


Advanced Computer Networks
Professor Dr Sameer Kulkarni
Department of Computer Science Engineering
Indian Institute of Technology, Gandhinagar
Lecture 47
P4 Programming

(Refer Slide Time: 0:17)


SDN FORWARDING ABSTRACTIONS



- Programmable data plane
 - "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN", SIGCOMM'2013.


 - "P4: Programming Protocol-Independent Packet Processors", SIGCOMM Computer Communications Review, July 2014.

IN THE BEGINNING...




Programmable Networks

Advanced Computer Networks



IN THE BEGINNING...




- OpenFlow was simple → A single rule table
 - Priority, pattern, actions, counters, timeouts

- Matching on any of 12 fields, e.g.,
 - MAC addresses
 - IP addresses
 - Transport protocol
 - Transport port numbers

Programmable Networks

Advanced Computer Networks




Now let us try to look into the P4 programming protocol-independent packet processors paper in a bit more detail. So to begin with, like I discussed before, what we had was a simple OpenFlow with a single rule table where you would have different aspects to match on like priority, pattern,

actions, the key counters that you would want to keep track of, the timeouts on which the flow rules would expire as simple rules but then the matching was confined to just 12 of the fields when the OpenFlow 1.0 began that were MAC addresses, IP addresses transport protocol and port numbers.

(Refer Slide Time: 1:07)

OVER THE PAST FIVE YEARS...



Proliferation of header fields


OpenFlow Version	Released on	#Match Fields (Headers)
1.0.0	December 2009	12
1.1.0	February 2011	15
1.2.0	December 2011	36
1.3.0	June 2012	40
1.4.0	October 2013	41
1.5.1	March 2015	44

Multiple stages of heterogeneous tables

Still not enough (e.g., VXLAN, NVGRE, STT, ...)

Programmable Networks

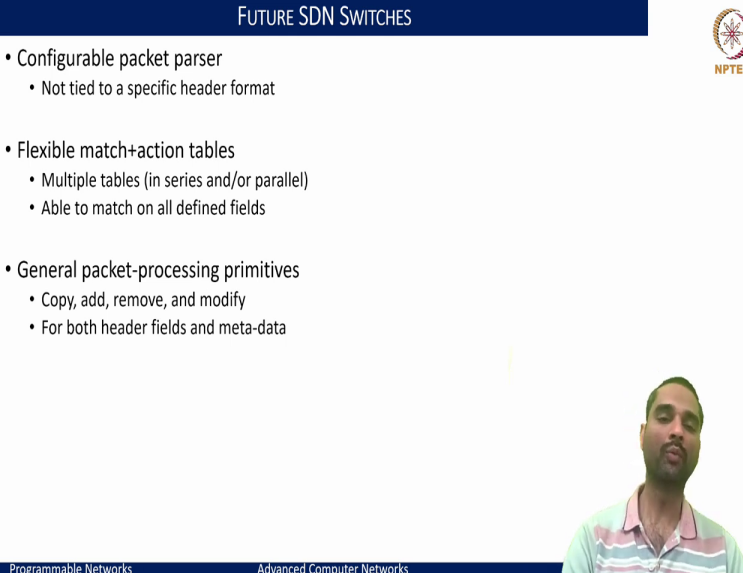
Advanced Computer Networks



But as the advancements happened, there were requirements to match on many of the other fields, and as we saw earlier, as the OpenFlow versions changed, we started to see a good ballooning in terms of the match fields that we would want to use, and it started from all the way 12 to 44 match action fields as in version 1.5. So there was a big proliferation of the header fields, and still, it was not sufficient and this proliferation also had another impact in terms of now you needed multiple of the tables to match on different kinds of these fields because each is a different bit width and you want different characteristics to be considered.

So all of this started to raise questions in terms of what we really need to rethink in terms of trying to match different sets of fields, and that is where the reconfigurable match action tables really paved the path, and if we consider network virtualization and overlay networks that we spoke of, we have a lot of encapsulation models like VXLAN, NVGRE, STT and so on. So looking at just one set of headers in one layer may not be sufficient. We may have to look further into the encapsulated header format to extract specific information. Hence, just where OpenFlow did not really help as much if we had to do any further.

(Refer Slide Time: 2:41)



The slide is titled "FUTURE SDN SWITCHES" in a dark blue header. It contains a bulleted list of features: "Configurable packet parser" (with a sub-bullet "Not tied to a specific header format"), "Flexible match+action tables" (with sub-bullets "Multiple tables (in series and/or parallel)" and "Able to match on all defined fields"), and "General packet-processing primitives" (with sub-bullets "Copy, add, remove, and modify" and "For both header fields and meta-data"). The NPTEL logo is in the top right corner. A video feed of a man in a striped shirt is in the bottom right, and a footer bar at the bottom contains "Programmable Networks" and "Advanced Computer Networks".

- Configurable packet parser
 - Not tied to a specific header format
- Flexible match+action tables
 - Multiple tables (in series and/or parallel)
 - Able to match on all defined fields
- General packet-processing primitives
 - Copy, add, remove, and modify
 - For both header fields and meta-data

And this is where the configurable packet parsers, which were not tied to any specific header format and allowed us to look for any of the custom header formats, became very meaningful and also the flexible match plus actions that we wanted to really deploy with multiple tables and not necessarily that all the tables are in a serial order they could even be set up as parallel tables that we would want to match like once we have built the packet header vector we could match on layer 2 layer 3 all of those at one go as we parse. So the ability to match on all of the defined fields in a go in either a serial or a parallel fashion would greatly be of use.

And likewise, when we match and we want to do the packet processing, the key primitives of copying a packet or adding certain headers, and removing or modifying specific headers are essential to make meaningful manipulations and process the packets within the data.

(Refer Slide Time: 3:46)

WE NEED A HIGHER-LEVEL INTERFACE



- To tell the programmable switch how we *want* it to behave

Programmable Networks

Advanced Computer Networks



And this is where the need for a high-level interface as abstractions like what we saw with RMT as the key configurations that we build within the hardware, but now what are the key abstractions that we would want to provide at a higher level so that the switch becomes programmable and it is easier to manage and flexibly do that, these were the key aspects that P4 tried to look at that is to tell the programmable switch how we exactly want it to behave and what is the means that we can try to build such a model or a behavior.

(Refer Slide Time: 4:26)

THREE GOALS FOR THE LANGUAGE



- Protocol independence
 - Configure a packet parser
 - Define a set of typed match+action tables
- Target independence
 - Program without knowledge of switch details
 - Rely on compiler to configure the target switch
- Reconfigurability
 - Change parsing and processing in the field

Programmable Networks

Advanced Computer Networks

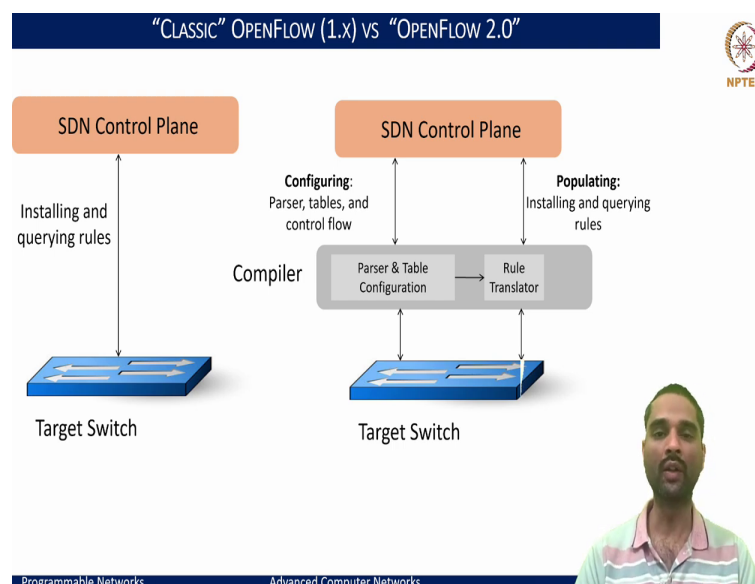


And for that, three of the aspects were considered quintessential. One is protocol independence, that is, when we try to configure a parser, it has to be basically defined on a set of typed match and action patterns or headers, and there is no notion of a fixed set of headers that you would want to match, every header that you would want to build is something that a switch should be able to learn and be programmed to understand the kind of headers that you would want.

Second, we want basically the target independence wherein it does not matter what kind of the actual hardware constraints or configurations are, but when we want to program, we want to make sure that they are expressed in a target-independent fashion just as we express in a high-level language which is agnostic to the underlying hardware architecture or ISA but a compiler would take care of translating the high-level language into the hardware specific language like ISA specific instructions to build the program. We would want similar aspects when we want to deal with the networks so that we rely on a compiler to configure the target-specific aspects but a program would be written without the knowledge of switch details. So we need the abstractions given out as a part of the programming language to build such a facility.

And third is reconfigurability, that is, once this system is up and running, we want to have the ability to change the parsing capabilities and processing in the field on the run. And these three key aspects were considered in developing what we now see as a P4 language.

(Refer Slide Time: 6:23)



And just to put an analogy of how it really differs from the classical SDN with OpenFlow 1.0 what it allowed is the SDN control plane to install and query for the rules on the target switch and like I said before this is basically on a closed control plane and closed ASIC specific APIs that would run and the agent that would run within a switch would then translate and set up based on what APIs there are six specific APIs to make sure that the queries and the rules are set correspondingly.

And now we are trying to go besides just installing and querying of the rules to even add the parse graphs and including the table configurations themselves. So earlier we considered tables to be given, and we would try to just fit the rules within those tables now we would want to configure and update the table aspect that is what we saw earlier like how the logical tables can be built on physical tables the same abstractions work here but then we are trying to give it out as a part of the configurations that can be built on a programming language site.

And second, we also want to build the parsers that can be configured and set up within the target switch, and again, we want to look up a system control plane to provide the right abstractions in building these configurations. So now we can think of two aspects: one is configurations that are parsers, the tables, and the control flow for each of these pipelines of tables that we want to build and the second, is the rudimentary SDN control of populating the rules on top of them. And to put together is what the P4 language tries to address.

(Refer Slide Time: 8:19)

P4 LANGUAGE

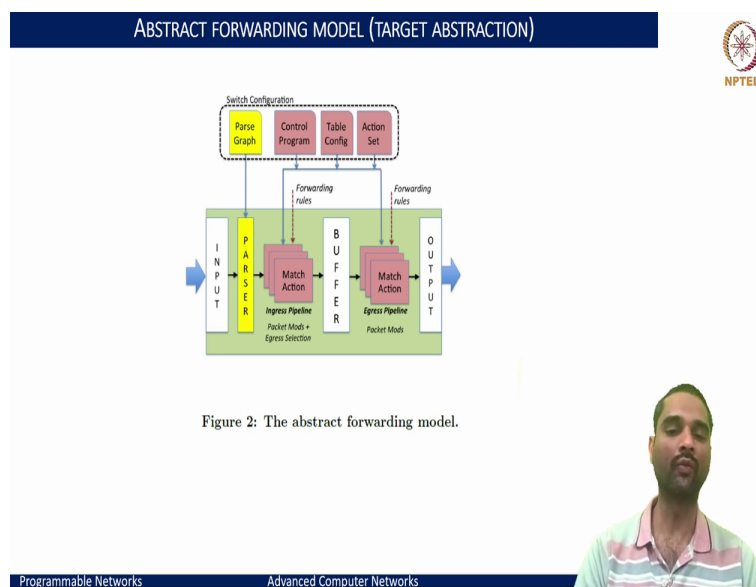


- Programming Protocol-Independent Packet Processing
- Telling programmable switches what to do



So P4 stands for programming protocol-independent packet processing, and that means you are actually writing the program to say what should be the protocols on which you want to build rather than relying on the standard protocols on which the switches operate and hence you are making this switching aspect to run on a protocol independent notion that means we could arbitrarily add any of the headers, any of the payload aspects and then make the processing to follow those headers and match and do take certain actions. And this is exactly the full power of telling programmable switches what we want to do.

(Refer Slide Time: 9:06)




And this comes in a very simple abstract forwarding model, and what we can put it up as on the switch side for the configuration we are trying to set up what we call as a parse graph as a specification that we want the switch to understand, second the control program that dictates what is the pipeline stages that for a given parse graph which are all the control or the pipeline stages that need to be processed both on the ingress as well as the egress side, third the table configurations themselves in terms of, for a given match action how would I want to set each of this configure, each of the table stages, what is the number of entries, number of tables that you want and what is the width of the table in terms of the bits that you want to match, what is the depth of the table in terms of the number of rules that you would want the table to occupy, all of these configurations could be set.


And then the last part is the action set that you would want to specify for each of the matches for each of the records within each of the tables in a different fashion, so a combination of these switch configurations the way we want them to be programmed along with the forwarding rules that we would want to apply on these action tables constitutes the P4 programming's the abstract forwarding model.

(Refer Slide Time: 10:34)

P4 CONCEPTS

- A P4 program contains definitions of the following
 - ❑ **Headers:** describes the sequence and structure of a series of fields (fields width and constraints on field values)
 - ❑ **Parsers:** A parser definition specifies how to identify headers and valid header sequence within packets
 - ❑ **Tables:** Match+action tables are the mechanism for performing packet processing. P4 program defines the fields on which a table may match and the actions it may execute.
 - ❑ **Actions:** P4 supports construction of complex actions from simpler protocol-independent primitives. Actions are available within match-action tables.
 - ❑ **Control programs:** the control program determines the order of match-action tables that are applied to a packet, describing the flow of control between match+action tables.





Programmable Networks

Advanced Computer Networks

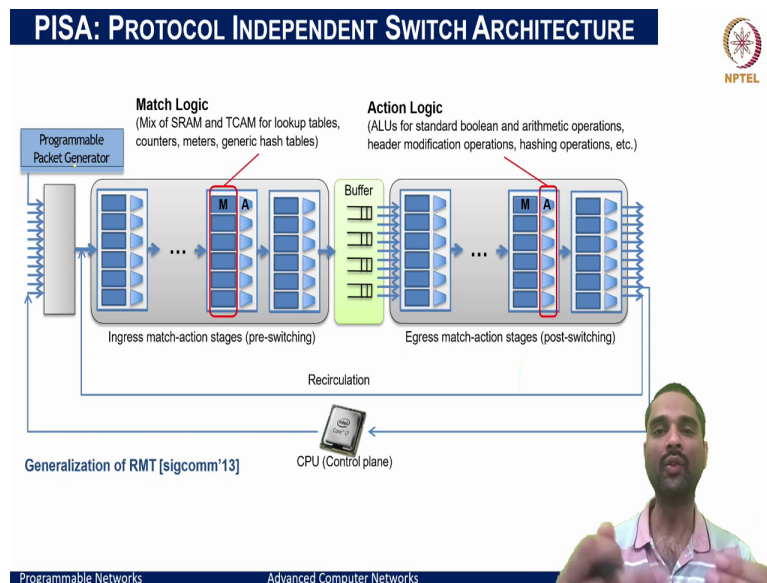
And to understand this better, we need to look at like the major aspects here in the P4. One is the headers, which describe basically the sequence and structure of the bits or the fields that we want to parse out or understand or give some notion of what aspects we want to treat distinctly for those bit fields and these field widths and can vary based on what our requirements would be and we would want to basically populate ground up saying we can define our own ethernet header model our own IPv4, IPv6, any of the custom headers and make the switch understand, these are the headers on which it would look for.

And second, is the parser that ties along with the header, header as a structure that you would build, and parser as the one that is looking for the entries in these structures and extracts out the information that you parse through the packet bits and identify which of the headers that you are matching or using a header sequence within the packets and extract this information and once we have these headers and parsers trying to extract all the match headers you would want to take specific actions on each of the matched headers and that is expressed through the use of tables

wherein you use the combination of the match action pattern in a table where the P4 program would define the fields on which a table would want to match and of the fields on which the table matches what is the kinds of what are the kinds of actions that it may want to execute.

And actions primarily constitute the construct of doing any manipulations on the packet headers, and again these constructs of complex actions can be built from very simple protocol-independent primitives and this can be specified as the entries within the match action table fit and the last part is the control program which precisely determines the order or the match action tables that are applied for a particular packet, In essence, they describe the flow of control between the match action tables and how the packet would parse its information including any of the metadata that you would want to tag as the packets get processed at different stages.

(Refer Slide Time: 13:16)



So to sum up what we just described and what we saw earlier in RMT, you would have on one side a programmable packet parser where the parser would look up and see what aspects of the packet you need to parse, and then you have the ingress match action stage pipeline which is a pre-switching entity that you are going to look up for and the match logic again as we discussed it would be a mix of SRAM and TCAM for lookup tables and we would be able to maintain the counters, meters for each of these and we could use for exact match to generate hash tables and look up the corresponding actions.

And the actions part corresponds to the typical ALU operations that you would do, like Boolean arithmetic operations header modifications, etc. And once you process through a set of stages you can buffer the packets and then process them on the egress pipeline likewise which is post-switching operations that you would want to do and if we want to circulate back and do the reprocessing we could do that at the end of the egress pipeline to start recomputing and redoing like the processing of ingress and egress matchings on the batch of packets. And this provides a very simple abstraction as what we saw earlier presenting us the means to build the switches that are programmable.

(Refer Slide Time: 15:02)

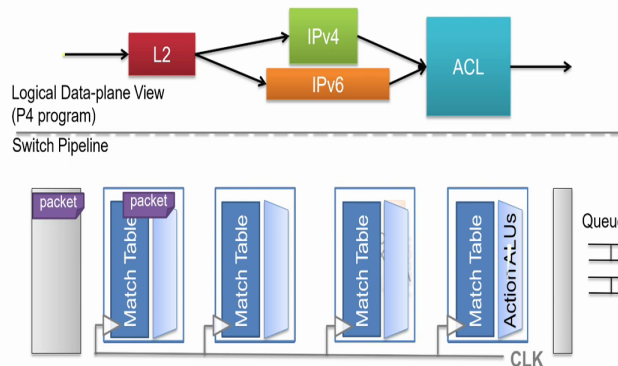


**WHY WE CALL IT PROTOCOL-INDEPENDENT
PACKET PROCESSING?**



Programmable Networks Advanced Computer Networks

DEVICE DOES NOT UNDERSTAND ANY PROTOCOLS UNTIL IT GETS PROGRAMMED



Programmable Networks

Advanced Computer Networks

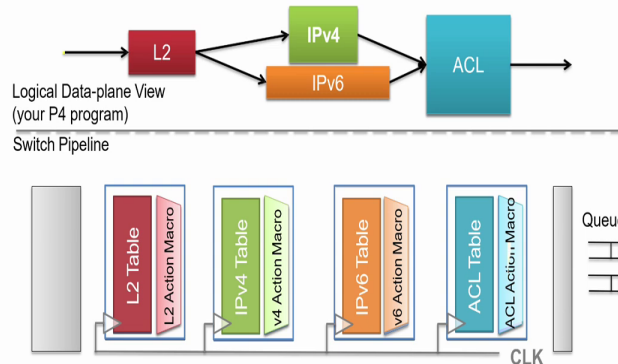


So the question comes why we would have to call it protocol-independent packet processing and in essence, what we have tried to build is that a device that is at the bottom here, the switching pipeline, has no notion of what L2 is what is L3 or L4 it is up to the programmer to define what L2 would be what L3 or L4 would be and thus because the devices now do not understand any protocols until they are really programmed that is why they become protocol independent and this defining the set of protocol now vests with the programmer to say how you want the match to take place.

So, at the hardware level within the match action pipeline, a packet is just raw bits and that is how it is going to be treated, and each of the tables dictates what that header is, not necessarily that you are matching on L2 L3 L4 of standard headers.

(Refer Slide Time: 16:07)

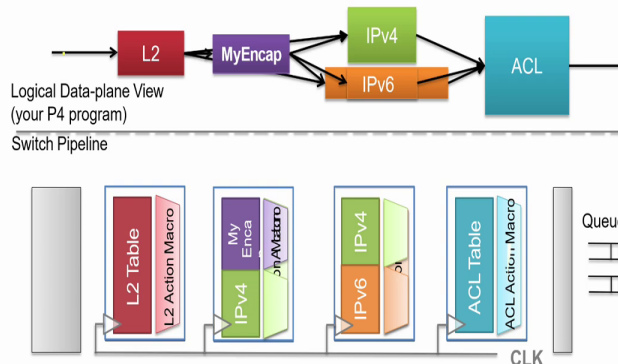
MAPPING LOGICAL DATA-PLANE DESIGN TO PHYSICAL RESOURCES



Programmable Networks

Advanced Computer Networks

RE-PROGRAM IN THE FIELD

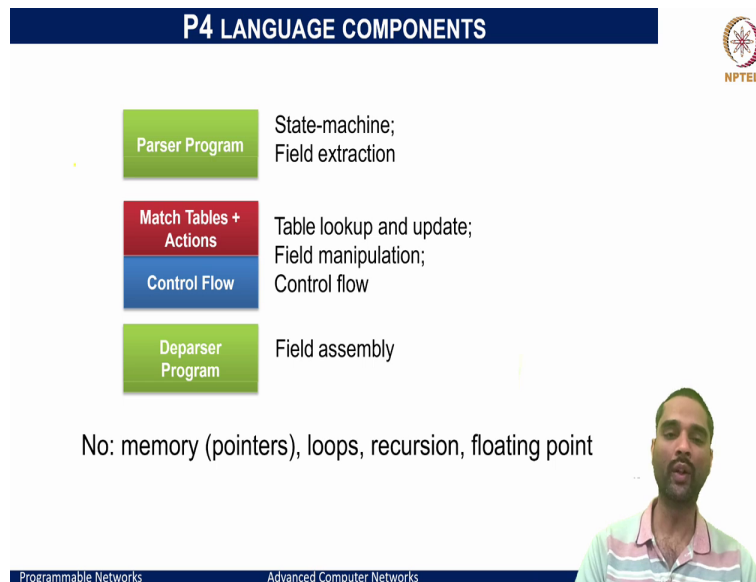


Programmable Networks

Advanced Computer Networks

And this gives us the flexibility to incorporate exactly the standard match actions that we saw earlier in terms of defining the L2 table in IPv4, IPv6, ACL tables and do the processing and also gives us the flexibility to shift any of the tables around based on our requirements so we may have a L2 table and then I may have a custom my encapsulation model that is being defined so I can push that matching into a subset of the table such as stage two and match those entries and then push the IPv4, IPv6 processing on another table and do the processing and each of these again could be done on each clock you may be able to process this switch stages and match on different tables.

(Refer Slide Time: 17:00)

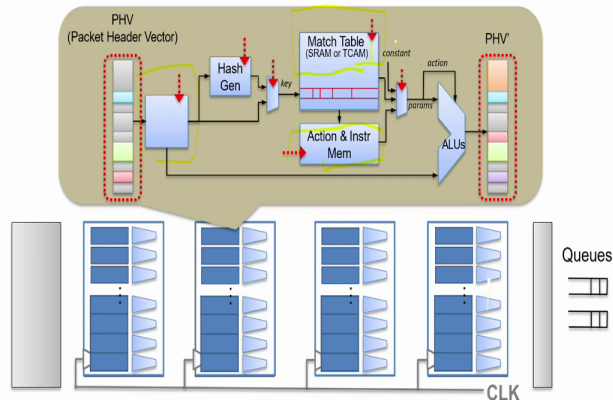


And what in essence this constitutes is to provide a P4 where we have a parser program that maintains the state machine where you are able to extract the field data, and you have the match action tables where you do the table lookup, update the fields manipulate on the contents that you would want to change in the headers, and you would define a control flow which dictates what match action tables and what is the sequence of match action tables that you would want to take and what is the metadata that you would want to carry forward as you go through the sequence of these match action tables.

And you also, in the end, have a deparser program which basically assembles back the packet header information with the payload after having taken all the actions and then processes the packets, but like we said earlier, the P4 language components but the language comes with certain restrictions in terms of avoiding the loops or recursions providing the floating points or even avoid memory or pointers because you would want not want to have any crashes in the hardware or any misinterpretation of content at a given location so it is much more easier to avoid the pointers and work directly on the data.

(Refer Slide Time: 18:29)

WHAT EXACTLY DOES A COMPILER DO?



Programmable Networks

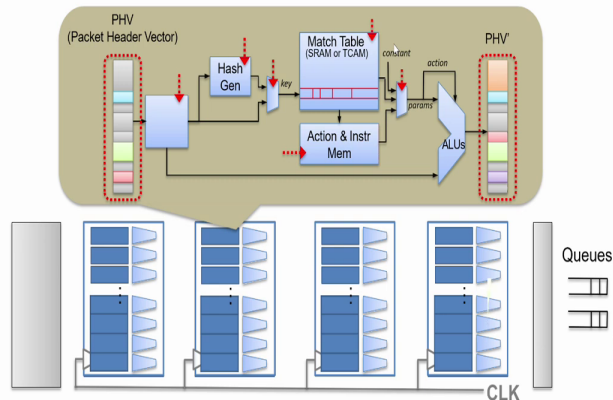
Advanced Computer Networks

And what exactly does a P4 compiler do is ensure that you extract out the packet header vector and generate the aspects of how to build this packet header vector, so it has to give you the constructs to build these packet header vectors and it has to give the constructs of how to define the hash generator what kind of a hash generation that you would want, what kind of header fields that we would want to match and build on the information and it will give us the constructs to say what kind of actions we need to take and what are the match entries that we want to look up so as a P4 language it has to give us the constructs to build these match tables and to build the instructions that we would want to execute on the action part on whenever there is a match.

And the kind of actions that we would want to take and any metadata that we would want to tag for packet processing and carry out the ALU operations in each of these stages so what we are looking at for each of these stages we may be able to P4 programming construct should allow us to populate these aspects.

(Refer Slide Time: 19:55)

WHAT EXACTLY DOES A COMPILER DO?



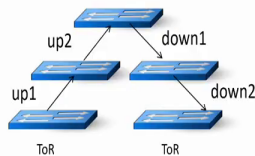
Programmable Networks

Advanced Computer Networks

P4 LANGUAGE BY EXAMPLE



- Data-center routing
 - Top-of-rack switches
 - Two tiers of core switches
 - Source routing by ToR
- Hierarchical tag (mTag)
 - Pushed by the ToR
 - Four one-byte fields
 - Two hops up, two down



Programmable Networks

Advanced Computer Networks

Let us try to look at an example scenario in terms of how we can actually write a P4 language to do a specific action, to specify specific parsing mechanism setup reactions, and see how we can really achieve this custom means to program a switch, so take for example a very simple data center routing where we have a very simple topology, so we will look into this in detail about data centers later but consider that we have a device that is connected to this top of the rack switch here and it would want to send a packet down here to this other end of the top of the rack switch.

And we may see that in a typical scenario of data centers, we will have multiple paths on which the packets can be chosen, but for a specific reason: if we want to ensure that we want the packets to carry a certain dedicated path like it has to go through this top of the rack and then go through up1 link to reach this device and then to the up2 link to reach this particular device and then down1 and down2. Our goal now is to specify such an intent where we can set up this particular kind of path and make sure that the switches are able to do this appropriately as the process at each of these stages.

So we want to write a P4 program on each of these devices which would honor and say if the packets to this ToR came for a particular destination, we would want it to take up1, and then if it came to this device, it would take the up2 link, and likewise down1 and down2 links and this can be achieved easily by saying we embed the path information within the packet itself.

So now we are trying to build a custom packet header that dictates what is the route that a packet needs to take in a given topology and that we call as a source routing which can be then honored by all of these racks so now you can see that this is not a standard header we are trying to add a different kind of header and this can be done in different ways and a very simple example that we can take is a 1-byte information for each identifier of what is the link that we want to take at each stage and let us call that as a hierarchical tag or a mTag that we want to be considered within the packet and based on this tag we want to take a decision which links that we would want to send.

So it is up, up down, down can be encoded as 4 bytes and be padded down to the packet header, and as each stage takes a packet, it would take out the first and then look at the next header and then forward it accordingly.

(Refer Slide Time: 22:56)

HEADER FORMATS



- Header

- Ordered list of fields
- A field has a name and width

```
header ethernet {  
  fields {  
    dst_addr : 48;  
    src_addr : 48;  
    ethertype : 16;  
  }  
}
```

```
header vlan {  
  fields {  
    pcp : 3;  
    cfi : 1;  
    vid : 12;  
    ethertype : 16;  
  }  
}
```

```
header mTag {  
  fields {  
    up1 : 8;  
    up2 : 8;  
    down1 : 8;  
    down2 : 8;  
    ethertype : 16;  
  }  
}
```

⏪ ⏩ ⏴ ⏵ 🔍

Programmable Networks

Advanced Computer Networks



So if we have to build such an aspect. One, we have to first look at the header format, which is basically the order list of fields or a field name that has the name and the width, and again we start with layer 2, which will be destination address, source address, ether type of 48, 48 and 16 bits and then if we can do it with the VLAN tags like what the traditional routing says we can set up the VLAN, say which VLAN, we could also think of VLANS with what is the PCP if it is 3 bits or CFI bit of 1 and VLAN ID of 12 bits and ether type considering of 16 bits overall to say we could embed the hidden VLAN fields in this order 4 bytes of a VLAN header.

And in our case, we would want the header mTag which would basically look up what are the upstream links that it has to take and what are the downstream links that it has to take, each being encoded as eight bits each, and again we would have the ether type that is being added later to say what is the following information that is coming after this header tag and this is how we have basically defined our header mTag that we would want in the construct of P4 language which basically says that we have 4 fields up1 8 bits width, and likewise up2 8 bit, down1 and down2 as 8 bit each and ether type as 16 bit entry.

(Refer Slide Time: 24:45)

- State machine traversing the packet
 - Extracting field values as it goes

```

parser start {
    ethernet;
}

parser ethernet {
    switch(ethertype) {
        case 0x8100 : vlan;
        case 0x9100 : vlan;
        case 0x800 : ipv4;
        . . .
    }
}

parser vlan {
    switch(ethertype) {
        case 0xaaaa : mTag;
        case 0x800 : ipv4;
        . . .
    }
}

parser mTag {
    switch(ethertype) {
        case 0x800 : ipv4;
        . . .
    }
}

```



And next, we would want to write a parser for such a header thing where we want to extract the field values as the packet progresses. So parser start for the ethernet, you would extract on the ethernet and say what to do, and the first thing as a packet comes, it would enter in the parser start and take ethernet as the next entry to parse, and it would enter the ethernet parse and look at the ether type, and if it is 8100 then it is looking for VLAN if it is 9100 then it would look for the other kind of a VLAN and then 800 ipv4 and so on.

And once it is in VLAN so if our packet, if you remember, we had ethernet followed by a VLAN and then our mTag so it would come with the value 8100 and enter into the parsing of the VLAN, and here in the parse VLAN we would switch on the ether type and if we coded our ether type for the mTag as aaaa then it would look up this ether type in the VLAN ether type field and match on the mTag and come to this mTag approach.

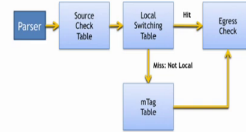
And once it parses the mTag again, we would see what is the next header field, we are going to the IPv4 that is 0x8000, and then it would go on for this field, and this is how we can write a parser which is now able to understand three of the headers, ethernet, VLAN and mTag and likewise ipv4 and so on, so we have built two of the constructs here now the headers as well as the parser.

(Refer Slide Time: 26:38)

MATCH-ACTION TABLES



- Describe each packet-processing stage
 - What fields are matched, and in what way
 - What action functions are performed
 - (Optionally) a hint about max number of rules



```

table mTag_table {
  reads {
    ethernet.dst_addr : exact;
    vlan.vid : exact;
  }
  actions {
    add_mTag;
  }
  max_size : 20000;
}
    
```

```

table local_switching {
  // read destination and checks if local
  // if miss occurs, goto mtag table
}
    
```

```

table local_switching {
  //verify egress is resolved
  // do not retag packets received with tag
  // read egress and whether packet was mtag
}
    
```



Next as the packets get parsed, they would be checked in the table in terms of what fields they are going to match and what actions they are going to take, and we may have to also worry in cases about how many max numbers of rules that we may put but let us consider for now that we are able to handle all the rules, what really would happen is, you parse the packet you check the source table, you look up the local switching table what is the hit then you want to do the egress check if there is a miss, that is, there is no information in the local you take the mTag table and then do the egress correspondingly what that means is either I am taking a decision based on local hit to say what egress port to take or if I do not have a hit on the local switching table I rely on the mTag table to dictate egress path to take.


And for this, we can say that if we write a table mTag table, this is the table that we want to populate, which says when I get to the mTag table and I match on the destination address ethernet, we read this and VLAN ID exact so if both are matched then the action that I want to take, is add mTag, and this max size of this table would be 20000 entries so the depth of this table mTag is defined as 20000 entries and all the matches that we want to do on ethernet and VLAN are exact matches.

And action here that we are trying to define is to add the mTag as an action, and on the table local switching, it would read the destination and check if local if miss then it will go to mTag table. So the decision of what a local table hit is to take default action, and the miss is to go to the mTag table where we would look up this information and do further processing and the table

local switching would basically verify egress is resolved and not take the packets that are received with a tag that is if there is a packet that is with the tag then it be handled by the mTag.


(Refer Slide Time: 29:13)

ACTION FUNCTIONS



- Custom actions built from primitives
 - Add, remove, copy, set, increment, checksum

```
action add_mTag(up1, up2, down1, down2, outport) {  
    add_header(mTag);  
  
    copy_field(mTag.ethertype, vlan.ethertype);  
    set_field(vlan.ethertype, 0xaaaa);  
  
    set_field(mTag.up1, up1);  
    set_field(mTag.up2, up2);  
    set_field(mTag.down1, down1);  
    set_field(mTag.down2, down2);  
  
    set_field(metadata.outport, outport);  
}
```



Programmable Networks

Advanced Computer Networks

And the custom actions that we would want to build like add, remove, copy, set, increment, checksum, and in this case, what we saw as an add_mTag, it takes four different fields, the up1, up2, down1, down2, and then it would add the header mTag, copy the fields of mTag ether type to the VLAN ether type and set the field VLAN ether type to aaaa, so that for the first time we are now basically populating all of these entries into the packet.

(Refer Slide Time: 29:52)

MATCH-ACTION TABLES



- Describe each packet-processing stage
 - What fields are matched, and in what way
 - What action functions are performed
 - (Optionally) a hint about max number of rules



```

table mTag_table {
  reads {
    ethernet.dst_addr : exact;
    vlan.vid : exact;
  }
  actions {
    add_mTag;
  }
  max_size : 20000;
}
    
```

```

table local_switching {
  // read destination and checks if local
  // if miss occurs, goto mtag table
}
    
```

```

table local_switching {
  //verify egress is resolved
  // do not retag packets received with tag
  // read egress and whether packet was mtagged
}
    
```

Programmable Networks

Advanced Computer Networks

ACTION FUNCTIONS



- Custom actions built from primitives
 - Add, remove, copy, set, increment, checksum

```

action add_mTag(up1, up2, down1, down2, outport) {
  add_header(mTag);

  copy_field(mTag.ethertype, vlan.ethertype);
  set_field(vlan.ethertype, 0xaaaa);

  set_field(mTag.up1, up1);
  set_field(mTag.up2, up2);
  set_field(mTag.down1, down1);
  set_field(mTag.down2, down2);

  set_field(metadata.outport, outport);
}
    
```

Programmable Networks

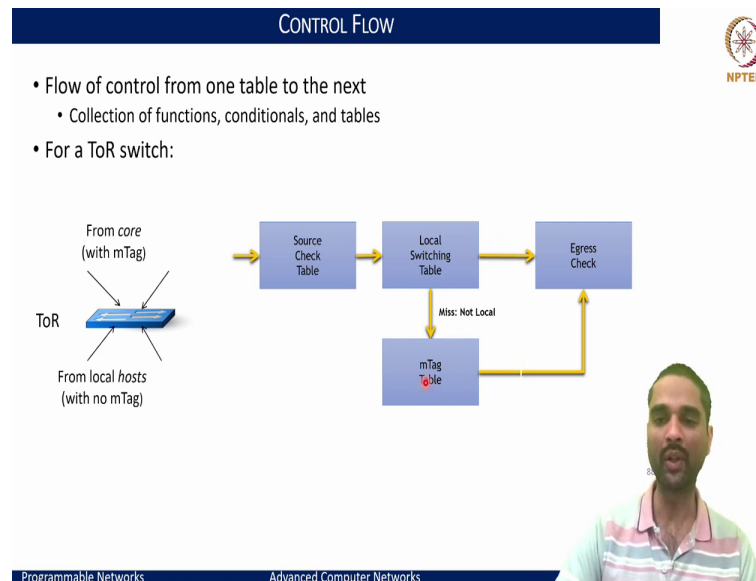
Advanced Computer Networks

In essence, now if you see whenever we saw this table mTag, and we saw this there is ethernet and VLAN exact match, we are calling this add_mTag action. So this add_mTag action now is basically setting up exactly the links that we would want to traverse for a given packet, so we are adding the header mTag with the field type as whatever the VLAN ether type is being which earlier would be pointing to IPv4 is now made to be pointing in the mTag ether type while the VLAN ether type is now made to point to the mTag itself.

So we have basically interspersed the mTag header between the VLAN and ipv4, and we have set the mTag width up1, up2, down1, and down2 that we would have received as a part of this

information that we set, and once these are all set, the set field as an action so that meta of the output port is set to the output port that we would want to take on the device.

(Refer Slide Time: 31:05)



And now the last part is exactly the control flow in terms of how this everything fits together, so from the top of the rack switch, we get with the connections where, like, if there are localhosts sending the packets, they do not have the mTag so you would add the mTag and send it up, but if the packets are coming towards the host then they would be coming with the mTag and then when there is a mTag you take it out and then send it to the destination correspondingly so these are the two different combinations that we are looking at.

So first, we will do the source check table and then take the local switching table, and then on the miss, do the mTag table and then process accordingly.

(Refer Slide Time: 31:57)

CONTROL FLOW

- Flow of control from one table to the next
 - Collection of functions, conditionals, and tables
- Simple imperative representation


```

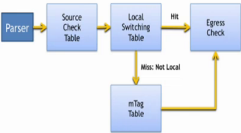
control main() {
  table(source_check);


  if (!defined(metadata.ingress_error)) {
    table(local_switching);

    if (!defined(metadata.outport)) {
      table(mTag_table);
    }
  }

  table(egress_check);
}
            
```







Programmable Networks
Advanced Computer Networks

So the control flow like is simple representation, this would mean is you first do the source check in the table, and then if there is no defined metadata, then basically we can see that that is an ingress error because we are not able to do, then we would want to see like the local switching table to take it further and in the local switching if there is a miss, then we know that there is no defined output port and if there is a hit there would be an output port that is going to be configured, and it would take the egress.

So if there is no defined metadata with the output port that is being set, then we would want to take the table mTag table, and once everything is done, it would then go to the table egress check, so to reiterate, we first started with the table source check, and if everything is fine that is no ingress error then we would go with table switching, local switching table to do the processing and again on this we are assuming if the local switching happens to be true then it would have set the output port and if the output port is not set then we want to do this mTag table and either way this mTag table should now set the output port, and then the last thing we would want to do is egress check.

(Refer Slide Time: 33:45)

- Parser
 - Programmable parser: translate to state machine
 - Fixed parser: verify the description is consistent
- Control program
 - Target-independent: table graph of dependencies
 - Target-dependent: mapping to switch resources
- Rule translation
 - Verify that rules agree with the (logical) table types
 - Translate the rules to the physical tables



And once we do this compilation of the program set it up, our intent of what we want to achieve is being done, so what this P4 compiler really now allows is to say that we are able to build the parser that is programmable, which can translate to a state machine and this we are able to do on top of a fixed parser which earlier was for fixed switches where we can verify that the description is consistent in those hardware.

And the control program that we have written is basically a target independent because now we have defined as a table graph of dependencies, but the compiler's job is then to match translate these table graphs of dependencies into the way that the target would accept, that is, map to the switching resources for each of the tables and set them up so that is where the target independence and target dependence aspects come in.

And the last part is the rule translation, which sets up like what are the rules that we want to verify agree with logical tables and translate those rules into the physical tables and entries correspondingly into these and like I said, a physical table can be mapped to multiple logical tables or multiple physical tables can be mapped to a single logical table.

(Refer Slide Time: 35:17)

- Compile into a state machine

Current State	Lookup Value	Next State
vlan	0xaaaa	mTag
vlan	0x800	ipv4
vlan	*	stop
mTag	0x800	ipv4
mTag	*	stop

Table 2: Parser state table entries for the *mTag* example.



And these aspects now, like when we compile all of that and set up the VLAN if our current state, if we consider how the packet parsing and processing would happen if our current state is VLAN, we look up for the value aaaa and the ether type and if that is true then we would take the next stage as mTag and if we have a current state VLAN and lookup value is 0x800 then the next stage is ipv4, and likewise when we look up VLAN, and there is next, the value is not defined then it is a parse error we would stop.

And if we are in the mTag and if we see that there is ipv4 as the next header, then we would go star 0x800 as a lookup value we would go ahead and do the ipv4 process, and if there is a mTag and the lookup value happens to be anything other than 0x8000 then again we have get a error, and we stop, and this is how our packet parser once we compile these are the set of possible states that we can basically the set of achieved the expected values that we can match and try to do the processing.

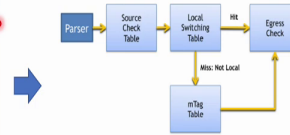
(Refer Slide Time: 36:38)

```
control main() {
    table(source_check);

    if (!defined(metadata.ingress_error)) {
        table(local_switching);

        if (!defined(metadata.outport)) {
            table(mTag_table);
        }

        table(egress_check);
    }
}
```



- And then to a target specific back-end

And now, to put everything into perspective, what we have achieved is to map whatever the control program that we defined to map it and set it up on the back-end based on whatever the physical tables requirements that have been set up by the switching device and this is where the compiler would basically translate all of these into the intents that we mentioned into a means that it would work on a given target or a given packet.

(Refer Slide Time: 37:23)

CONCLUSION

- OpenFlow 1.x
 - Vendor-agnostic API
 - But, only for fixed-function switches
- An alternate future
 - Protocol independence
 - Target independence
 - Reconfigurability in the field
- P4 language: a straw-man proposal
 - To trigger discussion and debate
 - Much, much more work to do!

So to summarize, what we have looked at is the way the P4 programs can be built where we start with vendor-agnostic APIs in the OpenFlow for how to dictate what actions to be done in the

OpenFlow 1.0 and built on top of that aspect of building how the intents can be specified to the hardware device in a protocol independent and target independent manner and be able to reconfigure that again using the SDN control plane.

And what we have achieved is a P4 language here with a proposal that is basically decoupling how the data plane programming can be done from the data plane's actual hardware implementation, and this has led to a lot of enhancements and improvements over the last few years, and there is a lot more active work that is going on in this field, and there is lot more to explore further in this space.