P4CEP: Towards In-Network Complex Event Processing

Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel
Institute of Parallel and Distributed Systems (IPVS), University of Stuttgart
Stuttgart, Germany
{firstname}.{lastname}@ipvs.uni-stuttgart.de

ABSTRACT

In-network computing using programmable networking hardware is a strong trend in networking that promises to reduce latency and consumption of server resources through offloading to network elements (programmable switches and smart NICs). In particular, the data plane programming language P4 together with powerful P4 networking hardware has spawned projects offloading services into the network, e.g., consensus services or caching services. In this paper, we present a novel case for in-network computing, namely, Complex Event Processing (CEP). CEP processes streams of basic events, e.g., stemming from networked sensors, into meaningful complex events. Traditionally, CEP processing has been performed on servers or overlay networks. However, we argue in this paper that CEP is a good candidate for in-network computing along the communication path avoiding detouring streams to distant servers to minimize communication latency while also exploiting processing capabilities of novel networking hardware. We show that it is feasible to express CEP operations in P4 and also present a tool to compile CEP operations, formulated in our P4CEP rule specification language, to P4 code. Moreover, we identify challenges and problems that we have encountered to show future research directions for implementing full-fledged in-network CEP systems.

CCS CONCEPTS

• **Networks → Programmable networks**; **In-network processing**; • **Software and its engineering → Publish-subscribe / event-based architectures**; *Message oriented middleware*;

KEYWORDS

In-network Computing, Data Plane Programming, P4, Complex Event Processing (CEP)

ACM Reference Format:
Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. 2018. P4CEP: Towards In-Network Complex Event Processing. In *NetCompute'18: ACM SIGCOMM 2018 Morning Workshop on In-Network Computing , August 20, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3229591.3229593

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
NetCompute'18, August 20, 2018, Budapest, Hungary
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5908-5/18/08…$15.00
https://doi.org/10.1145/3229591.3229593

1 INTRODUCTION

Recent developments in Software-defined Networking (SDN) have given rise to a new evolutionary step of network programmability. Exploiting data plane programming for offloading of application functionality from end-systems to programmable network elements while leveraging the performance of specialized forwarding hardware, capable of processing packets at line-rate throughput in orders up to Terabits per second with low latency, is a recent trend in networking called *in-network computing*. In-network computing has so far been proposed to support various distributed applications, ranging from consensus [8], over caching in distributed key-value stores [12] and network diagnostics, to aggregation functions in data-centric processing including machine learning and graph analytics [16]. It has been shown that in-network computing can yield significant performance improvements by increasing throughput, bandwidth-efficiency, or reducing latency.

In this work, we employ in-network computing to offload *Complex Event Processing* (CEP), a representative of stateful processing from the domain of message-oriented middleware. Traditionally, CEP has been implemented as an overlay of software middleboxes (operators) inferring higher-level knowledge (complex events) by evaluating specific combinations of incoming information (basic events). Packets convey basic events, which are typically comprising structured low-dimensional data, such as sensor data, stock market values for high-frequency trading, or data of network management, such as intrusion-detection systems or anomaly detection. In-network computing is best suited for processing of small data encapsulated in packet headers. Furthermore, CEP systems seek to reduce processing latency, e.g., in high-frequency trading, and to optimize bandwidth utilization. These goals are congruent to the performance gains of in-network computing.

We find [10] that a large class of applications, including CEP, is based on the *middlebox model*, where packets are processed on remote hardware appliances (middleboxes) or in virtualized environments on commodity server hardware (NFV). Although CPUs are cheap and allow for arbitrary packet processing in software, which has become remarkably fast, the middlebox model bears disadvantages: it increases network management complexity by the introduction of additional system components (remote, off-path entities) that can fail, and have to be managed (placement, dynamic configuration). By steering traffic through remote hardware, additional round trips are inherently inflicted, consequently increasing application latency, which is further exacerbated by *service chaining*. Thus, packets are ideally processed *in-situ* at high-performance network elements that they naturally traverse, consequently combining forwarding and processing, which resembles the rationale of in-network computing. Furthermore, the uniform interface of data plane programming, provided by the P4 language, greatly facilitates portability.

© ACM 2018. This is the author's version of the work. It is posted here for your personal use. Not for redistribution.
The definitive Version of Record was published in NetCompute'18, https://dx.doi.org/10.1145/3229591.3229593.

In previous work [3], we have offloaded content-based filtering in publish/subscribe middleware systems from middleboxes (*brokers*) to traditional SDN-enabled switches with a fixed pipeline. While this proved to be sufficient for *filtering* messages at line-rate, we now leverage in-network computing for the in-situ *processing* of complex events in the network, which is challenging due to current limitations of data plane programming for stateful packet processing.

Our contributions in this paper are as follows: we present P4CEP—our early work on an in-network implementation of CEP, including a proof-of-concept compiler from our P4CEP rule specification language to P4. We show that our design contains generic mechanisms, namely window-based aggregation functions (for reasoning over a window of events) and state-machine logic (for event detection), which are highly relevant and can be reused for the in-network implementation of other stateful packet processing applications. We discuss requirements from the perspective of CEP applications and provide feedback on useful data plane programming aspects and experienced limitations. We provide a preliminary evaluation of the performance properties of our implementation, which we deployed on programmable NIC hardware targets. Furthermore, we lay out a roadmap to a distributed in-network CEP implementation, addressing replication and partitioning strategies as well as in-network pre-filtering. The implementation of our prototype of P4CEP is available at: https://goo.gl/MEdPvv [11].

2 BACKGROUND

In this section, we give a brief introduction to data plane programming and describe limitations imposed by P4 and existing hardware targets, followed by an introduction to Complex Event Processing.

2.1 Data Plane Programming with P4

The paradigm of *data plane programming* subsumes the combination of (1) a quasi-standardized, hardware-agnostic domain-specific language (P4) implementing a uniform interface for defining the forwarding behavior of (2) emerging reconfigurable data plane hardware. Key elements of reconfigurable hardware are a parser defining header syntax and semantics and a match-action engine defining the semantics of processing. Both, parser and engine are software-definable, implementing a programmable multi-stage pipeline. Due to space constraints, we omit the description of the P4-language, while referring to [5].

For P4CEP, we consider the following targets: hardware targets residing in end-systems, such as (1) the Netronome Agilio NIC (NFP framework)[15], (2) the NetFPGA platform, as well as (3) reconfigurable ASIC-based (RMT) or FPGA-based (Corsa) data center switches. Furthermore, we consider (4) software-switch implementations, such as the P4 reference switch implementation bmv2 and PISCES, and (5) extended Berkeley Packet Filter (eBPF), which provide fast in-kernel processing within end-systems

In general, hardware targets face inherent limitations [9, 16]. (1) The size of both SRAM and TCAM memory is limited, which imposes bounds on the number of tables, their entries, and other held state. (2) Hardware switches are designed to unconditionally guarantee line-rate throughput. This places an upper bound on processing latency in the order of tens of nanoseconds, consequently bounding the number and complexity of packet operations in each pipeline stage. Hence, P4 models the control flow as an imperative program that specifies the execution sequence through the pipeline as a DAG, which rules out loops and thus renders P4 Turing-incomplete. (3) Stateful packet processing on programmable switches has been shown to be challenging [17]. Unsynchronized, concurrent access can lead to inconsistency effects, such as lost updates, which pose a severe threat for the correctness of stateful packet processing algorithms. The support of atomic register operations is target-dependent and not mandated by P4. However, Netronome's NFP SDK provides a pre-processor pragma for global synchronization of register access.

While reconfigurable switching ASICs are primarily designed for networking tasks like forwarding, FPGAs are much more flexible as they allow for the implementation of custom logic in hardware. To be able to exploit the extended programmability of such targets, $P4_{16}$ includes the `extern` primitive, which provides an interface to functions that are not part of the P4 specification, such as checksum computation and cryptographic operations. They can also be used for synchronization of register access. For instance, the NFP framework allows referencing to external functions written in C and executed in a C-sandbox running on NFP's micro-engines. It natively supports efficient atomic arithmetic operations and has a built-in mutex and semaphore library. Although external functions are a very powerful concept, they break target-independence and possibly lead to unbounded processing latency.

2.2 Complex Event Processing

Complex Event Processing (CEP) is a paradigm to infer the occurrence of situations of interest from basic events [6]. For instance, in the field of algorithmic trading, a situation of interest can be the detection of a leading market signal, whereas the basic event streams contain stock quotes of a stock exchange. An example from the field of sensor fusion is detecting fire (the complex event) by reasoning over measurements of networked smoke and temperature sensors (basic events). In doing so, a CEP system deploys a distributed *operator graph* between event sources and sinks, where each operator detects a specific event pattern in its input streams and emits output events when instances of the corresponding event pattern have been detected.

The pattern to be detected by a CEP operator is typically defined in an event specification language [6] as a *continuous query*. Such a query consists of a number of matching expressions, such as Sequence, AND, OR, NOT, etc., that specify the conditions under which a sequence of input events matches the query. Furthermore, a query can contain aggregation operations such as MAX, MIN, AVG, etc., that are known from stream processing systems [2]. In the fire detection example, these expressions are used to combine measurements of different sensor types (smoke, temperature) and allow the reasoning over their aggregated measurement values. Based on existing languages, we define a meaningful subset for in-network CEP, which we describe in detail in section 3.4.

CEP operators are often stateful, i.e., the processing of one event may influence the internal state of the operator, which in turn influences the processing of subsequent events. Usually, the state

relevant to a CEP operator is limited by a sliding window [13]. A sliding window restricts the infinite sequence of input events in an operator to a subsequence that can match the query. The extents of a sliding window are specified by a *window policy*, which defines the size of a window and its slide, i.e., by how much the window moves from one window instance to the next. In the example, sliding windows enable reasoning over time-series of measurements, e.g., allowing to infer trends. For instance, a fire can be defined to be inferred, when the averages over the last n measurements of the smoke and temperature sensors exceed a given threshold.

Thus, both pattern detection and sliding windows require holding state among the processing of incoming events. For an in-network implementation this consequently mandates stateful processing of packets (events), holding and processing per-packet state as well as inter-packet state. Required consistency semantics on reliability (lost events) and ordering (out-of-order events) in event processing may differ depending on the CEP application. We address consistency implications for P4CEP in section 3.2.

Typically, CEP operators are executed on end-systems. Typical performance figures show average processing latencies of about 200 µs, excluding the end-system's network stack latency, for detecting sequences of two states [7], which is the simplest form of stateful processing. In terms of throughput, highly parallel implementations of CEP operators on multi-core CPUs can reach up to 218,000 events/second for more complex patterns [13].

3 P4CEP: DESIGN & IMPLEMENTATION

In this section, we present our underlying system model, the P4CEP-compiler, and the P4CEP rule specification language along with an example illustrating our design.

3.1 System Model

P4CEP's system model, illustrated in Figure 1, assumes a set of **end-systems** that are interconnected by a set of programmable network processing elements (**P4CEP-targets**), forming a data plane topology. End-systems that host event-based applications (CEP end-systems) are differentiated into **event sources**, which observe **basic events** and disseminate them, e.g., networked-sensors or server reporting performance metrics or log data, and **event sinks**, which receive and react to **complex events**. P4CEP-targets (listed in §2.1) implement two types of functions: (1) network functions, which co-exist with CEP (**Co-NF**, dark-shaded), typically simple forwarding of non-CEP packets, and (2) CEP functions (light-shaded), which can be divided into **window operators**, which store the n last values of header fields in a FIFO-manner and offer aggregation functions over these values, and the **event detection engine**, which detects complex events based on a state machine implementation. Note that without loss of generality we henceforth consider just a single P4CEP-target and discuss the distribution of CEP onto multiple targets in the roadmap (§6). The **P4CEP runtime component** implements a control plane interface for an operator to P4CEP-targets. Besides deploying compiled P4CEP programs, it handles all runtime tasks: updating P4 table entries and state transitions in the CEP engine as well as acquiring statistics and other monitoring data from the targets.

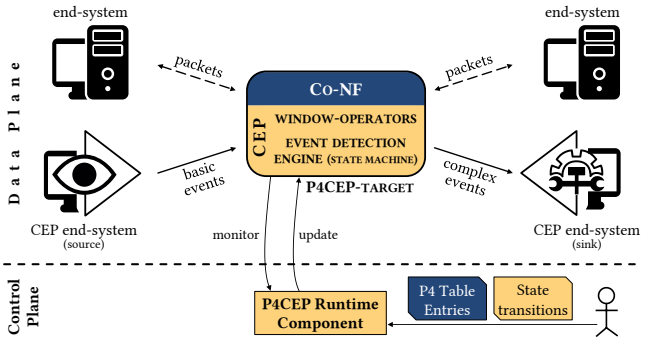

Figure 1: The P4CEP system model.

3.2 The P4CEP Workflow and Compiler

P4CEP's design-workflow is illustrated in Figure 2. It is mainly composed of our P4CEP compiler and an unmodified **P4 compiler** chain, consisting of a target-independent and target-dependent compiler, as well as **target-dependent toolchains**. Currently, we support target-specific external functions for the Netronome NFP target. The user-input to the P4CEP compiler (**CEP design config**) consists of P4-definitions of header fields and parser instructions for packets that are to be interpreted and processed as basic events as well as declarations of window operators and event definition rules, which describe how complex events are derived from basic events. We introduce our CEP rule specification language later (§3.4).

From these definitions, the **P4CEP compiler** creates corresponding P4 source code, supporting both P4 versions ($P4_{14}$ and $P4_{16}$). It comprises definitions of registers (inter-packet state, henceforth called global state), metadata structures (per-packet state), auxiliary tables for (multiple) windows, implemented as register ring-buffers, and (multiple) state machines, each associated with a distinct complex event pattern to detect. When required, our implementation ensures consistency of global data to avoid inconsistency effects such as lost updates, which would translate to unprocessed events or partially, i.e., non-atomically processed events. We protect global state from concurrent access by implementing critical sections in atomic control flow blocks in $P4_{16}$ and using NFP's atomic register access or it's synchronization library, as described in section 2.1. This inflicts execution overhead in terms of additional latency (***Limitation 1***).

The main logic is implemented in the CEP ingress **control flow**, which upon receiving a packet that transports basic events executes a sequence of actions implementing **window operations**, followed by state machine executions. Due to space constraints, we provide only a high-level description. We make a complete and annotated control flow example available in our P4CEP release [11]. First, the current instance count (ringbuffer head pointer) of the window is read from a register and incremented with overflow handling. One drawback of P4 (***Limitation 2***) is that registers cannot be directly referenced in arithmetic operations or as table keys. Thus, register values have to be copied into dedicated intermediate metadata fields and back, which bloats code space and execution overhead. Then, the header field value of the current event and instance count are stored in registers, i.e., are persisted in the window. For applying

the aggregation function on the window, our compiler has to unroll the window iteration, due to P4's lack of loops (***Limitation 3***). The aggregate value is stored in a metadata field m_{aggr}, as is the iteration counter m_{iter}. For each value r_i in the window, r_i has to be copied from the window register to a metadata field m_i. Then, the aggregation function is applied on m_{aggr}, referencing m_i. This sequence is repeated for all windows.

A pattern of basic events defines a complex event whose detection is modeled as a deterministic finite **state machine** $C = (\Sigma, S, s_0, \delta, F)$, as illustrated in Figure 3. It consists of a sequence of basic events (input symbols $x \in \Sigma$), where typically a basic event is specified by **predicates** P_x (simple or compound) on packet header fields, as we describe in greater detail in §3.4. For each pattern, the following actions are executed sequentially: all packet predicates are evaluated. If P_x evaluates to true, an id associated to that predicate ($P_x \rightarrow x$) is stored in a metadata field m_x. Then, the state machine is executed by first acquiring the current state $q \in S$ by copying from a register to a metadata field m_q, followed by performing a lookup with the key-pair $<m_q, m_x>$ on the **transition table**—a P4 table encoding δ. Upon a match, the returned value pair $<q_n = \delta(q, x),\ b_{is_accepting}>$ is written to registers if q_n is not an accepting state ($\neg b_{is_accepting} \equiv q_n \notin F$). If it is an accepting state, the state machine is reset, i.e., q_n is set to the initial state s_0, and the return value for the complex event is set, encoded in a header field, before the packet is sent to registered CEP sinks using the P4-`resubmission` mechanism. P4CEP allows the detection of multiple complex events by sequential execution of the corresponding state machines.

The P4 code generated by the P4CEP compiler is merged with the user-provided P4 program source file, which implements co-NF functionality, using P4's `include`-primitive. Additionally, runtime configuration files hold table entries and can be (re-)deployed at runtime by the P4CEP runtime control plane component. They are created in a target-compatible format by the P4CEP compiler and given as user-input for the co-NF part, respectively.

3.3 Limitations for Stateful Processing

Here, we discuss the encountered limitations of P4 and their implications for stateful packet processing. Additional to the aforementioned Limitation 1 (synchronizing access to global state), Limitation 2 (no direct operations on registers), and Limitation 3 (lack of a loop construct), another limitation lays in the fact that conditions in P4 can be *only* used within the control flow, not within actions (Limitation 4). Furthermore, $P4_{14}$-actions cannot be directly executed within a control flow, but have to be indirectly executed by using P4's `apply`-primitive to perform a lookup on an empty dummy-table where the action to be executed is specified as the default action (Limitation 5). We realize that some of these limitations are inherent design trade-offs in creating P4, which seemed to be driven by satisfying the intricate requirements of switch hardware architectures [9] to maintain line-rate processing, for instance ruling out loops (Limitation 3), rather than having stateful packet processing in mind. However, we observe that the evolution of P4 with $P4_{16}$ facilitates stateful packet processing, e.g., by the introduction of the atomicity primitive (Limitation 1) and corrects other seemingly unnecessary limitations like the action indirection (Limitation 5).

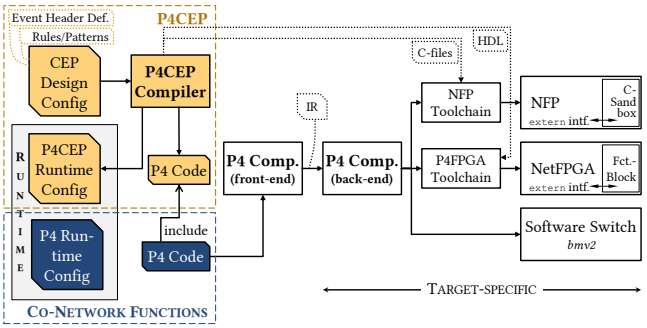

Figure 2: P4CEP workflow: design-time components and source files involved in building P4CEP for different targets.

3.4 P4CEP Rule Specification Language

The specification of the CEP-functionality (Listing 1) is split into the definition of window operators and event definition rules, which are compiled to a corresponding state-machine (Figure 3). Note that for sake of illustration, we simplified the example by using headers of common layer 3 and 4 protocols. Instead, custom headers possibly encoding type and value of basic events could be used. Moreover, we used simple instead of composed predicates.

As a side effect, the example shows that some co-NFs, such as anomaly/intrusion-detection, typically relying on sums or counts of specific (sequence of) packets, can be mapped directly to CEP-functionality. The illustrated example (Listing 1, Figure 3) enables the detection of the following anomaly pattern: a large IPv4 packet and an HTTP-packet, followed in sequence by an UDP-datagram or the sum of total lengths over all last $n = 8$ seen IPv4 packets exceeding 6 KB.

The following concepts of the P4CEP rule specification language are used to express such patterns:

`Window` definitions consist of the window **`size`** n and a field reference whose **`value`** is to be stored within that window (last eight IPv4 total lengths in the example). Field references are simple references to P4 headers or metadata that must have been parsed by the P4 program and thus be defined either as a CEP event header or as a co-NF header. Windows can be referenced by name within a pattern of a complex event definition or as its return value.

The definition of **`complex_events`** is structured as follows: (1) A return **`value`** to be set in a complex event packet, sent in case of

Listing 1: Exemplary P4CEP-rule definition of a window and a sequential pattern, composed of predicates on simple L3/L4-packets and on the window.

```
window sample_wnd {
  size 8
  value ipv4.totalLen
}
complex_event sample_evt {
  value sum(ipv4.totalLen)
  strategy skip-till-next-match
  pattern ([ipv4.totalLen > 500] && [tcp.dstPort == 80]) ;
          ([sum(sample_wnd) > 6000] ||
           [ipv4.protocol == 17])
}
```

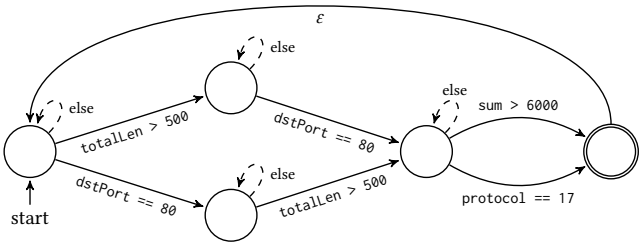

Figure 3: Generated finite state machine detecting complex event patterns as specified in Listing 1.

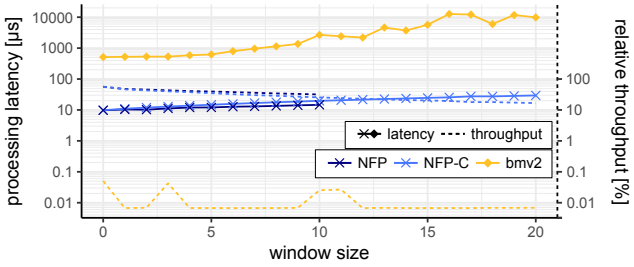

Figure 4: P4CEP's performance for increasing window sizes on an NFP smart-NIC and bmv2.

detection. This can be (1a) any valid P4 expression (static expression, field reference), or (1b) a reference to an aggregation function over a window (e.g., `sum(sample_wnd)`), or over a header field (see example).
(2) A **`strategy`** specifying the state transition if an incoming basic event does not match any predicate:
(2a) `skip-till-next-match` performs a transition to the same state, i.e., ignores the event, (`else`-branches in Figure 3 of the example); (2b) `strict` resets the state-machine by setting the next state to the initial state.
(3) A **`pattern`** of basic events defining a complex event. Basic events are specified by simple or compound predicates. A predicate can be any valid P4 condition on one or more field references, or a condition on an aggregation function over a window or over a header field. Predicates are demarcated by square brackets and combined to patterns using the following logical operators. (3a) Sequence `;`: the left predicate must hold true before the right. (3b) Conjunction `&&`: both predicates must hold true (in any order). (3c) Disjunction `||`: one of the predicates must hold true.

Finally, the following **aggregation functions** on windows or field references are currently supported: `sum`, `min`, `max`, and `count` (which counts how many times a predicate was true). We plan to implement `average`, which is not straightforward due to P4's missing float support and lack of a division operator, but can be approximated by fix-point and bit-shift operations on windows of sizes 2^n.

4 PRELIMINARY EVALUATION

In this section, we provide a first impression of P4CEP's practicability, evaluated on state-of-the-art P4 targets: a Netronome Agilio smart-NIC with two 10GbE-ports, on which we run a pure P4 implementation of P4CEP (**NFP**) and an optimized version employing `extern`-functionality through NFP's C-sandbox (**NFP-C**), as well as the software-switch **bmv2**.

We first provide a baseline analysis of a simple P4 program, implementing stateless forwarding based on parsing layer 2–5 headers of smallest-sized packets. We measured a baseline latency including serialization delay of 6.8 µs for NFP(-C) and 475 µs for bmv2, respectively. Baseline throughput is full line-rate (≈14.88 million packets per second (Mpps) for 10GbE) for NFP(-C) and 0.08% thereof (≈ 12 Kpps) for bmv2, denoted as relative throughput B_p.

We approximate the latency for CEP processing l_p by hardware-timestamping the egress of basic events, sent at a rate of 500 pps, and the ingress of consequently detected complex events. Propagation and serialization delay are negligible. Figure 4 shows the mean performance over 60,000 samples for varying window sizes n and one complex event pattern to be detected consisting of two basic events with one simple predicate each. In the depicted interval $0 \leq n \leq 20$, we measured a processing latency of $9.8\,\mu s \leq l_p \leq 29.5\,\mu s$ and relative throughput of $56\% \geq B_p \geq 16\%$ for NFP-C. The pure P4 implementation (NFP) performs slightly better, showing low overhead for the `extern`-mechanism. However, for $n > 10$, the generated P4 code exceeds the size limit of NFP and is hence rejected. With window handling implemented in the C-sandbox, NFP-C scales up linearly with $\Delta l_p \approx 1\,\mu s$ per iteration, up to $l_p \approx 969\,\mu s$ for $n = 1000$, where B_p drops to 0.4% (≈60 Kpps). Overall, NFP(-C)'s standard deviation of l_p (jitter) is quite low (tens to hundreds of nanoseconds), as is its deviation of throughput (≈0.02%).

While bmv2 has no code size restrictions, its performance is significantly worse. Starting with $l_p \approx 512\,\mu s$ and $B_p \approx 0.05\%$, it also shows inferior scalability properties, exceeding $l_p \approx 10\,ms$ for $n > 15$. Jitter is also increased.

We conclude that already in its early stage, P4CEP achieves good performance in particular on hardware targets. We expect even higher performance on more optimized implementations and targets (NetFPGA, PISCES) in the future.

5 RELATED WORK

In this section, we briefly discuss related work other than already mentioned.

SNAP [1] is a network-centric, high-level language for network programming, extending stateless packet processing with primitive stateful operations. It offers a stateful network-wise abstraction for packet processing by enabling access to a persistent global array in control programs, while making the distribution of that state in the data plane transparent to the programmer. Packet processing in a SNAP program depends on the current state of the network, which is held in variables within the global array and which is possibly changed as a result of processing. SNAP considers events as non-frequent changes in the network, such as traffic changes and failures, that trigger recompilation of the network program in the control plane.

Stateful NetKAT is a language for event-driven network programming [14], extending the NetKAT language by mutable state. It is similar to SNAP, however, it rather focuses on applying consistent network updates where consistency properties hold during the transition between two network configurations, which is triggered in response to events.

While both approaches enable network-centric stateful packet processing, P4CEP is tailored for complex event processing with a generic notion of events that includes but is not limited to network events. Since P4CEP is based on P4, it is more lightweight while still leveraging the full expressiveness of P4.

OpenState [4] is an implementation of a generic state machine in the data plane of an OpenFlow switch. It maps the state machine execution to a fixed match-action pipeline consisting of two tables, holding the current state and transitions on a per-flow basis. Since state is held in flow table entries, OpenState requires a custom OpenFlow instruction to be able to update the state after a transition. While the authors provide a modified implementation of an OpenFlow software switch, there are no hardware implementations. P4CEP uses P4 to implement its state machine logic without the need for modifications of software or hardware.

6 CONCLUSION & ROADMAP TO A DISTRIBUTED IN-NETWORK CEP

In this paper, we presented P4CEP, an in-network implementation of Complex Event Processing, and showed its practicability by experiment.

Based on our experiences gained from designing P4CEP, we argue that in-network computation, in particular for stateful processing, poses an interesting research question regarding the trade-off between portability (target-independence) and leveraging programmability, including application-specific custom functions (introducing target-dependence). For instance, while it was our design goal to stay target-independent through exclusive use of a uniform data-plane programming language (P4), implementing custom functions enabled mitigation of current limitations of P4 and enriched functionality at the cost of becoming target-dependent. In summary, we identified the following limitations of P4 for stateful in-network computing: (1) the overhead of state synchronization, (2) the inability to directly handling global state in registers, and (3) the indirection of action invocations. Although we understand some limitations as deliberate decisions in P4's design, we see great potential for constructs like bounded loops or more efficient primitives for synchronization. To further explore this trade-off, we plan to adopt more powerful operators from CEP and stream processing.

Another means to counter the observed limitations and to increase resource-efficiency is to leverage the distribution of in-network computation. For CEP, distribution bears the following benefits: (1) The early filtering of needless basic events, i.e., events that are not part of any complex event pattern, leads to reduced load in the event detection engine and to increased bandwidth-efficiency. Thus, we plan to adopt our in-network content-based publish/subscribe approach [3] using data plane programming to increase the expressiveness of filtering. (2) The disaggregation of event detection to multiple P4CEP-targets allows for the parallelization of CEP and hence increases performance. This requires strategies for replication and partitioning of basic events and a concept of splitting window handling and event detection, which we plan to develop. Ideally, distribution compensates some of the observed target limitations, such as code-size and pipeline-depth limits.

REFERENCES

[1] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 29–43. https://doi.org/10.1145/2934872.2934892

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (June 2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[3] S. Bhowmik, M. A. Tariq, B. Koldehofe, F. Dürr, T. Kohler, and K. Rothermel. 2017. High Performance Publish/Subscribe Middleware in Software-Defined Networks. *IEEE/ACM Transactions on Networking* PP, 99 (2017), 1–16. https://doi.org/10.1109/TNET.2016.2632970

[4] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. 2014. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev.* 44, 2 (April 2014), 44–51. https://doi.org/10.1145/2602204.2602211

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[6] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS '10)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/1827418.1827427

[7] Gianpaolo Cugola and Alessandro Margara. 2012. Complex Event Processing with T-REX. *J. Syst. Softw.* 85, 8 (Aug. 2012), 1709–1728. https://doi.org/10.1016/j.jss.2012.03.056

[8] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM, New York, NY, USA, 5:1–5:7. https://doi.org/10.1145/2774993.2774999

[9] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling Packet Programs to Reconfigurable Switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 103–115.

[10] T. Kohler, F. Dürr, C. Bäumlisberger, and K. Rothermel. 2017. InFEP - Lightweight Virtualization of Distributed Control on White-box Networking Hardware. In *2017 13th International Conference on Network and Service Management (CNSM)*. 1–6. https://doi.org/10.23919/CNSM.2017.8256045

[11] Thomas Kohler and Marius Maaß. 2018. P4CEP NetCompute Release. https://goo.gl/MEdPvv

[12] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 795–809. https://doi.org/10.1145/3037697.3037731

[13] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. 2017. SPECTRE: Supporting Consumption Policies in Window-based Parallel Complex Event Processing. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*. ACM, New York, NY, USA, 161–173. https://doi.org/10.1145/3135974.3135983

[14] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. 2016. Event-driven Network Programming. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 369–385. https://doi.org/10.1145/2908080.2908097

[15] Open-NFP.org. 2018. Open-Netronome Flow Processor (NFP) - Netronome. https://open-nfp.org/resources

[16] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*. ACM, New York, NY, USA, 150–156. https://doi.org/10.1145/3152434.3152461

[17] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/2934872.2934900