Parallelizing P4 Match Action Tables

Ryan Massicci
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
rmn9035@cs.rit.edu

Abstract — This paper focuses on the background of the P4 programming language, and how to invoke match-action tables in parallel. We describe how to identify match action table dependencies and how these dependencies effect the level of parallelism of the P4 program. The runtimes of three separate algorithms are analyzed and we discuss how the dependencies in the match action tables effect the parallelism of each algorithm. Finally we will present recommendations as to how P4 match action tables should be structured to optimize a P4 program to run in parallel.

I. INTRODUCTION

Traditional networking systems use hardware called "switches" to manipulate packets traveling through the data plane. Each switch has its own proprietary firmware that dictates how the switch handles the packets. The firmware dictates which networking protocols the switch supports. While switches are very fast, there are some downsarts to relying solely on the firmware built into a switch. If the switches completely defines the control flow of the system, the network administrator is unable to modify the system without replacing unwanted switches new switches that have the capabilities that the administrator wishes to add. This can become a time consuming and costly requirement if a system changes frequently or requires a large set of modifications.

An effective solution to this problem is to implement Software Defined Networking (SDN). SDN allows an administrator to define the data plane of a network system. By adding software control of the data plane, a network administrator can easily reconfigure the network to his specific needs without purchasing expensive new hardware switches. The system can adapt to new technologies by adding support for new protocols, and remove support for unused protocols. Adding new protocols via software is simple way to make the system more robust, while removing unnecessary protocols makes the system safer and easier to debug. One negative aspect of SDN is the speed degradation that results from moving from hardware defined control to software defined control. Using parallelism is an effective way to speed up the runtime of packet transfer operations in software.

In this paper, we will be investigating the possible speedup of executing match action pipelines for various P4 programs. Three P4 programs have been selected for analysis: IPv4 forwarding, Multi-hot route inspection, and ARP/ICMP messaging. IPv4 forwarding is a simple program that forwards a single packet from one host to another. The P4 program will define the data plane rules for how the packet is forwarded. The Multi-hop route inspection program is an extension of the IPv4 program, but adds an additional feature. Each time the packet is forwarded through a switch, a switch identifier is added to the packet header. Once the packet reaches its final destination, the packet header will contain an ordered list of all switches it has reached. The third program is ARP/ICMP messaging. This program implements a ping utility. A message from the originating host is send to another host on the same network. The message will be sent the the destination host, and the switch will send an acknowledgement back to the originating host.

Each program will be run in two separate environments. The first environment contains a singe core processor, forcing the program, and its match action table pipeline to be run sequentially. This will prevent the program from receiving any runtime benefits due to parallelizing the match action tables. The second environment has a duel core processor. This will allow the match action table pipeline to be parallelized. The speedup due to parallel execution will be calculated and analyzed for each program. We will also investigate any table dependencies that are found in the implementation of the match action tables. Finally we will provide recommendations for developers to keep in mind when designing match action tables that will help to optimize parallel execution of a P4 program.

II. THE P4 PROGRAMMING LANGUAGE

A. Language Background

The P4 programming language is intended for programming data plane applications for packet forwarding in a software defined network. The developers of P4 had three key goals in mind when designing the language: Target independence, protocol independence, and reconfigurability [1]. A P4 program is able to be compiled for a large number of platforms. The same P4 program is able to compiled to run on an FPGA, CPU, SOC, etc. This allows a P4 programmer the freedom to write a program for an intended use case, without the knowledge of what hardware the program will run on. P4 is protocol independent, meaning that there is no built in support for any networking protocol. Instead, the programmer must define the header and field formats for the protocol their program is implementing. The incoming packet is then parsed with respect the the header and field formats defined in the program. Reconfigurability enables an engineer to adjust
A fundamental concept in the P4 language is the pipeline. The P4 pipeline can be broken down into four key sections: parser, metadata bus, match-action tables, and deparser. The parser converts the serialized bits of a packet, and converts them into separate fields, referred to as the parsed representation. The metadata bus runs along the full pipeline and contains the parsed representation metadata and the match action table's intermediate results, and feeds that data to the match action tables and deparser. A match action table is a mapping of header field values to a given set of actions. The actions typically manipulate the packet metadata in some way. A pipeline can contain any number of match action tables, depending on the task at hand. Finally, after the match action tables have completed all required actions, the deparser takes the parsed representation and serializes it to be sent to the network switch.

### III. PARALLELIZING MATCH ACTION TABLES

Since the bulk of the execution in the P4 pipeline takes place in the match action table section, this is the section we will focus our efforts on parallelizing. In order to get the most efficient speedup, match action tables should be run in parallel. In theory, on a system with the number of cores equal to the number of tables in the P4 pipeline, executing all tables asynchronously should complete in the time it takes to run the longest running table. If the tables were run sequentially, the runtime would be the sum of all table runtimes. There are however some factors that dictate the degree to which a match action pipeline can be parallelized. If two or more tables match on or modify the same metadata field, it is considered a table dependency. Table dependencies restrict the degree to which tables can be run in parallel.

#### A. Match Dependency

Consider the case where a P4 program contains two match action tables. If one table modifies a field that is read by table two, we consider this a match dependency. The example code below illustrates an example of a match dependency.

```plaintext
1: action set_bd(bd) {  
2:     modify_field(metadata.bd, bd);  
3: }  
4:  
5: table port_bd() {  
6:     reads { metadata.ingress_port : exact; }  
7:     action { set_bd; }  
8: }  
9:  
10: table dmac {  
11:     reads { metadata.bd : exact }  
12:     action { set_egress_port; }  
13: }  
14:  
15: control ingress {  
16:     apply(port_bd);  
17:     apply (dmac);  
18: }
```

The control flow block "ingress" calls apply on each match action table. Tables "port_bd" and "dmac" are both applied. Ideally both tables would be run in parallel, but the tables must be examined to see if that is possible. Table port_bd reads the value of metadata.ingress_port, and then calls set_bd. The action set_bd, in turn, modifies the field metadata.bd. The next table reads the value of metadata.bd and then calls set_egress_port. Since port_bd modifies the field metadata.bd, the program must wait for the action to complete before the dmac table can read the metadata.bd field. This prevents the tables from being run in parallel. Whenever two tables contain a match dependency, the tables must be run sequentially [2].

#### B. Action Dependency

When two match action tables modify the same value during their action execution, it is considered an action dependency. In the code example below, the control block "ingress" applies the tables dmac and smac_filter. The read execution of each table is independent since each table reads a different value, and neither table modifies the read fields. Both tables do, however, modify the same field. Each table executes the "drop" action, which modifies the metadata.drop field. Two tables the modify the same field are considered to have an action dependency. Any two tables that have an action dependency can run in staggered execution [2]. Staggered execution allows the table's read executions to run in parallel, but the action executions must be run sequentially.

```plaintext
1: action drop() {  
2:     modify_field(metadata.drop, 1);  
3: }  
4:  
5: table dmac() {  
6:     reads { ethernet.dstAdr : exact; }  
7:     action { drop; }  
8: }  
9:  
10: table smac_filter {  
11:     reads { ethernet.srcAdr : exact }  
12:     action { drop; }  
13: }  
14:  
15: control ingress {  
16:     apply(dmac);  
17:     apply (smac_filter);  
18: }
```

#### C. Independent Tables

If two tables contain no match nor action dependencies, the tables are considered independent. Independent match action tables are able to execute completely in parallel [2]. When writing a P4 program to be used in a multi core environment, if possible, all match action tables should be independent. This

A fundamental concept in the P4 language is the pipeline. The P4 pipeline can be broken down into four key sections: parser, metadata bus, match-action tables, and deparser. The parser converts the serialized bits of a packet, and converts them into separate fields, referred to as the parsed representation. The metadata bus runs along the full pipeline and contains the parsed representation metadata and the match action table's intermediate results, and feeds that data to the match action tables and deparser. A match action table is a mapping of header field values to a given set of actions. The actions typically manipulate the packet metadata in some way. A pipeline can contain any number of match action tables, depending on the task at hand. Finally, after the match action tables have completed all required actions, the deparser takes the parsed representation and serializes it to be sent to the network switch.

III. PARALLELIZING MATCH ACTION TABLES

Since the bulk of the execution in the P4 pipeline takes place in the match action table section, this is the section we will focus our efforts on parallelizing. In order to get the most efficient speedup, match action tables should be run in parallel. In theory, on a system with the number of cores equal to the number of tables in the P4 pipeline, executing all tables asynchronously should complete in the time it takes to run the longest running table. If the tables were run sequentially, the runtime would be the sum of all table runtimes. There are however some factors that dictate the degree to which a match action pipeline can be parallelized. If two or more tables match on or modify the same metadata field, it is considered a table dependency. Table dependencies restrict the degree to which tables can be run in parallel.

A. Match Dependency

Consider the case where a P4 program contains two match action tables. If one table modifies a field that is read by table two, we consider this a match dependency. The example code below illustrates an example of a match dependency.

```plaintext
1: action set_bd(bd) {  
2:     modify_field(metadata.bd, bd);  
3: }  
4:  
5: table port_bd() {  
6:     reads { metadata.ingress_port : exact; }  
7:     action { set_bd; }  
8: }  
9:  
10: table dmac {  
11:     reads { metadata.bd : exact }  
12:     action { set_egress_port; }  
13: }  
14:  
15: control ingress {  
16:     apply(port_bd);  
17:     apply (dmac);  
18: }
```

The control flow block "ingress" calls apply on each match action table. Tables "port_bd" and "dmac" are both applied. Ideally both tables would be run in parallel, but the tables must be examined to see if that is possible. Table port_bd reads the value of metadata.ingress_port, and then calls set_bd. The action set_bd, in turn, modifies the field metadata.bd. The next table reads the value of metadata.bd and then calls set_egress_port. Since port_bd modifies the field metadata.bd, the program must wait for the action to complete before the dmac table can read the metadata.bd field. This prevents the tables from being run in parallel. Whenever two tables contain a match dependency, the tables must be run sequentially [2].

B. Action Dependency

When two match action tables modify the same value during their action execution, it is considered an action dependency. In the code example below, the control block "ingress" applies the tables dmac and smac_filter. The read execution of each table is independent since each table reads a different value, and neither table modifies the read fields. Both tables do, however, modify the same field. Each table executes the "drop" action, which modifies the metadata.drop field. Two tables the modify the same field are considered to have an action dependency. Any two tables that have an action dependency can run in staggered execution [2]. Staggered execution allows the table's read executions to run in parallel, but the action executions must be run sequentially.

```plaintext
1: action drop() {  
2:     modify_field(metadata.drop, 1);  
3: }  
4:  
5: table dmac() {  
6:     reads { ethernet.dstAdr : exact; }  
7:     action { drop; }  
8: }  
9:  
10: table smac_filter {  
11:     reads { ethernet.srcAdr : exact }  
12:     action { drop; }  
13: }  
14:  
15: control ingress {  
16:     apply(dmac);  
17:     apply (smac_filter);  
18: }
```

C. Independent Tables

If two tables contain no match nor action dependencies, the tables are considered independent. Independent match action tables are able to execute completely in parallel [2]. When writing a P4 program to be used in a multi core environment, if possible, all match action tables should be independent.
will allow the match action pipeline to be completed in the shortest time possible.

IV. TABLE DEPENDENCY GRAPHS

The P4 compiler has built in support for determining the dependency types between match action tables. The compiler examines the match action tables in the P4 program and determines whether the tables contain match dependencies, action dependencies, or are independent. Once this analysis has been completed, the compiler constructs a tables dependency graph. The table dependency graph allows the compiler to understand which tables are able to be run in parallel, and which tables contain match or action dependencies, limiting them to sequential or staggered execution [1].

A. Table Dependency Graph Example

The following code represents a sample P4 packet forwarding program[5]. We will investigate the table dependencies and construct a tables dependency graph to help visualize the possible parallelism of the match action pipeline.

```p4
1: table table1 {
2:  reads { ethernet.dstAddr : ternary; }
3:  action {
4:    malformed_outer_ethernet_packet;
5:    set_valid_outer_unicast_packet_untagged;
6:    set_valid_outer_broadcast_packet_untagged;
7:  }
8: }
9: 
10: table table2 {
11:  reads {
12:    l2_metadata.lkp_pkt_type : ternary;
13:    ethernet.srcAddr : ternary;
14:  }
15:  action { action2; }
16: }
17: 
18: table table3 {
19:  reads {
20:    l2_metadata.lkp_mac_type : ternary;
21:    ethernet.etherType : ternary;
22:  }
23:  action { action3; }
24: }
25: 
26: table table4 {
27:  reads {
28:    ingress_metadata.drop_reason : ternary;
29:    example_metadata.fldD : ternary;
30:  }
31:  action { action4; }
32: }
33: 
34: table table5 {
35:  reads {
36:    example_metadata.fldB : ternary;
37:    example_metadata.fldC : ternary;
38:    example_metadata.fldD : ternary;
39:    example_metadata.fldE : ternary;
40:  }
41:  action { nop; }
42: }
```

A table dependency graph can be generated for the above program by analyzing the common fields that each table reads and modifies. As illustrated in the code, table1 performs the actions malformed_outer_ethernet_packet, set_valid_outer_unicast_packet_untagged, and set_valid_outerroadcast_packet_untagged. These actions modify the fields ingress_metadata.drop_reason, l2_metadata.lkp_pkt_type, and l2_metadata.lkp_mac_type. These three fields are read values for table2, table3, and table4. As discussed in the match dependency section of this paper, if a table modifies the same field that is read by another table, the tables contain a match dependency. Since table2, table3, and table4 all have a match dependency with table1, they cannot be run until table1 has completed all actions. table2, table3, and table4 perform the actions action2, action3, and action4, respectively. Each of these actions modify different fields, so these three tables are independent from each other. The actions, table2, table3, and table4, perform in parallel. table2, table3, and table4, do however modify the fields example_metadata.fldB, example_metadata.fldC, and example_metadata.fldD, respectively. Each of these modified fields is a read value in table5. These are three more examples of match dependencies. This prevents table5 from running until tables2, table3, and table4, complete their actions. Below is the visual representation of the table dependency graph for this P4 program.
To investigate how parallelization can affect different algorithms, three separate P4 programs [4] were written and run in a single core environment and a dual core environment. The table dependency graphs of each program will be analyzed and we will discuss the positive and negative aspects of the match action table design with respect to parallelization. The runtime of each algorithm run sequentially and in parallel will then be examined.

A. IPv4 Forwarding

The first example is a simple IPv4 forward algorithm. This algorithm performs three actions. The first action is to update the source MAC address and the destination MAC address. Second, the time-to-live specified in the packet header must be decremented by one. Lastly, the program must forward the packet out of the correct network port.

```
1: table ipv4_lpm {
2:     reads { hdr.ipv4.dstAddr : lpm ; }
3:     action {
4:         ipv4_forward;
5:         drop;
6:         noAction;
7:     }
8: }
9:
10: action ipv4_forward(dstAddr, port) {
11:     standard_metadata.egress_spec = port;
12:     hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
13:     hdr.ethernet.dstAddr = dstAddr;
14:     hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
15: }
```

The implementation of this algorithm was quite simple. It contains one match action table and one user defined action that performs four modifications to metadata fields. The `ipv4_lpm` table reads the longest prefix match of the destination address in the ipv4 header. Depending on the control plane rules, the possible actions run would be `ipv4_forward`, drop, or `noAction`. The `ipv4_forward` action sets the output port of the egress metadata. The ethernet source address is set to the destination address found in the ethernet header, and the ethernet destination address is set to the destination address found in the packet header. Finally the ipv4 packet header’s time to live field is decremented by 1.

Sine this algorithm only contains one match action table, we don’t have an opportunity to parallelize the match action pipeline. Therefore the table dependency graph for this algorithm shows only the `ipv4_lpm` table being run. Unfortunately this algorithm provides no opportunity for parallelization, but on the bright side, it contains no dependencies.

B. Multi-Hop Route Inspection

The next algorithm under examination is Multi-Hop Route Inspection (MRI). This algorithm is similar to IPv4 forwarding, but with the added functionality of tracking the path the packet takes through the network. On each hop throughout the network, the program appends an identifier to the packet header. Once the packet reaches its destination, the packet contains an ordered list of identifiers.

```
1: table ipv4_lpm {
2:     reads { hdr.ipv4.dstAddr : lpm ; }
3:     action { ipv4_forward; noAction; }
4: }
5:
6: table swid {
7:     action { add_swid; noAction; }
8: }
9:
10: action ipv4_forward(dstAddr, port) {
11:     standard_metadata.egress_spec = port;
12:     hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
13:     hdr.ethernet.dstAddr = dstAddr;
14:     hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
15: }
16:
17: action add_swid(id) {
18:     hdr.mri.count = hdr.mri.count + 1;
19:     hdr.swids.push_front(1);
20:     hdr.swids[0].swid = id;
21: }
22:
23: action add_mri_option() {
24:     hdr.ipv4_options = IPV4_OPTION_MRI
25:     hdr.mri.setValid();
26:     hdr.mri.count = hdr.mri.count + 1;
27: }
```

The MRI algorithm is similar to the IPv4 algorithm in that they both share the `ipv4_lpm` table. This table performs the same actions as the `ipv4_lpm` table in the ipv4 forwarding algorithm; updating the output port, source and destination addresses, and decrementing the TTL. MRI expands on the IPv4 algorithm by adding an additional table and two additional actions. The new table `swid` is calls the `add_swid` action. `add_swid` increments the MRI count by one, adds the swid identifier to the front of the list of identifiers. By running...
this action at each hop through the network, once the packet reaches the final destination, it will contain an ordered list of swids. The final additional action add_mri_option sets the MRI option in the ipv4 header, sets the MRI status to valid, and increments the MRI count.

The MRI code contains two tables. The swid table modifies said fields in the packet header, while the ipv4_lpm table modifies ethernet and ipv4 files in the header. After comparing the fields being read and modified, we can determine that the tables are completely independent. This presents an opportunity to run the match action tables in parallel. The table dependency graph for MRI shows that the swid and ipv4_lpm tables can run in parallel.

C. ARP/ICMP Messaging

The final program discussed is ARP/ICMP Messaging. Address Resolution Protocol (ARP) and Internet Control Message Protocol (ICMP) are two networking protocols. ARP is typically used to map an IP address to a physical address of a device on the local network [7]. ICMP is typically used by network devices to report network error messages, such as unreachable servers [6].

This program is another extension of IPv4 forwarding, with the addition of adding the ability for switches to respond to any ARC/ICMP messages. The program ultimately allows a host machine to ping any other host that is connected to the switch, and the switch will respond to the ping request.

```
1: table ipv4_lpm {
2:     reads { meta.dst_ipv4 : lpm ; } 
3:     action { set_dst_info; noAction; } 
4: } 
5: }
6: table forward {
7:     reads { 
8:         hdr.arp.isValid() : exact; 
9:         hdr.arp.oper : ternary; 
10:         hdr.arp_ipv4.isValid() : exact; 
11:         hdr.ipv4.isValid() : exact; 
12:         hdr.icmp.isValid() : exact; 
13: } 
14:     action { 
15:         forward_ipv4; 
16:         send_arp_reply; 
17:         send_icmp_reply; 
18: } 
19: }
```

The full program code is too large to include in this paper, but can be found on the P4 github account found in [4]. The ipv4_lpm table reads the IPv4 destination address and its action sets the source and destination MAC addresses, as well as the output port. The forward table reads the validity state for several ARP, ICMP, and IPv4 fields. The table triggers the actions forward_ipv4, send_arp_reply, and send_icmp_reply. forward_ipv4 sets the ethernet source and destination addresses, sets the the output port, and decrements the packet TTL. send_arp_reply performs the same actions as forward_ipv4, but also adds several ARP settings to the header so that the packet will conform to the ARP protocol. send_icmp_reply performs the same actions as forward_ipv4 but adds several ICMP options to the packet header so that the packet will conform to the ICMP protocol.

Upon examination of the read and modified fields, its it determined that the ipv4_lpm table and the forward table are able to be run in parallel. The forward table performs many more field modification than ipv4_lpm, so the match action pipeline will be dependent on the speed of execution of that table. The table dependency graph shows how the ipv4_lpm and forward tables may be run in parallel.

VI. TIMING ANALYSIS

The runtime of each program was analyzed on two environments. In the first environment, each program was run on a system with a single core. This forced each program, including its match action pipeline, to run sequentially. This gave a baseline runtime for each program. The second environment enabled a second core on the host machine. This enabled the match action pipeline to run in parallel. The speedup for each algorithm’s runtime was calculated and compared. The runtimes were determined by running the P4 programs with the python scripts found in the P4 GitHub project [4]. Several Mininet instances were invoked on a host machine and a simulated network was run. Each program forwarded 10,000 packets between the virtual hosts with the data plane rules determined by the P4 program. The transfer speed was determined by taking the average speed of the 10,000 packets reported by Wireshark.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Synchronous</th>
<th>Parallel</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPV4 Forward</td>
<td>0.54 ms</td>
<td>0.53 ms</td>
<td>2%</td>
</tr>
<tr>
<td>MRI</td>
<td>0.63 ms</td>
<td>0.55 ms</td>
<td>12%</td>
</tr>
<tr>
<td>ARP/ICMP</td>
<td>0.76 ms</td>
<td>0.67 ms</td>
<td>12%</td>
</tr>
</tbody>
</table>
A. IPv4 Forward Analysis

The synchronous runtime for the IPv4 forwarding program was calculated to be .54 ms per packet transfer. This was calculated by averaging the Wireshark reported packet transfer speed of 10,000 packets. When the same program was run in the dual core environment, the runtime was barely reduced. The lack of speedup is expected because it was previously determined that the IPv4 match action pipeline is not able to be parallelized. The .01 ms speedup is likely negligible, and can be attributed to other system factors, rather than parallelizing the match action pipeline.

B. Multi-Hop Route Inspection Analysis

The synchronous runtime for the MRI program was calculated in the same fashion as IPv4 forwarding; taking the average runtime of 10,000 packet transfers. The synchronous runtime of the MRI program was clocked in at .63ms per transfer. This was slightly longer than the synchronous runtime of the IPv4 forward program because of the addition of the add_swid table execution. When executing the same program with two cores enabled, the runtime drops down the .55 ms per transfer. This was a 12% speedup for packet transfers over sequential execution. Notice that the parallel execution of the MRI program is nearly the same speed as that of the IP4V forward program. This reinforces the statement that parallel execution will allow the match action pipeline to complete in the time it takes the longest running table to run. The add_swid table was able to run in parallel with the ipv4_lpm table. From the analysis of the IPv4 forward program, we know that the ipv4_lpm completes in .53ms. Running the MRI program in a dual core environment allows the program to execute the full add_swid table while the ipv4_lpm table is executing. Comparing these two examples is a clear indicator that the using a multi core switch with an optimized P4 program will allow you to add more functionality to your routing algorithm without taking a performance hit.

C. ARP/ICMP Massaging Analysis

The ARP/ICMP Massaging runtime was also determined by taking an average transfer speed of 10,000 packets in a simulated network. The sequential packet transfer runtime was determined to be .76 ms per packet. This is .13ms longer than the MRI program. The longer runtime can be attributed to the additional actions that set the ARP and ICMP options in the packet headers. Running the ARP/ICMP messaging program in parallel dropped the runtime down to .67ms per packet. This as a 12% speedup from sequential runtime. Notice that the speedup is equal to that of the MRI program. This can be attributed to the fact that both programs were able to run their two match action tables in parallel.

VII. Conclusion

After analyzing the sequential and parallel execution of three separate P4 programs we can determine that designing a P4 program’s match action table pipeline properly can result in a significant speedup when executed in parallel. However, not all programs are candidates for parallel execution. The IPv4 program only has a single table, so it was not possible to parallelize the table execution. A program with only a single table will have the same execution time whether it is run sequentially or in parallel.

Analyzing a program’s table dependency graph will also shed light on the expected speedup of a parallelized program. If a match action pipeline contains many match dependencies, we know the program may see very little, and possibly no benefit in a multi core environment. The speedup will be slightly better if the table dependencies are action dependencies. The match action tables with action dependencies can run in staggered execution. Staggered execution is better than sequential execution, as the match execution of each table may be run in parallel, but will not see the runtime benefits that completely independent tables would see.

VIII. Recommendations

The major recommendations that developers should keep in mind when designing a match action pipeline are:

- Avoid table dependencies
- Prefer action dependencies over match dependencies
- Prefer multiple small independent tables over single table

When writing a P4 program, it is essential to design P4 tables with parallelization in mind. When possible the programmer should avoid any table dependencies. It may be possible to move specific actions out of one table and into another in order to avoid table dependencies. If a table dependency is unavoidable, the programmer should prefer an action dependency over a match dependency. An action dependency at least allows the match execution of tables to be run in parallel. This will result in some runtime speedup while a match dependency prevents any parallel execution between tables. The programmer should also be mindful of including too many actions in a single table. If a single table can be broken into two separate tables with no dependencies, the two tables could be run in parallel and the execution time of the match action pipeline could potentially be cut in half.

IX. Future Work

Future work on parallelizing P4 programs should be focused greater diversity of the programs being analyzed and introducing two key enhancements.

Future investigation of parallelizing match action tables should include a more divers array of programs. The programs should have a match dependencies, action dependencies, a mixture of both, and independent tables. Analyzing programs with more examples of the dependencies discussed would provide a better representation of how difference table dependencies affect the performance benefit of parallelizing the match action pipeline.

The parsing and deparsing of the packet header is required for every single packet transfer. Parallelizing these steps could potentially help to speed up the execution of the full P4 pipeline. Since the parser simply parses the packet header bitfield and assigns the values to individual fields. This presents an great opportunity to split up the packet header into equal chunks, and parse each chunk of the packet in parallel.

The other opportunity for a future enhancement is improving the P4 compiler’s optimization for parallel execution. The compiler can currently identify match and action dependencies. It would be a great benefit if the compiler could inform the P4 programmer of the dependencies so that they could design the match action tables in a better way. An even better enhancement would be if the compiler could
automatically redesign the match action table layout to remove the dependencies, or split overloaded tables up into smaller independent tables. These enhancements would optimize the match action tables for parallel execution, without the programmer having to go through the difficult task of finding and fixing table dependencies.

X. REFERENCES

5. https://github.com/jafingerhut/p4lang-tests