$P4_{16}$: Out of the Loop

Felix Sandmair, Henning Stubbe, Eric Hauser*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: felix.sandmair@tum.de, stubbe@net.in.tum.de, hauser@net.in.tum.de

***Abstract*—$P4_{16}$ is a programming language used for packet processing that was formulated without providing loops, such as for or while loops. This Paper is going to give a quick summary of $P4_{16}$ in order to show how it is still possible to implement loops in $P4_{16}$ and afterward compare the solutions in regard to performance.**

1. Introduction

$P4_{16}$ is a programming language that differs from many programming languages because it was designed without a concept for loops. The reasoning behind that is that if we eliminate iterations, the time needed for computing a P4 program is linearly dependent on the packet size. Having no loops can be limiting at times, e. g. for processing the payload of the packet, you might need some form of looping [1]. Another example would be TCP SYN proxies. TCP SYN proxies are a measure to protect a server against SYN flooding attacks [2]. To implement this functionality on software-defined networks looping would be very helpful. Furthurmore, this investigation shows the limits of the programming language $P4_{16}$. The following sections will discuss how it is possible to write loops in $P4_{16}$ even without concepts like while and goto. After explaining the approaches, it will focus on analyzing the performance and comparing our solutions.

2. $P4_{16}$ Introduction

$P4_{16}$ is a language that is used to program the data plane of software-defined networks. The programs written in this language are deployed on programmable networking hardware like routers, switches, network interface cards, or network appliances. Those devices are called targets. The manufacturer of those targets needs to provide the hardware or software implementation framework, an architecture definition, and a P4 compiler. $P4_{16}$ can only specify the data plane functionality of a device. After compilation, it generates an API for the control plane to interact with the data plane. The architecture definitions describe how the architecture is put together. Each $P4_{16}$ program can be divided into multiple Blocks, usually always consisting of a Parser, a Deparser, and a variable amount of Control Blocks. In this paper, the V1 Architecture will be the architecture that is used in the $P4_{16}$ programs, which is an architecture by the P4 Language Consortium. This architecture was designed in a way that is comparable to the old $P4_{14}$ architecture. Figure 1 shows how the architecture of the V1 Model is structured.
The Parser is always the first block a packet passes through. The Parser resembles a finite state machine that parses the packet headers and extracts the header data into a header struct. This data could be an ethernet header, but also personalized headers not following any popular protocol. Since it is possible to specify any header structure, it is technically possible to parse more than just the headers, e. g. a packet's payload [1].

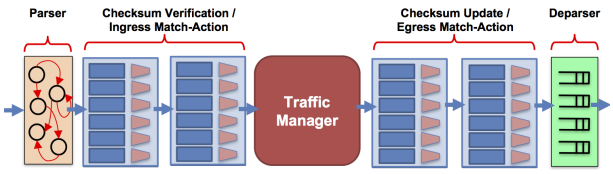

Figure 1: V1 Model Architecture [3]

After the parser, there are different programmable control blocks. In the case of the V1 Architecture, there are the MyVerifyChecksum, MyIngress, MyEgress, and MyComputeChecksum control blocks. Every target may also have extra functionality it can provide the user of the target. This functionality can be provided to the programmer with a so called extern function. These extern fuctions are methods that are unique to different targets. The Architecture can also limit the programmer to use certain extern functions only in certain control blocks [3]. In the end, the packet gets reassembled by the deparser. The deparser takes all the header data that was manipulated within the control blocks and maybe added new ones and puts them back together into a network packet.

3. Loop Concepts

In this section, the different approaches to implement loops in $P4_{16}$ will be discussed.

3.1. Parser Loops

Parser Loops are probably the most straightforward possibility to program loops in $P4_{16}$. At the beginning of the parser, there will be the opportunity to declare variables and constants. Also, the objects needed inside of the loop, can be instanciated here. After that, starts the definition of the finite state machine. The finite state machine always consists of a initial state, called start, where the parsing of every packet will begin and two

doi: 10.2313/NET-2023-06-1_05

different final states, accept and reject. Usually, each state will be used to extract the headers of the received packets. The headers extracted will be saved in a C struct like header struct. After the extraction of the header the transition will be invoked based on the information extracted from the header. E.g. if the header extracted would be an ethernet header and the EtherType of the header would indicate that the packet is an IPv4 packet it goes to the state for extracting IPv4 header. This decision will be done by using a select statement that matches an expression list to specific values. After the select statement matches the expression list, it will transition to the given state. Before the transition statement, an arbitrary amount of parser statements can be written. The syntax of Parser statements allows basic arithmetic operations on variables and constants and methods calls on objects instantiated before. Depending on the target, the parser statements might allow more or less functionality inside the parser statements. E.g. Not all targets will allow the use of if statements inside the parser. In case if statements are needed inside the loop it is possible to use the select statement to achieve the same functionality a if statement would provide. This will come at the cost of one state per if statement used inside the loop. A significant benefit of this approach to building loops in $P4_{16}$ is that it can be done on nearly every target, but there are exceptions. Some targets will not compile parsers that contain loops. The compiler will check if the parser can be unrolled into a graph without circles at compile time. If the parser can not be unrolled, it will further check if the loop advances the cursor of the packet in every iteration. If this is the case, the parser can be compiled since the packet size is finite and if the cicle always advances the cursor the loop can not loop infinitely [4].

```
bit<32> i = 0;

    state start {
        transition l1;
    }

    state l1 {
      //loop-body
      i = i + 1; //condition-update
      transition select(i<=CYCLES/*while-
    condition*/) {
          true:l1;
          default:parse_ethernet;
      }
    }
```

Source Code 1: do-while-loop in parser

```
bit<32> i = 0;

    state start {
        transition l1;
    }

    state l1 {
      transition select(i<=CYCLES/*while-
    condition*/) {
          true:l2;
          default:parse_ethernet;
      }
    }

    state l2 {
      //loop-body
      i = i + 1; //condition-update
      transition l1;
    }
```

Source Code 2: while-loop in parser

In Source Code 1 and Source Code 2 , two different implementations of parser loops can be seen. The code fragments show an implementation of a do-while-loop and the regular while-loop. For both solutions, the declaration of all variables needed and instanciation of all objects takes place. In this case, the variable i, which is used to cycle through the loop a constant amount of times, is the only variable needed. After that, the implementations differ. For the do-while-loop in Source Code 1, the body of the loop can be executed immediately before checking any condition. Afterwards, the breaking condition for our loop is written inside the select statement. If the condition holds, the select statement will evaluate to true and transition back to the same state. In order to get the more commonly used while-loop the usage of an additionale state is obligatory. This time instead of executing the loop-body inside the first state, the condition is beeing checks first. If the condition is met, the select statement transition to the second state of the loop l2, to execute the loop-body. After the loop execution is finished, l2 transitions back to state l1. The loop body of this example only consists of one line where the variable i gets incremented to iterate a fixed amount of times through the loop. Back in state l1 the condition check happens again. When the point that the condition evaluates to false is reached, it is possible to into another loop or in this case start with parsing of the headers of the packet.
Two different solutions were beeing implemented in hope of performance benefits if only one state is beeing used instead of two different states because of less transitioning between states. Another reason is to show that there are more then just one possibility to introduce loops into the parser.

3.2. Recirculate Loops

Another possibility to realize loops is by using extern functionality. In this case, the extern method recirculate_preserving_field_list() provided by the V1 architecture was beeing used. This extern simply takes the packet that is currently processed deparses it and introduces it back into the packet processing beginning with the parser. This method can only be used in the MyEgress control block of a P4 Program. When using this method, it receives a number as a parameter. This number represents a list of variables from the metadata that should be saved for the next pass through the packet processing. This means every variable that is needed for the loop, like a loop index, is stored inside the metadata and annotated using the @field_list annotation. The number used to annotate the varibales is given to the recirculate_preserving_field_list() extern as a parameter.

```
struct metadata {
    @field_list(RECIRC_FL)
    bit<32> i;
}
```

Source Code 3: metadata definition

doi: 10.2313/NET-2023-06-1_05

In this example, the variable i is stored inside the metadata as shown in Source Code 3. The the variable is annotated with the constant RECIRC_FL (constant with the value 0).

```
apply {
      if(standard_metadata.instance_type != 4)
     {
          meta.index = 0;
      }
        if (!(meta.index < CICLES) && hdr.ipv4
    .isValid()) { //condition check
          //if break condition is met the
    tables for routing need to be applied
           ipv4_lpm.apply();
       }
    }
```

Source Code 4: Ingress control

In the Ingress control, the first thing that is beeing done is to check if the packed arrives here for the first time or if it is a packet that was already recirculated. This is done by checking the field instance_type of the standard_metadata. If the field does not contain the value four [5], it is not a recirculated packet, and the metadata fields need to be initialized. In this example, the variable i gets set to 0 in order to keep track of the number of iterations the packet has already done. If the packet that arrived in the Ingress control is recirculated, the condition will be checked to make sure if the packet needs continue looping or if the loop should be stopped and the packet should be routed.

```
apply {
      if(meta.index < CICLES) { //condition
    check
          //loop-body
          meta.index = meta.index + 1; //
    condition-update
          recirculate_preserving_field_list(
    RECIRC_FL);
      }
    }
```

Source Code 5: Egress control

When entering the egress control, the break condition of the loop will be checked. If the condition holds, the execution of the loop body will start. At the end of the execution, the recirculate_preserving_field_list() method is used. The reason why the recirculate_preserving_field_list() method is used inside the if statement is because otherwise, the routing done by the ipv4_lpm.apply() in Source Code 4 would be overwritten, and the packet would continue looping. Like mentioned before, the recirculate extern takes an integer value in order to save the annotated variables inside the metadata. Here the constant RECIRC_FL is given to the method to save the value of the variable i. After the Egress control, the packet will be reassembled by the deparser and sent back to the parser.
One problem with this approach of building a loop in $P4_{16}$ is that it will create a large overhead for each iteration since it has to pass through the whole package processing every iteration. Also, this solution is unique to the V1 architecture, or other targets that implement a recirculate extern function. [6]

3.3. Custom Loop Header

This solution is similar to the loops introduced in Section 3.2. Instead of relying on an extern method for recirculating, a physical connection between one of the output ports to one of our input ports of the device is beeing established, in order to recirculate the packets. Also a personalized header struct is introduced where all variables used inside the loop will be stored. This means variables needed for the calculations and the variables needed for the break condition check will be saved inside this header. Once a packet arrives, all headers of the packet will be parsed, including the custom loop header. After the parsing is done the program would need to check in the Ingress control if the packet headers include the custom loop header or not. If the packet does not arrive with the custom header that header will be generated and added to the packet headers. If the packet headers already include the custom header it means the packet is looping and the break condition needs to be checked. If the condition is met and the loop needs to be continued the loop body is executed. After the execution of the loop body the variable defining the output port needs to be updated to the port connected to with one of the input ports. If the loop needs to be stopped the custom header needs to be removed again and the packet will be routed. This approach comes with the same problem of the recirculate loops. Since the packet needs to go through the whole packet processing for each iteration, the loop overhead is quite extensive. An advantage of about this approach is that it can be run on every P4 programmable device since the device has no idea it is sending a packet back to itself.

4. Loop Performance Analysis

The test setup that was beeing used for the performance analysis is based on a mininet simulation for $P4_{16}$. The Mininet setup is running on a Virtual machine using Virtual Box. It was set up with the repository [7] and vagrant. In the Virtual Box settings, the VM had four cores and 4096 MB of RAM assigned. The Computer the tests were run on is a laptop with an Intel Core i7-9750H (2.6Ghz) and 8GB of 2667MHz RAM. The mininet topology used has two hosts, h1 and h2, connected with a switch s1 in between. The P4 Program is run on the switch s1. The solution shown in Section 3.3 will not be coverd in the performance analysis. The simulation of the physical connection from one of the outgoing port to one of the ingoing ports was attempted but not succesful. Further studies could try to either verify the solution presented in the section 3.3 outside of a simulation on a real device or find another solution inside the simulation. Because of this, the solution from section 3.3 is only a concept but is not verified if it would acually work.

Figure 2: Mininet topology [8]

doi: 10.2313/NET-2023-06-1_05

To get the measurements shown in Figure 3 below, 50 identical packets were sent from host h1 to host h2 and computed an arithmetic mean of the measured times. The time it took to execute the loop inside switch s1 was calculated by capturing the pcaps of the input and output interface. Pcaps are files that are generated by sniffing on an interface of a network device and they store information like the time a packet left the interface or the IP adress of the packet. Since this Paper focuses on the performance of the different loop types no calculations were done inside the loops to compare just the performance of the loop concepts. As seen in Figure 3, six measurments were beeing done for each approach, starting from 5000 loop iterations to 50000 loop iterations. It was observed that the first packets that were sent had very long times. Because of that the decision was made to send ten packets before measuring the time of the 50 packets used to calculate the average. Another observation that was observed is that the times of most packets were pretty close, but a few of the packets had high variances. This could be a scheduling problem since the p4 program is run inside a mininet simulation inside a VM.

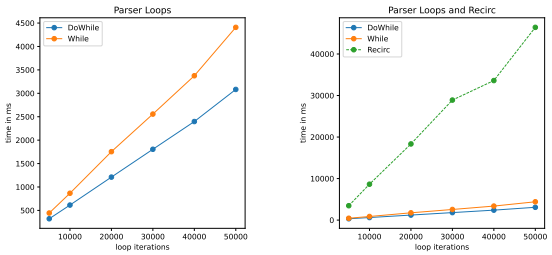

Figure 3: execution time graphs

As expected, the performance of the loops implemented in the parser is better than the one using the recirculate_preserving_field_list() method. This is expected because the recirculate loop has to do the whole package processing for each loop. The performance of the do-while-loop is close to the performance of the while-loop but the do-while-loop is still the faster of the two. This should be because the do-while-loop only uses one state instead of two. The difference between the do-while-loop and while-loop is pretty small and will get even smaller once more states are introduced to the loop for realizing if statements with the select statement. Another thing that can be seen very clearly in the plot is that the time grows linearly the more iterations are added to the loop. This is good to see because otherwise, it would not be suitable for algorithms that need a high iteration count.

5. Conclusion

We saw it is possible to write loops in $P4_{16}$, with some solutions being more costly than others. It can be said that loops used in the parser are a better solution than the recirculate loop. Unfortunately, it was impossible to compare our solutions to the approach mentioned in Section 3.3. One question that remains unanswered is whether it is a good idea to use loops in $P4_{16}$ in general. To answer this question, it would be necessary to do more studies that use the concepts introduced in this paper to implement algorithms for real-world use cases and compare them to solutions from other languages.

References

[1] M. Budiu and C. Dodd, "The p416 programming language," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 1, pp. 5–14, 2017.

[2] D. Scholz, S. Gallenmüller, H. Stubbe, and G. Carle, "Syn flood defense in programmable data planes," in *Proceedings of the 3rd P4 Workshop in Europe*, 2020, pp. 13–20.

[3] B. N. Vladimir Gurevich, "P416 Introduction," https://opennetworking.org/wp-content/uploads/2020/12/p4_d2_2017_p4_16_tutorial.pdf, 2017, [Online; accessed 25-September-2022].

[4] T. P. L. Consortium, "P416 Language Specification," https://p4.org/p4-spec/docs/P4-16-v1.2.3.pdf, 2022, [Online; accessed 25-September-2022].

[5] p4language, "intrinsic," https://github.com/p4lang/p4c/blob/main/testdata/p4_14_samples/switch_20160512/includes/intrinsic.p4#L60-L66, 2019, [Online; accessed 25-September-2022].

[6] A. Fingerhut, "v1model special ops," https://github.com/jafingerhut/p4-guide/tree/master/v1model-special-ops, 2021, [Online; accessed 25-September-2022].

[7] p4lang, "tutorials," https://github.com/p4lang/tutorials/, 2022, [Online; accessed 25-September-2022].

[8] "Working with P4 in Mininet on BMV2," https://opennetworking.org/wp-content/uploads/2020/12/p4_d2_2017_p4_16_tutorial.pdf, [Online; accessed 25-September-2022].

doi: 10.2313/NET-2023-06-1_05