Public Review for

P4RROT: Generating P4 Code for the Application Layer

Csaba Györgyi, Sándor Laki, Stefan Schmid

In this paper, the authors propose a new code generation mechanism to streamline application-level offloads expressed in the P4 programming language. Specifically, they present P4RROT, a new library that allow developers to write application layer logic in Python which is then converted in P4. The authors discuss the pain points and challenges for automatic code generation and show the applicability of P4RROT in two different contexts: a publish-subscribe sensor data processing system and a real-time data streaming engine, supporting MQTT-SN and MoldUDP traffic.
The reviewers appreciated the quality of the paper both in the style of the presentation as well as the thought provoking idea. They convened that this work does a great job in to explaining the issues and challenges in writing P4 programs alongside proposing a solution that is actually open-sourced and available for the entire community to build upon: https://github.com/Team-P4RROT/P4RROT. A special mention for the repo, that provides instructions on how to start and how to contribute to the project.

Public review written by
Gianni Antichi
Politecnico di Milano

P4RROT: Generating P4 Code for the Application Layer

Csaba Györgyi
Eötvös Loránd University, Hungary
University of Vienna, Austria
gycsaba96@inf.elte.hu

Sándor Laki
ELTE Eötvös Loránd University, Hungary
lakis@inf.elte.hu

Stefan Schmid
TU Berlin, Germany
stefan.schmid@tu-berlin.de

ABSTRACT
Throughput and latency critical applications could often benefit of performing computations close to the client. To enable this, distributed computing paradigms such as edge computing have recently emerged. However, with the advent of programmable data planes, computations cannot only be performed by servers but they can be offloaded to network switches. Languages like P4 enable to flexibly reprogram the entire packet processing pipeline. Though these devices promise high throughput and ultra-low response times, implementing application-layer tasks in the data plane programming language P4 is still challenging for an application developer who is not familiar with networking domain. In this paper, we first identify and examine obstacles and pain points one can experience when offloading server-based computations to the network. Then we present P4rrot, a code generator (in form of a library) which allows to overcome these limitations by providing a user-friendly API to describe computations to be offloaded. After discussing the design choices behind P4rrot, we introduce our proof-of-concept implementation for two P4 targets: Netronome SmartNIC and BMv2. To demonstrate the applicability of P4rrot, we investigate case studies in the context of publish-subscribe sensor data processing and real-time data streaming, supporting, in particular, MQTT-SN and MoldUDP packets.

CCS CONCEPTS
• **Networks** → **In-network processing**; **Programmable networks**; • **Software and its engineering** → **Source code generation**;

KEYWORDS
offloading, programmable data planes, code generation, P4

1 INTRODUCTION
The compute infrastructure is becoming increasingly distributed, offering multiple locations to serve requests and execute applications. This introduces interesting opportunities for spatial optimization: by bringing computation and data storage closer to the requester (or client), response times and bandwidth can often be greatly improved.

Early examples of this paradigm are the distributed domain name system and the content distributed networks created in the late 1990s. [3]. More recent examples include the edge computing paradigm as well as the trend towards in-network computing.

This paper is motivated by a novel offloading opportunity introduced by software-defined networks [10], and in particular programmable data planes [12]. Programmable networks have recently received much attention both in academia and industry, for their support of fast networking innovations: while developing new semiconductors is a time-consuming and expensive process, a programmable data plane provides an efficient and flexible way to support new protocols and new requirements. The data plane programming language P4 (Programming Protocol-Independent Packet Processors) is independent of the forwarding hardware design, and relies on a compiler specific to the hardware (e.g., SmartNICs, NetFPGAs, programmable ASICs), thus allowing the fast implementation of new protocols and networking algorithms. Over the last years, several interesting use cases for programmable data planes have been demonstrated that are related to resilience, security, resource allocation, among others [12]. Like how people started using GPUs for non-graphics-related use cases (e.g., crypto mining and deep learning), many research projects focus on how programmable data planes could be used for executing application layer tasks (e.g., emergency stops, robot control, or even key-value stores [2, 7, 11]).

P4 has proven useful for implementing application-layer tasks, but even simple functionalities can quickly result in complex software projects. This is due to the limitations of the underlying devices and the fact that P4 was not designed to support this kind of computations. For example, consider the following scenario which highlights some of the challenges that developers can face when implementing L7 logic. Suppose that in order to precalculate an aggregated value to reduce the CPU load, a programmer wants to calculate and insert an additional integer at the beginning of the UDP payload. This seemingly simple task requires to parse the usual headers, adjust the total length field in the IPv4 and UDP headers, recalculate the checksum, and implement a simple static forwarding. Even if the programmer already wrote similar and relevant code in the past (e.g., a ring buffer made of registers), it is hard to reuse this implementation since P4 does not easily support to encapsulate such a high-level abstraction. Along the way, the programmer might further need to use workarounds and different tricks to avoid resource limitations and compiler bugs. Whether calculations are peformed with a table or if-else statements may appear to be an implementation detail, is actually a design decision from a P4 point of view, requiring additional effort. If the programmer then wants to further extend the functionality, e.g., by inserting a second integer, she must modify the existing code in multiple different places. Also testing is complex, and may require a complete pipeline with whole packets (not only specific headers or fields), as there is typically no good way to unit test implementation parts.

We argue that automatic code generation can greatly simplify implementing application-layer tasks if we narrow down the scope of target features. By application-layer tasks, we mean functionalities concerned about the payload rather than other networking layers. Against this backdrop, our main ***contribution*** in this paper is P4rrot, a code generator (in form of a library) for P4, to support and speed up the offloading process. With P4rrot, the developer

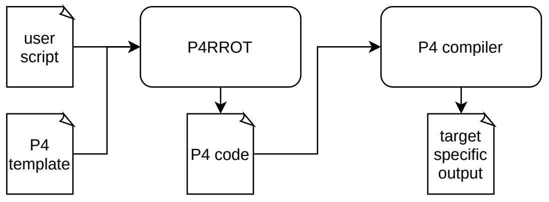


Figure 1: The high-level overview of the use of P4RROT.

can describe the application layer logic using our (Python3) library[1] and generate the equivalent P4 code. P4RROT does not require a new programming language, which also simplifies the adoption of new P4 features. In this paper, we discuss the main design aspects of P4RROT and show its early-stage Python3-based implementation for BMv2 (Behavioral Model v2) and Netronome NFP (Netronome Flow Processor)targets. We have chosen BMv2 and Netronome NFP targets as they both use the same V1Model architecture. While BMv2 can be considered as an ideal target where the generated code can be executed without any restrictions, NFP smartNIC is a more sensible target with hardware and compiler constraints to be considered. Among the possible hardware targets we selected this smartNIC since it seems better suited for server offloading than constrained ASICs. Note that we also work on adding support for Intel Tofino ASICs because of their widespread use, high throughput and ultra-low latency, having promising initial results at the time of writing this paper.

To demonstrate the improved development experience given by P4RROT, we discuss two in-network computing projects in the context of publish-subscribe sensor data processing applications and real-time data streaming, supporting, in particular, MQTT-SN and MoldUDP packets (relevant, e.g., for high-frequency trading). These case studies show that our approach indeed allows to make simplifications and assumptions, and automate tasks that the original P4 language cannot do. We also demonstrate the viability of our approach using a Netronome SmartNIC and the BMv2 software switch.

Fig. 1 illustrates the architecture of our proposed solution. The code generator takes two inputs: a P4 template describing the usual parts of a pipeline, and the application-specific logic using P4RROT's API. After running the code generator, we get a P4 code that can be further compiled to the desired target using the vendor-provided compiler.

2 PAINPOINTS

While P4 is a great language for implementing new protocols and networking features, and also sufficiently expressive to describe even application layer logic, it was not designed with offloading applications in mind. In this section, we elaborate on the previously mentioned hindering factors and provide some examples.

Boilerplate code. P4 allows the programmer to define and parse every header in a customised way. This flexibility is a cornerstone for implementing new protocols or innovatively reusing fields. Accordingly, implementations often rely on many lines of boilerplate code, e.g., defining and parsing standard and well-known headers, checking and recalculating checksums or implementing basic forwarding rules. However, when we only care about the application layer, the standard implementation of these functionalities is still necessary even if we do not wish to change them. This can be time consuming and also render the code long and hard to read.

Cross-layer dependencies. Certain header fields depend on the encapsulated content, for example the total length field in the IPv4 header, the length field in the UDP, and the checksum in the UDP header. These cross-layer dependencies might be hard to maintain while we are focusing on higher-level responsibilities. When we change the size of the processed packet by adding and removing parts of it, we have to adjust all of these fields one by one (e.g., IPv4 total length or UDP length). Moreover, we need to check the adjustments over and over again when we introduce new functionalities during the project's life-cycle, which can be cumbersome.

Lack of high level encapsulation. Technically, we can separate different functionalities in different control blocks, but there is no good way to encapsulate reusable data structures and algorithms efficiently. For example, we might want to use a simple ring buffer or a bloom filter based on registers in different projects. Ideally, this would be achieved by defining different control blocks for different methods for the same data structure. However, registers and other stateful elements are local to the enclosing control block. As a workaround, one might consider creating a (parametrised) control block and passing the requested operation and parameters in metadata. This however is fairly cumbersome and requires additional checks and memory to execute the code for the requested operation.

Implementation details become design decisions. Sometimes, the exact same functionality can be implemented with different P4 building blocks to avoid the resource limitations of the given target. For example, we can check whether two values are equal either with a simple if-statement, or we can calculate their difference and check whether it is 0 or not with a match-action table. Switching between these implementations is not easy.

Fragmented code. If a programmer wants to modify or extend the current solution, she might need to modify the code at multiple different places. Consider the following example. Initially, an extra integer was inserted before the UDP payload using a custom header with a single `bit<32>` field. Later, it was decided to insert two integers. To achieve that we extend the custom header definition with a second `bit<32>` field, then adjust the IPv4 and UDP length fields, assign the value in the control block, and finally add the extra field to the checksum calculation in the deparser.

Hard to test. Testing P4 code is often considered difficult, and we generally still lack efficient unit testing frameworks, that support early testing of just parts of the code. Even a simple functionality (adding two numbers) might require a long and complex code in order to be tested (e.g. using PTF tests [14]). Once we have this code, whole packets have to be generated and set to exercise our solution. Finally, if an error is detected, it is still hard to debug and find the faulty code section.

Tricks and workarounds. Last but not least, implementations often use technical tricks and workarounds to avoid resource limitations or overcome compiler bugs. Technical manipulations are necessary when there are slight differences between the target devices. For example, the length of the physical port numbers on the

[1]https://github.com/Team-P4RROT/P4RROT

BMv2 and a Netronome smartNIC are different, although they both use the V1Model architecture. Some compilers might also be buggy, thus requiring additional effort from the programmer. Furthermore, the programmer has to deal with resource limitations. For example, if the comparison is limited to 16-bit long values, one may use two nested if statements to check the equality of two 32-bit values. We argue that these kinds of problems may occur more often during an in-network computing project, since these applications aim to stretch the boundaries of the target device.

3 GOALS AND REQUIREMENTS FOR P4RROT

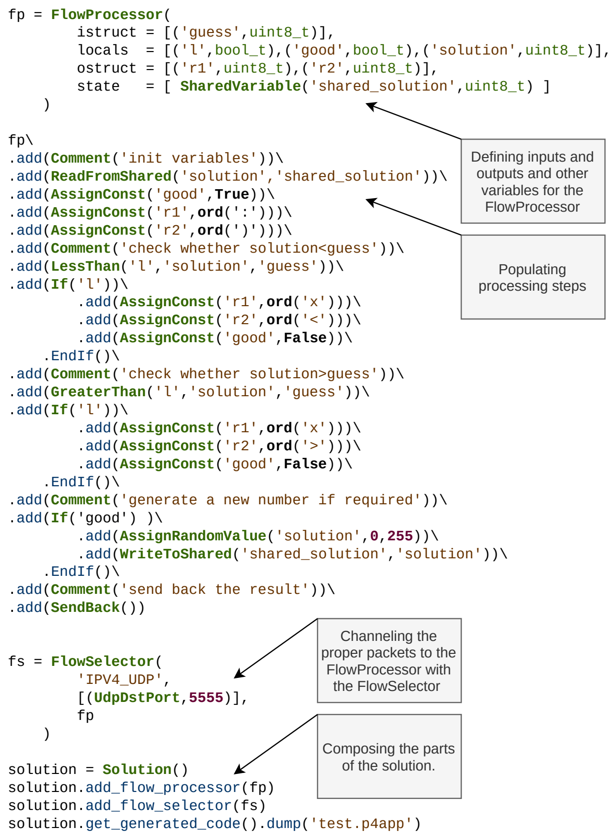
This section lists the main principles the design and implementation of P4RROT follows, divided into two categories. First, we highlight the features that overcome the previously described pain points. Second, we describe additional requirements improving the adaptability of our solution.

3.1 Painkillers

An important decision underlying P4RROT's design is that we limit our target programs to the set of offloaded application layer logic, thus making the design simpler in many ways. Having to deal only with a limited amount of commands and data types (e.g., a 3-bit long field does not make much sense in this context), we can make powerful assumptions and automate many tasks. Simplicity and structure allow the developers to keep their work compact and prevent code fragmentation. Furthermore, we require that implementation details should remain implementation details, preferably in the form of "hints". For example, checking the equality with an if-else statement or using a table can be simply an optional parameter. Our solution should further be able to encapsulate complex, often used logic, and hide data structures and algorithms (like a ring buffer made of registers) behind a straightforward API. Adding similar extensions should also be easy. Last but not least, once we fully or partially described the business logic, we want to simulate the behaviour of the application layer without even generating the P4 code. Being able to account for overflows and other low-level details is crucial.

3.2 Adaptability

The code generator should leverage an existing and well-known language in an easy-to-understand way, so programmers can work with a familiar syntax and are productive quickly. Generally, the generated P4 code should be human-readable and reasonably easy to understand. The programmer should still have the opportunity to override the code generator's decision, e.g., for further performance optimizations. The generator only performs simple semantic checks, and for instance, does not override constant values. By avoiding complex verifications, the implementation of possible extensions is simplified. We also note that target-specific checks (e.g. resource constraints) might be even impossible to implement because of legal restrictions. Also error messages should be easy to understand and point to the line where the programmer made the mistake. Since P4RROT is essentially a library that allows programmers to describe the abstract syntax tree (AST) of the offloaded solution by defining a complex object, the semantic checks are run after adding each and every node. By doing so, the generator library can raise an exception at any line as early as possible.



Defining inputs and outputs and other variables for the FlowProcessor
Populating processing steps
Channeling the proper packets to the FlowProcessor with the FlowSelector
Composing the parts of the solution.

```
fp = FlowProcessor(
        istruct = [('guess',uint8_t)],
        locals  = [('l',bool_t),('good',bool_t),('solution',uint8_t)],
        ostruct = [('r1',uint8_t),('r2',uint8_t)],
        state   = [ SharedVariable('shared_solution',uint8_t) ]
    )

fp\
.add(Comment('init variables'))\
.add(ReadFromShared('solution','shared_solution'))\
.add(AssignConst('good',True))\
.add(AssignConst('r1',ord(':')))\
.add(AssignConst('r2',ord(')')))\
.add(Comment('check whether solution<guess'))\
.add(LessThan('l','solution','guess'))\
.add(If('l'))\
        .add(AssignConst('r1',ord('x')))\
        .add(AssignConst('r2',ord('<')))\
        .add(AssignConst('good',False))\
    .EndIf()\
.add(Comment('check whether solution>guess'))\
.add(GreaterThan('l','solution','guess'))\
.add(If('l'))\
        .add(AssignConst('r1',ord('x')))\
        .add(AssignConst('r2',ord('>')))\
        .add(AssignConst('good',False))\
    .EndIf()\
.add(Comment('generate a new number if required'))\
.add(If('good') )\
        .add(AssignRandomValue('solution',0,255))\
        .add(WriteToShared('shared_solution','solution'))\
    .EndIf()\
.add(Comment('send back the result'))\
.add(SendBack())


fs = FlowSelector(
        'IPV4_UDP',
        [(UdpDstPort,5555)],
        fp
    )

solution = Solution()
solution.add_flow_processor(fp)
solution.add_flow_selector(fs)
solution.get_generated_code().dump('test.p4app')
```

Figure 2: Sample usage of P4RROT implementing a number guessing game

4 ARCHITECTURE AND DESIGN

This section describes the internal working of P4RROT's code generator through the main components of an offloading project. For a better understanding, we provide a simple example first. Then we explain the main design decisions and architectural elements behind the scenes.

4.1 A simple example

To illustrate the various features and the style of an offloading project, let us consider a simple example. Fig. 2 shows the implementation of a simple number guessing game. Although it is not particularly useful, it provides an easy-to-understand task that does not require any background knowledge. The client has to figure out a randomly generated number. After each guess, there are three possible outcomes: the solution is lower than the provided number, the solution is greater than the provided number, or the client wins (and a new number is generated).

First, we create a `FlowProcessor` in a declarative input-output style. The input is a single byte representing the client's guess, and the output is two bytes (treated as characters in the later stages).

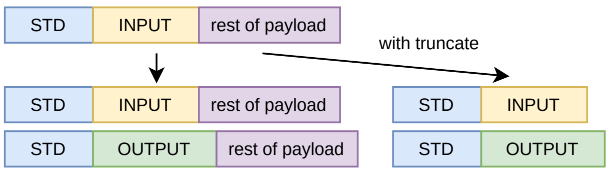


Figure 3: The different ways the user can transform a packet.

Additionally, we use some local variables and a shared variable to store the correct solution. After that, we populate the processing steps with various commands to define our application-level algorithm. We assume that the client has the correct answer, and then we change it if the guess is less or greater than the right solution in the `SharedVariable`. In the end, we send back the packet where it came from with a single `Command`. Using a `FlowSelector`, we also need to define which packets should be processed by the previously described `FlowProcessor`. In the end, we assemble the parts on a single object and generate the P4 code using the provided template.

4.2 Processors

`FlowProcessor` objects are the essence of the offloaded solution describing the application-layer calculations. During the instantiation, the programmer defines the input and the optional output structures. By default, the code generator sets the input header invalid and the output header valid, thus leaving room for modifying the packet structure. Moreover, if the `truncate` extern or similar functionality is available, the remaining payload can also be removed. If the output structure is not defined, the original input header is not invalidated. Fig. 3 depicts the different ways the user can transform a packet. The use of local variables, temporary headers and, stateful elements are also possible.

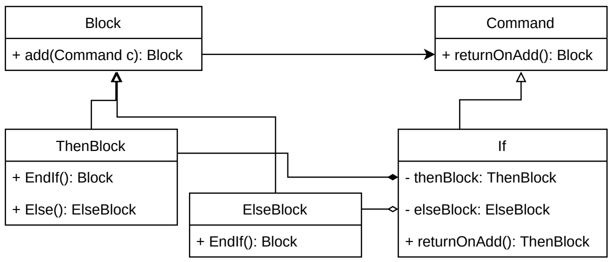
A `FlowProcessor` contains a `Block` object which encapsulates a sequence of `Commands`. `Commands` are responsible for describing different operations performed on the packets. They can vary from low-level casting to high-level data structure manipulations. Besides the necessary input and output variables, they might provide different implementation hints using optional variables.

When a new `Command` is added to a `Block`, it returns itself or another `Block`. Thanks to this design, it is possible to describe algorithms reasonably intuitively, similar to the JavaScript Promises or the LINQ library in C#. Comfortably defining if-else statements is also possible. Once we add an `If` command, we return a `ThenBlock` (which inherits from the `Block` class). The `ThenBlock` instance can return an `ElseBlock` using its `Else` method. Finally, the `ElseBlock`'s `EndIf` returns the original parent `Block`. Fig. 4 depicts the class diagram of the previously mentioned objects with a simple example. We define `Switch` statements and `Atomic` blocks similarly.

Leveraging this design, the code generator library can provide simple semantic checks upon adding `Commands` and simulate the behavior of a `FlowProcessor`.

4.3 Selectors

To define what kind of packets are processed in the offloaded solution, the user script can create `FlowSelectors`: simple objects using



```
blockA.\
.add( ConstAssign('x',5) )\  <- returns blockA
.add( If('b') )\  <- returns  a new ThenBlock
        .add( ConstAssign('y',5) )\ <- returns  the same ThenBlock
    .Else()\  <- returns  a new ElseBlock
        .add( ConstAssign('y',5) )\ <- returns  the same ElseBlock
    .EndIf()\  <- returns blockA
.add( ConstAssign() )   <- returns blockA
```

Figure 4: UML class diagram and an example explaining the internal workings of the if-else statement's implementation.

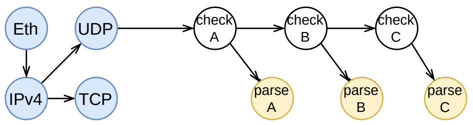


Figure 5: A simple parser-chain.

the code generator library. First, the `FlowSelector` defines the underlying standard protocols with a single constant (e.g. `IPV4_UDP`). Since one might have different selectors for the same underlying protocols, the code generator organizes the P4 parser states in a chain-like structure (see Fig. 5). If the condition of the selector is met, then we extract the application-layer data, otherwise we check the next selector in a different state. If there is no next selector, we proceed to the next desired state, e.g. `accept`. To deal with unused, empty chains, one can use `#define` pragmas. The P4 template always expects an empty chain unless a particular macro is defined. The generator can easily insert this `#define` pragma into the generated parser code.

The programmer can define the selection criteria using a list of pairs. Each pair consists of a field name and a required value. The used field can be a standard field of a standard header (e.g., UDP destination port) or the member of a user-defined struct describing the beginning of the application data. The latter one leverages the lookahead capabilities of the P4 parser.

4.4 Templates

Templates serve as a static starting P4 code for an offloading project. It can be used to describe the parsing of standard headers or implement basic forwarding rules. The template only references the generated code parts using the `#include` preprocessor pragma. Theoretically, the developer can use as many extra generated files as she wishes. However, at least one include pragma in the custom headers, parsing, header-struct body, ingress declarations, and ingress `apply` block seem necessary. A template must also provide an interface for the code generator by defining certain metadata variables and macros. For example, we maintain the number of added and removed bytes using metadata variables and provide workarounds for the atomic block using macros. One might use macros for every purpose, thus making the template codes more flexible and more complex at the same time.

5 CASE STUDIES

This section shows the application of P4rrot with two case studies. The complete source code of these case studies (and other simple examples) is also available under an MIT license.

5.1 Publish-subscribe sensor data

We consider MQTT-SN, a simplified version of the MQTT publish-subscribe protocol, and a popular solution for Internet-of-Things protocols. In this architecture, the clients subscribe to different topics represented by a string or a topic ID. Once someone (e.g., a sensor) publishes a message, the central component, called broker, distributes the data to every subscriber. The question arises of how topic names convert to topic IDs. MQTT-SN has an extra REG and REG-ACK packet querying the ID for a given topic name.

In our scenario, we have a sensor emitting the temperature value and a safety component that publishes an emergency “stop” message if this value exceeds 70. The safety component runs inside a server using a smartNIC (or other suitable P4 programmable device). Our goal is to reduce the response time of the safety component.

To implement this functionality, We create a `FlowSelector` prescribing the “publish” messages to our `FlowProcessor` performing this simple check. This processor gets the required topic ID from a `SharedVariable` (a long register array in P4). If the message is a temperature value above the threshold, we transform the packet and send it back as an emergency stop message. This transformation overrides the topic ID to the “alarm” topic’s ID (also got from a `SharedVariable`) and the message payload. Finally, the `SendBack` Command takes care of swapping the addresses and sending back the packet through the physical port it came from.

Our implementation also catches the REG and REG-ACK packets to store the previously mentioned topic IDs, thus making our solution entirely plug-and-play. The responsible `FlowSelectors` and `FlowProcessors` are also simple and easy to understand. With this extension, our solution is 150 lines of code with comments and empty lines included. We can also imagine a template specifically made for MQTT-SN traffic, thus enabling even simpler implementations.

5.2 Real-time data streams

Our second case study revolves around a service supporting real-time data streams, as they for example appear in the context of high-frequency trading where interested parties monitor market events (e.g. buy and sell orders, executed transactions).

Specifically, we consider the ITCH application layer protocol provided by NASDAQ. ITCH messages are sent to the customer using MoldUDP as a transport layer, which is similar to the original UDP protocol but with a special payload structure. After the standard UDP header, there are three additional fields: session (identifying the stream), sequence number (the ID of the first message), and message count (the number of messages included in the MoldUDP packet). Finally, we find the message blocks consisting of two parts: the length of the ITCH message and the ITCH message itself.

Clients can detect packet loss by observing the MoldUDP sequence numbers. In case of a loss, they can request the missing messages from a different retransmission server. Our task is to request retransmission automatically, thus saving time. A retransmission request has the same structure as every MoldUDP packet with a slightly different meaning: session, sequence number (the ID of the first requested message), and message count (number of consecutive messages requested).

In our implementation, we have created a `FlowSelector` for the MoldUDP packets. The corresponding flow processor maintains a `SharedVariable` containing the expected following sequence ID. If the next ID is as expected, we simply update the desired next ID using the “message count” field. Suppose the sequence ID is higher than expected. In that case, we remove the message blocks (using the `TruncateRemainng` command) and send back the packet to a retransmission server and deliver the copy of the original packet using the `ClonePacket` command. If the sequence number is less than expected, then it is a retransmitted packet; if the packet ordering retransmission was already requested, we do nothing.

We tested our solution using Python scripts mimicking the MoldUDP protocol, and interestingly, the P4rrot based solution has a similar length to the script mocking the client.

6 EVALUATION

6.1 Observability of the solutions

One of the main questions arising is the functional correctness of the generated code. It is important to test if the generated code works as expected, e.g., that the responses are indeed coming from the switch or smartNIC and not from the host machine.

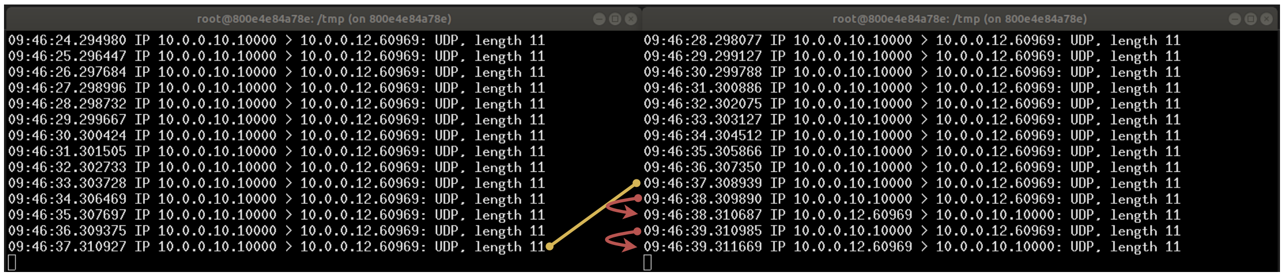
In the case of the MQTT-SN use case, we can quickly work around the problem by introducing a slightly different payload indicating the source of the emergency stop. The operational behavior can be checked by analysing the packet traces captured by tcpdump. Fig. 6 depicts that the packet carrying sensor values higher than the threshold does not arrive at the host since the reply is generated by the smartNIC.

MoldUDP offers less marking opportunities. Besides observing the tcpdump logs, our client introduces an option to disable retransmission requests. Thanks to this option, every retransmission request we observe must came from the activity of the smartNIC. The performance measurements of Section 6.2 was performed this way.

6.2 Response times

We compared the reaction time of the offloaded and the original non-offloaded MoldUDP implementations. Our test MoldUDP server sent out a packet in every second where each packet triggered a retransmission request.

Fig. 7 depicts the observed response times on a 2x25G Netronome smartNIC between two servers. We used tcpdump to capture the relevant packets and iperf to generate background traffic. The offloaded solution requests retransmission faster. Moreover, it is less affected even by the 10 Gbps background traffic. Our measurements


root@800e4e84a78e: /tmp (on 800e4e84a78e)
root@800e4e84a78e: /tmp (on 800e4e84a78e)

```
09:46:24.294980 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:25.296447 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:26.297684 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:27.298996 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:28.298732 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:29.299667 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:30.300424 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:31.301505 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:32.302733 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:33.303728 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:34.306469 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:35.307697 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:36.309375 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:37.310927 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
```

```
09:46:28.298077 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:29.299127 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:30.299788 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:31.300886 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:32.302075 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:33.303127 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:34.304512 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:35.305866 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:36.307350 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:37.308939 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:38.309890 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:38.310687 IP 10.0.0.12.60969 > 10.0.0.10.10000: UDP, length 11
09:46:39.310985 IP 10.0.0.10.10000 > 10.0.0.12.60969: UDP, length 11
09:46:39.311669 IP 10.0.0.12.60969 > 10.0.0.10.10000: UDP, length 11
```

Figure 6: Tcpdump before and after the P4-switch using BMv2. Packets are passing through up until the yellow pair because the sensor value is below the threshold. After that point, the sensor packets are returned as alarm messages (red arrows).

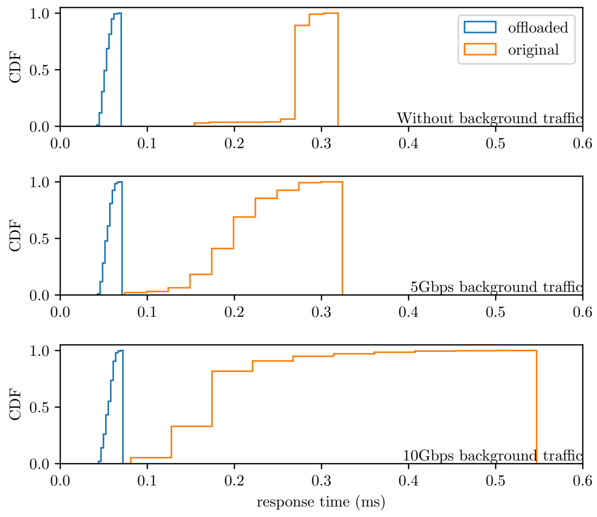


Figure 7: The observed response times on a 2x25G Netronome smartNIC between two servers

confirm, that P4RROT-generated solutions can indeed take advantage of offloading.

6.3 Code generator performance

The time required to run the code generator is negligible. It is a fraction of a second since we do not run any heavy optimisations or other complex algorithms. The generated P4 code is well-formed and easy to read.

P4RROT can keep the flow of the processing logic together. Fig. 8 depicts a switch-case logic described in Python and the generated P4 code using tables. The arrows show how the business logic gets rearranged to fit P4's abstractions. Implementing a switch-case branching with a table instead of nested if-else statements can yield a more efficient and less resource-intensive solution depending on the target device. However, describing a piece of code in a table fragments the flow of logic, thus tearing apart parts that the programmer might wish to keep together.

We also analysed the code length required to describe the same behaviour. The P4RROT description (Python3) was 54 non-empty lines of code in the stock market data monitoring use case while 113 lines in the MQTT-SN use case. The total length of the P4 code was 343 and 382 lines, approximately three times more. After optimising the generated parts, we can reduce these values to 330 and 337. If we count only the automatically generated code parts by excluding the template, we get 58 and 139 non-empty lines of code that can be manually reduced to 45 and 94. These values are already satisfying. However, we want to highlight that these targets and use cases do not require excessive table usage, which could further increase the usefulness of the code generator.

7 ADDITIONAL RELATED WORK

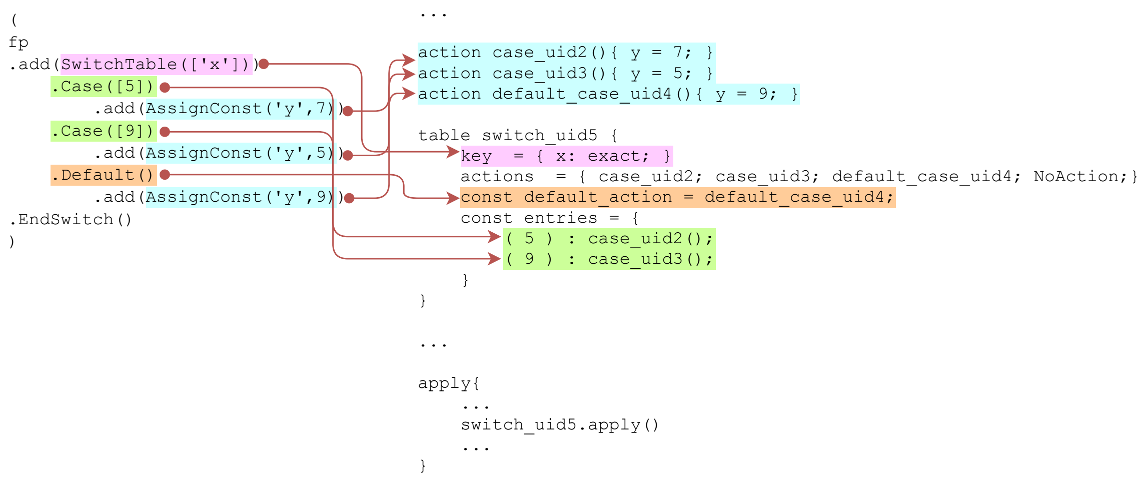
Generating lower-level code for a specific use case from a high-level language is a common theme for simplifying application development and used in many contexts. For example, TensorFlow [1] allows Python users to create computation graphs and run them on GPU, and Keras [9] allows to define and use neural networks using a simple API. A similar networking related example is MoonGen[4] which facilitates performance measurements using DPDK and Lua. However, these do not focus on P4.

We are also not the first to present a code generator which outputs P4 code. For example, Graph-To-P4 [17] is a boilerplate code generator for parser graphs. Also different packet filtering solutions convert logical expressions (concerning application data) to P4 code, such as CAMUS [6] and FastReact [16]. LUCID [15] is an entirely new language meant to implement control plane functionality in P4 data planes. However, these and other examples [5, 8, 13] do not focus on offloading arbitrarily defined application tasks. To best of our knowledge, we are the first to provide an interface that is capable of describing general application-layer features.

8 DISCUSSION AND FUTURE WORK

We presented a code generator, P4RROT, which allows to provide a familiar and straightforward interface to the P4 programmer, hence simplifying application offloading. P4RROT's simple interface is possible by narrowing down the scope to offloading application functionalities. We note that our approach is not limited to the Python language and the same design can be used for other high-level languages, e.g. Java, C#.

While P4RROT can already be useful in its current form, our project is still in an early stage. P4RROT is open source and we can imagine even complex behaviors defined in tens of lines of code,



```
(
fp
.add(SwitchTable(['x']))
    .Case([5])
        .add(AssignConst('y',7))
    .Case([9])
        .add(AssignConst('y',5))
    .Default()
        .add(AssignConst('y',9))
.EndSwitch()
)
```

```
...

action case_uid2(){ y = 7; }
action case_uid3(){ y = 5; }
action default_case_uid4(){ y = 9; }

table switch_uid5 {
    key   = { x: exact; }
    actions   = { case_uid2; case_uid3; default_case_uid4; NoAction;}
    const default_action = default_case_uid4;
    const entries = {
        ( 5 ) : case_uid2();
        ( 9 ) : case_uid3();
    }
}

...

apply{
    ...
    switch_uid5.apply()
    ...
}
```

Figure 8: A switch-case logic described in P4rrot (left) and the generated P4 code (right).

using bloom-filters, heaps, floating-point operations and other high-level abstractions. We also plan to add extra functionalities to the existing system, including a built-in way for the interaction between the data plane and the non-offloaded server components.

Furthermore, our approach is not architecture-specific. Adding new targets is easy since target-specific behavior is encapsulated in separate objects and templates. Thanks to this encapsulation, one can easily add new targets by providing templates and target-specific `Commands` and `SharedElements` and reuse the core code generator functionalities. From our experiences, TNA (Tofino Native Architecture) developers can greatly benefit from quickly switching implementation alternatives. Supporting the Tofino ASIC is already a work in progress with promising results. In the case of the support for BMv2 and Netronome smartNIC, we only experienced minor workarounds because of minor technical differences and bugs since both targets use the v1model as their architecture.

P4rrot may also open a business opportunity for companies. We can imagine scenarios in which the templates and some additional extensions are proprietary and in which the end-users write python scripts. After the code generation, the P4 code is automatically compiled and loaded to the company's device.

ACKNOWLEDGMENTS

Research supported by the German Bundesministerium für Bildung und Forschung (BMBF) project, 6G-RIC: 6G Research and Innovation Cluster, 2021-2025. S. Laki thanks the support of National Research, Development and Innovation Office - NKFIH, FK_21 138949. The research was co-funded by the project Strengthening the EIT Digital Knowledge Innovation Community in Hungary (2021-1.2.1-EIT-KIC-2021-00006), implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the 2021-1.2.1-EIT-KIC funding scheme.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Fabricio E Rodriguez Cesen, Levente Csikor, Carlos Recalde, Christian Esteve Rothenberg, and Gergely Pongrácz. 2020. Towards Low Latency Industrial Robot Control in Programmable Data Planes. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 165–169.

[3] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. 2002. Globally distributed content delivery. *IEEE Internet Computing* 6, 5 (2002), 50–58.

[4] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*. 275–287.

[5] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 357–371.

[6] Theo Jepsen, Ali Fattaholmanan, Masoud Moshref, Nate Foster, Antonio Carzaniga, and Robert Soulé. 2020. Forwarding and routing with packet subscriptions. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 282–294.

[7] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.

[8] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable {In-Network} Security for Context-aware {BYOD} Policies. In *29th USENIX Security Symposium (USENIX Security 20)*. 595–612.

[9] Nikhil Ketkar. 2017. Introduction to keras. In *Deep learning with Python*. Springer, 97–111.

[10] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (2015), 14–76.

[11] Sándor Laki, Csaba Györgyi, József Pető, Péter Vörös, and Géza Szabó. 2022. In-Network Velocity Control of Industrial Robot Arms. In *Proceedings of 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)*.

[12] Oliver Michel, Roberto Bifulco, Gabor Retvari, and Stefan Schmid. 2021. The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications. In *Proc. ACM Computing Surveys (CSUR)*.

[13] Vikram Nathan, Srinivas Narayana, Anirudh Sivaraman, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Demonstration of the marple system for network performance monitoring. In *Proceedings of the SIGCOMM Posters and Demos*. 57–59.

[14] P4Lang. 2015. Packet Test Framework. (2015). https://github.com/p4lang/ptf

[15] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: a language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 731–747.

[16] Jonathan Vestin, Andreas Kassler, and Johan Åkerberg. 2018. FastReact: In-network control and caching for industrial control networks using programmable data planes. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. IEEE, 219–226.

[17] Eder Ollora Zaballa and Zifan Zhou. 2019. Graph-To-P4: A P4 boilerplate code generator for parse graphs. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 1–2.