more or less would be similar for whether you had the match or no match.

(Refer Slide Time: 8:41)





And these were exactly the concerns, and the question then was how to design a flexible switch chip and also when we want to design a flexible switch, what are the characteristics that we want to look for and also, at the same time, look for the cost to be minimal. And this paper exactly tries to address these aspects, and it came up with what it says as an ideal RMT which should allow for a set of pipeline stages, each with a match table of arbitrary depth and width. That means the configurations, if I said that I needed a layer 2 with more than 8k entries, I should be able to configure them as I need like if I said earlier that I need to deploy the switch in a core network and I want to ensure to have lot more of IP entries IPv4 entries to be matched with 64k,

I should be able to configure that those many numbers of entries in terms of the depth and I also want to match on IPv6 not just IPv4 then I should also have the control to configure the width in terms of not just 32 bits but 128 bits and likewise.

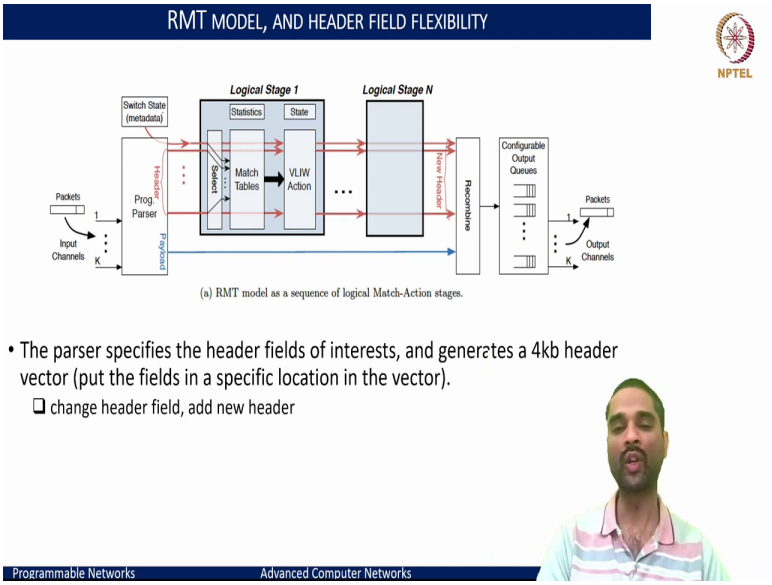
And this is the innate characteristic where you want to have the match tables to be configurable, and once they are configured what this differs, how this differs in the MMT approach where multiple match tables were there, is one, the num field definitions for each of the tables in the stages those could be changed, and new fields could be supported rather than just the fixed set of fields that the MMT would support as level 2, level 3 and ACL that we saw in the earlier part.

Second, the number of tables that we would want to support and the topology in terms of what stages that each table would want to pick in terms of when it tries to process, including each table's width and depth characteristics, could also be controlled and configured rather than saying that we only confine to what is being given are all having the same width would not be a good case because that would lead to a lot of wastage of resources. So if these parameters could be configured provided that we are within the resource budgets of the underlying hardware capabilities then it would enable a better utilization and better usage of the resources.

And third, and most important is again that we want to build on the action sets, so if I want to add any custom action that we want we can build on the existing primitives to define and add new sets of actions, and fourth we want to also have arbitrarily modified packets that could be placed in specified queues, what this really means is to ensure that whatever the packets that we want to modify we want them to not just follow the pipeline as what we just saw but rather make them to jump to a specific pipeline make them wait at specific queues, pick them up in different orders so that the pipelines can be defined for each kind of a packet processing that we want to do.

And these were, in essence, the distinctions that reconfigurable match tables tried to bring besides what we saw in the SMT and MMT approach,es and all of this, if it were to be done, we want this to be done from SDN control plane so that all of these configurations once we want to deploy we can update and change on the runtime and make these changes to vary using an SDN controller so that it can be programmatically controlled. So these were the core ideas with which the reconfigurable match tables emerged.

(Refer Slide Time: 13:09)



And the model looks somewhat like this, where you process a bunch of packets, and each of these packets could be processed in different channels. Here, what each of the packets as we want to process, you want to look at the packet headers and consider the packet payloads to be just add-ons without trying to look into those but you are really concerned about the header part of the packet that you would want to work and for each packet, you would want to build a state wherein you want to tag specific metadata with each of these packets as it gets processed and this processing can happen through multiple of the logical stages.

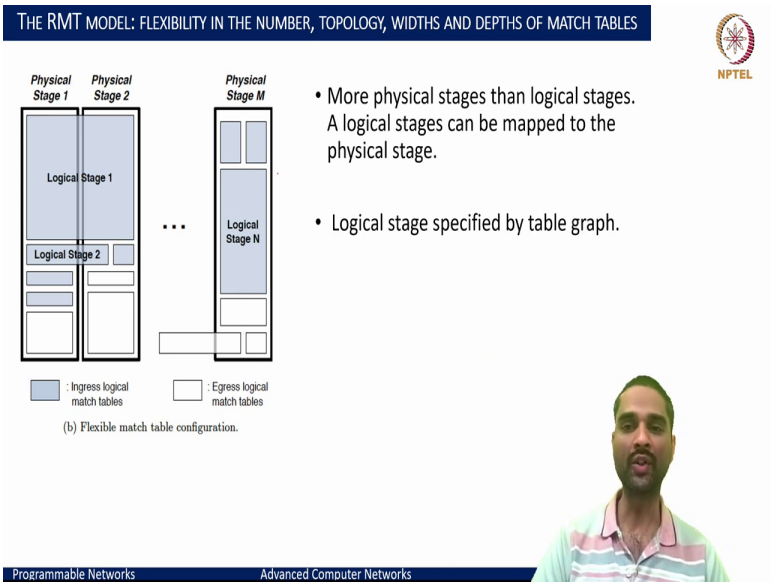
And when we say this processing for the header fields of interest that we want to do, we want to basically look at specific header length, and here the paper tried to say that we can do it as a 4-kilobit header vector that can be put for a given packet and this header vector can be processed through this multiple pipelines of these logical stages where you want to do the match action and take specific actions as updating of the state and metadata of the packets as you progress and process the data.

And once you process the data, you recombine the header with the payload, the header note that this need not necessarily be the same as what header you started with because the headers may have transformations because of the actions that you would take in each of the logical stages but add on the payload and recombine the packets and then set it appropriately for the outgoing

configurations and again the output channels or the ports on which you would want to send out the packets can be set up.

And this readily now allows us to change the header fields without worrying about the contents of the payload as long as we are able to match them up properly and also add newer headers or modify, alter the headers completely at any of the stages and still be able to stitch the entire packet and this is where the first of the flexibility in terms of what header fields that we would want to look and in fact any of the custom headers that we would also want to put because now we are trying to keep a 4kb header vector disregard of what header spaces the standardized headers that you want to look this can be accommodated in this space.

(Refer Slide Time: 16:03)



So now let us look at like what other aspects when we think of the logical stage versus the physical stage like when we looked at the hardware ASICs, they may have a certain set of physical stages that are hardwired but as a logical stage that we would want to define different widths on which we want to match different sets of actions that we want to take that means the width of a logical stage may encompass multiple physical stages.

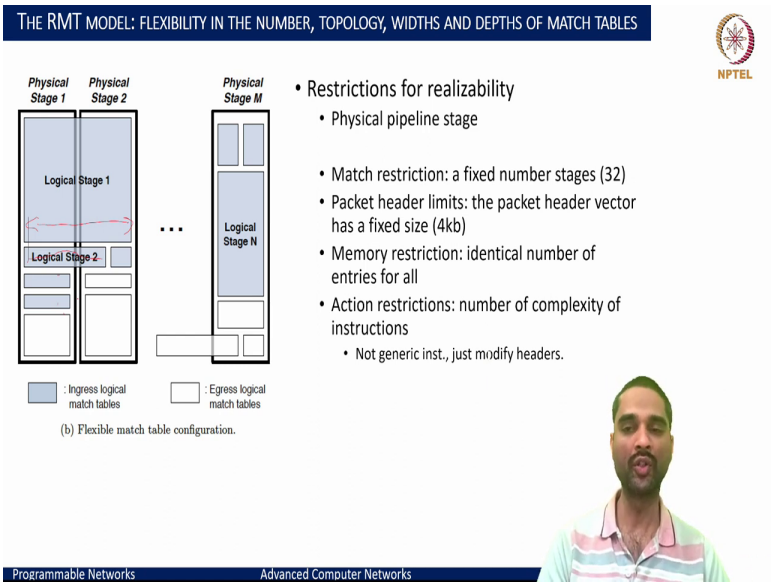
So if the physical stage comes with width support for 24 bytes or 32 bits and we want to match on 64 bits, then two of the 32 bits would allow us to do that. So a logical stage of 64-bit matching and processing could be realized through the use of two of the physical stages as we

see in this diagram here and the processing of these packets also is done in two phases, one is when the packets arrive you want to process the packets as the packets arrive and then you take the necessary actions and as you want to send the packets out you want to take specific actions.

And these were classified into what were called as the ingress where, as the packets arrive you want to match on the ingress or the input and then on the input side what all processings you would want to do logical match actions would be performed and that is shown by the gray color and the egress is when the packets want to be sent out, that you deparse and you want to take specific connections you would want to do the egress processing which could also be part of multiple stages that could be combined.

So here in essence each logical stage binds a state machine, with which you are parsing and taking specific actions, not the physical stages. And the logical stage can be mapped to a multiple physical stage.

(Refer Slide Time: 18:12)



And when we want to do this, and if you see that your hardware has a fixed number of stages, how do you want to define on the logical side like this also needs to have certain limits in terms of how many stages we can support and all the physical stages are basically the pipeline stages that you would go through but the analogical stage you may have different stages that you may

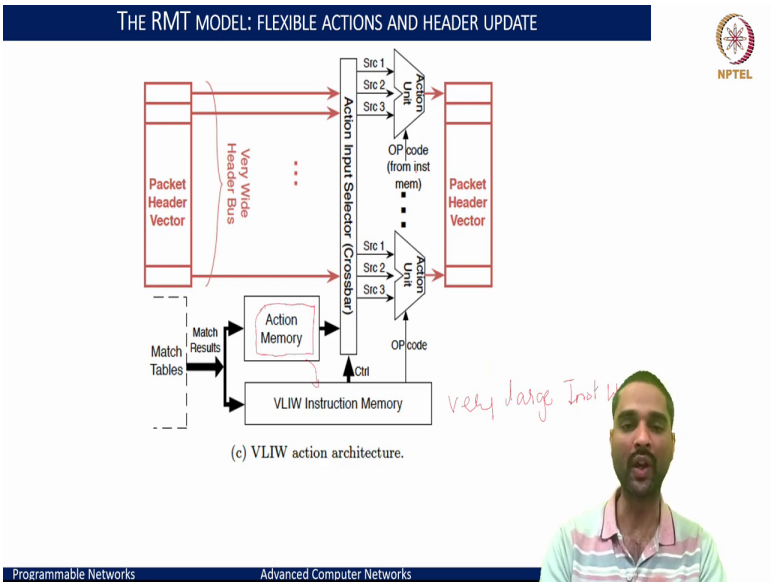
either combine or you may skip. In essence, the logical stagings need not necessarily be the cycle of physical stages chained one after the other.

And the other aspect also to remember is that when we have these flexible match table configurations, the memory restrictions and action restrictions need to be also considered in terms of what all entries we have in a given physical stage or a given logical stage, would they have all as the same memory restriction in terms of the number of bits, in terms of the size that you would want to accommodate or when I combine the two physical stages I can have multiple logical stages in a same physical stage notation that way I may have a width of one logical stage span in different bits as opposed to other logical stages which may have different bit width.

So we can see that we can have logical stage 1 with different width characteristics as opposed to logical stage 2 and maybe multiples of the logical stages which may have their own different aspects, but we cannot have a logical stage that will have width more than the combined physical stages that we can have, so that comes as the basic restriction and action restrictions also follow in terms of what kind of complex actions that is given ASIC could support, we cannot go beyond.

We may be able to do multiple of those but not any more complex than what is already being supported, but we should also have like multiples of header modifications if it supports for header modifications, we may be able to do multiple of the header modifications very easily, but if it does not support for swap or any of the complex modes of changing of the headers, then we may be confined to those.

(Refer Slide Time: 20:52)



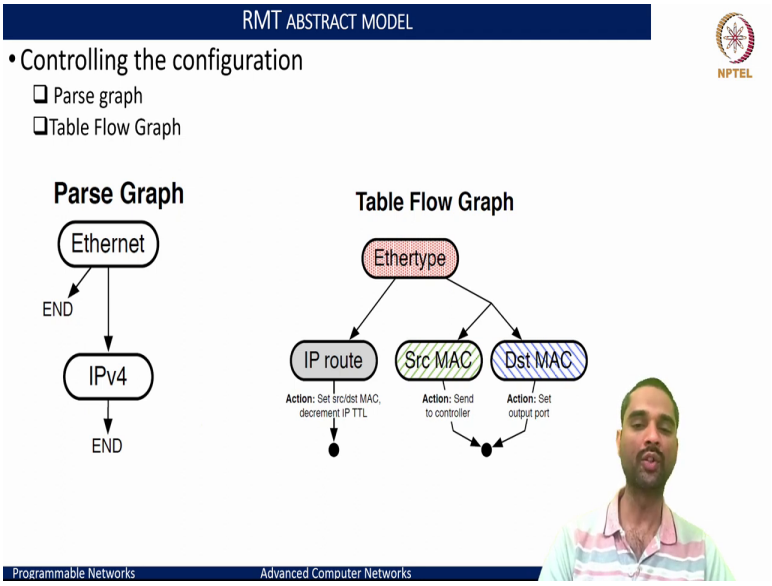
So how this processing really happens is, as the packets come, you logically separate out the packet header vector, which can be of 4k bits that you want, and separate out the payload from them, and this packet header vector would also be padded with the state information or the metadata that you would want to tag so that you can associate between the header as well as the payload and which of the stage, what processings are being done can also be carried as metadata.

And this packet header vector would then use the bits, and each of the bits that you want to match on, may result in a specific set of actions so in stage 1, if you see that you have to replace the source MAC with the current switch MAC ID or the port switch port's MAC ID that would be one of the actions, the other action being decrement the TTL, so all of these is what is contained within the action memory.

So a bunch of actions can be plugged and written as what we call as a very large instruction word which is a combination of multiple of the instructions that you would want to execute or actions that you want to take on a specific packet. So instruction 1 could be: change the source MAC, instruction 2: decrement the TTL, instruction 3: could be replace source IP with public IP and likewise. So we may have a bunch of actions that we would want to take and each of these action units would operate on this packet header vector translating the packet header vector from the original packet header to the new packet header that would go through as we process in each of the stages.

And this architecture, in a way, makes it a lot more flexible to add specific actions and update the header at one go trying to do different actions as we have different matches result in different sets of actions.

(Refer Slide Time: 23:13)



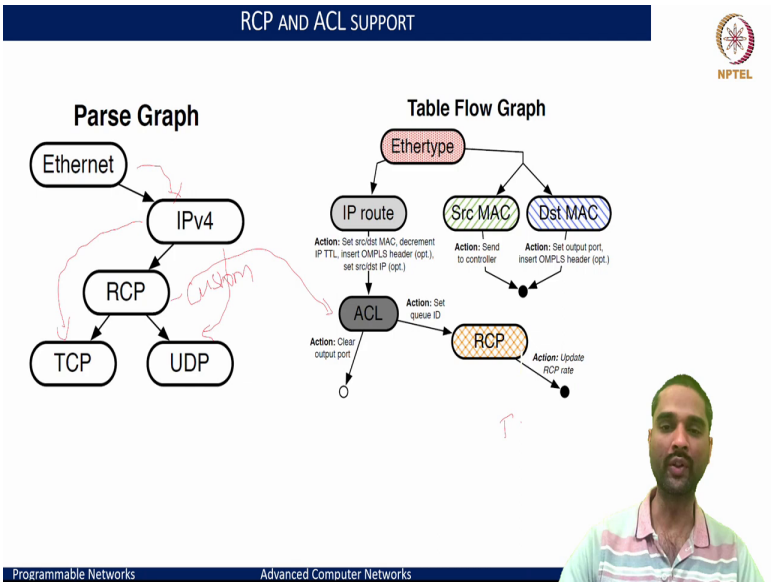
And for enabling this kind of operation, we require what we call as a state machine that is being built in two distinct graphs, one is called a parse graph, and the other is a table graph. And what a parse graph says, you start with the layer 2 header or ethernet and as you parse the packet you may see that you reached the IPv4, then you would want to process continue and process the IPv4 and then be done, and for each of the parsings you want to have specific match actions that is being done.

So when I parse the ethernet, I got the ethernet source address and destination MAC address and now I would want to change the ether type or based on the ether type, I want to look up what is next and follow and make some specific changes then there is an association of the state at ethernet to do an action at the ethernet layer like when I have said we want to change the source MAC then whatever the ethernet data that we have parsed, the source ether we have to replace that with the action would be then to take a replacement for that particular thing of source MAC with a new MAC address.

And when we parsed the IPv4, then the action could be that I route or that is decrement TTL or set source and destination MACs, and likewise so the table flow graph dictates what kind of actions you want to take at each of these parsed entities, while the parse graph maintains the state of what headers that you have matched and for each of the header what kinds of action that you want to take.

So if it were just the ethernet packet and no IPv4, you would on the table graph look at the ether type and take the actions on source MAC destination MAC like either set the output port or send to the controller as the specific actions or if it were IPv4 packet then based on the ether type you come to the IP route and take the actions as set source destination MAC and decrement IP TTL and so on.

(Refer Slide Time: 25:32)



And this parse graph and table graph have to be there as state machines that need to ensure that for each packet you are embedding this metadata to know what each stage is trying to do likewise when I said like consider a simple parse graph which says we start at a layer 2 with the ethernet and next up at layer 3, you match on the IPv4 and the layer 4 typically is either a TCP or UDP but we want to match on a custom header, RCP that is interspersed between IPv4 and TCP or UDP then we would be able to do that with this kind of a parse graph which says ethernet, IPv4, RCP and then it could be either of the TCP and UDP.

And this is how we can now get the benefit and flexibility to match on the custom headers, which need not necessarily be the standardized headers that are interspersed, like RCP here corresponds like the rate control protocol, and here it is interspersed between the layer 3 and layer 4 header, and we will be able to parse and process them at much more ease by mentioning what would be our parse graph and for the corresponding parse graph we could have the table flow graph which says if the packet were the ethernet type with the IPv4 match, we would take the action set as setting the source destination MAC, decrementing the IP TTL or inserting the MPLS headers and likewise.

And on the actions, if there is access control setup that we want to do for saying set the QID as RCP for if it were RCP, we would want to jump and do the access control setting the QID of RCP to ensure that we are rate controlling that specific packet and then correspondingly take the TCP or UDP set of actions that we would want to do in further processing. So this way, we can now support different kinds of protocols and facilitate the processing for each of these packets ready.

(Refer Slide Time: 28:04)

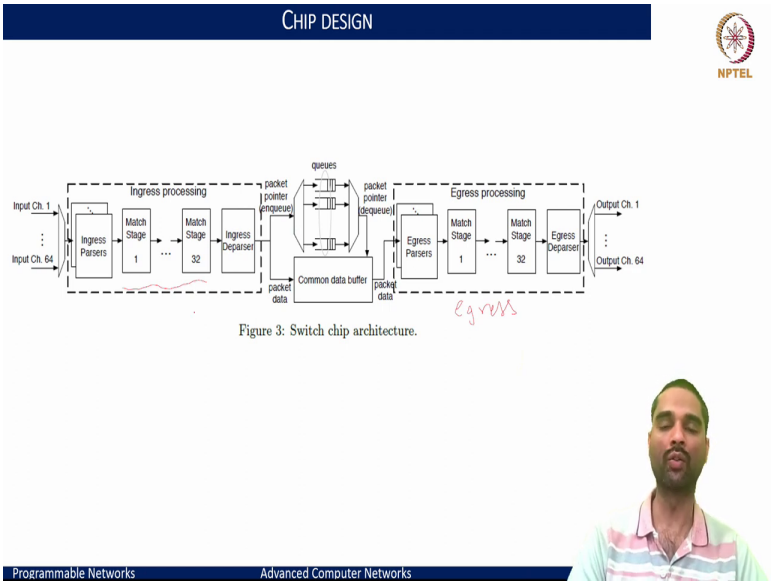
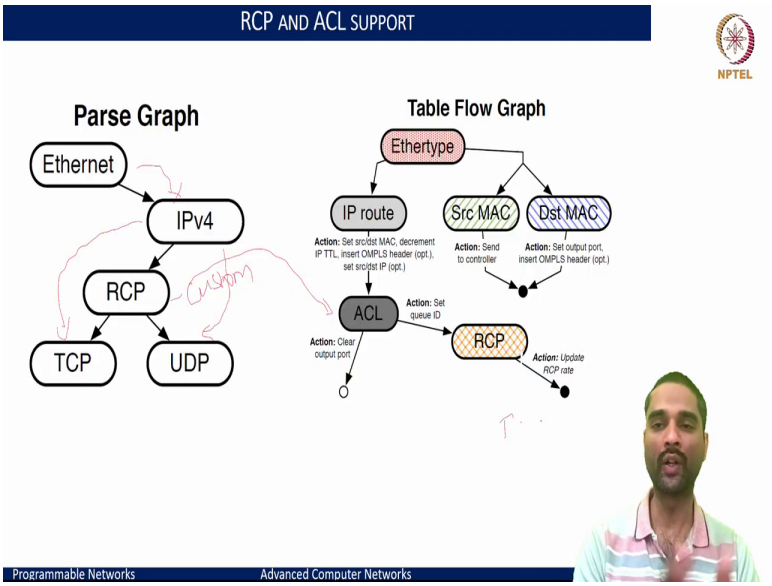


Figure 3: Switch chip architecture.



And this design overall, like when we envision supporting this kind of a parse graph and a table flow graph having the match action tables in pipelines, the design looks like this where you have a bunch of input channels on which the packets would arrive, and as the packets arrive they go through the ingress parser where all the packet headers are being extracted and we have the packet header vector that is being set up and then you start doing the ingress processing as the packets arrive, you start doing multiple stages, the match stage 1 match stage 2 and likewise up to 32 match stages through which the packets would be processed.

And once the packets are processed, and specific actions are being taken at each stage, you would deparse or basically reassemble the packet along with the content that it needs to process and enqueue it on any of the queues for further processing and once the packet pointers are updated in any of these queues, then you look up what is the port on which the data needs to be sent and what is the specific processing on the output that you would want to do being carried out in the egress pipeline, and this is basically the egress pipeline processing, and that is done before emitting out the packet on a specific output channel.

And this generic framework of processing with input and output also enables us to think, if I did particular processing and I wanted to be done re-queue it in the ingress for some other mode, we could also basically dequeue and send it packet back onto this for reprocessing of the packet and we may wonder why we need separate ingress processing and egress processing with 32 different match stages in each case.

Consider the very simple aspects like when we want to do a multicast, you would have the same kind of a packet that needs to be sent out on different ports, and here if we want to customize like for the congestion bed or any of the MAC destination addresses we definitely cannot do it on an ingress because it is the same packet now that is going to be sent out on multiple ports.

So we need to do that in the egress stage where you would want some customizations or flexibility when you want to send packets same set of packets on different modes. Hence egress processing, although manually used, is of importance when we look at the processing for different cases otherwise, we will not be able to distinguish what kinds of packets would be sent out on different aspects on different ports. And thus, this kind of chip design was achieved with different hardware aspects of 10Gb NICs or a 40Gb NICS, different 64 channels, and so on.

(Refer Slide Time: 31:31)

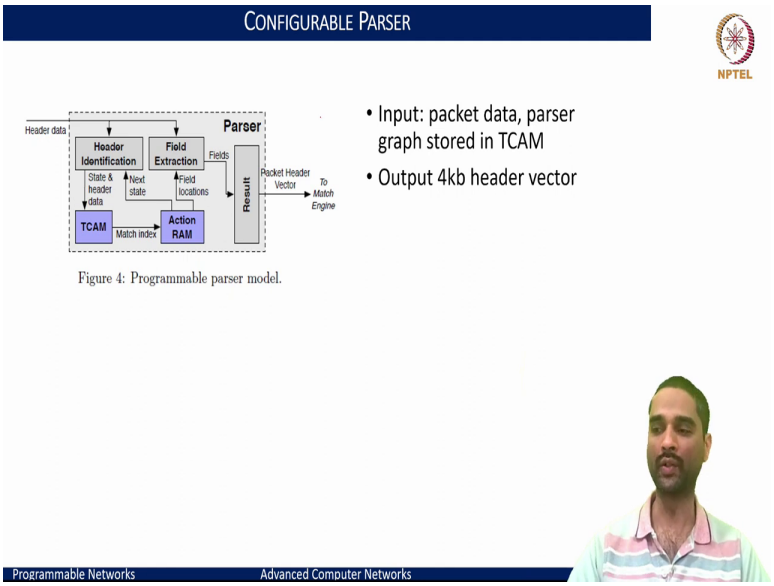
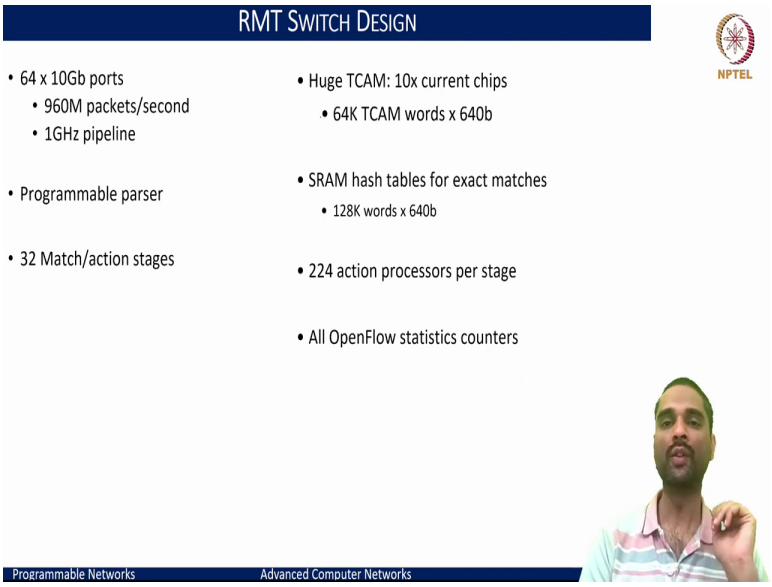


Figure 4: Programmable parser model.

But let us try to see what are the major components here, and their design aspects and understand the components. So first is the configurable parser, this parser accepts basically the incoming packet data and produces the 4k bit packet header vector, and this packet header vector is what is then used in the match action pipelines to be acted upon, and the output, in this case, is fixed to the 4kb header vector and how this really processes is basically you get the header data and identify the set of header fields.

And you may do like TCAM using the TCAM to match on the specific bits it could be an exact match it could be the longest prefix match, and you would then identify, I parse the packet, found out the ether type based on the ether type I would want to see what is the next state that I want to set, what are the next fields that I want to extract so if it is ethernet layer I would want to extract source MAC, destination MAC, ether type and all, as the fields get extracted they are going to be written into this packet header vector which would then be sent to the match action. And in general, this loop repeats to parse each packet and do the parsing repeatedly for the set of packets till you extract all the headers.

(Refer Slide Time: 33:11)



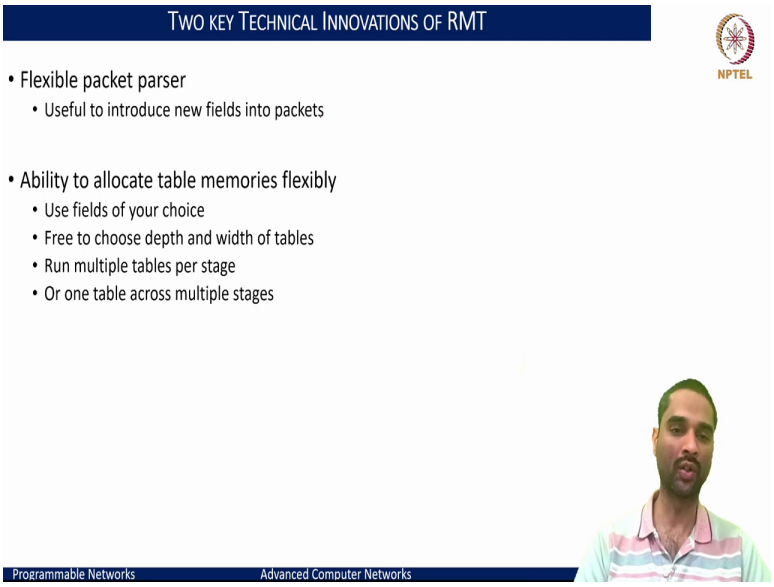
And the switch design, in this case, corresponded for the RMT to have 64 of the 10 Gb ports, which had the ability to match 960 million packets per second with a pipeline rate of 1 GHz processing where the clock for each of the stages that he would process is clocked at a 1 GHz and this programmable parser with 32 match action stages which basically had the major cost coming in terms of a huge TCAM which is like 64k TCAM words of 640 bits each which is almost 10 times of typically what you see as 4k or 2k TCAM words that are accommodated in the custom ASICs.

And also this adds to the memory overhead in terms of the SRAM hash table where if you want to manage the exact matches, you would hash, keep the entries as a hash and try to see how many things you can match on, and that is done as 128k words of 640 bits each and overall this

processing at each match action with 32 stages on the ingress 32 stages on the egress, all require 224 action processors that were set up per stage.

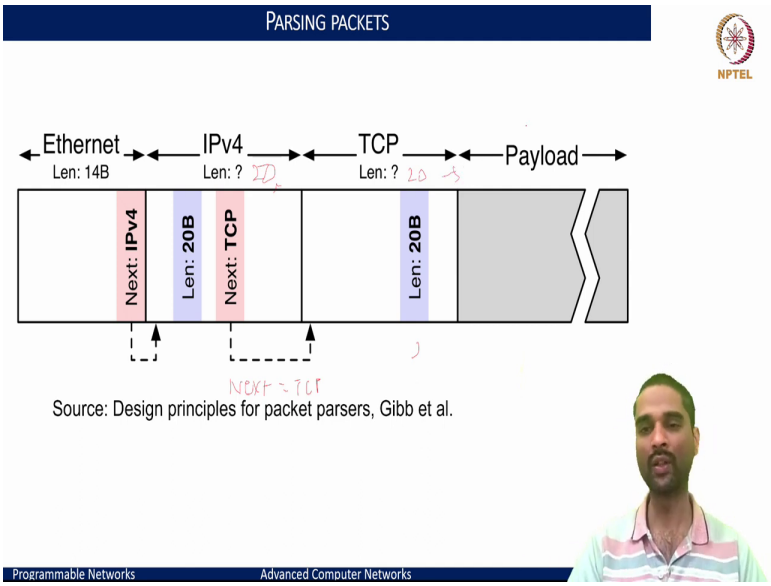
So on the ingress per stage, you had 224 action processors that they would use to keep either the counters taking specific actions and likewise, and this was the switch design.

(Refer Slide Time: 34:52)



But the key innovative aspect that we need to consider is one with respect to the flexible parsing, which were useful to introduce the newer fields into the packets and enable readily to act on any of the new kinds of data that otherwise were not standardized headers, second is the ability to allocate table memories much more flexibly because now we dictate the choice of the table width and depth that we would want and combine multiple of the physical entries into logical entries or in a single physical stage we could define multiple logical stages in better utilizing the memory at each of the stage.

(Refer Slide Time: 35:37)

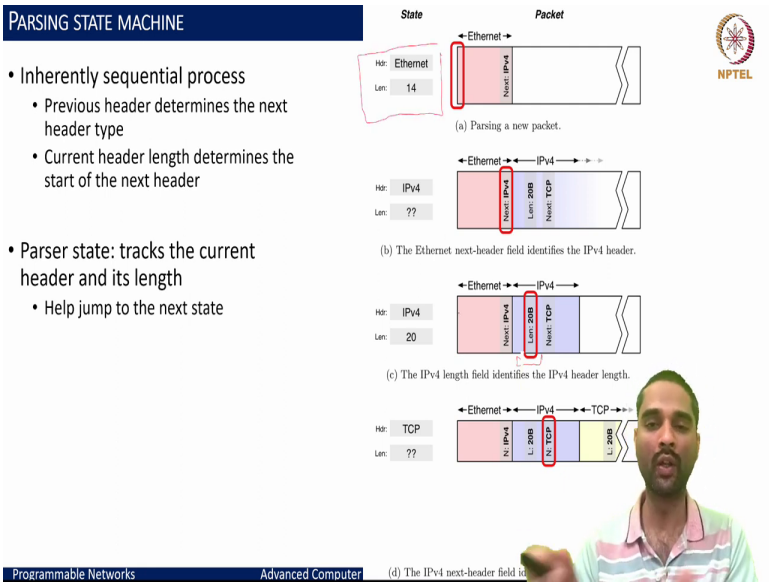


So to look at how the parsing really progresses like if we have a typical packet with ethernet of 14 bytes followed by IPv4 and then TCP and payload, the first aspect is now when you start, you start with an offset 0 and progress towards looking at what is the ethernet header, so you always often parse the first 14 bytes identify what is the next that is the ether type and if the ether type happens to be IPv4 0x8000 or the value corresponding to that value you would see what is the next entry that you would want to look up.

And likewise, your parse graph would then say switch from ethernet to IPv4, and when you look into the IPv4 you look at a fixed 20-byte length or look at the field with a length and say in the IPv4 header, the length field corresponds to 20 bytes then you would push further by 20 bytes to go and check what is the next header and to do that you would see in the IP header what is the next type with either TCP or UDP and correspondingly say what should be the next value.

So as you progress in each stage, you would mark, length currently is 20 bytes, and next is TCP, and that is how you basically keep on adding the offset and then move again, again at a TCP layer, you would see next length is 20 you would add that much and go further if there were any other headers that you would want to parse.

(Refer Slide Time: 37:23)



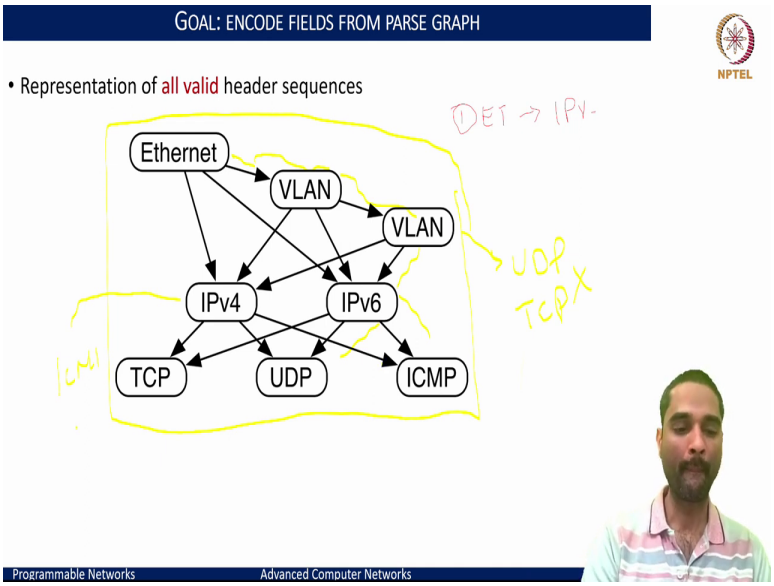
And this readily helps us to see that we can process the packets in a sequential stage starting from offset 0, adding on the sets of offsets until we reach and get the entire packet, and that also means that we are no more confined to a given sequence of packets. After ethernet, if this were to be VLAN, then the information in the next would point to VLAN, and we know the bytes that we would want to look for in the next VLAN header has 16 bytes and then take it from there how to vary looking up different packets.

So this parsing mechanism, I mean, we can see that it is inherently sequential because you have to start with the base header ethernet and then look up what are the next subsequent headers and start catching up on them, but the current header length in a way determines what is the next header or where is the start of the next header that you are really looking so these two important informations that we would need is the current header type and the length of the header associated with each as a state that needs to be maintained in the parse graph, so that we are able to process and do all the header parsing correctly.

And also if when you say that we start with a new packet first, you extract the ethernet from the ethernet, you see the next field see, the IPv4, and you go to the IPv4, and here you need to deduce the next length of the IPv4 from the field within that means we need to know exactly the fields or what is the offset for this length that I need to look up to know how many bytes I need to jump further.

And likewise when you look up the TCP, I need to know the exact information of the header in data that I need to look for to say how many bytes I need further to go and look for the next, and this model of parsing the state machine makes it much more simpler and easier to look for the contents and adapt it for any of the headers that we would want to incorporate as long as we are able to extract what is the type of header and what is the length information, at each stage we would be able to add process each of these headers appropriately.

(Refer Slide Time: 40:09)

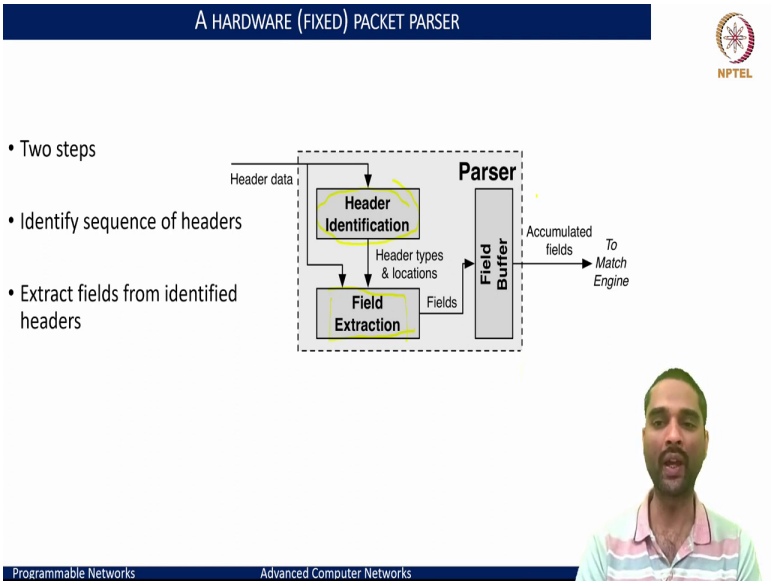


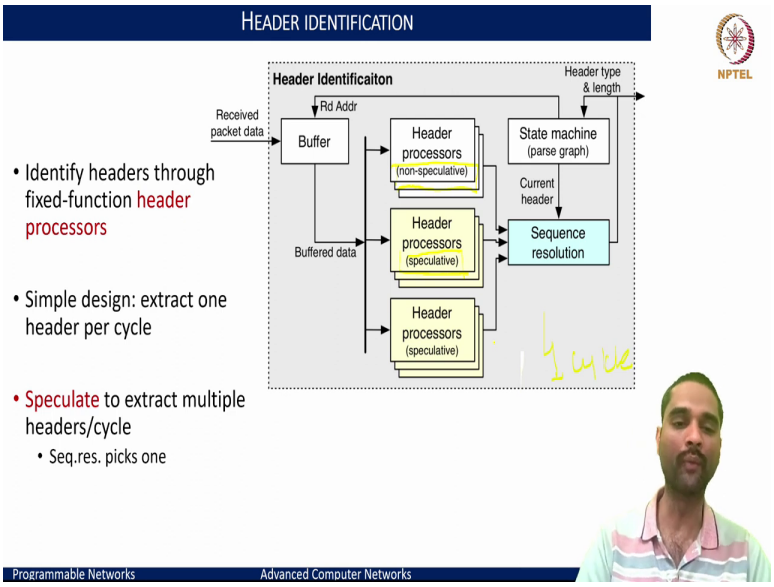
And here, if we say that we have a parse graph, and this looks like a big mesh, but what it really says is we have a bunch of valid header sequences and one of the sequences is basically ethernet followed by IPv4 followed by TCP; the other is we have the ethernet followed by VLAN followed by IPv6 and then it could be either a UDP or ICMP and likewise we may have basically ethernet VLAN followed by another VLAN and then IPv6 and then ICMP or UDP.

So we can have so many of the parse graphs, which basically help us understand what are all the valid sequences that we can build, and given this kind of a graph, we would be able to say that we want to match on ethernet first followed by VLAN and if VLAN has another VLAN encapsulation we can look up on that VLAN and then go on to the next but if we had any of the combinations saying VLAN jumping to UDP or jumping to TCP this would not work.

So all the combinations that are being expressed in terms of what all sequences that we can facilitate or support would be used, and in this case, if we had IPv4, TCP, UDP, and ICMP and ICMP is not in the parse graph list that we are looking then we are only saying I see IPv6 with ICMP or IPv4 with ICMP are as to combinations that will be supported in this case.

(Refer Slide Time: 42:14)





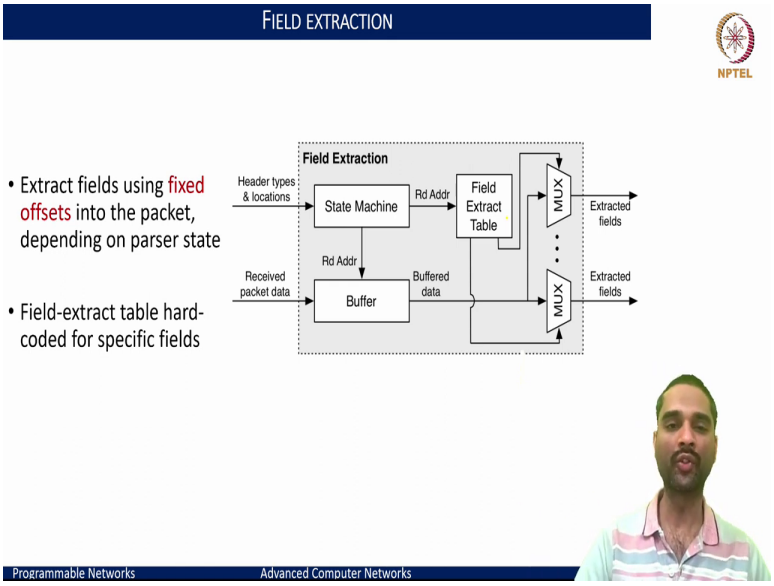
So more generally, this parsing happens in basically two steps first, as we get the header data, you identify the header information, and second, you extract the associated fields that you would want to look up and add them to the field buffer, and every time there is an edition of these

headers that you match, you would then add the corresponding actions in the bucket. So there is a common buffer where the headers are going to be processed for the ones that you have matched in a non-speculative manner, or these are the ones that are confirmed headers that we have seen, and for all of these, we would have the sequence of operations in terms of what matches what actions need to be taken.

And once we identify the header and we say there is a length and we want to look up, those can basically be added as speculative header operations processors that you would want to do and, in a simple essence, what every time there is a sequence resolution that happens, all of this happens in a one cycled unit and each, for each cycle you would go and do one stage of operation and add the set of actions that you would want to process.

So in a very simple design, you would extract one header per cycle, but in a more optimized design, we can do the in a single cycle we can also extract multiple headers and speculatively extract the earlier rather than just doing in a sequential fashion, and this is where the things add up as optimizations that can be performed in header parsing ahead of time in one cycle we may be able to do multiple sequences.

(Refer Slide Time: 44:15)

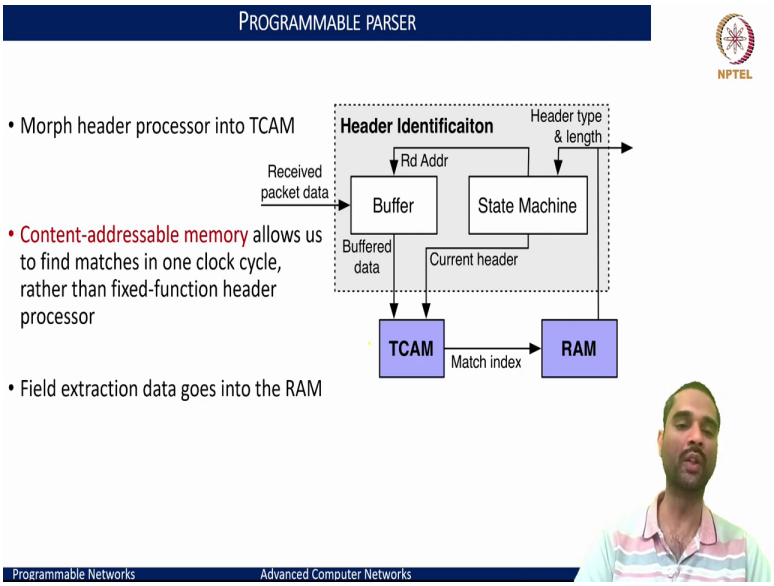


And once we have extracted the fields, we need to look up the match, and for this, we would use the state machine that is being built on the parse graph, and then we use the read address to

correspondingly do the table matches and if the extracted fields we see that tables that are being hard coded are able to see a match in any of the entries that you would see then you would basically go ahead and do the processing on the action side.

So there is a state machine that is updated for the parse graph, and there is a state machine that is updated for the table flow graph, and once we extract these fields that is dependent on the current bar state as we go to the next stage this current bar state updates could reflect back in the additional changes.

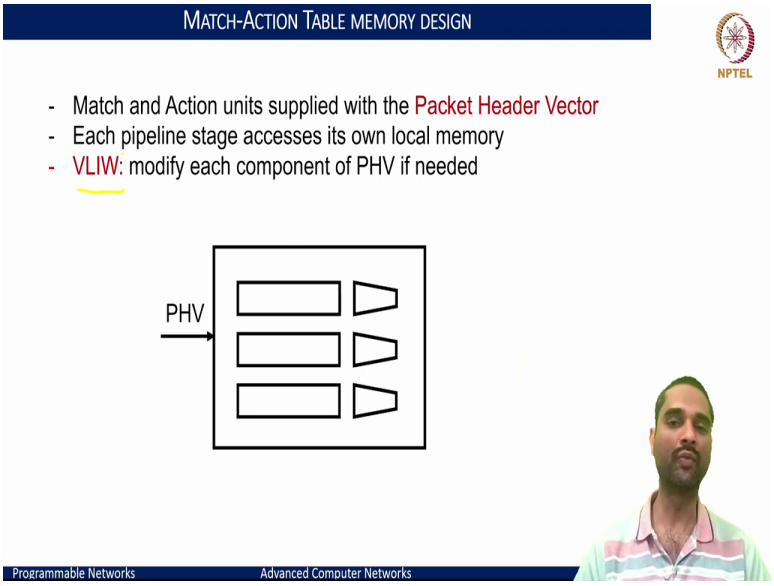
(Refer Slide Time: 45:15)



So this matching can be correspondingly done with basically the TCAMS where you specify what is the TCAM entry and then also allow it to be matched with the RAM or the corresponding entry in the RAM so that you can extract the data from the RAM and then match and then take corresponding actions and the utility of TCAMS, although they are expensive power hungry, they allow us to find the matches in one clock cycle, so when you want very high performance you would go for the TCAMS but maybe at the expense of cost and power.

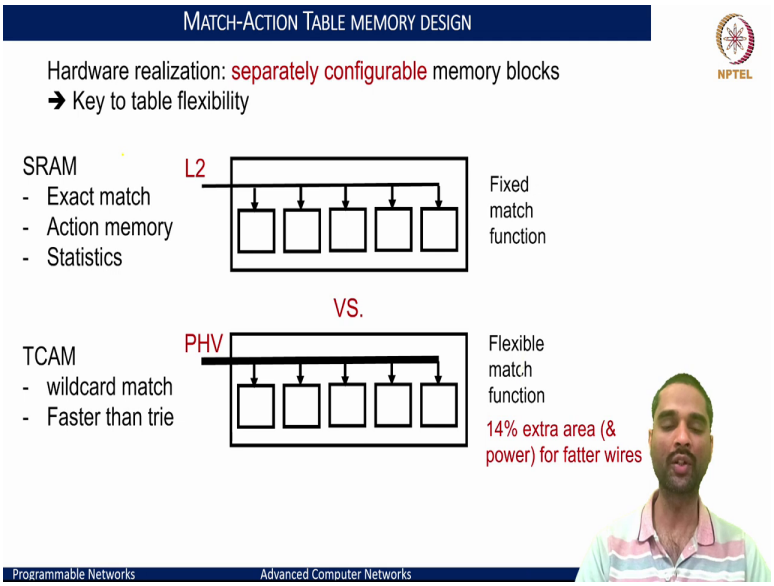
But this process when once we save like the most optimal parts can be put in a TCAM while the less optimal ones can be put in the RAM where you know that 90 percent of the hits are happening for a specific entry, then you would want those to be in the TCAM, and this is how the things can further be optimized.

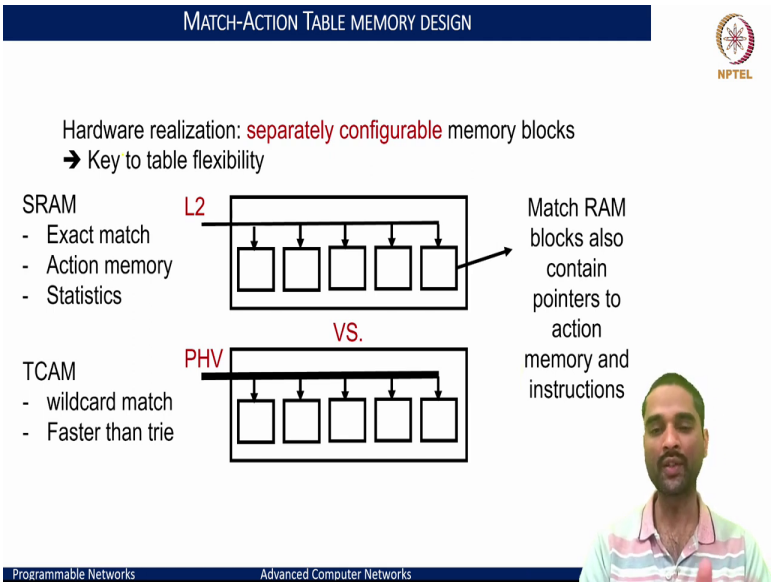
(Refer Slide Time: 46:27)



And the last part is about how to build the match action tables in the memory design, and like we said, for each of the match and actions, the unit is supplied with the packet header vector, which basically is the manipulation of the packet header that is being extracted and as the actions are taken you would have a modified header information, and for this, the actions have to be set up as a part of the information that you would give on a pipeline for each of the stages as the instructions like add, modify, remove or updates that you would want to build on this particular header bits. So this is encoded as what we call as the very large instruction words, which basically modify each of the components of the PHV as necessary.

(Refer Slide Time: 47:24)



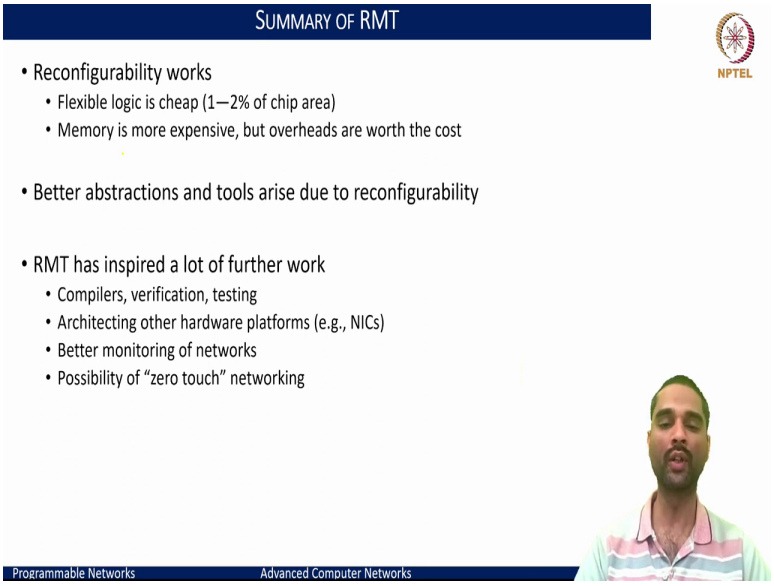


And this, again like, can be realized in hardware in different ways one just using the SRAM where you can have an exact match on each of the fields, and once this fixed match happens, you would be able to take necessary actions, or as on TCAM you could have like the packet header vectors having a wildcard matches which can basically now allow you to take flexible match functions based on what number of entries really matched.

And even LPM, in this case, can allow you to have flexible match functions and but this TCAM with this incurs extra power like I mentioned, with extra area requirements as well nonetheless, these match blocks now basically what they would contain is a pointer to the set of actions that

you would want to take and those actions basically encoded in memory as the set of functions or instructions that you would want to execute and this usage of TCAM like you learned earlier about the trie approach and it is that the TCAM wildcard match is much more faster than the trie based approach and nonetheless they are power hungry.

(Refer Slide Time: 48:49)



So, in summary, what we have looked at is the reconfigurable match tables and the way they work and enable for the switching to be made much more dynamic is, you know very cheaper way in as opposed to, like saying the logic now is much more flexible, but it comes at a cost but that is very minimal. Second like, there are trade-offs that can be made in terms of what components can be chosen with respect to having increased die area, increased power, etc, with respect to how the TCAMS are used or SRAM or DRAMS can be used.

But overall, what it really provides is a better abstraction and reconfigurability that is needed with the switching chips, and in fact, the RMT was the very pioneer work that led to the development of how to build the compilers, what are the means to build the verifications and testing for such frameworks and how to build the programmable network interface cards, how to better monitor the networks and also are paving in terms of what we call a zero-touch networking where like the entire automation of the networking systems can be done much more readily with the programmability rather than having a human intervention or configurations to be done.

And this, in fact, paved the way for what we call as the P4 programming, and most of the aspects that we just discussed are borrowed in the P4 programming.