Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

In-Network Complex Event Processing using Advanced Data-plane Programming

Marius Maaß

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	M.Sc. Thomas Kohler, Dipl.-Inf. Ruben Mayer
Commenced:	December 1, 2017
Completed:	June 1, 2018

Kurzfassung

Die Latenz der Ereigniserkennung ist ein wichtiger Aspekt der Verarbeitung komplexer Ereignisse (*Complex Event Processing*), der durch die Ausnutzung existierender Fähigkeiten von Netzwerkkomponenten verbessert werden kann. Aus diesem Grund wird in dieser Arbeit eine neue Sprache für das Ausdrücken von Regeln für die Erkennung komplexer Ereignisse und p4cep, ein Compiler, der diese Sprache in P4 Code übersetzt, vorgestellt, womit die Erkennung komplexer Ereignisse in die Datenschicht (*data-plane*) des Netzwerks integriert werden kann. p4cep liest eine Regeldefinition zur Erkennung komplexer Ereignisse in dieser Sprache ein, verarbeitet diese Definitionen weiter zu einem endlichen Automat und konvertiert diesen in P4 Code und Tabelleneinträge für die Weiterleitungstabellen des Switches. p4cep wurde mit dem Netronome SmartNIC getestet, evaluiert und mit dem CEP System Apache Flink verglichen. In einem einfachen Vergleichsfall wurden 18 µs Latenz mit p4cep und 232 µs mit Apache Flink gemessen, dies stellt eine Verringerung der Latenz um 93% dar.

Abstract

Event detection latency is an important aspect of complex event processing which could be improved significantly by utilizing the capabilities of networking devices, which already exist in computer networks. For this reason, this thesis presents a new language for expressing complex event detection rules along with p4cep, a compiler which translates these rules into P4 code, which can be used for integrating complex event detection into the data-plane. p4cep reads a CEP rule definition, builds a finite state machine for event detection and then converts the rules into P4 code and forwarding table entries. p4cep was tested and evaluated with the Netronome SmartNIC and compared against the state-of-the-art CEP framework Apache Flink. In a simple test case and with the Netronome SmartNIC, an event detection latency of 18 µs was measured for p4cep compared to 232 µs with Apache Flink, which is a 93% decrease in latency.

Contents

List of Abbreviations

AST Abstract Syntax Tree. 35

bmv2 Behavioral Model Version 2. 51

CAM Content Addressable Memory. 67

CEP Complex Event Processing. 9

DAG Directed Acyclic Graph. 21

NFP Netronome Flow Processor. 50

NIC Network interface controller. 50

PISA Protocol Independent Switch Architecture. 12

SDN Software Defined Networking. 9

TTL Time To Live. 15

1 Introduction

Complex Event Processing (CEP) is a method for inferring complex circumstances or *events* from multiple smaller, less informative, *basic* events. A common example of a real-world application of CEP is high frequency trading which uses basic events in the form of price changes and makes complex decisions based on several of these smaller events. CEP applications frequently rely on low processing latency and high throughput. In the case of high-frequency trading, this is obvious since here even single micro seconds of additional latency can cause significant profit losses. The currently existing, software based CEP systems usually have latency figures of approximately 200 µs [CM12] in the best case since they rely on the *middlebox model* where the CEP system receives events by inserting itself into the network stream which adds latency for every packet and decreases throughput.

Data-plane programming is a new development in the Software Defined Networking (SDN) research area which allows customizing forwarding operations executed on SDN switches by allowing the network operator to program the switch with custom code. This opens up many possibilities for utilizing the existing forwarding hardware for tasks which are not directly related to pure packet forwarding.

This thesis aims to improve the latency and bandwidth of CEP applications by moving the required processing operations into the networking hardware. Since this forwarding hardware is commonly not designed for general purpose computing tasks it will be necessary to determine what capabilities are available with data-plane programming and how those features could be used for implementing complex event detection operations. Once those capabilities are determined, this knowledge could be used for hand-writing CEP code for the data plane which. However, since the languages used for data-plane programming are specialized for this specific use case, manually writing the code has a high probability of resulting in wrong detection semantics if the data-plane language semantics are not mapped correctly to the CEP semantics.

To address this problem, this thesis presents a new language for expressing complex event processing rules which is designed to be expressed by the programming languages available for data-plane programming. Furthermore, a compiler will be designed and implemented which translates rules written in this new language to P4, a widely supported data-plane programming language. This compiler, called P4CEP, will allow a network operator to integrate CEP rules into the existing forwarding hardware without significant additional overhead.

The latency and throughput of the code generated by this compiler will then be compared to the state-of-the-art CEP framework Apache Flink by using real-world forwarding hardware such as the Netronome SmartNIC. In this evaluation it will be shown that the code generated by P4CEP has an end-to-end event detection latency of 18 µs. The latency of Apache Flink was measured in a similar environment which showed an end-to-end latency of 232 µs for the same use-case. This is a significant latency reduction of 93%.

Parts of this thesis have been accepted for publication [KMD+18].

Structure

This thesis is structured into the following chapters:

Chapter 2 – Background This chapter will explain the background information required for understanding the topics which are discussed in this thesis. It will go into detail about what Software defined networking is and how the data-plane can be configured with P4, a language specifically designed for data-plane programming. The chapter will also briefly explain the basic concepts behind CEP.

Chapter 3 – Related Work This chapter will show and summarize existing works which are related to this thesis.

Chapter 4 – System Model and Requirements The following chapters will work with a model of a system which will be described in this chapter.

Chapter 5 – Design This chapter will show the design of p4cep and the rule definition language used for expressing the complex events. It will also introduce a model for the basic and complex events.

Chapter 6 – Implementation Based on the design chapter, this will show how p4cep was implemented.

Chapter 7 – Evaluation This chapter will use the implementation and evaluate it by comparing it to Apache Flink, an existing CEP framework, and evaluating the overhead caused by the inclusion of p4cep.

Chapter 8 – Conclusion The thesis is summarized, final results are presented and possible future topics are described.

2 Background

This chapter will introduce some basic concepts which will make understanding the following chapters easier.

2.1 Software-defined Networking

A traditional network is composed of multiple special-purpose devices, that are designed for fast forwarding of packets based on headers of multiple well-established and standardized communication protocols. If at all possible, the forwarding tables and other configuration settings need to be changed using a proprietary protocol which is very often not compatible with other forwarding hardware which makes a homogeneous control software hard to implement.

SDN is a term used for describing a technique which allows to influence the forwarding information of switches with SDN support with standardized protocols and from a central location. Since the early SDN protocols such as OpenFlow were designed to be usable with forwarding hardware which already existed at that point, the possible operations were limited. However, if all switches were configured with firmware which supported the OpenFlow API, then every switch in the network could be configured with OpenFlow which made the management of the network easier than it was before.

When a packet arrives at an SDN switch, it is matched to a *flow* using a *Flow table*. A flow is defined by a set of header values (for example the IPv4 destination address and the TCP destination port) which identify what communication flow a packet belongs to. Each flow has a set of actions, that are applied to every packet of a specific flow. These action can include setting the forwarding port which determines on which port the packet will exit the switch again. It is also possible to manipulate the values of certain header fields such as the Ethernet source and destination addresses or the IP address fields. The exact set of actions, that are supported by a switch is dependent on which OpenFlow version it supports.

For example, a controller could add a flow rule for packets with a destination IP of `10.10.0.0/16` and a TCP port of 80. If a packet of this flow is identified, then the port out of which the packet should be sent is specified with an OpenFlow action. Traditionally the switch would have needed to determine that information using various distributed algorithms which may or may not yield an optimal path or even a solution at all. By using the separation between a controller and the forwarding- or *data-plane* this process can be optimized. This separation of *control-plane* and *data-plane* is one of the central aspects of SDN and also an important aspect for this thesis.

The basic structure of an SDN enabled network is shown in Figure 2.1. The control plane communicates with the data plane through a standardized control protocol such as the OpenFlow protocol mentioned above. Using this protocol, the control plane sends *flow table* entries to the *data-plane*. The routers then use this configuration and forward packets according to the specified rules. The data plane also has a communication channel for sending data back to the control plane. This can include meta data such as changes in the link states between routers and also entire packets when the control plane configured the data plane so that unknown or special packets are sent to the control plane for further processing.

When using OpenFlow, the rules for packet forwarding can be configured from a central location but the header fields these rules operate on are still largely static since the SDN switch may use specialized hardware for efficiently parsing the headers from a byte stream. This means that while new forwarding algorithms can be deployed easily it is nearly impossible to introduce new protocols into the network without also changing the forwarding hardware. This gave rise to Protocol Independent Switch Architecture (PISA) and *data plane programming* which is the topic of the next section.

2.2 Data Plane Programming

The original SDN concept revolved around controlling the data flows based on existing Internet protocol headers such as IP, TCP or UDP. While this covers a lot of possible use cases, it is not flexible enough for all applications. If a new protocol header is introduced in a network all routers

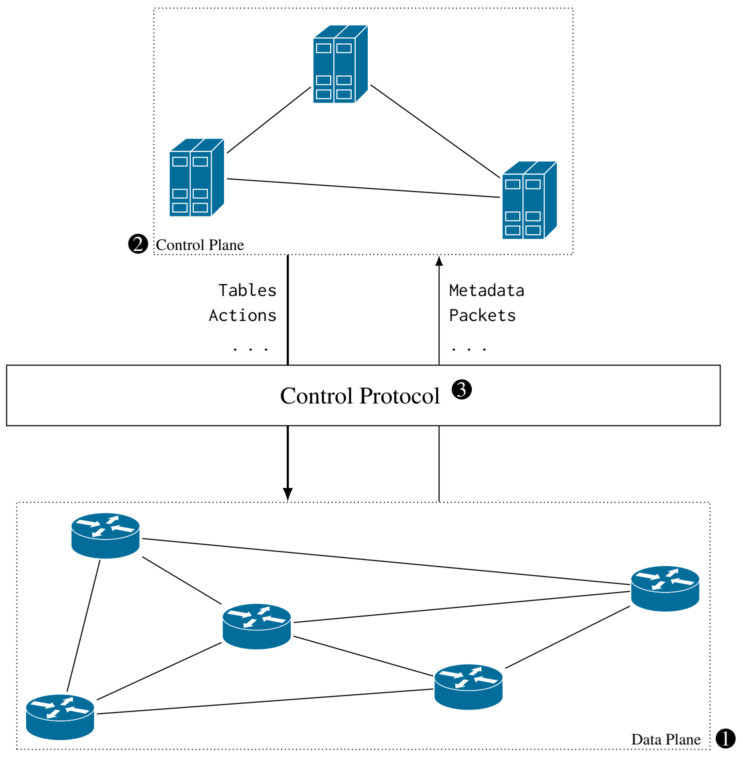


Figure 2.1: Schematic overview of an SDN network. The data-plane (❶) contains the switch hardware conducting all forwarding operations. The control plane (❷) controls how this happens by changing table entries through a standardized API (❸), for example OpenFlow.

need to understand the fields of this protocol before it can be implemented successfully. This is cumbersome to implement in software switches since the software of all software switches needs to be adapted to the new protocol. If hardware switches are involved this is effectively impossible since the hardware cannot be changed anymore. In this case, a new protocol needs to be added by the hardware vendor which may take multiple years.

For this reason the domain specific language P4 [BDG+14] was designed to fix this inherent issue of previous SDN designs by introducing a language for programming the Protocol Independent Switch Architecture concept.

2.2.1 P4

P4 is a domain specific language specifically designed as a language for programming the packet forwarding behavior of SDN switches. It is capable of expressing custom header formats and specifying code for parsing these headers dynamically from a packet. It features support for custom forwarding tables and various other forwarding related constructs such as header field manipulation, counters and registers.

This allows P4 to be completely protocol independent. If a new protocol needs to be deployed in a network it is as easy as changing the P4 program to include support for the new header structures. Another defining feature of P4 is reconfigurability. New P4 programs can be deployed to switches when required instead of waiting for new hardware. The final aspect is target independence. Most switches can be programmed, at least partially, using low-level software, often called firmware. This was used for the initial OpenFlow versions where existing hardware was upgraded to support OpenFlow. However, this software is always highly target specific and firmware written for one switch is most likely not compatible with a switch from another hardware vendor. P4 aims to improve that situation by keeping the language free from any target specific features and instead relies on a general set of features that needs to be supported by a target. Then a target specific compiler will be written which translates the P4 code to a representation suitable for that target, for example new firmware or a new hardware design for FPGAs.

This section will discuss the $P4_{16}$ [P4 17] language version which has a number of improvements over the language presented by Bosshart et al. [BDG+14] which was later known with some minor changes as $P4_{14}$.

Header format definition

Headers are specified in a syntax which is similar to a C `struct`. Listing 2.1 shows the definition of the IPv4 header without an options field. The `typedef` functionality is similar to C which allows to create an alias for a type to make reading types easier by giving specific fields easy to understand names. `bit<n>` is the basic unsigned integer type of $P4_{16}$. It represents an unsigned integer type with a width of n bits. Internally, every header also has a single bit value which indicates whether the header instance is holding valid data. This bit is initialized to 0 and set to 1 if the header is parsed. It can be checked with the `isValid()` method and can be modified with the `setValid()` and `setInvalid()` methods.

Listing 2.1 P4$_{16}$ definition of a simple IPv4 header without options fields.

```
typedef bit<32> ip4Addr_t;

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
```

P4 Program Pipeline

A P4 program is organized in a pipeline of function blocks which process input data and pass it to the next block in the pipeline. A schematic overview of this pipeline is shown in Figure 2.2. The following section will describe how these function blocks are programmed using P4.

Parser The `parser` construct allows a P4 program to specify how the headers of a packet will be parsed. A parser definition of a standard Ethernet and IPv4 packet is shown in Listing 2.2. The parser is represented by a finite state machine where the states are responsible for extracting the individual header structs into runtime variables. Transitions can be made conditionally based on the value of a parsed header fields which can be seen in line 10 of Listing 2.2 where the parser

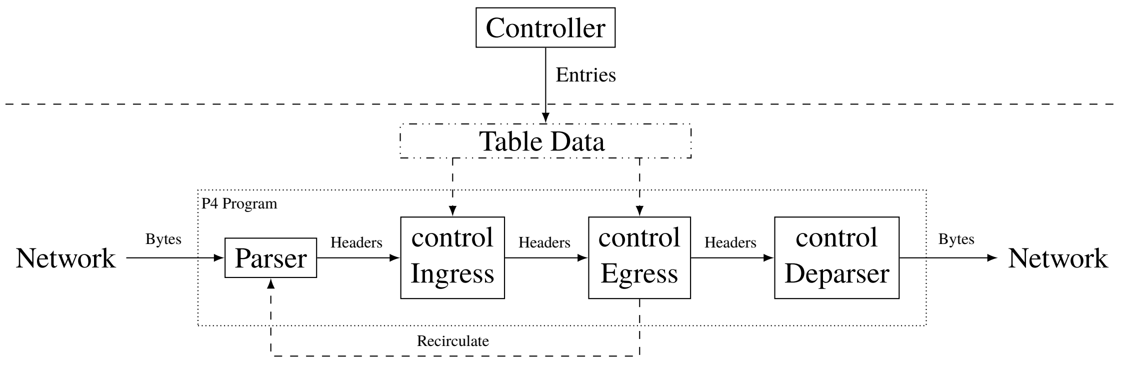


Figure 2.2: The abstract structure of a P4 program. Packet data is received from the network, parsed into the header structures in the Parser and passed between control stages until it is sent out into the network again. The ingress and egress stages may use table data set by the control plane for making forwarding decisions. A special recirculate path is available for sending packet from egress back to the parser.

checks the etherType field of the Ethernet header and continues parsing the IPv4 header if the type matches the defined value of IPv4. Once the parser has finished all extraction operations, it returns the accept state which terminates the parsing code.

Control blocks In a P4_{16} program control blocks are the constructs responsible for deciding what happens to a packet. This section will discuss individual characteristics of the control block. The full example is available in Listing A.1 on Page 81. This example was written for the bmv2 P4 Software Switch [p4la].

control blocks are defined similarly to a C function which is shown in Listing 2.3. It receives the headers structure as a parameter which is the structure of headers parsed by the parser block. Additionally, some metadata for the packet is supplied. The meta field is a defined type which can contain arbitrary, user defined data for usage in the forwarding operations. standard_metadata is a structure which contains various standard metadata values for the packet such as through which port it was received or how much data is contained in the packet.

action statements are similar to C functions which can have parameters but no return value. Actions are typically used in combination with table lookups for matching input data to output data. Listing 2.4 shows the two action definitions used by this example. drop_action is an action without parameters which only calls the function mark_to_drop built into P4 which sets a flag for the current packet that it should be dropped when the Ingress control block is finished. ipv4_forward receives the values for the next hop of the packet as parameters and uses them to set on which port the packet should be sent out and what the next ethernet address should be. It also decrements the Time To Live (TTL) value of the packet.

Tables were already mentioned and are used for conducting data lookup based on some input parameters. They are declared using the table structure seen in Listing 2.5. A Table can have key, actions, size and default_action fields. The P4_{16} standard also specifies more fields which are not

Listing 2.2 P4_{16} parser definition for parsing a packet header consisting of the Ethernet and IPv4 headers.

```
parser Parser(packet_in packet,
              out headers hdr,
              inout metadata meta,
              inout standard_metadata_t standard_metadata) {
    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```

Listing 2.3 Definition of the P4$_{16}$ `Ingress` control block signature.

```
control Ingress(inout headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata)
```

Listing 2.4 Action definitions defined within the `Ingress` control block. See Listing A.1 for a complete example of how these actions fit into the control block.

```
action drop_action() {
    mark_to_drop();
}
action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
```

relevant here. `key` specifies a list of header fields which should be used for conducting the table lookup. Every field can have a specific lookup type. `exact` tells P4 that the input value must match the table value exactly. `range` allows to specify a maximum and minimum value and `lpm` conducts a Longest-Prefix-Match and receives values such as `10.10.1.0/24` if the value is an IPv4 address such as shown in the example.

`actions` specifies which actions may be called from this table. This is only a hint to the compiler what action calls are valid from this table. The actual action called for a specific entry in the table is specified by the control plane at runtime and not the P4 program. `size` is the number of entries that should be available in the table. `default_action` configures what action should be called in case no entry in the table matches the input values. If this action has parameters they must be specified in the P4 program.

After the actions and tables have been declared, the P4 program can use them to control the forwarding behavior. This is done using the `apply` block shown in Listing 2.6 which may contain `if` statments, table `apply()` calls and also code which is not wrapped in an `action`.

Most P4 targets also expose a second “Egress” control block type which is executed after the “Ingress” control block.

Listing 2.5 Table definition for a Longest-Prefix-Match (`lpm`) of a IPv4 address.

```
table ipv4_lpm {
    key = {
        hdr.ipv4.dstAddr: lpm;
    }
    actions = {
        ipv4_forward;
        drop_action;
        NoAction;
    }
    size = 1024;
    default_action = NoAction();
}
```

Listing 2.6 The actual code for the `Ingress` control block. It checks if there is a valid IPv4 header and uses the table from Listing 2.5 for executing forwarding actions.

```
apply {
    if (hdr.ipv4.isValid()) {
        ipv4_lpm.apply();
    }
}
```

Deparser The Deparser control block specifies how the packets parsed and modified by previous steps should be serialized when it is sent to the next switch. This function (shown in Listing 2.7) receives a `packet_out` variable and a parameter containing the header values produced by previous steps in the P4 pipeline. The `packet_out` type has functions for emitting header structures which can be used for specifying in which order the headers should be serialized. This order can be independent of how the headers were parsed by the `parser` block. If a header is not valid then it will not be emitted which allows supporting multiple mutually exclusive headers such as IPv4 and IPv6 at the same time.

Differences between $P4_{16}$ and $P4_{14}$

Before $P4_{16}$ there was the language version $P4_{14}$. This version is still used and supported by many P4 targets without support for the newer $P4_{16}$ language version. The new version $P4_{16}$ introduced some radical changes which are not backwards compatible with source code written for $P4_{14}$. However, most of the changes are not relevant here since they do not affect the design or capabilities of this project. For this reason, only the changes relevant to the implementation as described in Section 6.3.2 will be discussed here.

No code in control flow blocks As mentioned before, $P4_{16}$ allows writing code directly into the `apply` part of a `control` block without needing an intermediary `action`. This was not allowed in $P4_{14}$ and code could only be written in actions.

Different boolean operator names Some boolean operators were renamed when switching from $P4_{14}$ to $P4_{16}$ in order to move the language closer to the syntax known from C. The “Or” and “And” operators were called `or` and `and` in $P4_{14}$, but they were renamed to `&&` and `||` in $P4_{16}$.

Listing 2.7 $P4_{16}$ deparser definition. This writes the parsed Ethernet and IPv4 headers in the same sequence they arrived but with the changes applied in the `Ingress` control block.

```
control Deparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

No binary operators P4_{16} changed the syntax of how many operations are expressed. For example the P4_{16} code `hdr.ipv4.ttl = hdr.ipv4.ttl - 1;` is not valid in P4_{14} and needs to be written as `subtract(ipv4.ttl, hdr.ipv4.ttl, 1);`.

No local variables In the control blocks of P4_{16} it was possible to declare variables local to that control block without declaring them in a header or metadata field first. This is a new feature in P4_{16} and is not available in P4_{14}.

No custom deparser functionality The deparser functionality shown in Listing 2.7 is a new feature of P4_{16}. In P4_{14}, the headers were always written in exactly the order they were parsed in which made separating the incoming and outgoing header structure impossible.

2.2.2 Runtime Data and Design Time Data

In the context of data-plane programming two distinct data types exist which are both generated by the control plane. OpenFlow only used runtime data which was encoded entirely in the flow tables of the switch. This data can be changed at any time by the control plane. In P4 this data exists as the entries of the tables and the contents of the registers.

In addition to this, data-plane programming also has design time data which is the data that was generated when compiling the data-plane program for the target. To differenciate these two types, this thesis will use the term “Runtime control plane data” when referring to table and register contents and “Design time control plane data” when referring to any data that is contained within the data-plane program. Design time control plane data cannot be changed without recompiling the data-plane program and reconfiguring the target switch.

2.3 Complex Event Processing

Complex Event Processing describes the process of inferring more **complex events** from smaller, less informative **basic events**. For example, if a building was equipped with temperature sensors in every room then an increase in temperature could indicate either a fire or a problem with the heating or cooling system. The temperature value alone does not contain this information but if the Complex Event Processor combines the information returned by the sensors in the past with the current state it could determine whether the increase was sudden in which case it would suggest a fire or more slowly if the problem was caused by a misconfigured heating system. It is also possible to examine events from different event sources such as a smoke detector. If the smoke detector detects smoke in the air and there was a sudden temperature increase it is highly likely that a fire has broken out. However, if there is no smoke but still a sudden temperature increase the problem may not be a fire but something else which would require a different reaction.

The generalization of this paradigm is centered around three core concepts:

- Event sources
- Operators
- Event sinks

Event sources are abstract objects that generate *basic events*. A basic event may contain some information (for example a sensor value or an image). An operator receives basic or complex events and uses them for detecting a more complex event pattern. There can be multiple operators in the same CEP network where each operator detects a separate event pattern. Typically, operators are deployed in a graph where the output events of one operator are passed to one or more other operators or event sinks. Typical operators define an *event pattern*, for example `A; B; A` which means that first the event `A` must occur, then `B` and then `A` again. Recognizing these patterns is usually done using a state machine which consumes events and determines if an event sequence forms a complex event. Additionally, CEP systems have the capability of storing values from previous events and reexamining them at a later time. This is usually done using *windows* which provide a view of a limited amount of events that will be processed together. The size of this window can either be defined by a time span (for example the events of the last ten minutes) or an absolute number of events (for example the last 100 event regardless in what time they arrived). Event sinks are the opposite of event sources. They accept events (either basic or complex) and process them. This processing could be a logging operation where the detected event is written to a logging system for later examination or it could be a system which sends an E-Mail to an administrator if an abnormal condition was detected based on some sensor values.

The event sources, operators and event sinks communicate with each other. Depending on the context of the CEP applications this could be done by passing object references to each other if all operations happen within one process. It could also be done using a local socket when the complex event processing happens in multiple processes but on the same machine. However, if the framework uses distribution for increased performance it is always necessary to use a network protocol such as TCP or UDP for transmitting the messages between sources, operators and sinks.

There exists a multitude of different CEP systems such as Apache Flink [Apa] that can be used to develop custom event processing systems that detect complex events as specified by the user. Most of these systems use a centralized system where one or more instances of the framework work together which increases the amount of events that can be processed by the system at the same time. However, this also increases the hardware requirements since a fast CEP system needs adequate processing power to guarantee low latency and high throughput.

3 Related Work

There are some works related to the topic being discussed in this thesis. These will be discussed in the following chapter.

3.1 CEP

There exist numerous works discussing various aspects of CEP. In this section some works will be discussed which are, for various reasons, relevant for the topic of this thesis.

3.1.1 TESLA

TESLA [CM10] is an event definition language used for expressing complex events. The presented language is similar to SQL and defines complex events by combining predicates on primitive events with logical operators such as $\wedge$ (*and*), $\vee$ (*or*) and $\neg$ (*not*). The language also allows using time based operators which check if a certain condition has been true within a time span. Aggregate functions, such as the average, which collect previous values and compute a new value from them are also available.

Cugola and Margara also show how these complex events could be detected by describing how a complex event definition in the TESLA language could be detected by converting the rules specified within a complex event into multiple state machines that evaluate events incrementally as new basic events arrive.

This is related to this thesis since TESLA was used for determining what operators are important for a CEP application. This information was then integrated into the design of P4CEP.

3.1.2 Apache Flink

Apache Flink [Apa] is a popular open-source CEP framework. For users, it provides two different kinds of APIs. The *DataSet* API can be used for processing large data sets in an offline setting where processing jobs may run as batch jobs. The *DataStream* API is designed for handling real time processing applications and is more useful for traditional CEP applications where events should be processed as they arrive.

Carbone et al. [CKE+15] describe the inner workings of the framework in more detail. The basic primitives Apache Flink uses for processing data are the *Dataflow Graphs*. These Directed Acyclic Graph (DAG) structures encode how data flows between sources, operators, sinks and external data storage devices. Flink provides a high level of fault tolerance which allows it to guarantee strict exactly-once-processing guarantees. It does this by using checkpoints which store the internal state of stateful operators and partial re-execution in case of failures. The authors also describe and evaluate how the internal buffer management configuration affects the latency and throughput of events. If the internal buffers are small and have a low timeout value they pass their data to the next operator more frequently which achieves a low latency of 20 ms but also with a decreased throughput of 1.5 million events per second. Increasing the buffer timeout increases the throughput while at the same time increasing the latency which may be desirable for some applications. When setting a buffer timeout of 50 ms the throughput is increased to 80 million events per second but the latency is also increased to 50 ms. This is an important configuration parameter for evaluating the minimum latency of Apache Flink which is a useful information for latency comparisons with Apache Flink.

The framework is extensible and allows the user to add custom data sources, data sinks and operators which can provide the same level of fault tolerance as the built-in operators if used properly.

Since this project is widely used and already shows relatively low processing latency, it was used as a comparison basis when evaluating the performance of p4cep in Section 7.2.

3.2 P4 / SDN

SDN is a widely discussed topic which includes a lot of academic works and commercial developments. However, data-plane programming is a relatively new trend in the SDN community which is also supported by hardware vendors such as Barefoot and Netronome who support P4 on their hardware platforms. This section will discuss some general SDN and data-plane programming topics which are related to this thesis.

3.2.1 SNAP

SNAP [AKG+16] is a high-level language for expressing stateful packet operations. The language provides operations for managing the state of the program. The SNAP code is translated to the NetASM [SF15] language which can then be compiled to executable operations for various targets such as FPGAs and NPUs.

The SNAP compiler intelligently distributes the operations of the SNAP program in the network by analyzing the dependencies within the program and mapping them to a modified integer linear program which can then be optimized to yield rules which are distributed in the network. SNAP also supports atomic operations.

Since complex event processing requires a lot of stateful operations, SNAP is related to the topic of this thesis. However, SNAP is aimed more at implementing stateful networking operations instead of complex event processing. Furthermore, instead of P4 it uses a lower-level language as the output language of its compiler.

3.2.2 OpenState

OpenState [BBCC14] is a proposed addition to the OpenFlow protocol which would extend it by allowing the control plane to execute stateful operations on incoming packets. This can be used for implementing actions ordinarily only possible on more powerful end-systems such as port knocking. Port knocking is a technique where a service only becomes available to a client if it sends a specific sequence of packets to the host. This is commonly used by firewalls for protecting an otherwise vulnerable service from port scanning. Since ordinary OpenFlow systems do not allow the controller to store values directly on the switch, it is not possible to implement this technique or similar operations directly in the switch. In this case, an external, more powerful machine is needed for checking the network traffic for these port knocking sequences.

OpenState proposes an additional, stateful table which stores the state of a finite state machine. Since a single state machine would not be very useful, this state table also contains a *flow key* which identifies which flow a stored state belongs to. If a packet arrives the flow key is looked up in the state table which retrieves the current state of the state machine. The state is then combined with the value of a header field (for example the TCP port) which forms the state lookup key. This uses the familiar *match-action* semantic with an additional step after executing the action. The action can be one of the standard OpenFlow action such as *drop* or *forward*. In addition to the action, the table lookup also returns the next state according to the configured finite state machine. The final step is to write the new state back to the state table at the correct position according to the flow key which was used for the initial state lookup.

In addition to an OpenFlow specification, an implementation of OpenState also exists for the P4 language [San]. It uses the $P4_{14}$ language version for implementing the state machine semantics defined in the original paper.

P4CEP uses state machines for recognizing complex events and uses a mechanism similar to what OpenState proposes. OpenState manages individual state machines for multiple flows which is not necessary for P4CEP since it does not operate on a per-flow basis.

4 System Model and Requirements

In this section, a formal definition of the system model and the requirements for that system will be given. Unless stated otherwise, the remaining sections will assume that this model is used when referring to the system.

4.1 System Model

A representation of the system model is shown in Figure 4.1. The system consists of a set N of one or more network nodes which are connected by a set of switches S (Figure 4.1 only contains one of the switches as an example). There is a subset of network nodes $E_o \subseteq N$ which also act as *event sources* which generate basic events as defined in Section 5.1. Another subset $E_i \subseteq N$ acts as *event sinks* which receive event notifications from the event sources. An event source may also act as an event sink at the same time.

The network nodes will provide common services such as web servers or database APIs and will communicate with each other using standard network protocols such as TCP or UDP via IP. The event sources send their event notifications into the network using the same networking protocols.

There is a controller C which controls the forwarding behavior of the switches in the network by configuring the flow tables of the switches S present in the network. This controller has a global view of all switches and can individually configure the flow tables of each switch.

4.2 Requirements

The switch needs to support dynamic forwarding table configuration and it needs to be able to be programmed using the P4 data-plane programming language. Using P4, the switch implements traditional forwarding using the information supplied by the controller. Additionally, the P4 code contains parts that detect complex events in the event stream generated by the event sources in the network. Network traffic that does not encode basic or complex events is forwarded using the standard forwarding behavior and is not affected by the complex event processing code.

Basic event notifications sent by event sources need to be encoded in a way that can be understood by the P4 parser system. The general capabilities of the P4 parser system (how many headers and how much data) are determined by the switching hardware which is in use in the network. If a basic event notification is too complex to be parsed by the switches (for example complete JPEG or PNG images) then the basic events will not be processed by the CEP code present in the switches.

The CEP code will only receive the events that arrive at the switch which means that it does may not have a global view of all events generated by all event sources.

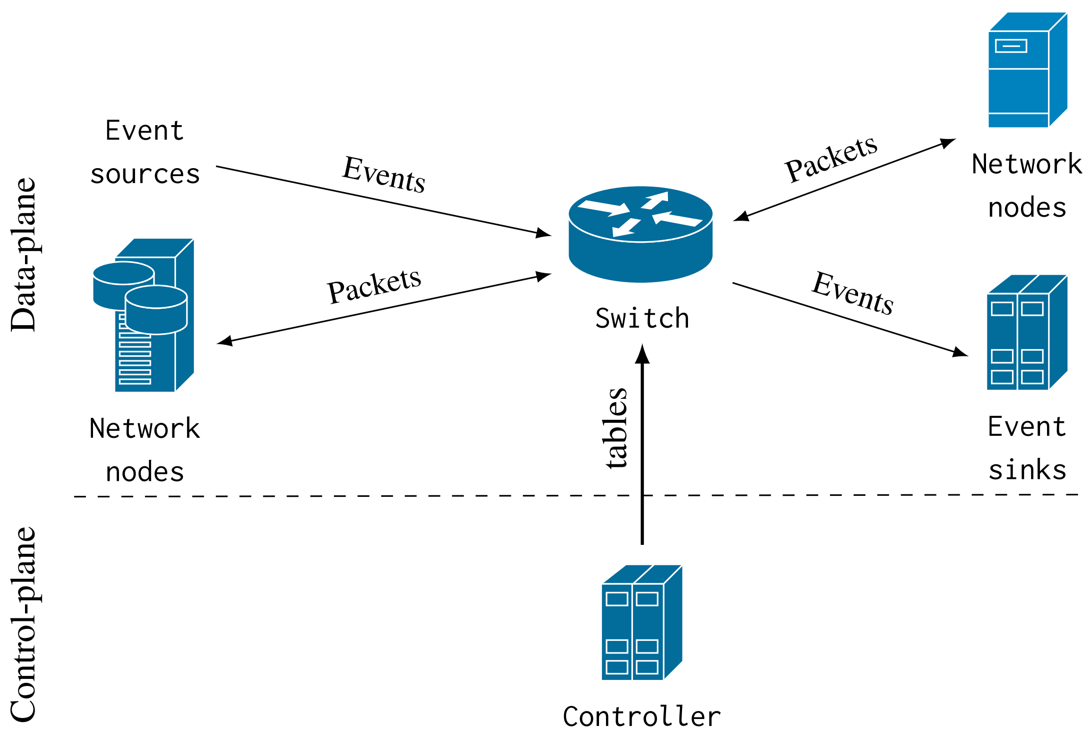


Figure 4.1: Basic system model used for this thesis. In the system there are multiple network nodes which may also act as event sources or event sinks. The nodes in the network are connected via one or more switches which are configured by a central controller which controls the data flows in the network.

5 Design

Detecting complex events using an in-network implementation requires that it is known what is considered to be an event and a complex event. Furthermore, while the P4 code required for detecting these events and their processing could be written by hand it would be easier for the user to have some sort of rule definition language which would then be translated to P4 by a compiler.

This section will first present what exactly is considered to be an event and then it will present the design of a rule definition language and a compiler designed for translating this language to P4 code.

5.1 Event-model

There are two general kinds of event that are being considered separately here. The first are **basic events** and the second are **complex events**.

Basic events are generated by event sources present in the network. An event is defined by a predicate p which is applied to event data d. The event data d is an array of byte data which was received by the switch which is processing the events. This byte array is then parsed into a form which is usable by the predicate. The complexity of the parsed form of the data is determined by the parsing capabilities of the switch. Since this thesis is based on P4, the parsing capabilities of P4 determine the possible data formats. The parsing capabilities of P4 are based on a finite state machine so the theoretical complexity of the parsed data format is limited. However, in the context of header parsing in dedicated switching hardware, the parsing capabilities of P4 are extensive which will allow using most event data formats.

The predicate p, when applied to d, determines if a packet is an instance of a basic event. There may be multiple separate predicates which each define their own basic event. If the predicate evaluates to true, then that packet is considered to be an instance of this **basic event**. The possible complexity of p is defined by the capabilities of the used target language (such as P4) and may include logical operators for combining multiple conditions into one event predicate. Section 5.2 will go into more detail about how the predicates may be defined and what limitations they have.

The source of the event data may be a simple temperature sensor which sends its data into the network using UDP datagrams or it could be a load balancer software in a data center which reports the loads on individual nodes in the network. If there was a basic event for when the temperature is above 50 °C then the predicate would be $temp.val > 50$. If the temperature was encoded in a UDP datagram then the switch would parse the UDP payload as a single value and make that value available to the predicate as $temp.val$. While basic events are defined as predicates, the basic event data is separate from that and can be used multiple times which may also include forwarding the event data to another switch or event sink.

A **complex event** is an occurrence of an event which was inferred from several basic events by using a set of user defined rules. Complex events are sent to event sinks in the network for further processing or logging. The user defined rules consist of basic events and more complex, stateful operators. These stateful operators are represented by function calls with parameters which define their return value. It is also possible to use a sliding window over previous values for making decisions based on a limited view of the past.

5.2 Rule Definition Language

Writing P4 code for detecting events as described in Section 5.1 by hand would be very error prone and work intensive due to the restrictions of the P4 language. For this reason, a specialized CEP to P4 compiler was developed which reads the CEP rule definitions written in a custom language and translates them into standard P4 code. This tool is called P4CEP and is the focus of this capter and Chapter 6.

In Listing 5.1 an example of a rule file can be seen. This example demonstrates all the features available in P4CEP which will be described below. The capabilities of this language were defined based on the CEP event model described in Section 5.1 and a survey of the currently available features in other CEP frameworks.

The rule definition language frequently makes use of P4 conditions which are used with only minor changes in the resulting P4 code. These must be syntactically and semantically valid at the point in the P4 source code where the P4CEP generated code is inserted. P4CEP can not validate the names or types of most P4 expressions given in the rule file since it does not have the necessary information available. All P4 operators which are valid in a P4 `if` condition are also valid in P4CEP patterns.

5.2.1 Complex Events

Complex event definitions define when P4CEP should detect a complex event. Two examples of complex events are shown in Listing 5.1 in the lines 14 to 19 and lines 21 to 25. Since complex event patterns are defined across multiple packet instances, P4CEP needs to keep some state about what events it has already seen for that complex event. For this reason, a finite state machine is used which is built from the specified pattern. The operators supported in the pattern are defined in such a way that converting the pattern into a finite state machine is efficient to do while still maintaining the capability for expressing complex patterns. The finite state machine defines when the pattern is detected by reaching its end state.

The `value` of a complex event is an unsigned 32-bit integer which will be included in the complex event notification when the event is detected. The bit size of the value is defined by the size of the value field in the event notification header. This can be used for assigning custom metadata to a detected event. The `value` expression can be any valid P4 expression which has a 32-bit integer type or it can be a P4CEP function call which will be evaluated before sending the event notification.

Since CEP applications may have different requirements for when an event should be detected, P4CEP supports the `strategy` specifier for complex events. This specifies how packets should be handled that are not in the sequence of events as specified in the complex event pattern. The two

Listing 5.1 An example of a rule definition file which defines two windows, two events, two variables and uses functions to evaluate windows.

```
var len_threshold = 50;
var protocol_var = 17;

window test_wnd {
    size 5
    value hdr.ipv4.totalLen
}

window test_wnd2 {
    size 10
    value hdr.ipv4.protocol
}

complex_event test_evt {
    value count(test_wnd, $value > 20)
    strategy skip
    instances 3
    pattern [count(hdr.ipv4.totalLen > $len_threshold) > 5]
}

complex_event test2_evt {
    value min(hdr.ipv4.totalLen)
    strategy strict
    pattern ([hdr.ipv4.totalLen > 50] && [hdr.ipv4.protocol == $protocol_var]); ([min(test_wnd) >
     600] || [hdr.ipv4.protocol == 17])
}
```

available options are `strict` and `skip`. The names and semantics are based on the definition provided by Flouris et al. [FGD+17, Section 2.3]. Here, two strategies are particularly interesting for P4CEP. “Strict contiguity” requires that all events occur together and no unrelated event may occur in between them. This strategy is available as the `strict` strategy in P4CEP. “Skip till next match” is a more relaxed strategy which allows unrelated events to occur between relevant events. This is available as `skip` in P4CEP. For example, a complex event should be detected if the events `A` and `B` are detected in sequence. If the basic events arrive in the sequence `A C B` arrive, the detected complex event depends on the specified strategy. When the first `A` arrives both strategies behave the same and consume the event since it is expected as the first event in the sequence. If the second event `C` arrives, `skip` ignores the event since the complex event does not require that all events occur without interruption and the following `B` event triggers an event notification. As specified above, `strict` requires that all events occur together which means that the `C` event above means that the complex event `A - B` did not occur.

When the strategy is set to `skip`, the option `instances` can be set to a number higher than 1. The number of instances controls how many parallel event matchers will be active for this event. This allows detecting complex event instances even if the events that make up this complex event are interleaving in the event stream. When there is a complex event with a pattern that first expects the event `A` to appear and then `B` in exactly that sequence and the events `A A B` arrive it is obvious that the pattern should appear twice. However, if there is only one matcher of the complex event active it will consume the first `A` event and then expect a `B` event. The second `A` event will be ignored since it does not fit in the pattern established by the `A - B` sequence expected by the matcher. If there were two instances of the matcher the first `A` would match for the first instance of the matcher and instead of being ignored, the second `A` would start a new instance of the matcher that does expect a `B`. When the `B` event arrives the pattern of both instances is fulfilled and two complex events of the sequence `A - B` are recognized by P4CEP.

The `pattern` is the central part of the complex event definition. It defines what basic events need to happen in order to trigger the complex event. This uses the predicates as defined in Section 5.1 for determining what basic events have been detected in the current packet. This is done by specifying boolean P4 expressions that are executed on incoming events to determine if an event related to this complex event has occurred. In this pattern, brackets (`[` and `]`) are used to separate individual event predicates from the overall complex event pattern. The expressions between the brackets will be checked for every arriving packet and then be used as the input for the finite state machine. The pattern `([hdr.ipv4.totalLen > 50] && [hdr.ipv4.protocol == 17]); ([min(test_wnd) > 600] || [hdr.ipv4.protocol == 17])` from Listing 5.1 demonstrates the usage of all three available logic operators:

Sequence (`a ; b`) First the condition `a` must be true and then `b` in exactly that sequence.

And (`a && b`) Both `a` and `b` must be true but the sequence does not matter.

Or (`a || b`) Either `a` or `b` must be true.

As seen in the example, the operands for the logical operators are allowed to be more complex expressions or they can be single predicates surrounded by `[` and `]`, for example `[hdr.ipv4.protocol == 17]`.

5.2.2 Windows

Windows are a construct for storing multiple header values of previous packets and then accessing them at a later point. Windows have two parameters, `size` and `value`. Multiple windows may exist in a single rule definition but the names need to be unique. Two examples of a window definition can be seen in Listing 5.1 in the lines 4 to 7 and 9 to 12.

`size` specifies how many values should be stored in the window. This has to be a constant value since a dynamic upper limit cannot be implemented due to the language restrictions of P4, which will be described in more detail in Section 6.2.2. `value` is header reference (for example `hdr.ipv4.totalLen`) which will be used to retrieve the value to store in this window when a packet arrives.

5.2.3 Variables

As described in Section 2.2, the P4 concept includes the ability to reconfigure switches at runtime with new P4 programs. However, loading a new P4 program may take some time in which the switch is unable to forward packets. To support data which needs to be changed frequently, P4, and OpenFlow before it, have support for dynamic forwarding tables which can be changed by the control plane at runtime without causing any service interruptions.

In the case of P4CEP, most of the rule definitions are represented as P4 code. For example, the event predicate `[hdr.ipv4.totalLen > 50]` appears in that form in the P4 program and cannot be changed without also changing and recompiling the P4 code. However, the state machines used for detecting complex events are implemented using standard forwarding tables which can be changed at runtime. For example, it is possible to change the pattern `[hdr.ipv4.totalLen > 50] && [hdr.ipv4.protocol == 17]` to `[hdr.ipv4.totalLen > 50] ; [hdr.ipv4.protocol == 17]` by manipulating the state transition table. The same is not true for the predicate `[hdr.ipv4.totalLen > 50]`. If the threshold of 50 needs to be changed to 100, the entire P4 program needs to be recompiled and the switch needs to be reconfigured with the new program. This situation can be improved by providing dynamic values which can be changed at runtime without reconfiguring the entire switch. This is supported by the variables feature in P4CEP.

An example of a variable declaration is shown at the start of Listing 5.1. A variable has a name and a default value which is assigned to it in the rules file. The default value will be used if it isn't changed later but apart from that has no special meaning for P4CEP. In a pattern, a variable can be referenced by prefixing the name with a `$`. In the example above, the predicate could be changed to `pattern [hdr.ipv4.totalLen > $len_threshold]` which would allow the control plane to later change the value of `$len_threshold`.

Variables are declared using the `var` keyword. The name can then be used as describe above like a normal number in the pattern definition of a complex event.

5.2.4 Functions

Complex event pattern need to access data stored in windows or some other information that is not readily available using the standard P4 syntax. For that reason P4CEP provides certain built-in functions that can be used to execute more complex operations that would be impossible using standard P4. Functions typically can be used in a free form and a window form. The free form does not operate on a window but instead uses a single accumulator register for keeping state from previous packets. Window functions can evaluate the values in a window and do complex computations with those values.

At the moment, free functions have no way of resetting their internal accumulators so there is no way of clearing the state of the function in a controlled way. Such a feature could be added with a new syntax construct but that is not included in the current version of P4CEP.

sum(*operand*)

The `sum` function provides a way for summing up all the values stored in a window or computing the sum of all previous values of a header field.

If used with a P4 header reference as *operand* (for example `sum(ipv4.totalLen)`) then P4CEP will generate code so that the previous sum of the header field is kept in a register and whenever a new packet arrives the new value is added to the sum and provided for the rest of the P4 code. There is no special handling of integer overflows so in case this happens it will use the standard P4 semantics.

If *operand* is the name of a window, then P4CEP will instead compute the sum of all values currently stored in that window.

min/max(***operand***)

Determining the minimum or maximum value of a header field or a window can be very useful for CEP applications in some circumstances like determining if a sensor value has crossed a specific threshold in the past.

P4CEP supports this use case with the `min` and `max` functions which can compute the minimum and maximum values of both a header field or a window.

Count

`count` is a different kind if function than the previously described functions. Instead of operating on a header field, it accepts a boolean condition and then counts how many times that condition has been true.

The free function (for example `count(ipv4.totalLen >= 50)`) simply checks the condition for every packet and increments an internal counter if the condition evaluates to true. The new value of the count can then be used in a complex event pattern.

When used in the form of a window function, the basic operation stays the same but instead of checking a generic condition it will now count how many times a condition is true for all the values in the window. Since P4 code cannot access the value of a specific window element directly, P4CEP provides a special syntax construct for accessing the value within the condition. The window function has the form `count(<window_name>, <condition>)`. Within `<condition>` the special variable `$value` is available which will hold the value of the window element that is currently being checked. For example `count(test_wnd, $value > 120)` will count how many elements of the window `test_wnd` have a value greater than 120.

5.3 Workflow

The general work flow when using P4CEP can be seen in Figure 5.1. This assumes that there already is an existing P4 program, that processes non-CEP related traffic. The CEP rules are written in the custom rule definition language described above and given to the P4CEP compiler. This step will generate four files. The existing and the generated P4 code are then combined by the standard P4 compiler and compiled to the target specific representation.

Header File The first file is a P4 header file which contains definitions for the rest of the P4 code. It contains a header struct which defines the structure of the CEP event detection header which is used when a complex event has been detected. It also contains a `struct` for meta data used by P4CEP which must be added to the global meta data struct used by the P4 program with the name `cep`. This struct contains fields for storing global data used by P4CEP in multiple pipeline stages.

The header also defines the two macros `CEP_INGRESS_HANDLING` and `CEP_EGRESS_HANDLING`. `CEP_INGRESS_HANDLING` needs to be added to the `apply` block of the ingress control block before any forwarding operations have been executed. This macro expands to code that will handle adding the CEP header when an event has been detected and it also contains a check for when a

CEP packet from another switch has been received. By default, these packets are ignored by the CEP code P4CEP generates. `CEP_EGRESS_HANDLING` needs to be placed in the egress control block and handles creating new packets which is required for sending the CEP detection packet. It is possible to do this without dropping the original packet by first cloning the packet and then recirculating it. If the original packet should be dropped it is recircled without cloning it first. When the recircled packet arrives at the ingress control again `CEP_INGRESS_HANDLING` will execute the proper actions for adding the CEP header and setting the correct destination address header values.

Declarations The second file generated by P4CEP contains *declarations* required for the CEP code. This file must be included in the ingress control block and defines things local to this block. This includes local variables, actions and tables used by P4CEP.

Code The third file contains the actual CEP code generated by P4CEP. It must be included in the `apply` block after all usual forwarding operations have been executed. This is the code that will actually do the CEP operations.

Runtime Control Plane Data All previously mentioned files were P4 files which were included and processed by the standard P4 compiler. The fourth file contains meta data collected by P4CEP while generating the P4 code. This data is encoded in the JSON format and contains a data structure which describes what tables were generated by P4CEP and how they need to be configured. Section 6.6 goes into more detail about what information this file contains. The control plane can use this data for configuring the tables used by the code generated by P4CEP.

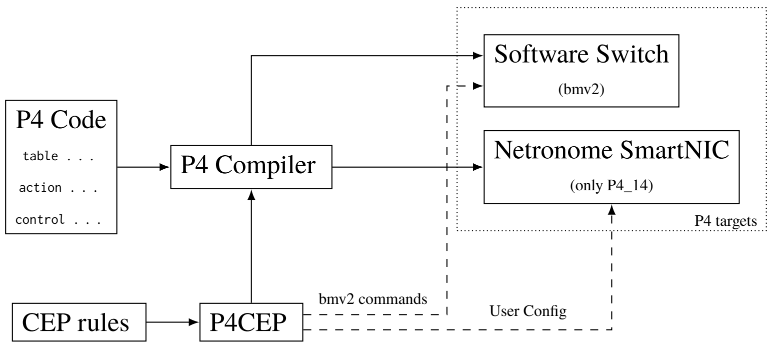


Figure 5.1: Workflow of integrating CEP rules into a P4 program using P4CEP. The existing P4 code is combined with the code generated by P4CEP from the CEP rules specified by the user. Additionally, it generates runtime control plane data for a specific target which is then used for dynamically configuring the tables and registers generated by P4CEP.

6 Implementation

Based on the event model and language design shown in the previous chapter a compiler was implemented which translated the P4CEP rule definition language to P4 code. This chapter will describe how exactly the rule files are translated to P4 code.

First a general overview of the compiler is given in Section 6.1 which will describe the general steps which are taken for translating from the rule definition language to P4. Section 6.2 will go into detail about how certain parts of the code generator work. In Section 6.2.1 the algorithm used for translating event patterns to finite state machines is described while Section 6.2.2 will detail how the remaining parts of the AST are transformed into P4 code. Section 6.3 describes certain parts of the implementation that affect the entire compiler. The limitations of the current compiler version are discussed in Section 6.4. Section 6.5 lists the supported target platforms and any describes target specific code. Finally, Section 6.6 will describe the format in which the runtime control plane data is written.

6.1 Overview

The data flow within P4CEP is shown in Figure 6.1. The implementation of P4CEP is separated into three steps. The first step (❶) parses the rule file given to the compiler to an internal Abstract Syntax Tree (AST) representation which is used by all other steps. The second step is generating the P4 code and collecting the control plane information. This step is split into two parts. The first part (❷) generates a finite state machine from the complex event pattern specified in the rules and the second step (❸) translates the AST into P4 code. The final step (❹) takes the data generated by the code generator and writes it in a generic format to a file so that this file can be used by the control plane for configuring the tables P4CEP uses. The project was implemented using C++.

Parser The parser uses the widely used Bison parser and Flex lexer. These are configured using custom grammar files written for P4CEP which build an AST while parsing the file. When the parsing code finishes successfully the result is an internal representation of the AST which is syntactically correct but has not been checked for semantic correctness.

Code generator The code generator then uses the AST and generates the corresponding P4 code from it. However, before any code can be generated the AST needs to be checked to make sure that it is semantically correct. This check includes a simple type check which ensures that the most obvious errors are found without running the P4 compiler. This step also checks if the variables used by the expressions in the tree are valid. For every complex event P4CEP also builds the state machine corresponding to the pattern that is used by the event. The exact procedure of how this is done will be explained in Section 6.2.1.

After doing preliminary semantic checks the AST is ready to be transformed to P4 code. Since P4CEP supports both $P4_{14}$ and $P4_{16}$ this step is implemented by two different classes, one for each language version. The differences that requires this distinction are listed in Section 6.3.2. How the individual components of the rule definition file are translated into P4 code will be described in Section 6.2.2.

While generating the code, the generator also populates an internal list of forwarding table definitions which will be used for generating the control plane data later. When a new table definition is generated depends on the generator target. The $P4_{16}$ generator only needs to generate the state lookup tables for the complex events and some ancillary tables which are required for rewriting packets or passing variable data to the P4 program.

Runtime Control Plane Data generator After the code has been generated the only thing left is the generation of the control plane data. This is done by examining the AST produced by the previous steps and the list of table definitions which the code generator built while generating the P4 code. This information is then encoded in a JSON file which can be parsed by the external control plane code. The format and contents of this file are described in Section 6.6.

6.2 Code Generator

The code generator is the central part of P4CEP which translates rule files to P4 source code. The details of this process will be described in the following section.

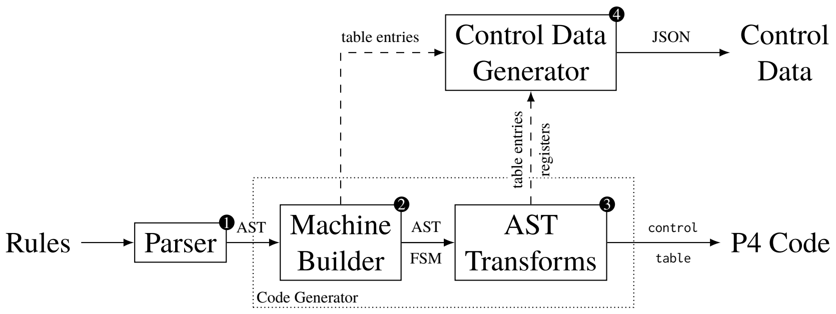


Figure 6.1: Flow of data and control within P4CEP. The rules are parsed by the Parser (❶) into an AST which is passed to the finite state machine builder (❷). This component constructs the FSM and passes that and the AST to the AST transformation step (❸). This step converts the AST into P4 code and writes the generated code to a file. Additionally, any required runtime control data is passed to the runtime control data generator (❹), which converts the data into JSON.

6.2.1 Building the state machine

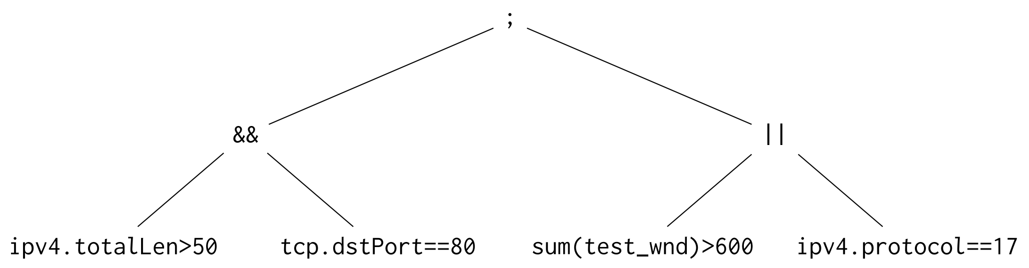
Before any code can be generated for the complex event detection, P4CEP first needs to process the patterns specified in the complex event definitions and convert them to conventional state machines. The formal definition of the state machines is the standard quintuple $(\Sigma, S, s_o, \Delta, F)$ where Σ is the alphabet of inputs accepted by the machine, S is the set of states in the machine, s_0 is the starting state, Δ is a state transitioning function $S \times \Sigma \rightarrow S$ and F is the set of final states.

The machine generated by P4CEP has only one final state which also immediately resets the machine back to the starting state s_0. The input alphabet is the set of expressions (enclosed in `[` and `]`) used by the complex event pattern.

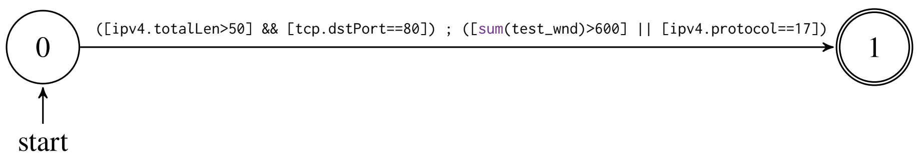
The pattern will be converted to a state machine using the algorithm shown in Algorithm 6.1. The individual steps taken by the algorithm will be explained on the basis of the following complex event example:

```
complex_event test_evt {
    value sum(hdr.ipv4.totalLen)
    pattern
    ([ipv4.totalLen>50] && [tcp.dstPort==80]) ; ([sum(test_wnd)>600] || [ipv4.protocol==17])
}
```

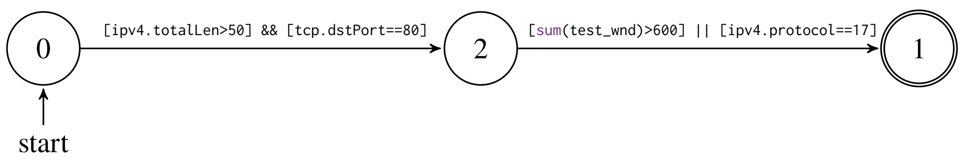
The `build` function is called with the pattern expression (`([ipv4.totalLen>50] && [tcp.dstPort==80]) ; ([sum(test_wnd)>600] || [ipv4.protocol==17])`) as its parameter. The syntax tree of the binary operator expressions is structured like this:



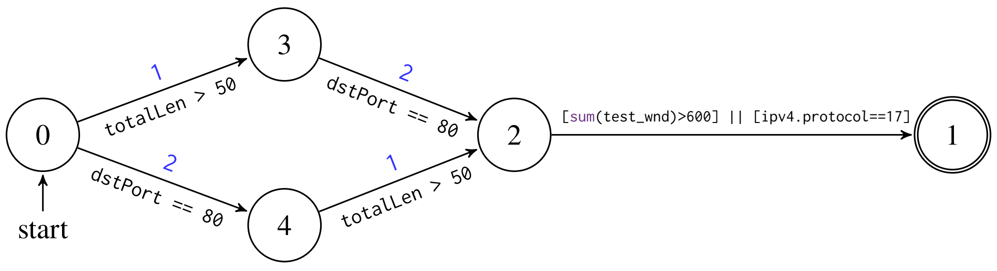
When `build` of Algorithm 6.1 is called with this expression it will first construct the initial state and the end state before calling `convert`. Conceptually, the state machine has two states and a single transition between them which contains the entire pattern expression:


0
([ipv4.totalLen>50] && [tcp.dstPort==80]) ; ([sum(test_wnd)>600] || [ipv4.protocol==17])
1
start

The first `convert` call splits the state machine since the top-level expression is a sequence operator (;):


0
[ipv4.totalLen>50] && [tcp.dstPort==80]
2
[sum(test_wnd)>600] || [ipv4.protocol==17]
1
start

Since the left part of the expression (`[ipv4.totalLen>50] && [tcp.dstPort==80]`) is processed first, the `&&` branch in `convert` is executed which results in the following intermediary state machine:



The final step is converting the right part of the expression (`[sum(test_wnd)>600] || [ipv4.protocol==17]`) which takes the *Or* path in the `convert` function:

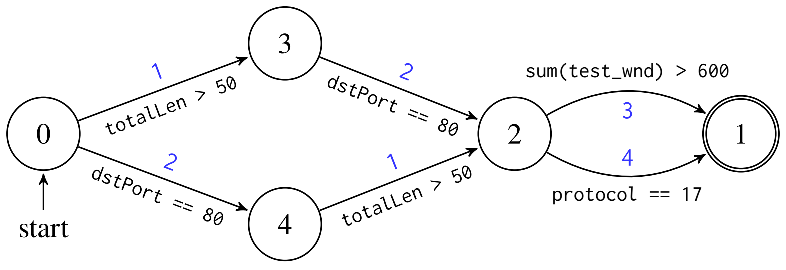


Figure 6.2: Finished state machine generated from a complex event pattern.

After the algorithm is finished, P4CEP has a correct state machine definition in memory which can then be transformed into P4 code and table entries. Since P4 table lookups only operate on integers instead of generic expressions, P4CEP maps the precondition expressions to unique integer identifiers which can be used by the table lookup functions provided by P4. Since the same predicates may appear multiple times in the state machine (for example `totalLen > 50` in the state machine above), P4CEP needs to ensure that the same predicate is always assigned the same identifier. Otherwise, a predicate may appear twice in the input alphabet with different integer identifiers but only one of them will be chosen when mapping the P4 expressions to their integer identifiers which would lead consistency issues. To avoid this, P4CEP compares the AST of all expressions added as transitions via `addTransition` and reuses an existing identifier if the same expression has already been used. In the diagrams above these identifiers have been added opposite to the predicates in blue. They are assigned in ascending order but have otherwise no special meaning. The expressions which appear multiple times in the machine are assigned the same identifiers. Based on this information, P4CEP can generate the control plane data for the table which can be seen in Table 6.1.

State	Transition	Next State
0	1	3
0	2	4
2	3	1
2	4	1
3	2	2
4	1	2

Table 6.1: Lookup table for the example pattern shown in Section 6.2.1. **State** will be the state the machine is currently in. **Transition** is the integer encoding of the predicate for that transition. **Next State** is the value returned by the table which indicates what the next state is based on the state and transition. The state and transition values correspond to the values given in Figure 6.2.

Algorithm 6.1 Algorithm used for converting complex event patterns to state machines. `createState` creates a new state and adds it to the machine. `createEndState` does the same but also marks it as an end state. `addTransition(expression, start, end)` adds the given predicate as a transition from state `start` to `end`.

```
function CONVERT(expression, start, end)
    if expression is binary then
        if expression is sequence then        // Both expressions appear in the specified sequence
            mid ← CREATESTATE
            CONVERT(expression.left, start, mid)
            CONVERT(expression.right, mid, end)
        else if expression is and then // Both expressions must appear, sequence is not important
            stateA ← CREATESTATE
            CONVERT(expression.left, start, stateA)
            CONVERT(expression.right, stateA, end)
            stateB ← CREATESTATE
            CONVERT(expression.right, start, stateB)
            CONVERT(expression.left, stateB, end)
        else if expression is or then                    // One of the expressions must appear
            CONVERT(expression.left, start, end)
            CONVERT(expression.right, start, end)
        end if
    else                                  // Expression is an expression with a single predicate
        ADDTRANSITION(expression, start, end)
    end if
end function
function BUILD(patternExpr)
    start ← CREATESTATE
    end ← CREATEENDSTATE
    CONVERT(patternExpr, start, end)
end function
```

6.2.2 AST transformations

Some syntax constructs available in P4CEP pattern expressions are not valid P4 source code since they express stateful CEP operators. To make these expressions and syntax elements usable in a P4 environment they need to be transformed so that they conform to the P4 syntax and the P4 semantics reflect what the rules intend to detect. There is a wide array of transformations that need to happen before the rules become usable in P4. The semantics of how these syntax elements should work have been specified in Section 5.2. This section will concentrate on how these elements are transformed into P4 code.

Free Functions

Free functions (for example `sum(hdr.ipv4.totalLen)`) are typically accumulator functions which do not need more than one slot of memory. Free functions do not require any information from the other parts of the P4 code but can be used in all other rule elements which means that they are required to be evaluated first.

All free functions work very similarly. Before any code is generated, the function is assigned a unique variable name which is used for declaring a local 32-bit unsigned integer variable. Then, at the beginning of the code generated by P4CEP, for every free function call, the current value is read from a register slot. With this value and the expression given as the parameter of the function call the new value is computed as described below and written back to the register. The newly computed value remains in the local variable used by this function which allows the rest of the code to use this variable whenever the value of the function call is needed.

sum(*expression*) The `sum` function has the simplest implementation. It simply takes the current function value, adds the value of the expression to it by using this exact expression as the right side of the operation and writes the result back to the function variable. Since header values can have an arbitrary bit-width the value first needs to be cast to the correct bit type to avoid P4 compilation errors. The following is an example for the P4 code that may be generated by the function call `sum(hdr.ipv4.totalLen)`:

```
@atomic {
    cep_memory.read(cep_function_2_value, 15);
    cep_function_2_value = cep_function_2_value + (bit<32>)hdr.ipv4.totalLen;
    cep_memory.write(15, cep_function_2_value);
}
```

min/max(*expression*) The minimum and maximum functions compute the new value by comparing the current value of the expression to the value stored in the register. If the value is less than for the `min` function or greater than for the `max` function, the current value of *expression* is assigned to the function variable and written to the function register. This code was generated by the function call `max(hdr.ipv4.totalLen)`:

```
@atomic {
    cep_memory.read(cep_function_2_value, 15);
    if ((bit<32>)hdr.ipv4.totalLen > cep_function_2_value) {
        cep_function_2_value = (bit<32>)hdr.ipv4.totalLen;
    }
    cep_memory.write(15, cep_function_2_value);
}
```

count(*expression*) The `count` function is different from the functions described so far. Instead of accepting an expression which contains a simple header reference, this function needs a P4 condition as its parameter. When the condition is true then it increments the value of the function variable and if it is not, no action is taken. The following was generated from the P4CEP expression `count(hdr.ipv4.totalLen >= 120)`:

```
1  @atomic {
2      cep_memory.read(cep_function_2_value, 15);
3      if ((hdr.ipv4.totalLen >= 120)) {
4          cep_function_2_value = cep_function_2_value + 1;
5      }
6      cep_memory.write(15, cep_function_2_value);
7  }
```

The inner expression `hdr.ipv4.totalLen >= 120` is clearly visible in line 3. P4CEP always encloses all expressions in parentheses to avoid problems with differences in the operator precedence of P4 and P4CEP.

Window updates

Window contents are managed using a ringbuffer system. This minimizes the amount of register operations needed for adding a new element to the window. This means that a window uses $n + 1$ register slots when n is the size of the window. The first register element contains the register index of the last element that was added to the window. When a new element is added to the window the code first reads the current head value from the assigned register location. It then increments that value and checks if the new value points outside the assigned register range. If it does, it resets the value back to the first register element:

```
cep_memory.read(cep_head_value, 0);
cep_head_value = cep_head_value + 1;
if (cep_head_value >= 6) {
    cep_head_value = 1;
}
```

This could be done in other languages using the modulo operation but that operator is not available in $P4_{16}$. The `cep_head_value` variable now contains the register index where the new item should be written. The new index is also the value which should be written to the register used for storing the ring buffer head index. Writing the new value involves two register operations. The first to write the new value to the window and the second for updating the head register:

```
cep_memory.write(cep_head_value, (bit<32>) hdr.ipv4.totalLen);
cep_memory.write(0, cep_head_value);
```

At this point the window is fully updated. However, having the values in multiple registers isn't particularly useful since the rule expressions can not access those values directly. For this reason, P4CEP provides functions that targets windows which will be evaluated when a window is updated.

Before the previous window values can be evaluated, P4CEP needs to set the initial values of the variables which are assigned to the window functions. This initial values are chosen so that they reflect the state of the function as if it was executed with only the newest value of the window. For the `sum`, `min` and `max` functions this means that the initial value is simply the newest window value since the sum of one window element is exactly the value of that window element. The same holds for `min` and `max` where the newest value is always both the minimum and maximum value if there are no other values present:

```
cep_function_2_value = (bit<32>)hdr.ipv4.totalLen; // sum(test_wnd)
cep_function_3_value = (bit<32>)hdr.ipv4.totalLen; // min(test_wnd)
cep_function_4_value = (bit<32>)hdr.ipv4.totalLen; // max(test_wnd)
```

`count` functions look differently but are fundamentally the same since the initialization simply checks if the condition is true or not and initializes the counter accordingly:

```
if (((bit<32>)hdr.ipv4.totalLen > 10)) { // count(test_wnd, $value > 10)
    cep_function_5_value = 1;
} else {
    cep_function_5_value = 0;
}
```

After the window functions are initialized, the values stored in the window registers can be evaluated. To do that, the previous ring buffer head value is reused since that already points to the position where the newest value was added to the window. The head index must be incremented to point to the next window element. Like previously, the new value needs to be checked for an overflow to avoid reading outside the window bounds. Once the correct head value has been computed the window value from that position can be read into a local variable:

```
cep_head_value = cep_head_value + 1;
if (cep_head_value >= 6) {
    cep_head_value = 1;
}
cep_memory.read(cep_tmp, cep_head_value);
```

Here `cep_tmp` is the local variable which contains the value of the window at the current position. It will be used wherever P4CEP needs to use the window value. For example, in `count` function expressions `$value` will be replaced with `cep_tmp` in the generated P4 code. The code that is actually generated is very similar to the code shown in Section 6.2.2:

```
cep_function_2_value = cep_function_2_value + cep_tmp; // sum(test_wnd)
if (cep_tmp < cep_function_3_value) { // min(test_wnd)
    cep_function_3_value = cep_tmp;
}
if (cep_tmp > cep_function_4_value) { // max(test_wnd)
    cep_function_4_value = cep_tmp;
}
if ((cep_tmp > 10)) { // count(test_wnd, $value > 10)
    cep_function_5_value = cep_function_5_value + 1;
}
```

This window evaluation code needs to be executed $n - 1$ (when n is the size of the window) times to process all window elements since the first element was already processed in the initialization step. Normally this would be done with a simple `for` loop but since that is not available in $P4_{16}$, P4CEP needs to unroll the loop which means that it is repeated $n - 1$ times without changes to the code.

Updating the window and reading the values needs to happen atomically since otherwise there would be lost updates or dirty reads when the access to the window registers is not synchronized properly. In the $P4_{16}$ standard there is support for this by using the `@atomic` annotation which is used for all window operations.

State machine input mapping

As described in Section 6.2.1, the state machine transitions are mapped by the state machine builder to integer identifiers. However, P4 still needs to map the success of a predicate check to the corresponding integer identifier. This is done in a single `if – else if` block before applying the state lookup table. The following code was generated from the example pattern used in Section 6.2.1:

```
if ((hdr.ipv4.totalLen > 50)) {
    meta.cep_data.transition_input = 1;
} else if ((hdr.tcp.dstPort == 80)) {
    meta.cep_data.transition_input = 2;
} else if ((cep_function_2_value > 600)) { // cep_function_2_value is the variable of sum(test_wnd)
    meta.cep_data.transition_input = 3;
} else if ((hdr.ipv4.protocol == 17)) {
    meta.cep_data.transition_input = 4;
} else {
    meta.cep_data.transition_input = 0xFFFFFFFF;
}
```

`meta.cep_data.transition_input` is later used as a lookup value for the state machine table. If none of the predicates match then the special value `0xFFFFFFFF` is used which will not be used as the identifier of any transition. This makes sure that the table lookup will always result in a miss which will trigger the default action of the lookup.

State machine implementation

The integer identification of the next transition is now available in `meta.cep_data.transition_input`. In order to do the correct lookup in the state transition table the current state of the machine is needed. Since all state machine operations need to happen atomically or else an inconsistent state may be created, the state machine code is enclosed in an `@atomic` block.

The basic idea behind the implementation of the state machines is based on OpenState [BBCC14] and its P4 implementation [San]. However, since P4CEP does not use flow specific state machines it is not necessary to use a second table for storing the mapping from the flow to the current register state.

The first action inside the atomic block is reading the current state of the machine from the assigned register:

```
cep_memory.read(meta.cep_data.state, 0);
```

This completes the final step required for the state table lookup which can now be executed:

```
cep_test3_evt_state_lookup_0.apply();
```

`cep_test3_evt_state_lookup_0` is defined in the P4 declarations as follows:

```
table cep_test3_evt_state_lookup_0 {
  key = {
    meta.cep_data.state : exact;
    meta.cep_data.transition_input : exact;
  }
  actions = {
    cep_state_advance;
    NoAction;
  }
  size = 6;
  default_action = NoAction();
}
```

The table has two possible actions. `NoAction` is used as the default action when there is no match in the table. This implements the `skip` strategy of a complex event definition which simply skips over unrelated events since no values are changed by that function which leaves the state machine in the same state as it was before the table lookup. `cep_state_advance` is the function used for when there is a match in the lookup table. The definition of this function can be seen in Listing 6.1. This action receives as its inputs the next state and a boolean bit that indicates whether the next state is an end state in the finite state machine. The received values are written to metadata fields from where they can be processed outside the action.

The processing code for this data can be seen in Listing 6.3. This code first checks if the boolean flag `is_end_state` has been set. This indicates that the final state of the state machine has been reached which means that an event has been detected. If that is the case, the machine is reset to the starting state (which is represented by the integer 0) and the values for sending the complex event detection packet are written. `event_id` specifies which event has been detected by assigning it the integer identification P4CEP chose for this event. The runtime control plane data written by P4CEP contains a mapping for these identifications which allows external applications to map this number back to the original event name. `event_value` is the field which will hold the value specified by the `value` option given in the `complex_event` definition. This value will then be included in the event detection packet later. `event_detected` is a boolean flag that will be used by other parts of the code for determining if an event has been detected by any of the pattern matchers.

If the complex event used the strategy `strict` instead of `skip` then the only thing that changes is the default action of the table. Instead of using `NoAction`, which does nothing, it uses `cep_state_reset`. The definition of this action can be seen in Listing 6.2. It sets the state to 0 which means that the processing code will reset the state machine to the starting state if an event occurs that was not expected by the pattern.

This concludes the complex event detection. Now, the metadata contains a boolean flag which indicates if a complex event was detected. The `egress` code can use this data for triggering the code which rewrites the packet with the event information and sends the packet towards the CEP host which will process this notification.

6.2.3 Multiple complex event instances

If a complex event uses multiple instances, some aspects of the code shown in Section 6.2.2 are changed to implement the different behavior. The semantics of multiple event instances, as described in Section 5.2.1, is that an event that moves a state machine from its initial state should be consumed by that operation and not processed by other instances. In all other circumstances the event should

Listing 6.1 Definition of `cep_state_advance`.

```
action cep_state_advance(bit<32> next_state, bit<1> is_end_state) {
    meta.cep_data.state = next_state;
    meta.cep_data.is_end_state = is_end_state;
    meta.cep_data.event_consumed = 1; // only present if there is an event with multiple instances
}
```

Listing 6.2 Definition of `cep_state_reset`.

```
action cep_state_reset() {
  meta.cep_data.state = 0;
  meta.cep_data.is_end_state = 0;
}
```

Listing 6.3 This P4 code is called after a state table lookup for processing the data returned by the table lookup. An end state causes the machine to be reset to the starting state and the values for the event detection are written to the corresponding metadata fields. If the state was not an end state then the new state is simply written back to memory.

```
if (meta.cep_data.is_end_state == 1) {
    cep_memory.write(0, 0);
    meta.cep_data.event_id = 0;
    meta.cep_data.event_value = cep_function_6_value;
    meta.cep_data.event_detected = 1;
} else {
    cep_memory.write(0, meta.cep_data.state);
}
```

be processed by all instances. In order to be able to make this decision the P4 code needs additional information. It needs to know the state the machine instance was in before the lookup table was applied and it needs to know if an event was consumed.

The first information can be retrieved by saving the state before the lookup table was used in a local variable. The second bit of information can be determined by adding another local variable which indicates whether the event that is currently being looked at was consumed by the lookup table action. This variable is initialized to `false` and is set to `true` (or 1 since P4 has no boolean variables) when an event triggers a state advance operation which can be seen in Listing 6.1, line 4. The changed initialization code can be seen in Listing 6.4.

The first instance of the state machine is executed unconditionally since there is no previous state machine which would have initialized the local variables properly. All other instances are surrounded by an `if` block with the condition `cep_original_state != 0 || meta.cep_data.event_consumed == 0`. This expression is the logical negation of the check if the original state was the starting state of the state machine and if the event was consumed which means that a new state was reached by the lookup table action. The full code for one such event instance can be seen in Listing 6.5. This code is repeated $n - 1$ times if the instance count of the event is n since the first event was already handled without the surrounding `if` block.

Since P4 code is commonly represented as a DAG of table applications, using the same table for all state lookups of the individual machine instances would create a cycle in the table graph of the program. To fix this, P4CEP uses multiple table instances with the same contents but different suffixes to avoid creating cycles in the program graph. This is not optimal but necessary due to the limitations of the P4 language.

Listing 6.4 Initialization code for multiple event instances.

```
cep_memory.read(meta.cep_data.state, 12);
cep_original_state = meta.cep_data.state;
meta.cep_data.event_consumed = 0;
```

Listing 6.5 Full code example of an state machine instance evaluation.

```
if (cep_original_state != 0 || meta.cep_data.event_consumed == 0) {
    @atomic {
        cep_memory.read(meta.cep_data.state, 13);
        cep_original_state = meta.cep_data.state;
        meta.cep_data.event_consumed = 0;
        cep_test_evt_state_lookup_1.apply();
        if (meta.cep_data.is_end_state == 1) {
            cep_memory.write(13, 0);
            meta.cep_data.event_id = 0;
            meta.cep_data.event_value = cep_function_0_value;
            meta.cep_data.event_detected = 1;
        } else {
            cep_memory.write(13, meta.cep_data.state);
        }
    }
}
```

6.3 Implementation Details

Some details of the implementation which were omitted in the previous sections are noteworthy and will be described in the following section.

6.3.1 Registers

Most of the features offered by P4CEP need to keep some state between packet arrivals. For example, the state machine needs to know what state is currently is in and when a state change has occurred the new state needs to be saved somewhere.

To implement this, P4CEP uses the register feature provided by the standard P4 language. All required registers are allocated in a single block and all values are stored as 32-bit unsigned integers. When preparing to generate the P4 code, P4CEP examines all syntax construct which need memory for storing their state (for example windows, functions or complex events). The syntax construct is then assigned a register range which will be reserved for that element. A range of registers is needed since some constructs (for example windows) need more than one element of memory. If only one register is needed (for example for free functions) then a range with size 1 is used. Register ranges are represented by two integers which point to the first register index belonging to that syntax element and a size which specifies how many elements from that point on belong to this register range. When P4CEP then generates the P4 code it can insert the register ranges as integer constants which will make it easier for the compiler to optimize the generated code.

6.3.2 Differences between $P4_{16}$ and $P4_{14}$

There are some major differences between $P4_{16}$ and $P4_{14}$ which impact how P4CEP has to generate code for the two targets. This section will list what differences impacted the code generation and how these differences were worked around to allow P4CEP to generate code for $P4_{14}$.

No code in control flow blocks

A major difference between P4$_{16}$ and P4$_{14}$ is that P4$_{16}$ allows code to appear in the body of the `apply` part of a `control` statement without being wrapped in an action. That is not allowed by P4$_{14}$ where all code must be within an `action` and be called by applying a table. This is a problem for P4CEP since the generated code often does not need a table match and was therefore written directly into the `apply` part of the P4$_{16}$ `control` statement. Since P4$_{14}$ does not allow that, P4CEP needs to make some adjustments to how the code is generated to still support the P4$_{14}$ target. Since P4$_{14}$ supports default actions for tables this mechanism was used for executing actions without actually doing a table match. When P4CEP needs to generate code it will put that code into the body of an action, generate a table with this new action as the default action and then it will insert an `apply` call where the original code should have been. The following code might be generated if the lines 2 to 4 would have appeared in the P4$_{16}$ source code:

```
1  action cep_tmp_action_48() {
2      register_write(cep_memory, 13, 0);
3      modify_field(cep_data.event_id, 1);
4      modify_field(cep_data.event_value, cep_locals.cep_function_2_value);
5  }
6  table cep_tmp_action_48_call_table {
7      actions {
8          cep_tmp_action_48;
9      }
10 }
```

At the point where the original code would have appeared only a singly `apply` would appear:

```
apply(cep_tmp_action_48_call_table);
```

P4CEP will then make sure that the table is configured properly via the runtime control plane data output so that `cep_tmp_action_48` will be set as the default action of `cep_tmp_action_48_call_table`. Another problem of this approach is that control flow operations such as `if` and `else` may not appear in action code which means that any control flow structures will require breaking up an action block. P4CEP will group as many operations as possible together into one action to minimize the amount of generated action blocks. Since P4CEP needs to make a lot of decisions with `if` statements this optimization can not be used in many circumstances but where possible it reduces the amount of generated tables.

Different boolean operator names

Between P4$_{16}$ and P4$_{14}$ there was a change to how boolean operations were called in the P4 language. In P4$_{14}$ `and` and `or` were used for the conjunction and disjunction respectively while P4$_{16}$ uses the character sequences `&&` and `||` which are more common in C like languages. P4CEP supports this difference by always requiring the P4$_{16}$ syntax in the rule files but then generating the P4$_{14}$ operators when needed. This is possible since P4CEP parses the used P4 expressions into an AST instead of copying the code without changes to the output file.

No binary operators in $P4_{14}$

In $P4_{14}$ most operations and computations use syntax constructs which are similar to function calls. For example, setting the value of a header field is done with `modify_field`: `modify_field(ipv4.totalLen, 20)`. $P4_{16}$ uses the more common syntax found in C like languages where a `=` is used for assignments and mathematical operations such as addition use the correct characters instead of function calls. This is not a problem for P4CEP since it simply uses the function calls instead of the $P4_{16}$ syntax when in $P4_{14}$ code generation mode.

No local variables

$P4_{16}$ has support for local variables which only exist within a specific `control` block (for example `ingress` or `egress`). Since the scope of these variables is limited to one control block, the P4 compiler can use this information to better optimize the code. Metadata headers are used for passing data between different stages of the P4 program. However, that is not needed for most data P4CEP uses so it uses local variables where possible to allow more optimization opportunities for the P4 compiler.

$P4_{14}$ does not have this feature but it still supports metadata headers since they were part of the original P4 specification [BDG+14]. P4CEP uses these metadata headers for storing information that would have been stored using local variables in $P4_{16}$.

No custom deparser functionality

P4CEP needs to add new headers to packets when it wants to send an event notification. In $P4_{16}$ this can be done by using `.setValid()` on the header instance and then adding that header to the deparser control block used by the P4 program:

```
1  control deparser(packet_out packet, in headers hdr) {
2      apply {
3          packet.emit(hdr.ethernet);
4          packet.emit(hdr.ipv4);
5          packet.emit(hdr.udp);
6          packet.emit(hdr.cep);
7      }
8  }
```

This code block instructs the switch how it should serialize the parsed header values before sending the packet out of the switch. The `emit` function is defined so that it does not emit a header if it is not valid which allows the program to include all possible headers and not worry about sending invalid headers in case a header was not parsed. Line 6 in the source code above contains the `emit` call needed by P4CEP which is fully supported by $P4_{16}$.

However, $P4_{14}$ did not support this functionality yet. This language version needs to know the sequence in which headers are serialized from the parse graph that was constructed by the parser code. Since the CEP header never needs to be parsed by the switch this is a problem since then P4 will never deparse the CEP header which makes sending complex event notifications impossible.

Fortunately, P4 does not need the header to be actually parsed but instead it simply needs it to be in the parser graph [Bas16]. By using a condition in the parser which is always false the CEP header can be inserted into the graph without actually parsing it:

```
parser parse_udp {
    extract(udp);
    return select(current(0, 64)) {
        0 : parse_cep_header;
        default: ingress;
    }
}
```

In this code the `parse_udp` state selects the next parsing state based on the next 64 bytes of the packet currently being parsed. If they are all 0 then the cep header will be parsed, otherwise the parsing will be finished and the ingress pipeline will be executed next. Since the probability that 64 bytes after the UDP header are all 0 is very small, it is almost certain that the CEP header will not be parsed but it will still be inserted into the parse graph at the correct position. This circumvents the deparser limitation if P4$_{14}$ and allows the language target to be fully supported by P4CEP.

6.4 Limitations

While designing and developing P4CEP some issues were discovered that could not be solved easily. Those problems will be discussed here.

6.4.1 Limited target platform capabilities

The biggest problem that was encountered during the development of P4CEP were the limited language and target capabilities. Since P4 is a language designed for hardware which uses match-action tables for executing code, most of the code generated by P4CEP is not ideal for these targets. A large portion of the code simply executes code without using the *match* semantics of the *match-action* model. This code can still be expressed using *match-action* semantics but this is not an efficient representation of this code. Due to this limitation the usefulness of P4CEP depends largely on how well the target can cope with the style of code generated by P4CEP. Ideally, the switch hardware would continue to incorporate more features already found in general-purpose CPUs.

6.4.2 Atomicity

Most features provided by P4CEP involve reading and writing to P4 registers. Since most switch architectures use some sort of multi threading or pipelining techniques, concurrent execution can cause inconsistency issues if two threads manipulate the same memory locations. Normal operating systems provide features such as mutexes or semaphores for avoiding these issues. However, typical SDN hardware may not have support for such operations which can cause consistency issues in the code generated by P4CEP.

The P4$_{16}$ language version provides the `@atomic` feature which can be used for instructing the compiler that this code block should only be executed atomically. P4CEP emits this annotation when in P4$_{16}$ mode but P4$_{14}$ does not have native support for such a functionality.

Using this language feature will decrease the performance on targets which make use of concurrent packet processing but this is unavoidable since consistent results are required for any CEP application.

6.5 Target Platforms

P4CEP supports both the $P4_{16}$ and $P4_{14}$ language versions but in order to actually execute the code generated by P4CEP it needs to be tested on actual P4 target hardware. This section will discuss the targets used for developing and evaluating P4CEP.

6.5.1 Netronome SmartNIC

The Netronome SmartNIC [Net] is a programmable Network interface controller (NIC) platform. It uses a custom processor design called the Netronome Flow Processors (NFPs) which are based on a RISC architecture. They are fully Turing complete which means that the platform is more powerful than the P4 language.

It is possible to program this NIC with a pure P4 program which is first compiled to C and then to microcode which can be executed on the NFP. The P4 SDK also provides the capability of adding custom actions which can be called from P4 and are implemented in C. The C implementation of these actions can use the full capabilities of the C language which includes constructs not available in P4 code such as dynamic loops. As described in Section 6.4.2, atomicity of various operations done by P4CEP needs to be ensured for a consistent program behavior. The $P4_{14}$ langauge version does not provide any feature that would allow to implement this. However, using the external C function feature provided by the NFP P4 SDK it is possible to implement this functionality in an external C function. The code for this is provided in Listing A.2. It uses a simple spin lock algorithm which makes use of the atomic memory access feature provided by the NFP C API. When enabled, P4CEP will insert this function call wherever a critical region is required. Since the C code is not changed by P4CEP, the section which needs to be locked is provided to the function via a metadata field since parameters are not supported for primitive functions.

The Netronome P4 SDK supports some additional annotations on register and action objects in the P4 source code. In order to provide the best support possible for this target, P4CEP can be configured to add these annotations to the generated source code where necessary. When enabled, all registers are annotated with the `@pragma netro reglocked cep_memory` pragma which tells the Netronome P4 compiler that all access to this register should be locked to reduce concurrency issues. This does not solve the atomicity problems already described in Section 6.4.2 since P4CEP generated register read and write operations which need to be executed atomically which is not what this annotation guarantees.

Furthermore, when generating an action which uses a register operation (`register_read` or `register_write`) P4CEP will add the `@pragma netro no_lookup_caching` annotation to this action. This prevents the NFP from caching the register contents since this behavior would violate the consistency requirements of P4CEP.

6.5.2 Behavioral Model Version 2 (bmv2) Software Switch

The Behaviorial Model Version 2 (abbreviated as bmv2) [p4la] is a P4 software switch designed for the development and debugging of P4 programs. Due to this focus on P4 program testing it is not a high performance software switch such as Open vSwitch [Ope].

When running bmv2, bypasses most of the networking code provided by the operating system and uses raw sockets instead. This allows Behavioral Model Version 2 (bmv2) to execute the parsing steps specified by the P4 program itself.

When compiled for the bmv2 target, a P4 program is represented as a JSON file which contains the header and parser definitions of the program. An action is represented as a sequence of primitive actions. For example the statement `modify_field(ipv4.protocol, 17)` will be compiled to the JSON shown in Listing 6.6. These instructions will be executed in sequence similarly to how a CPU would execute a normal program. bmv2 has support for jump instructions and even conditional backwards jumps within an action [p4lb]. This makes it theoretically more powerful than the P4 language but this capability is not exposed in the P4 language. A P4 program for this target is commonly compiled by p4c [p4lc] which can process both $P4_{16}$ and $P4_{14}$ source code. This target has not special support for implementing special actions outside the P4 program similar to the primitive actions supported by the NFP.

The ingress and egress pipelines of the P4 program are executed in separate threads. However, while the egress pipeline can be run in parallel, that is not done for the ingress pipeline[1]. This reduces the scalability of this software switch and is especially noticeable when testing the maximum bandwidth this implementation can support (see Section 7.3.1).

Listing 6.6 bmv2 JSON format equivalent of the expression `modify_field(ipv4.protocol, 17)`.

```
{
    "op" : "assign",
    "parameters" : [
        {
            "type" : "field",
            "value" : ["ipv4", "protocol"]
        },
        {
            "type" : "hexstr",
            "value" : "0x11"
        }
    ]
}
```

[1] https://git.io/vpPe6

6.5.3 PISCES

PISCES [SCP+16] is a prototype compiler for translating P4$_{14}$ code to C code which is then included in a changed version of Open vSwitch [Ope] to create a high performance software switch which processes packets according to the P4 source code. It supports the P4 parser construct for specifying how packet headers should be parsed.

This is done by replacing the traditional code Open vSwitch uses for parsing the packets and the *match-action* pipeline, with the C code generated by the PISCES P4 compiler. It also provides additional support for flow rules for configuring the forwarding tables. However, the format used for this is completely undocumented which makes it hard to write configuration files for custom P4 source code. An example of such a flow rule is provided in Listing 6.7 and the corresponding P4 code is shown in Listing 6.8. The specified flow rule has some resemblance with the P4 code. However, writing such a rule for a custom P4 program without any documentation would be time-consuming and error prone.

The PISCES P4 software switch was considered as a target for P4CEP. However, since there is no documentation on how the needed control plane data could be converted from the format P4CEP outputs to the format PISCES accepts, it was not possible to test the generated code with this target. PISCES supports the P4$_{14}$ language version so the code generated by P4CEP should be compatible with this project.

6.6 Runtime Control Plane Data

While generating the P4 code required for detecting complex events, P4CEP will accumulate data which will be required by the control plane later for properly using the added P4 code. For example, the state lookup table shown in Section 6.2.1 would need to be sent to the switch through the control plane. For this reason, P4CEP will write a JSON encoded structure of all the data required for using the generated P4 code. This includes initial register contents, table contents and complex event identifier mapping information.

Tables are stored in the `"tables"` array where each array element contains the information for a single table. Table objects have a `"name"` key which contains the name the table has in the P4 program. The entries of the table are specified with the `"entries"` array where every element is a single entry in the table. An example of a state lookup entry can be seen in Listing 6.9. This object specifies which action to execute and what parameters should be passed to this action. It also specifies the keys which should be used for matching the input data. `"default_entry"` uses the same format without specifying keys since that concept does not apply to the default action of a table.

Listing 6.7 Open vSwitch flow rule specification for an example program using a table for longest prefix matching if an IPv4 address.

```
$DIR/ovs-ofctl --protocols=OpenFlow15 add-flow br0
                            "table=1,priority=32768,ipv4__dstAddr=0x0A00000F/0xFFFFFF00 \
                             actions=set_field:0x0B00000F->routing_metadata_nhop_ipv4, \
                                     set_field:2->reg0, \
                                     set_field:63->ipv4__ttl, \
                                     resubmit(,2)"
```

Listing 6.8 P4 action code for IPv4 forwarding used by an example of PISCES. This is the P4 code which corresponds to the flow rule specified in Listing 6.7.

```
action set_nhop(nhop_ipv4, port) {
    modify_field(routing_metadata.nhop_ipv4, nhop_ipv4);
    modify_field(standard_metadata.egress_spec, port);
    add_to_field(ipv4_.ttl, -1);
}
```

Listing 6.9 Single table entry encoded in JSON.

```
{
  "action": "cep_state_advance",
  "keys": [
    {
      "name": "cep_data.state",
      "value": 0
    },
    {
      "name": "cep_data.transition_input",
      "value": 1
    }
  ],
  "parameters": [
    {
      "name": "next_state",
      "value": 3
    },
    {
      "name": "is_end_state",
      "value": false
    }
  ]
}
```

The control plane can parse this information easily since JSON is a widely supported format. P4CEP provides Python scripts for doing this conversion for the bmv2 and NFP targets. The bmv2 conversion script reads the JSON metadata and outputs a sequence of commands which can be passed to the command line interface of `simple_switch`. The NFP conversion script will output a User-Config suitable for usage with the P4 SDK provided for the NFP.

7 Evaluation

Once the implementation of the design was complete the project was evaluated for latency and bandwidth. This chapter will describe how the performance was measured and will show and discuss the results.

The comparison has been done with Apache Flink which was be used to measure the event detection latency and compare it with the latency of the code generated by P4CEP. Furthermore, the latency and the bandwidth was be measured for various different test cases. This includes the effect of the complexity of complex event pattern, the window size, the number of separate complex events and the number of event instances.

7.1 Methodology

The methods used for measuring the latency and bandwidth of P4CEP are important for ensuring that the results are representative. This section will describe how the latency and the possible bandwidth was measured and evaluated.

For both the latency and bandwidth measurements two identical host machines were used. The Intel Xeon CPU E5-1650 v4 (6 Cores, 12 Threads, 3.60 GHz) CPU with 32GB of RAM were used. The used Netronome NIC was the “Agilio CX 2x10GbE”.

7.1.1 Latency

Packet transmission and event detection latency was measured using Solarflare NICs (SFN8522PLUS with 10 Gb/s Ethernet) with hardware timestamping capabilities. The topology of the measurement setup can be seen in Figure 7.1. On the **Latency** host the *Ping* process sends UDP packets containing the packet ID at a fixed interval out of port 0 of the Solarflare NIC. After sending the packet, the timestamp hardware of the NIC is used for determining the egress time of the datagram which is then read by the *Ping* process. The retrieved timestamp is written to a file along with the packet id for later examination. For all latency tests the packets were sent at an interval of 500 µs between packets, which translated to 2000 packets per second, for 60 seconds resulting in a sample size of $\approx$ 120000 packets. The size of the UDP payload was 8 bytes. This size was chosen because the resulting data layout is compatible with the event notification header defined by P4CEP.

On the **CEP System** the packet arrives through the physical port 0 and can take multiple different routes depending on what test is currently being run. If P4CEP on the NFP is being evaluated, then it will stay completely in the NFP and be processed by the P4CEP instance currently running within it. The packet will then be sent back through the physical port 1 without being passed to the host system at all. When the bmv2 software switch is being tested, then the NFP is configured with a simple pass-through P4 design which acts as a simple bridge between the NFP and the host. This makes the physical port 0 available to the host system as *v0.0* and the physical port 1 as *v0.1*. The bmv2 instance on the host handles all forwarding and P4CEP operations by reading packets from *v0.0* and *v0.1* and processing them according to the rules specified by the P4 file. If the latency of Apache Flink was measured, then the setup will be similar to when bmv2 is used. However, instead of P4CEP handling the packets using raw sockets, the packets will be passed to Apache Flink for processing by using UDP sockets. After either bmv2 or Apache Flink is done with the packet it is sent back through the virtual port *v0.1* of the NFP which forwards the packet to the physical port 1. A second process called *Reflector* runs on the **Latency** host and receives the packet by reading from the pyhsical port 1 of the Solarflare NIC. The packet payload contains the packet ID which is retrieved in addition to the packet ingress time stamp determined by the timestamping hardware found on the NIC. Both the ID and the retrieved timestamp are written to a second file for later examination.

Since both the egress and ingress timestamps originated from the same hardware it is possible to compare these values without requiring any special clock synchronization algorithms. This will be used for computing the full end-to-end latency of the packet by matching the IDs found in the two files written by the *Ping* and *Reflector*. This allows to compute the end-to-end latency in nanoseconds of each packet and, when combined with all other packets sent during the test, various statistical values such as the average and the median can be computed based on the latency data.

In the following sections the latency measurements are shown using histograms with error bars. The height of the histogram boxes shows the median latency of all packets in the data set. The error bars show the minimum value the first quartile and as the maximum the third quartile.

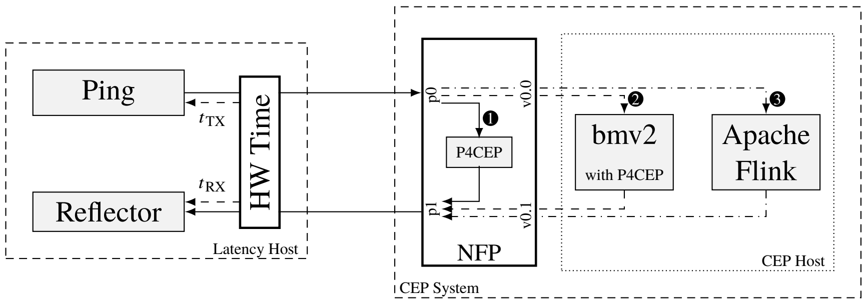


Figure 7.1: Topology of latency testing. The Solarflare NIC equipped **Latency** host sends packets with an assigned ID out of port 0 of the NIC and records the hardware timestamp t_{TX}. When the packet returns on port 1 the ingress timestamp t_{RX} is recorded along with the ID. Within the CEP system, multiple packet paths exist depending on the evaluation mode. When evaluating the NFP performance (❶), the packet stays within the NIC and is processed by P4CEP. When using bmv2 (❷), the packet is sent to the bmv2 instance on the host by using one of the virtual network ports provided by the NFP. The same is done evaluating Apache Flink (❸). In the context of CEP applications, *Ping* acts as an *Event source* while *Reflector* is the *Event sink*. The CEP system acts as the complex event processor.

7.1.2 Bandwidth

The bandwidth topology shown in Figure 7.2 is similar to the latency topology. The source host and the NFP host are connected via two Ethernet ports each. All packets received by the evaluation host on port 0 are processed and then sent back via port 1. The source host will then determine how many packets were received per second. The bandwidth setup uses a FreeBSD system with the Netmap tool pkt-gen [Riz] for sending and receiving packets.

The measurements will be shown using histograms where the height of the bars shows the average relative bandwidth supported by the target. The relative bandwidth is $\frac{b_m}{b_b}$ where b_m is the measured bandwidth for a given scenario. b_b is the baseline bandwidth which was measured in a separate test case (see Section 7.3.1). This improves the comparability of different targets even if the actual bandwidth values are different. No error bars are provided since no significant test variance was encountered in any of the test cases. Each bandwidth test was executed for 60 s which resulted in 60 bandwidth data points for every test case.

pkt-gen has some issues with receiving low packet rates where it only reports that $\approx$ 1014 packets per second were received instead of reporting a more accurate number. This causes issues for the bmv2 software switch evaluation since such low packet rates are usual for some evaluation scenarios when using bmv2. When this happens the graphs will show a value of $\approx$ 0.078 as the relative bandwidth.

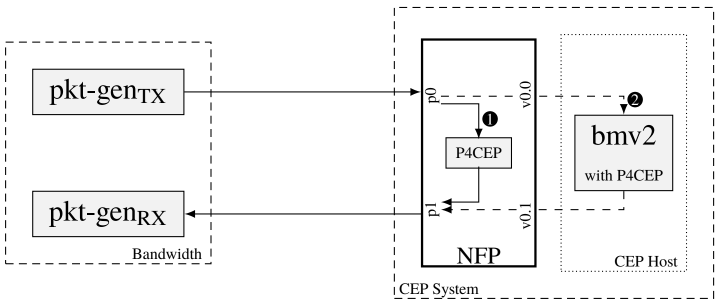


Figure 7.2: Topology of bandwidth testing. The source host uses pkt-gen of netmap (pkt-gen$_{TX}$) for sending frames at maximum rates to the CEP system. In the context of CEP applications, this pkt-gen instance acts as the *Event source*. The second instance (pkt-gen$_{RX}$) acts as the *Event sink* and counts how many packets have arrived. The CEP system processes the arriving events and then sends them to an event sink. Within the CEP system, the packet path is determined by the target that is being evaluated. If the NFP is the target, the packets stay within the NFP and will be processed by P4CEP (❶). If bmv2 is in use (❷), then packets will be passed to the bmv2 instance running on the host and will be processed there.

7.1.3 Limitations

During the evaluation multiple problems were discovered that will be described here:

Erratic behavior of bmv2 The behavior of the bmv2 target was often erratic which made definitive statements about the performance of this target hard to make. This is very obvious when looking at the error bars used in the latency evaluation graphs. The error in the NFP target was very small in most cases while the error range of the bmv2 target was large ($> 80\,\mu s$) and varied widely when conducting a different test. Since this target is not designed as a high performance software switch this was not unexpected.

Bandwidth testing failed some times For either the NFP or bmv2 target the bandwidth tests sometimes failed which resulted in bad data. When repeating the same test with the same input data again the results returned to the expected ranges. Since this problem occurred with both the NFP and bmv2 targets it is unlikely that this was caused by the code added by p4cep. It may be possible that there is a software bug in the tools used for capturing the bandwidth data which caused it to report wrong data. It may also be possible that the NFP has problems with high throughput situations but that could not be verified here. The NFP is relevant to the bmv2 case since even if the software switch was in use the NFP functioned as a bridge which connected the host system with the bandwidth test system.

Low bandwidth values As mentioned before, the bmv2 bandwidth test cases had issues when only a very low bandwidth was available. This made the test results mostly useless in those cases. This is likely an inherent issue present in the *pkt-gen* tool.

7.2 Comparison with Apache Flink

In order to make any statement about the performance of p4cep it needs to be compared to an existing CEP framework in a series of comparable scenarios. In this section, the latency performance of p4cep will be compared with Apache Flink, a popular, Java based open-source CEP framework.

7.2.1 Simple pattern

The performance of both Apache Flink and p4cep is an important comparison scenario. For this reason a simple test scenario was used which executes a simple pattern detection operation which always passes. This would generate an event for each incoming packet which could then be used for measuring the latency from the event generation on the **Latency** host until the event notification arrived. The p4cep rules shown in Listing 7.1 use a simple pattern which utilized the `event_id` header which was parsed from the UDP packets sent by the *Ping* process on the **Latency** host. The Apache Flink equivalent shown in Listing 7.2 uses a Java UDP Server for receiving the UDP packets from the **Latency** host and then parses the IDs from their binary representation. A simple filter equivalent to the filter used by p4cep is inserted in the pipeline before the packet is sent back via UDP to the **Latency** host.

The measurements that were taken from this setup are shown in Figure 7.3. The P4CEP case which was run on the NFP had a median latency of 17.3 µs while Apache Flink had a median latency of 205 µs which means that P4CEP in combination with the NFP has a latency that is 90% less than the established CEP framework Apache Flink. When using the bmv2 software switch the performance is worse than Apache Flink with a median latency of 522 µs which is not surprising since bmv2 is not particularly optimized for high performance situations since the goal of bmv2 is easy P4 development and not performance.

7.2.2 Window evaluation

Sliding window operations are a common feature found in many CEP frameworks. P4CEP also supports this feature which makes it possible to compare the event detection latency when using this feature.

The P4CEP test case used a rule definition (see Listing 7.3) which evaluated an expression which was always true and a window of varying size.

This test case showed an issue with the NFP P4-Compiler. The compiler and the hardware are not capable of handling large P4 programs. If the size of the window (from now on referred to as n) is increased past 10 the program became too large to be able to fit into the available memory on the NFP. When increasing n further, the compilation times increased exponentially until it was no longer practical to conduct the tests. This made it impossible to collect data from the pure P4 target with values $n > 10$.

As described in Section 6.5.1, the P4 code for the NFP is capable of being extended using custom functions implemented in C. These functions were not constrained by the limitations of the P4 language and could use advanced language constructs such as loops. Since the need for loop unrolling in the P4 program was the primary reason for the increased program size an equivalent C-function was implemented and is shown in Listing A.3. It makes use of the registers defined by the P4 program to improve compatibility with the existing P4CEP code. It also takes the window parameters from metadata fields which makes it possible to use the same function for multiple windows with different sizes. However, since P4CEP is only designed to generate P4 code and not C-code it was not possible to dynamically generate this code. For this reason, a static version of the code was written which implements a single function evaluation equivalent to `sum(test_wnd)`. Since the C language is more powerful than the P4 language it is possible to implement all P4CEP operators in C if necessary.

The bmv2 version of the test did not suffer from such problems and was evaluated without problems up to a window size of 100.

Listing 7.1 P4CEP rule definition used for this comparison with Apache Flink. The expression `event_id.id >= 0` is always true to allow measuring the time it takes if a packet generates an event.

```
complex_event test_evt {
    value ipv4.totalLen
    strategy skip
    pattern [event_id.id >= 0]
}
```

Listing 7.2 Apache Flink code equivalent to the P4CEP rule definition shown in Listing 7.1. `ids` is a `DataStreamSource` which generates events from single arriving UDP packets. `IdPacketSink` is a `SinkFunction` which sends the data contained in the packet back to the **Latency** host via a UDP socket.

```java
ids.filter(idPacket -> idPacket.getId() >= 0)
   .addSink(new IdPacketSink(<...>));
```

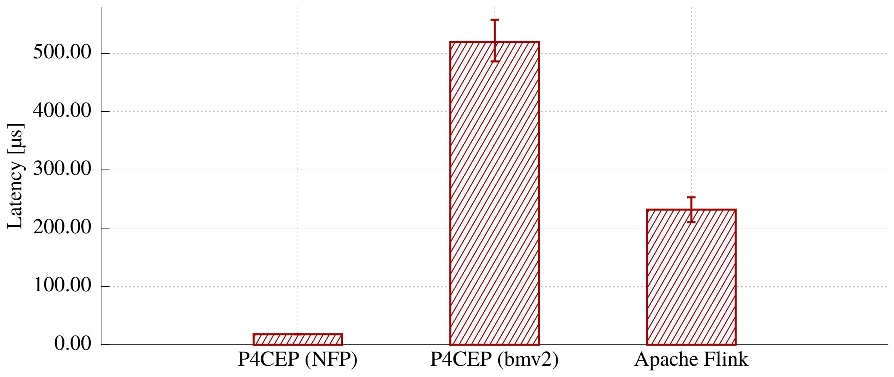


Figure 7.3: Median latency of the simple filter test case. The NFP with P4CEP has a latency of 18 µs. The bmv2 software switch has 520 µs latency. The Apache Flink comparison has 232 µs end-to-end latency.

Apache Flink does not offer the same kind of windowing mechanics as P4CEP. Instead it is focused on time based windows which are not possible with the current version of P4CEP. However, it is possible to use event based sliding windows which are similar to what P4CEP offers by using special triggers and evictors. The used data flow definition is shown in Listing 7.4. `ids` is a `DataStreamSource` which generates events from single arriving UDP packets. `IdPacketSink` is a `SinkFunction` which sends the data contained in the packet back to the timing host via a UDP socket. The `windowAll` and subsequent `trigger` and `evictor` calls set up a window which will keep n previous values in memory and `SumWindowFunction` will sum the values stored in the events and generate an output packet with that value. It is important to make sure that the packet which is sent to the timing host has the largest ID in the window since that is the last packet which entered the window. This means that this packet is responsible for generating the complex event which means that the latency of this packet has to be measured. Otherwise, the timing information may compute the latency between two different packets which would distort the results. The default configuration of Apache Flink uses buffer mechanisms which buffer data for up to 50 ms before passing it to the next operator. This mechanism can be changed by using the `setBufferTimeout` method of the `StreamExecutionEnvironment`. When set to 0 the buffering mechanism is disabled completely which results in the best latency but the worst throughput. For the window tests Apache Flink was configured so that this buffer mechanism was disabled to ensure that the comparison with Apache Flink is as fair as possible. Due to the way Apache Flink works, it was not possible to replicate the P4CEP semantics exactly. With the data stream configuration specified in Listing 7.4 the window is cleared when it reaches a size of n whereas the window of P4CEP only removes the last value. This means that for the Apache Flink test case the sample size was reduced to $\frac{120000}{n}$ where n is the tested window size.

Unfortunately, the Apache Flink setup from above could not be used with the bandwidth evaluation system since the packets generated by *pkt-gen* are not formed in a way that can be received by the available Java networking classes. The available bandwidth of other P4CEP targets will be evaluated in Section 7.3.

The overhead of using this feature will be evaluated in Section 7.3.6 with a greater window size range.

Latency

The latency measurements were conducted for multiple targets. As described before, the NFP is not capable of accommodating pure P4 programs with window sizes larger than 10. For this reason the variant of the P4 program which uses a C function for evaluating the window is also included in this test. The range of the window size ranged from a window with 1 element to a window with 50 elements. The window size was incremented by 1 from 1 to 10, then by 2 until a size of 20 and finally by 5 until reaching 50.

Figure 7.4b shows a graph of latency measurements in µs for both the NFP and NFP-C targets. The base latency for both NFP targets started at 19.6 µs. The latency grows linearly with the window size at $\approx$ 0.5 µs per additional window element when using the pure P4 code. The C function target is less efficient and grows at $\approx$ 1 µs per window element. When evaluated with increasing window sizes the latency of the NFP-C target continued the linear trend which is shown in Figure 7.4c.

The latency of the bmv2 software switch shown in Figure 7.4a starts at 550 µs and grows linearly at $\approx$ 11 µs per added window element until the window reaches a size of 20. At that point it suddenly jumps to $\approx$ 600 ms and grows linearly again at 12.5 ms per window element. This behavior is likely caused by the bmv2 program using more memory for the internal representation of the P4 program than is available in the CPU cache. This would lead to many cache misses when executing the program which would lead to worse performance than when the program could be stored in its entirety in the CPU cache.

The latency graph of Apache Flink shown in Figure 7.4d shows a slightly worse latency than either of the NFP targets with $\approx$ 270 µs latency regardless of the window size. Since the latency of the Apache Flink appears to be constant at least in the evaluated value range it will be faster than the

Listing 7.3 P4CEP rule definition used for the window evaluation comparison with Apache Flink. The expression `event_id.id >= 0` is always true to allow measuring the time it takes if a packet generates an event. The `sum` call has been inserted to ensure that the window evaluation works correctly.

```
window test_wnd {
    size 2
    value ipv4.totalLen
}
complex_event test_evt {
    value event.id
    strategy skip
    pattern [event_id.id >= 0] || [sum(test_wnd) > 0]
}
```

Listing 7.4 Apache Flink code equivalent to the P4CEP rule definition shown in Listing 7.4.

```
ids.windowAll(GlobalWindows.create())
        .trigger(CountTrigger.of(2))
        .evictor(CountEvictor.of(2, true))
        .process(new SumWindowFunction())
        .addSink(new IdPacketSink(<...>));
```

NFP targets if large windows are required. If the buffer timeout value is left to the default value chosen by Apache Flink then the latency is constant at 50 ms which is much worse than the latency measured without having the buffer mechanism enabled.

7.3 Overhead

In order to evaluate what the impact of the code added by P4CEP has on the performance of the target, a number of scenarios will be tested where both bandwidth and latency are measured. Each scenario will vary one aspect of a P4CEP rule file and then measure the latency and bandwidth. The results will then be compared with a baseline test (see Section 7.3.1).

For latency, this comparison will happen by looking at the *additional* latency caused by the P4CEP code. This is done by substracting the median latency of the baseline from all latency measurements. This does not include the minimum and maximum values used for the error bars.

The bandwidth graphs are drawn by dividing the bandwidth measured for a specific target and scenario by the bandwidth measured in Section 7.3.1 for this target. This results in a relative bandwidth measure which is easier to compare between different targets.

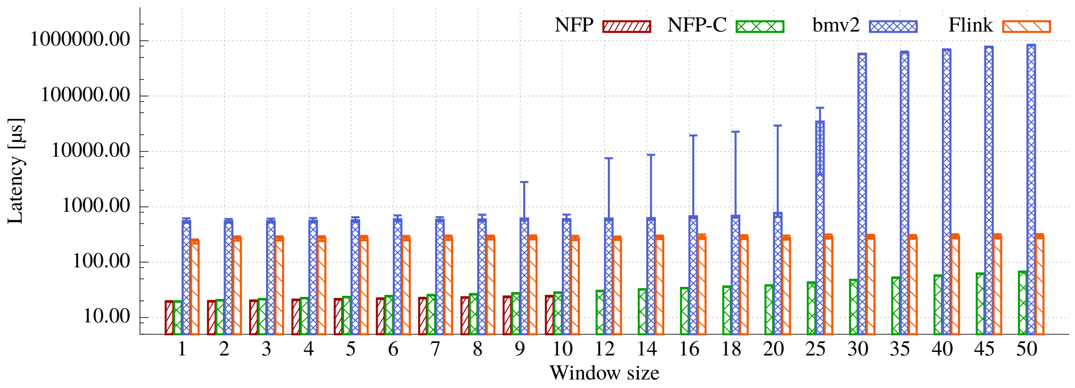
7.3.1 Baseline measures

The baseline testing was done using the same evaluation setup already described in Section 7.1 and by using a P4 source file without any added CEP code.

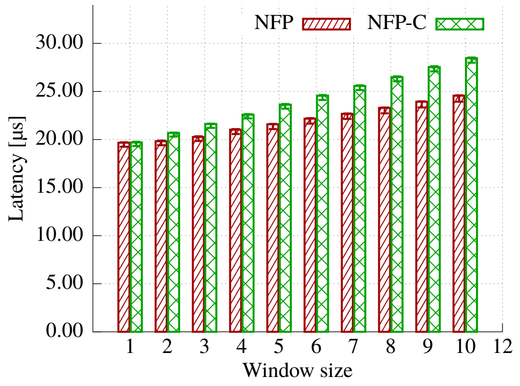
Latency

The baseline latency measurements of the two basic targets (NFP and bmv2) are shown in Figure 7.5. For the NFP, a median latency of 6.85 µs with a minimum of 6.73 µs and a maximum of 6.89 µs was measured. The bmv2 has a median latency of 464 µs with a minimum latency of 427 µs and a maximum of 508 µs.

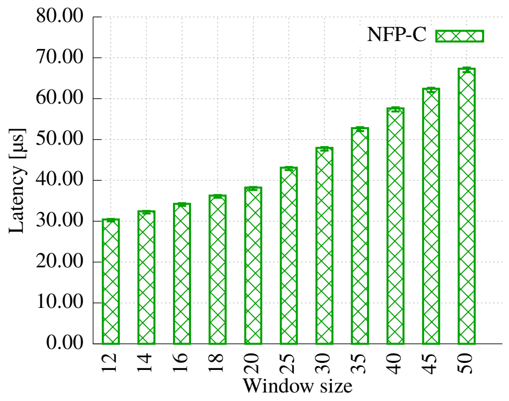
This shows that the baseline latency variance of the bmv2 is much higher than the NFP latency variance. Since the bmv2 is not optimized for performance this is expected.



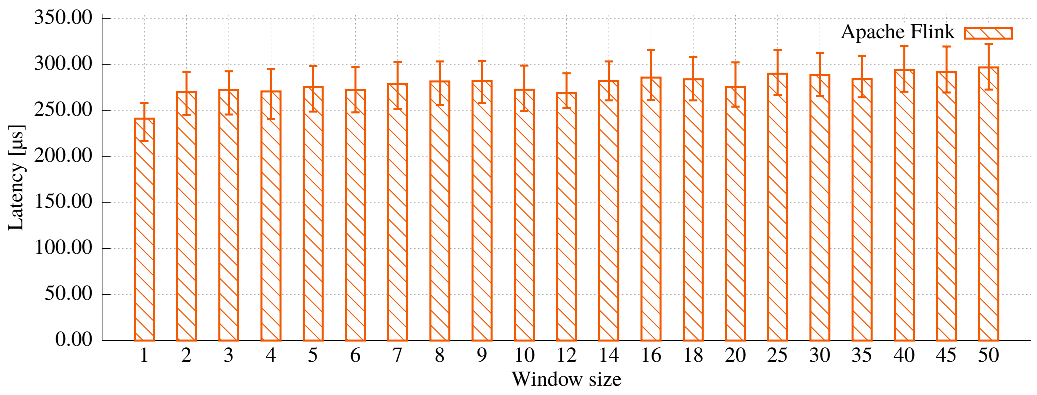
(a) Comparison of latency when using p4cep with the NFP, NFP with C extensions, bmv2 and Apache Flink.



(b) Latency comparison of the default NFP code and the NFP code with C extensions.



(c) Latency measurement of the P4 NFP code with C extensions from a window size of 12 to 50.



(d) Latency measurements of all different window sizes when using Apache Flink.

Figure 7.4: Latency comparison between p4cep targets NFP, NFP-C and bmv2 and the CEP framework Apache Flink when using a sliding window.

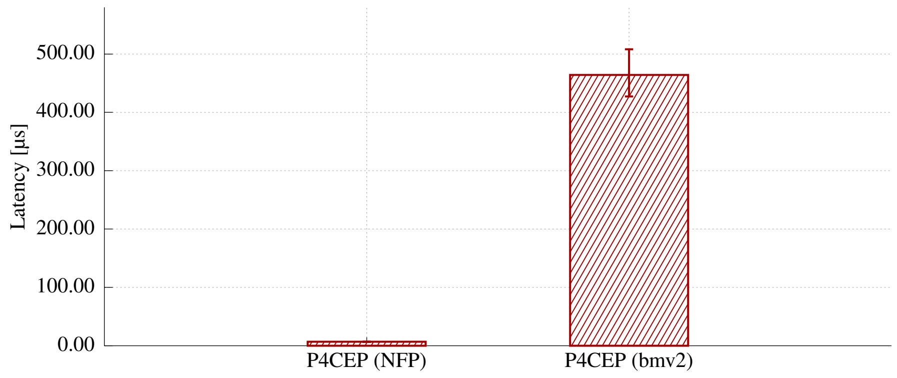


Figure 7.5: Baseline latency of the NFP and bmv2 targets. The NFP has a median latency of 6.85 µs latency. The bmv2 software target has a median latency of 464 µs

Bandwidth

Figure 7.6 shows the baseline bandwidth measurements of the NFP and bmv2 software switch. While the NFP manages to maintain the 10 Gb/s line-rate bandwidth the bmv2 is much worse with 12 633 pps which corresponds to a maximum bandwidth of 8.4 Mb/s.

7.3.2 Multiple expressions

This scenario evaluates the impact the `pattern` complexity has on the performance of the system. This will use multiple simple expressions in a sequence as shown in Listing 7.5. The **highlighted part** of the code will be repeated *n* times where each expression will be a slightly different expression to ensure that all expressions are checked. Every expression will be designed to evaluate to false so that the overhead of executing multiple conditional expressions is measured. The size of the state machine lookup table will also increase linearly with the number of expressions.

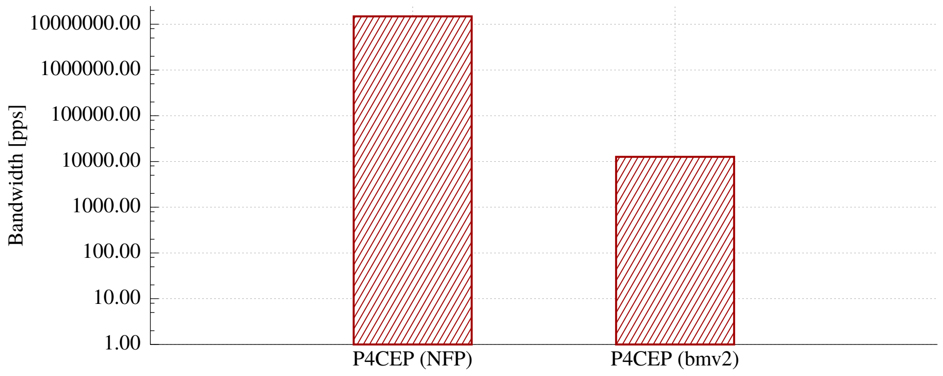


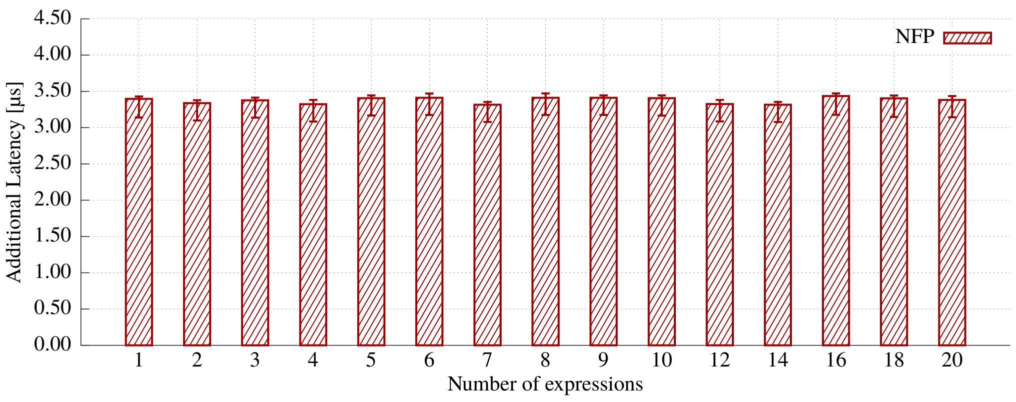
Figure 7.6: Baseline bandwidth of the NFP and bmv2 targets in packets per second. The NFP has an average bandwidth of 14 876 061 pps which corresponds to the line-rate bandwidth. The bmv2 software target has an average bandwidth of 12 634 pps.

Listing 7.5 Rule file for different expression evaluation. The **emphasized** part of the code was parameterized to be repeated *n* times with different expressions each time.

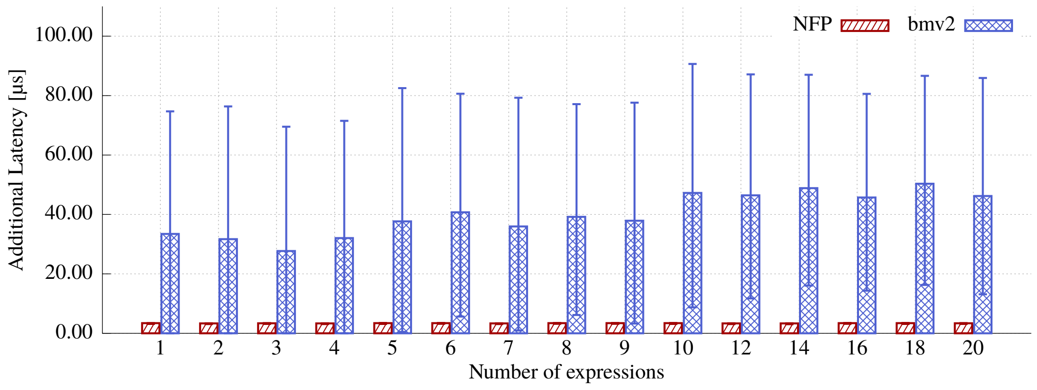
```
complex_event test_evt {
    value sum(ipv4.totalLen)
    strategy skip
    pattern [ethernet.etherType == 2049];[ethernet.etherType == 2050]
}
```

Latency

The latency measurements of this scenario are shown in Figure 7.7. For the NFP the additional latency shown in Figure 7.7a is constant at approximately 3.4 µs throughout all tested scenario values. The behavior of the bmv2 software switch shown in Figure 7.7b is much more erratic but shows a slight tendency for increased latency when increasing the number of different expressions. The NFP seems to be able to execute the additional transition predicates without adding any measurable latency while the bmv2 software switch seems to have a linear latency increase in this case.



(a) Additional latency of the NFP target.



(b) Additional latency of the bmv2 software switch target with the NFP data for comparison.

Figure 7.7: Additional latency when using multiple different expressions in an event pattern.

Bandwidth

The bandwidth results shown in Figure 7.8 show similar trends to the latency results. The NFP target stays at a constant 60% of the original bandwidth while the bmv2 bandwidth decreases linearly from 64% to 53%.

7.3.3 Complex expressions

The complexity of expressions defines how much data can be processed for a single primitive event since the *predicate* of an event determines when a basic event has occurred. The performance impact of this was tested by building increasingly complex expressions. These expressions must be built so that they must be evaluated in their entirety to determine the truth value. For this reason this test checks the value of `ethernet.ethertype` with different values which were all chosen so that the value never appears in the network traffic used by the bandwidth or latency topology. The rule code shown in Listing 7.6 uses disjunctions to ensure that no short-circuiting could happen which ensures that the switch would evaluate all parts of the expression.

When evaluating this scenario a problem with the NFP P4 compiler was discovered. The compiler can not handle complex expression properly and failed with an error message when an expression with more than 8 conditions was used. The bmv2 target used the never p4c compiler which did not suffer from this problem which allowed more complex expressions to be tested with the bmv2 target.

Latency

In the range of possible evaluation values (from 1 to 8), the NFP target (shown in Figure 7.9a) does not show any major latency changes when increasing the complexity of the used expression. The latency stays constant at ≈ 3.15 µs.

The same is true for the bmv2 target shown in Figure 7.9b where the additional latency stays at ≈ 40 µs regardless of the expression complexity.

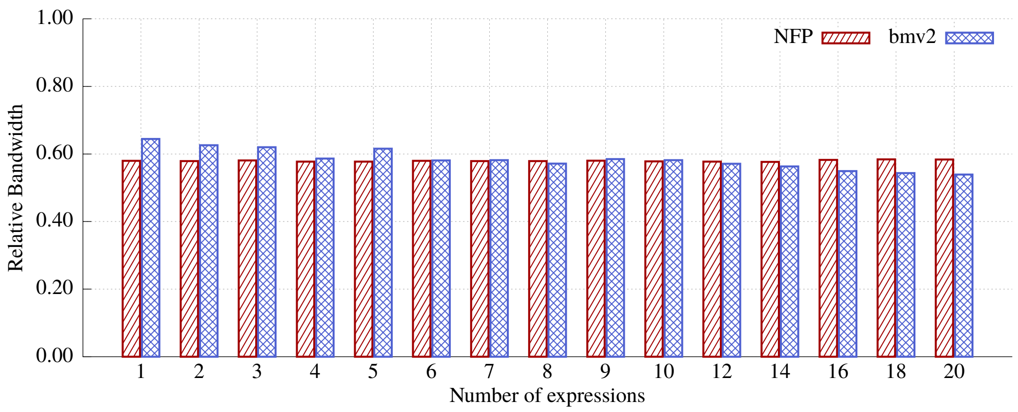
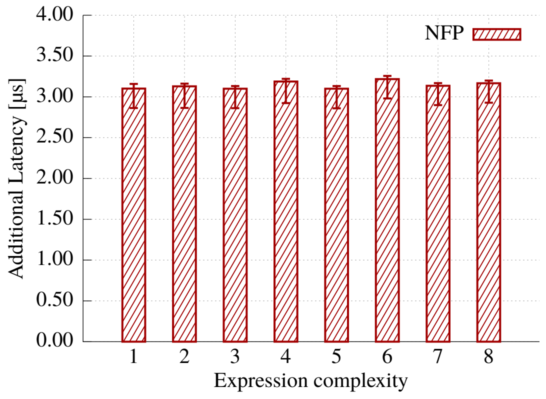


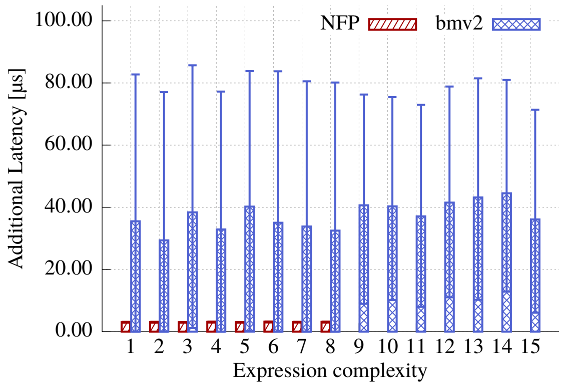
Figure 7.8: Relative bandwidth to the baseline test case when using multiple different expressions in an event pattern.

Listing 7.6 Rule file used for evaluating the impact of complex expressions in rule definitions. The **highlighted part** was repeated parameterized for this test so that it would check *n* expressions as a single event.

```
complex_event test_evt {
    value sum(ipv4.totalLen)
    strategy skip
    pattern
    [ethernet.etherType == 2049 || ethernet.etherType == 2050]
}
```



(a) Additional latency of the NFP target



(b) Additional latency of the bmv2 software switch target

Figure 7.9: Total additional latency when using a single complex expression in an event pattern.

Bandwidth

The bandwidth measurements shown in Figure 7.10 are similar to the values shown in the latency measurements. Both the NFP and bmv2 targets maintain a constant bandwidth of ≈ 60% of the original bandwidth.

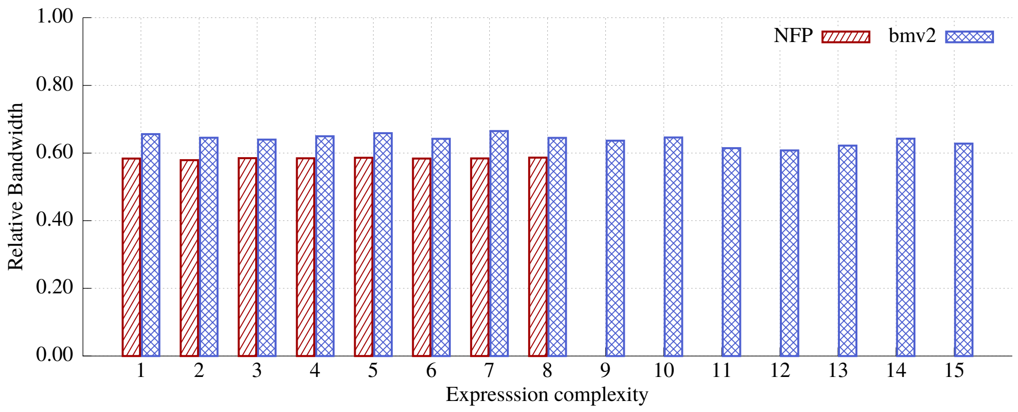


Figure 7.10: Relative bandwidth when using a single complex expression in an event pattern.

7.3.4 Multiple of the same expressions

The effect a table lookup has on the latency is an important measurement for how many states and transitions an event pattern may have. Since dedicated forwarding hardware commonly uses Content Addressable Memory (CAM) for table lookups the time a single lookup takes does not increase if the size of the table itself increases. The same is true for hardware with only simple RAM where hash tables offer $O(1)$ access times.

To test this theory the rule template shown in Listing 7.7 was used. It increases the number of `[ipv4.protocol == 1234]` expressions in the pattern which will increase the size of the lookup table. Since P4CEP eliminates duplicate expressions found in the event pattern, the size of the transition mapping block does not increase when adding more of the same expressions to the pattern. This will test how many times a single event can be included in a complex event pattern since an event is defined by the predicate it uses.

Latency

The latency measurements of both targets shown in Figure 7.11 support the theory from above. The NFP target shows no increase in latency when increasing the size of the state lookup table. The same is true for the bmv2 target although the usual high variance in the test results of this target make it hard to make a definitive statement.

Bandwidth

The bandwidth measurements shown in Figure 7.12 show similar results to the latency tests. The relative bandwidth of the NFP and bmv2 targets does not change when increasing the size of the state lookup table.

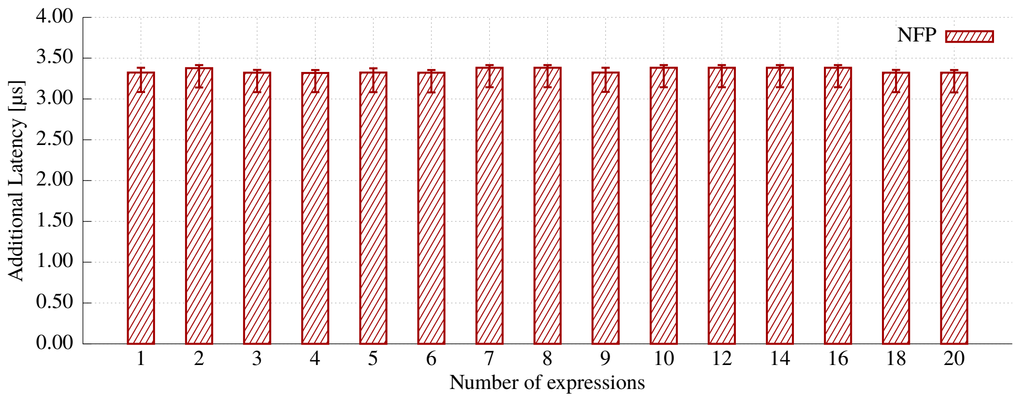
7.3.5 Number of variables

The effect the number of used variables has on the performance is very important to determine how much mutable external data can be added to a P4CEP definition without impacting performance. The rule definition template used for this scenario is shown in Listing 7.8.

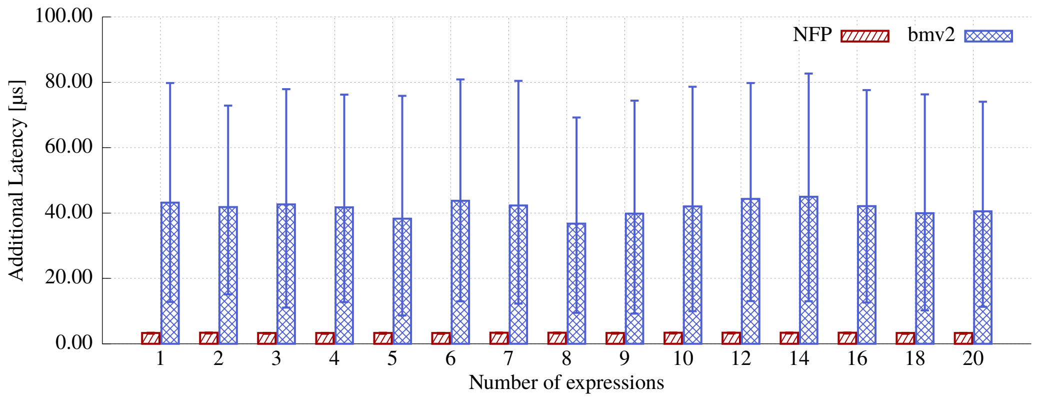
It was only possible to measure the latency and bandwidth up to 15 variables for the NFP since the Netronome P4-Compiler would not accept a program which used more variables. This was due to how P4CEP passes the variables to the P4 program. It uses a table without entries and only a

Listing 7.7 Rule file for same expression evaluation. The colored part of the code was parameterized to be repeated *n* times.

```
complex_event test_evt {
    value sum(ipv4.totalLen)
    strategy skip
    pattern [ipv4.protocol == 1234];[ipv4.protocol == 1234]
}
```



(a) Additional latency of the NFP target



(b) Additional latency of the bmv2 software switch target

Figure 7.11: Additional latency when using the same expression multiple times in an event pattern.

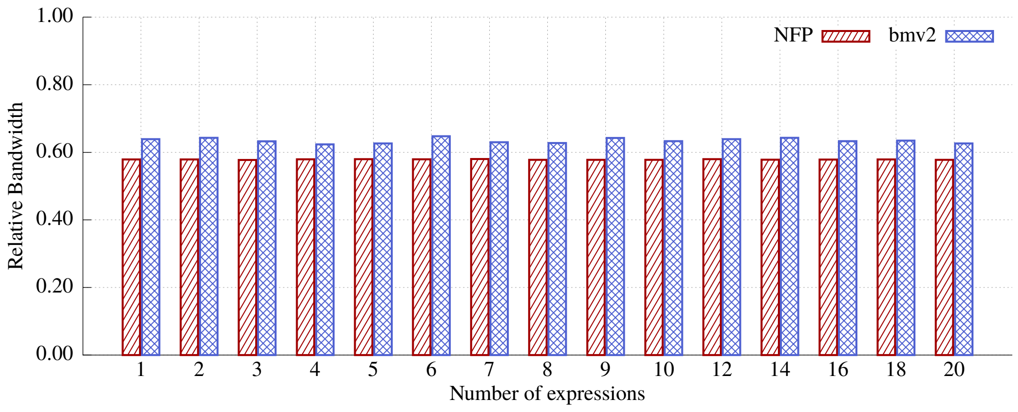


Figure 7.12: Relative bandwidth when using the same expression multiple times in an event pattern.

default action with the associated variable values. If more than 15 variables are used the compiler would need too much space for the action parameters and aborted compilation. This can be worked around by using multiple tables and actions but that was not implemented in the version of P4CEP evaluated here.

Latency

The latency measurements for the variables scenario are shown in Figure 7.13. The results of the NFP target shown in Figure 7.13a show that adding the initial variable increases the latency slightly but not further if more variables are added. This increase from zero to one variable is caused by the addition of a table to the P4 program since P4CEP does not include the variable table if there are no variables in the rule file.

The bmv2 target shown in Figure 7.13b has a similar trend although it appears as if every added variable increases the latency slightly. This suggests that the NFP can handle multiple parameters for a table action better than bmv2.

Bandwidth

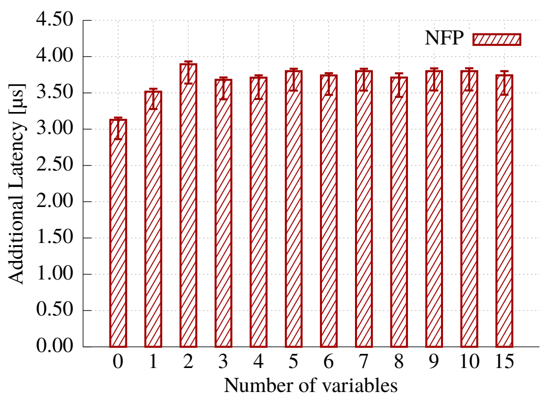
The bandwidth results shown in Figure 7.14 are similar to the latency results. The relative bandwidth of the NFP does not seem to be affected by the number of variables. The bandwidth supported by the bmv2 decreases slightly when more variables are added.

7.3.6 Window size

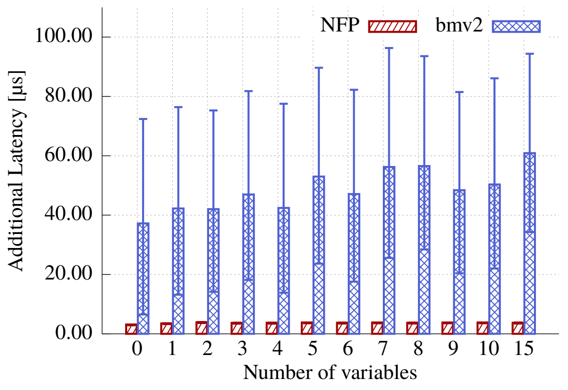
The effect of the window size was already partially examined in Section 7.2.2 when comparing the performance of P4CEP to Apache Flink. In this section the overhead of iterating through the window will be measured by increasing the window size. The rule file used for this test is shown in Listing 7.9. Only the window size was varied while all other aspects of the rule file were left unchanged. Compared to the Apache Flink test case this test case will use a larger range of window size values.

Listing 7.8 Rule file for the variable evaluation. For the measurements n variables were declared and used in a equality check in the event pattern.

```
var test_var_1 = 1;
var test_var_2 = 2;
complex_event test_evt {
    value sum(ipv4.totalLen)
    strategy skip
    pattern
        [ipv4.protocol == $test_var_1];
        [ipv4.protocol == $test_var_2];
        [ipv4.protocol == 1234];[ipv4.protocol == 1234];[ipv4.protocol == 1234];
        [ipv4.protocol == 1234];[ipv4.protocol == 1234];[ipv4.protocol == 1234];
        [ipv4.protocol == 1234];[ipv4.protocol == 1234];[ipv4.protocol == 1234]
}
```



(a) Additional latency of the NFP target



(b) Additional latency of the bmv2 software switch target

Figure 7.13: Additional latency when using multiple P4CEP variables in a rule file.

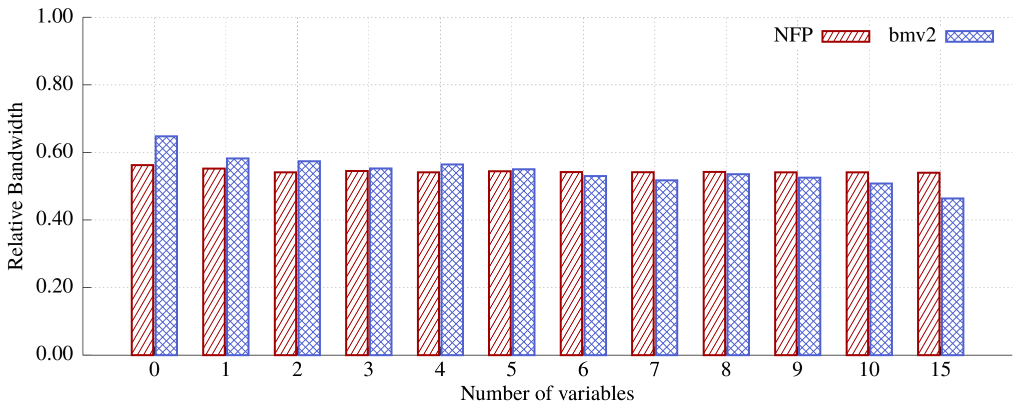


Figure 7.14: Relative bandwidth when using multiple variables in an event pattern.

This test case had the same issues already encountered in the Apache Flink comparison evaluation where a window size larger than 10 could not be supported by the NFP when using pure P4. For this reason the same C extensions as before were used in this case for evaluating larger window sizes.

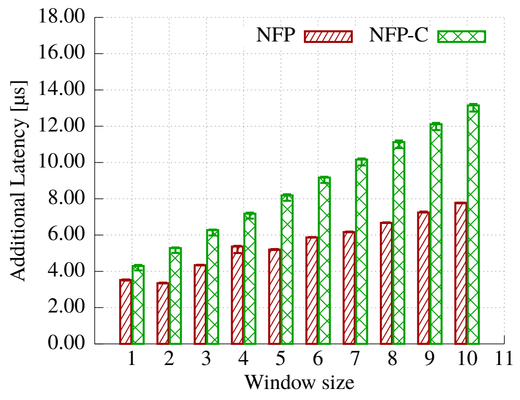
Listing 7.9 This template was used for evaluating the impact of the window size. The **highlighted** part was varied for the test.

```
window test_wnd {
    size 2
    value ipv4.totalLen
}
complex_event test_evt {
    value sum(test_wnd)
    strategy skip
    pattern [ipv4.protocol == 1234]
}
```

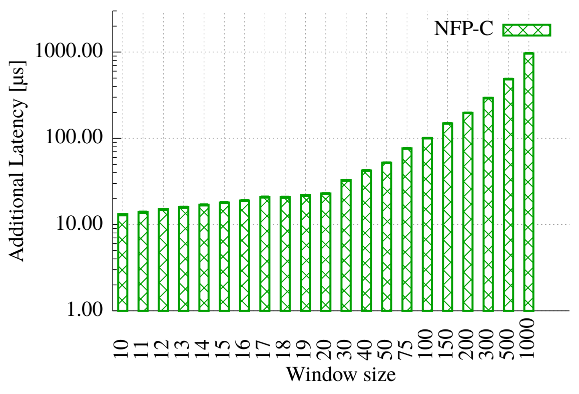
Latency

The latency results shown in Figure 7.15 are similar to the trends of the latency results from the Apache Flink comparison (Section 7.2.2). The additional latency of the NFP starts at 3 µs and then increases by 0.5 µs for every additional window element. The C extensions show a constant increase of 1 µs per window element which is consistent over the entire tested value range.

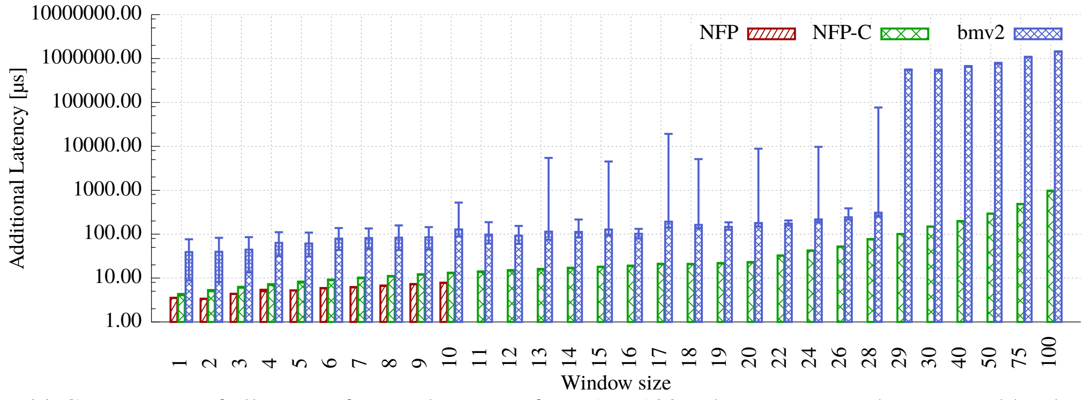
The bmv2 target shows the same sudden jump which was already observed in Section 7.2.2. The latency of the individual packets captured while conducting the window test with bmv2 and a window size of 100 is shown in Figure 7.16. The first packets have a relatively low latency of ≈ 9 ms. After that, the latency rises linearly to the median latency of 1.4 s shown in Section 7.2.2.



(a) Additional latency of the NFP target and the NFP P4 code with C extensions up to a window size of 10.



(b) Additional latency of the NFP P4 target with C extensions. This contains the measurement for window sizes of 10 to 1000



(c) Comparison of all targets for window sizes from 1 to 100. The pure NFP P4 target could only support window sizes of up to 10 elements.

Figure 7.15: Additional latency when increasing the size of a single sliding window.

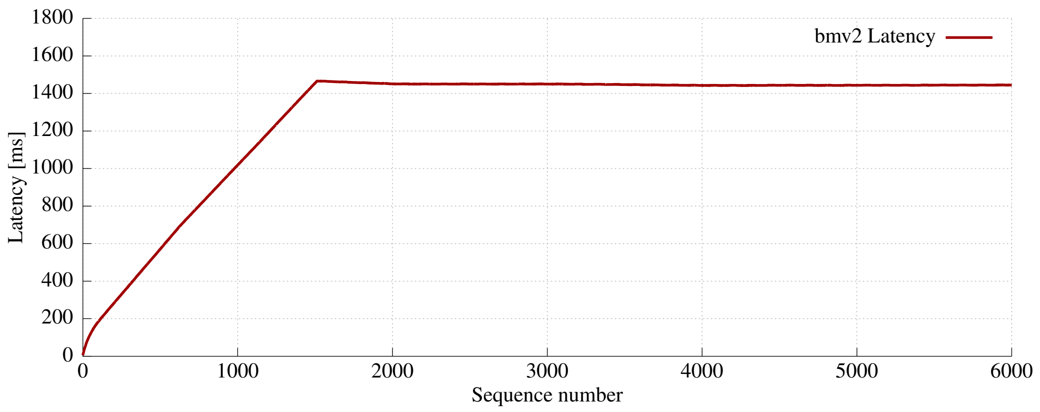


Figure 7.16: Latency of individual packets when conducting the latency test with a window size of 100. The first 6000 packets of 120000 total are shown.

Bandwidth

The bandwidth results shown for window sizes from 1 to 10 shown in Figure 7.17a show a linear decrease in the available bandwidth when increasing the window size. Windows in rule files seem to be especially harmful for the available bandwidth since even a window with a single element reduces the available bandwidth on the NFP to 47% with similar results for all other targets.

When increasing the window size further as shown in Figure 7.17b this trend continues. As mentioned before, since the packet rate for the bmv2 case is too low, *pkt-gen* does not yield useful information for this target with these window sizes.

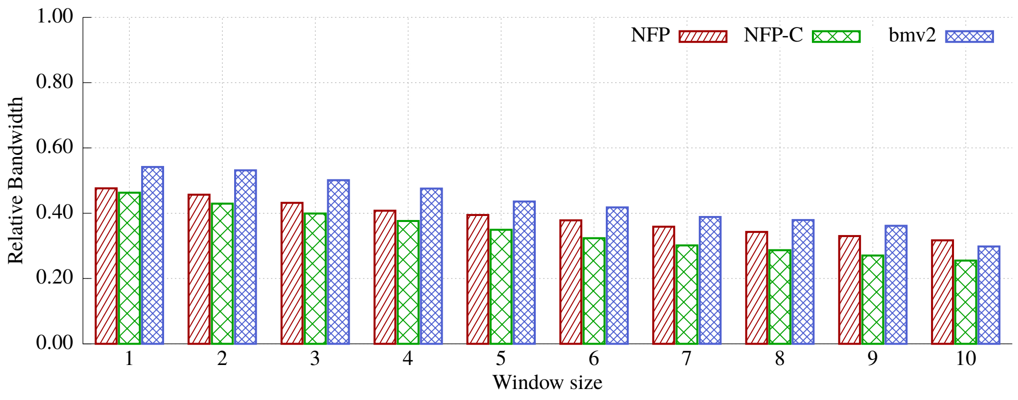
7.3.7 Number of complex events

The number of complex events that are possible to be used simultaneously determines how many information streams can be followed at the same time. For this test, the rule definition shown in Listing 7.10 was used which increases the number of complex events linearly.

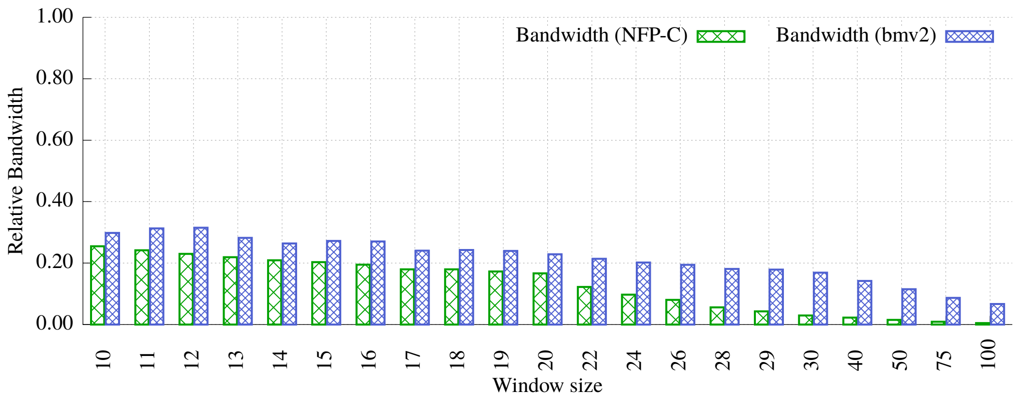
The NFP P4-Compiler was not able to load programs which use more than 4 events at the same time onto the NFP. The same error as in the window size evaluation was encountered here when using 5 or more events at the same time. The bmv2 software switch did not suffer from such an issue and could be evaluated with more events.

Listing 7.10 This shows the basic rule definition used for this scenario. The number of events was increased by using the same event multiple times but with different names.

```
complex_event test_evt_0 {
    value sum(ipv4.totalLen)
    strategy skip
    pattern [ethernet.etherType == 2049]
}
```



(a) Bandwidth measurements of the NFP, NFP with C extensions and bmv2 software switch targets for window sizes from 1 to 10.



(b) Bandwidth of the NFP with C extensions and bmv2 software switch targets for window sizes from 11 to 100.

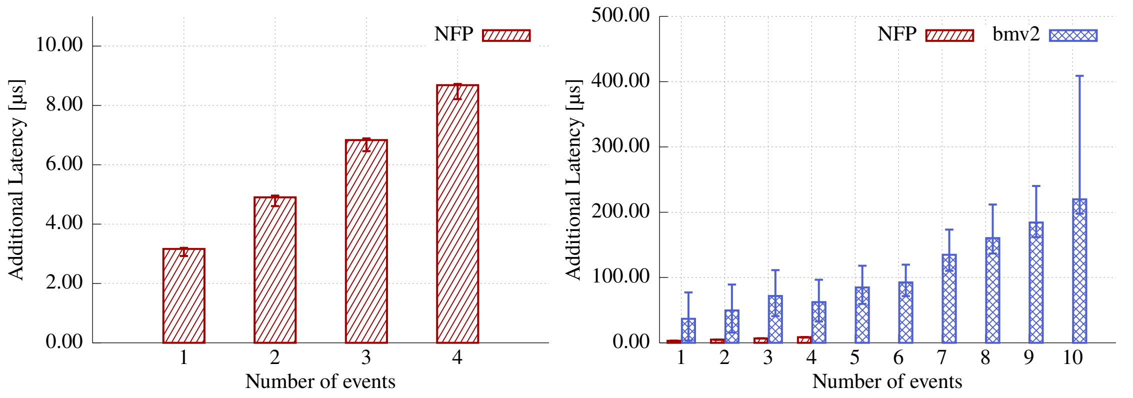
Figure 7.17: Relative bandwidth when using a sliding window.

Latency

As described above it was not possible to evaluate the performance of multiple events on the NFP platform past 4 events. Within this range (shown in Figure 7.18a) a linear increase of 1.3 µs per added event was observed. The bmv2 software switch (Figure 7.18b) had a similar linear increase of 18, 3 µs per added event.

Bandwidth

The bandwidth measurements shown in Figure 7.19 shows a linear decrease in the available bandwidth for both the NFP and bmv2 targets. The available bandwidth on both targets decreases linearly with the number of complex events which is similar to how the latency measurements evolved when adding more complex events.



(a) Latency of the NFP target **(b)** Latency of the bmv2 software switch target

Figure 7.18: Additional latency when using multiple different events in a single rule file

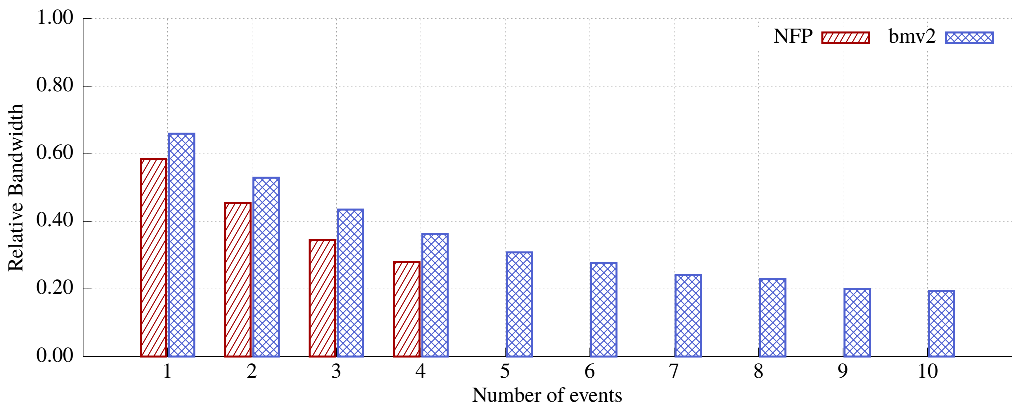


Figure 7.19: Relative bandwidth when using multiple events in a single rule file

7.3.8 Number of event instances

Multiple event instances are required for detecting interleaving event instances. This scenario tests the overhead of this feature by increasing the number of event instances when using a pattern that made sure that every instance was in a state different from the starting state so that all the code added by the additional instances was executed. The rule file used for this test can be seen in Listing 7.11.

When evaluating the NFP target the same issue as with the window and multiple event scenarios was observed. When trying to compile and load P4 code which had more than 5 event instances the upload or compilation process failed since the program was too large to fit into the program memory available to the NFP.

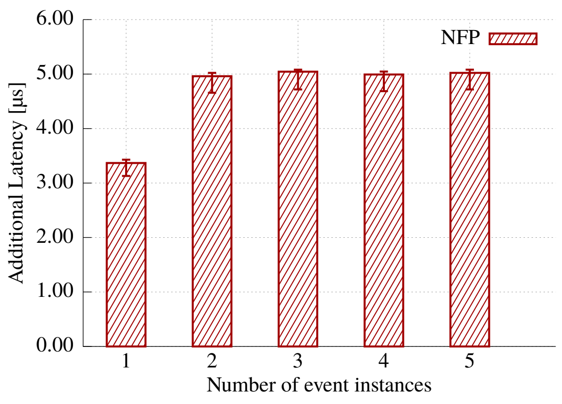
Latency

The additional latency (shown in Section 7.2.2) of the NFP target shows an increase of 1.5 µs when using a second instance. However, all further instances do not seem to incur a significant overhead. A similar trend can be observed when conducting the same test with the bmv2 software switch.

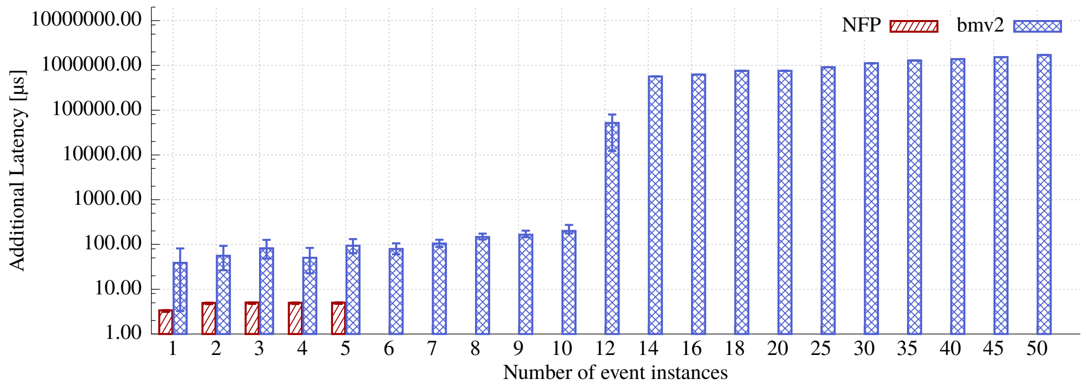
Listing 7.11 P4CEP rules for testing event instance overhead. The first part of the pattern was always true in the used test setup which made sure that all instances were in a state different from the starting state.

```
complex_event test_evt {
    value sum(ipv4.totalLen)
    strategy skip
        instances 2
    pattern [ipv4.protocol == 17] ; [ipv4.protocol == 1234]
}
```

Up to 6 instances, the latency seems to be mostly constant. A linear increase of ≈ 50 µs per added instance was observed after that point up until 10 event instances. After that point the same pattern as already seen in other bmv2 test cases can also be observed here. The additional latency suddenly jump to 50 ms and then rises linearly with ≈ 6 ms per added event after that point.



(a) Additional latency of the NFP target



(b) Additional latency of the bmv2 software switch

Figure 7.20: Additional latency when specifying multiple event instances for one event

Bandwidth

The bandwidth measurements largely agree with the results shown by the latency measurements. The bandwidth (shown in Figure 7.21) of the NFP target stays at 58% for all tested event instance numbers with only minor fluctuations.

The bandwidth tests of the bmv2 shows the inverted pattern of the latency tests. The available bandwidth of the bmv2 decreased linearly with each added event until hitting the minimum value measurable by *pkt-config* at 12 complex event instances.

7.4 Findings

This section evaluated various aspects of the performance P4CEP offers. The latency when using the NFP is a 90% improvement over the established CEP framework Apache Flink.

The overhead testing has shown that certain features P4CEP provides have a different impact on the performance. It was shown that the complexity of the complex event pattern itself did not impact the latency. However, every data structure which requires memory caused a significant performance penalty although the latency never went over the latency of Apache Flink. This suggests that P4CEP is less suited for processing past data for every packet and particularly well suited for detecting complex event patterns which do not use stateful operators.

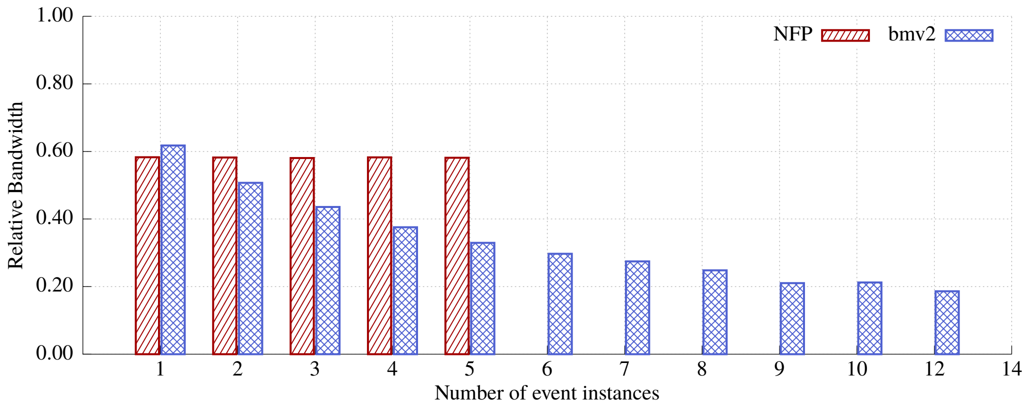


Figure 7.21: Relative bandwidth when using a single event with multiple instances

8 Conclusion

This thesis presented p4cep, a compiler for translating Complex Event Processing rules into P4 code which can be used for detecting complex events in the data-plane with only minor overhead. Based on a simplified CEP event model, a specialized language for specifying CEP rules was designed. p4cep was developed for transpiling this language to P4 which could then be included in standard P4 source code and be compiled for a specific target using the standard P4 compilers.

The generated code was compared with the popular CEP framework Apache Flink and evaluated for the additional overhead the code generated by p4cep incurs. In the comparison with Apache Flink, a significant latency decrease was possible with p4cep when the generated code was executed using the Netronome SmartNIC (also known as NFP). In the tested cases a latency of 18 µs was measured compared to 232 µs when using Apache Flink which is a 93% decrease. The event detection latency when using the NFP increased with the usage of stateful operations which means that with this target p4cep is more suited to operations which require less memory.

The overhead evaluations showed that for all evaluated targets the amount of register operations significantly impact the latency and available bandwidth while the complexity of the complex event pattern has no significant impact on the performance of the code generated by p4cep. This means that the current version of p4cep is best suited for detecting complex event patterns if the amount of stateful operators can be reduced.

8.1 Future work

While p4cep was developed, some possibilities for further expansion of the project were discovered. These were not implemented in the current version of p4cep but could be integrated into the compiler in future works.

Currently, the code generated by p4cep applies the CEP rules for all packets that arrive at the switch. In a network where standard traffic and event packets coexist, this may not be desirable since the network traffic is subjected to additional processing latency even though it is not part of the event stream. Furthermore, it is possible that this traffic changes the pattern detection state of the complex events which could lead to false negatives or false positives in the event detection. By adding support for conditionally enabling the p4cep code based on the header values of a packet, this could be avoided.

At the moment, header values are added to windows unconditionally which is fine for the initial implementation but it limits the capabilities of the window feature. It could be possible to allow the user to specify a condition which is evaluated before adding an element to a window. This would increase the flexibility of the window feature.

How to distribute CEP operators in a CEP operators graph across multiple processors and machines is an ongoing research area. A similar issue exists when conducting CEP by utilizing data-plane hardware. If there is a CEP operator in the network which ignores most of the events generated by an event source then it would make sense to install a filter close to the event source which would only forward events if they are actually going to be processed by other CEP operators. Deciding what switch would be the most optimal location for the installing of this filter is not a trivial question and could be examined in the future.

The current version of p4cep has only a local view of the events that arrive at one switch. If two related events occur in two separate routers, then this complex event is not detected at this moment. It would be beneficial if there was a mechanism for exchanging the basic events between different p4cep instances to gain a global view of all events that arrived in the network.

A Appendix

Listing A.1 P4$_{16}$ control block for ingress packet handling.

```
control Ingress(inout headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {
    action drop_action() {
        mark_to_drop();
    }
    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }
    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop_action;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }
    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}
```

Listing A.2 This code provides two P4 primitive actions which can be used for ensuring mutual exclusion for a certain area of P4 code. It uses a standard spin lock algorithm with atomic memory access functions for implementing this feature. This is based on the code presented by Wu and Luo [WL17].

```c
int pif_plugin_cep_critical_section_begin(EXTRACTED_HEADERS_T *headers, ACTION_DATA_T *action_data)
{
        uint32_t sec_id = pif_plugin_meta_get__cep_locals__cep_cection_id(headers);

        __xrw uint32_t xfer = 1;
        mem_test_set(&xfer, &pif_register_cep_memory[sec_id], sizeof(uint32_t));
        while(xfer == 1) {
                mem_test_set(&xfer, &pif_register_cep_memory[sec_id], sizeof(uint32_t));
        }

        return PIF_PLUGIN_RETURN_FORWARD;
}

int pif_plugin_cep_critical_section_end(EXTRACTED_HEADERS_T *headers, ACTION_DATA_T *action_data) {
        uint32_t sec_id = pif_plugin_meta_get__cep_locals__cep_cection_id(headers);

        __xwrite uint32_t xfer_out = 0;
        mem_write32(&xfer_out, &pif_register_cep_memory[sec_id], sizeof(uint32_t));

        return PIF_PLUGIN_RETURN_FORWARD;
}
```

Listing A.3 This code was used for implementing the loop based window evaluation in C.

```c
int pif_plugin_cep_window_evaluate(EXTRACTED_HEADERS_T *headers, ACTION_DATA_T *action_data) {
        uint32_t first = pif_plugin_meta_get__cep_locals__cep_window_first(headers);
        uint32_t last = pif_plugin_meta_get__cep_locals__cep_window_last(headers);
        PIF_PLUGIN_ipv4_T *ipv4_header = pif_plugin_hdr_get_ipv4(headers);
        uint32_t len = PIF_HEADER_GET_ipv4___totalLen(ipv4_header);
        uint32_t size = last - first, head_id = first - 1;
        // head_id is the register index where the head ring buffer value  value is stored
        __xread struct pif_header_cep_memory in_xfer_cep_head, in_xfer_cep_wnd_value;
        __gpr struct pif_header_cep_memory out_reg_cep_head, out_reg_cep_wnd_value;
        __xwrite struct pif_header_cep_memory out_xfer_cep_head, out_xfer_cep_wnd_value;
        uint32_t current_head, sum, i, wnd_pos;
        mem_read32(&in_xfer_cep_head, &pif_register_cep_memory[head_id], sizeof(struct
        pif_header_cep_memory));
        out_reg_cep_head = in_xfer_cep_head;
        // Store this value so we can use it later for our for loop
        current_head = out_reg_cep_head.value - first;
        // Increment old head value and write new window value to that location
        current_head = ((current_head + 1) % size) + first;
        out_reg_cep_wnd_value.value = len;
        out_xfer_cep_wnd_value = out_reg_cep_wnd_value;
        mem_write32(&out_xfer_cep_wnd_value, &pif_register_cep_memory[current_head],
    sizeof(uint32_t));
        // Now also write the new head value to the correct register
        out_reg_cep_head.value = current_head;
        out_xfer_cep_head = out_reg_cep_head;
        mem_write32(&out_xfer_cep_head, &pif_register_cep_memory[head_id], sizeof(uint32_t));
        // Now we are done with the management part of the code so we can begin actually
        // evaluating the values in the window
        // Initialize with the current value of the header
        sum = len;
        // We start at 1 since we already have the current header value in the sum variable
        for (i = 1; i < size; ++i) {
                // Increment actual
                wnd_pos = ((current_head + i) % size) + first;
                // Read the value from that position from the registers
                mem_read32(&in_xfer_cep_wnd_value, &pif_register_cep_memory[wnd_pos], sizeof(struct
                pif_header_cep_memory));
                out_reg_cep_wnd_value = in_xfer_cep_wnd_value;
                sum += out_reg_cep_wnd_value.value;
        }
        // The sum has now been computed so we can write it in the correct meta data value
        // and finally return
        pif_plugin_meta_set__cep_locals__cep_function_0_value(headers, sum);
        return PIF_PLUGIN_RETURN_FORWARD;
}
```

Bibliography

[AKG+16] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, D. Walker. "SNAP: Stateful network-wide abstractions for packet processing". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM. 2016, pp. 29–43 (cit. on p. 22).

[Apa] Apache. *Apache Flink*. URL: https://flink.apache.org/ (cit. on pp. 19, 21).

[Bas16] A. Bas. *[P4-dev] where to add_header?* May 2016. URL: http://lists.p4.org/pipermail/p4-dev_lists.p4.org/2016-September/000481.html (cit. on p. 49).

[BBCC14] G. Bianchi, M. Bonola, A. Capone, C. Cascone. "OpenState: programming platform-independent stateful openflow applications inside the switch". In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 44–51 (cit. on pp. 23, 43).

[BDG+14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker. "P4: Programming Protocol-independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: http://doi.acm.org/10.1145/2656877.2656890 (cit. on pp. 13, 48).

[CKE+15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas. "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015) (cit. on p. 21).

[CM10] G. Cugola, A. Margara. "TESLA: a formally defined event specification language". In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM. 2010, pp. 50–61 (cit. on p. 21).

[CM12] G. Cugola, A. Margara. "Complex event processing with T-REX". In: *Journal of Systems and Software* 85.8 (2012), pp. 1709–1728 (cit. on p. 9).

[FGD+17] I. Flouris, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Kamp, M. Mock. "Issues in complex event processing: Status and prospects in the big data era". In: *Journal of Systems and Software* 127 (2017), pp. 217–236 (cit. on p. 29).

[KMD+18] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, K. Rothermel. "P4CEP: Towards In-Network Complex Event Processing". In: *Proceedings of ACM SIGCOMM 2018 Workshop on In-Network Computing (NetCompute'18). ACM, New York, NY, USA* (2018) (cit. on p. 10).

[Net] Netronome. *SmartNIC OVerview - Netronome*. URL: https://www.netronome.com/products/smartnic/overview/ (cit. on p. 50).

[Ope] Open vSwitch. *Open vSwitch*. URL: https://www.openvswitch.org/ (cit. on pp. 51, 52).

[P4 17] P4 Language Consortium. *$P4_{16}$ Language Specification*. May 2017. URL: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf (cit. on p. 13).

[p4la] p4lang. *behavioral-model*. URL: https://github.com/p4lang/behavioral-model (cit. on pp. 15, 51).

[p4lb] p4lang. *BMv2 JSON input format* - actions. URL: https://github.com/p4lang/behavioral-model/blob/master/docs/JSON_format.md#actions (cit. on p. 51).

[p4lc] p4lang. *$P4_{16}$ prototype compiler*. URL: https://github.com/p4lang/p4c (cit. on p. 51).

[Riz] L. Rizzo. *pkt-gen*. URL: https://github.com/luigirizzo/netmap/tree/master/apps/pkt-gen (cit. on p. 57).

[San] D. Sanvito. *openstate.p4*. URL: https://github.com/OpenState-SDN/openstate.p4 (cit. on pp. 23, 43).

[SCP+16] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, J. Rexford. "Pisces: A programmable, protocol-independent software switch". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM. 2016, pp. 525–538 (cit. on p. 52).

[SF15] M. Shahbaz, N. Feamster. "The case for an intermediate representation for programmable data planes". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM. 2015, p. 3 (cit. on p. 22).

[WL17] X. Wu, Y. Luo. *Network Measurement with P4 and C on Netronome NFP*. Open-NFP. 2017. URL: https://open-nfp.org/documents/63/Network_Measurement_with_P4_and_C_on_Netronome_NFP_UML.pdf (cit. on p. 82).

All links were last followed on May 28, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature