Window-based Parallel Operator Execution with In-Network Computing

Bochra Boughzala
University of Groningen
Groningen, The Netherlands
b.boughzala@rug.nl

Christoph Gärtner
Technical University of Darmstadt
Darmstadt, Germany
christoph.gaertner@tu-darmstadt.de

Boris Koldehofe
University of Groningen
Groningen, The Netherlands
b.koldehofe@rug.nl

ABSTRACT

Data parallel processing is a key concept to increase the scalability and elasticity in event streaming systems. Often data parallelism is accomplished in a splitter-merger architecture where the splitter divides incoming streams into partitions and forwards them to parallel operator instances. The splitter performance is a limiting factor to the system throughput and the parallelization degree. This work studies how to leverage novel methods of in-network computing to accelerate the splitter functionality by implementing it as an in-network function. While dedicated hardware for in-network computing has a high potential to enhance the splitter performance, in-network programming models like the P4 language are also highly limited in their expressiveness to support corresponding parallelization models. We propose P4 Splitter Switch (P4SS) which supports overlapping and non-overlapping count-based windows for multiple independent data streams and parallelizes them to a dynamically configurable number of operator instances. We validate in the context of a prototypical implementation our splitting strategy and its scalability in terms of switch resource consumption.

CCS CONCEPTS

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

KEYWORDS

Data Parallelism, In-network Computing, Load Balancing, Complex Event Processing (CEP), P4 Language, Data Plane Programming.

1 INTRODUCTION

Today's high-speed communication networks interconnect a growing number of data sources. Observed event rates are consequently ever increasing. A study on managing financial data reports around 18 billion notifications per day and an average of 700,000 per second with a peak rate of more than 1 million events/sec [9]. Data parallel processing is an important concept to enable improved elasticity and increased scalability for handling the growing data rate in distributed event-based systems such as Complex Event Processing (CEP) [5]. With data parallelization, multiple functionally identical operators are deployed in the system to perform the same computational task e.g., pattern matching on a subset of the events, called *windows*. Often the parallel operator execution is based on the splitter-merger model [21, 23]. In this architecture, one important limiting factor of the overall system performance is the splitter as it is responsible for partitioning the incoming event streams and load balancing the identified event partitions towards the operator instances. The event partitioning operation might be window-based [8] or key-based [11]. In the case of window-based operators, the splitter defines the event partitions by applying *windowing semantics* to determine the start and the end of the windows and delivers them to a set of operators. Therefore, the splitter represents a bottleneck as it determines the maximum achievable throughput at which the splitter-merger architecture can process events. Even with compute-intensive operators involving longer processing time, the splitter remains the bottleneck since decreasing the service time requires more operator instances. Hence, the splitter imposes an upper bound for the system throughput.

In this paper, we aim at exploiting novel methods of in-network computing [4] for supporting data stream partitioning and load-balancing. We propose a network-centric approach by moving the splitter function to new programmable switches [3] in order to benefit from their performance. Due to the development of in-network computing and Software-Defined Networking (SDN) [17], programmable data planes and their corresponding programming language P4 [2] allow the reconfiguration of switching devices to deliver customizable in-network functions at high-speed, e.g., Intel Tofino 3 delivers 25,6 Tbps throughput [13].

While in-network computing nodes offer the potential of higher performance for the splitter function, programming the *parallelization models* in such devices is challenging. The P4 language is limited in its expressiveness and support for stateful processing. Moreover, it is important for the splitter function to be adaptive to the dynamic nature of parallel operator execution. Therefore, we propose P4SS to address these challenges. We show methods to create multiple parallelization models and support their dynamic updates for the reconfiguration of *window specifications* or changing the number of the operator instances. In summary, we make the following key contributions. We explain how the splitter function can be executed in the network while still complying both with the splitter-merger model and the SDN principles. We describe how P4 abstractions can

be leveraged to build an in-network splitter function supporting distinct windowing semantics, we present the design of P4SS and we give the algorithmic description of two windowing methods (i) *overlapping i.e., sliding* and (ii) *non-overlapping i.e., tumbling* count-based windows. Finally, we provide an evaluation of our implementation using a virtual testbed.

The remainder of the paper is organized as follows. In Section 2, we present the system model and the problem description. Then, we explain the design of our solution in Section 3. Next, we provide an evaluation of our implementation in Section 4. In Section 5, we discuss the related works. In Section 6, we give the conclusion and future work.

2 SYSTEM MODEL AND PROBLEM DESCRIPTION
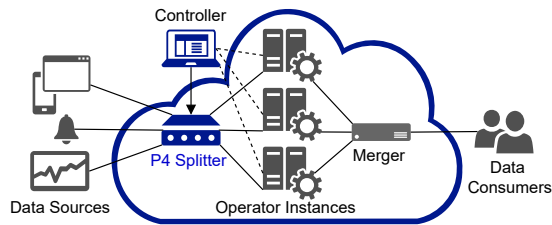
2.1 System Model

Figure 1: P4-based Data Parallelization Framework.

Data parallel operator execution comprises three components, the splitter, a dynamic set of operator instances and the merger [23]. The splitter executes a parallelization model to split incoming data streams and assigns them to dedicated operator instances. The streaming system can dynamically adapt the parallelization degree by allowing the operator execution environment to dynamically add new operator instances.

In this paper, we assume a network-centric approach for data parallel operator execution. While operator instances are still executed on traditional server nodes, the splitter is executed on the network path by a dedicated network switching device. The process of deploying a parallelization model requires to perform switch reconfigurations that support the coexistence of multiple parallelization models with corresponding window specification. Traditionally, network switches are based on fixed-functions Application-Specific Integrated Circuit (ASIC) with predefined behaviour according to standard protocols. Introducing customized in-network functions in such devices is not possible or would require highly specialized application interfaces mostly known to the hardware vendor. However, programmable switches [13] allow their reconfiguration with the P4 programming language. Then the deployed network function, when mapped to the P4 switch model, can leverage line-rate performance characteristics very close to the traditional ASICs.

In our model (Figure 1), the splitter function is executed by a P4 programmable switch. The P4 splitter is configured by an external controller via the control plane interface. According to SDN, the controller has a global view on the system. It can monitor the dynamic workload of the operator instances and collect their operational status. Therefore, according to a feedback loop the controller can adapt the parallelization degree by updating the configuration of the window specification in P4SS when the workload increases or an operator is overloaded or down. The resulting parallelization model is pushed to the splitter which has the P4 program running.

A P4 program typically applies at the packet level. In our notion, we assume that an event corresponds to a packet. The switch receives multiple input data streams arriving from different data sources in the form of event packets on dedicated ingress ports. The basic switch model comprises a set of ingress ports, a programmable match-action pipeline and a set of egress ports. The P4 program starts by an ingress parsing stage of the packet headers. Then, an ingress match-action pipeline composed of match-action tables is executed. A match-action table is a lookup table populated by the controller with match-action entries consisting of (i) a lookup key on which the match is performed (ii) an action data which is executed when there is a table lookup hit. The ingress pipeline is ended by a deparser to reconstruct the packet header fields that were potentially altered during the match-action process. At this point, the event packet is either assigned to an egress port in the case of unicast, or to a multicast group in the case of multicast. An egress processing is then applied to the packet or packet replicas through egress parsing, egress match-action pipeline then egress deparsing. Finally, the packet exits the P4 switch and is routed to the identified operator destinations.

2.2 Problem Description

The splitter divides incoming data streams according to a given window specification [8, 26]. There are time-based, count-based and marker-based windows [7]. The resulting windows can be overlapping or non-overlapping. While splitting the data stream into windows, the splitter assigns each window to an operator instance. Therefore, the splitter plays the role of a load balancer. It is important that the splitter provides guarantees of even load distribution among the operator instances. Also, it needs to maintain a per-stream per-window consistency so that for each independent data stream all events within the same window are sent to the same operator. With this property, we ensure the correctness of the results of operators. The splitter must be adaptive to changes in the system and to new configurations, e.g., instantiating a new parallelization model with its corresponding window specification, or scaling up or down the number of operator instances for a given stream. The dynamic updates are important since it would allow reconfiguring the splitter at run-time without requiring to reload a new P4 program.

Implementing expressive windowing semantics requires the ability to keep a persistent or semi-persistent state within the splitting function. While this is widely supported in general purpose programming languages such as GO or Java, this is not the case with programmable data planes and their programming language P4. In such devices, the high-performance comes at the cost of losing some programming capabilities. First, in-network computing nodes have limited on-chip memory. Second, the hardware primitives are very simple to ensure that the processing remains efficient and within a bounded latency. For example, arithmetic operations such

as multiplication, division, and floating points operations in general are not supported. It is also not possible to have a loop in a P4 pipeline. While stateful processing is limited with programmable data planes, it is still possible to a certain extent by the means of stateful objects such as *Registers* and *Counters*. Unlike stateless objects which have their state reinitialized for each incoming packet, stateful objects keep their state between packets.

3 P4SS DESIGN

In this section, we present the design of P4SS (pronounced pass), we explain its data plane pipeline processing and we describe the different windowing semantics it supports.

3.1 Design Decisions

In P4SS, each data stream has its associated parallelization model installed in the switch. A generic parallelization model is defined by a window specification with a window size n and a window shift δ, in addition to the maximum number of operator instances which are assigned to this particular stream. The P4SS program exposes these parameters to the controller via the control plane interface, e.g., gRPC-based P4Runtime API [25]. For a specific parallelization model of a given stream, the controller adds at run-time the proper match-action entries to instantiate the values of these parameters in the action data. Therefore, an update to the windowing scheme does not require recompiling a new P4 program. Since each event is encapsulated in a packet whose header can be processed by the P4 switch, we define the event header (see Listing 1) to include a stream type e.g., weather or stock price, a sequence number and a timestamp. Hence, by having the stream type retrieved from the event packet and used as part of the lookup key, P4SS can maintain independent parallelization models for each separate input stream. In our current prototype of P4SS, we support *tumbling* (Algorithm 1) and *sliding* (Algorithm 2) count-based windows. For load balancing the identified windows among the operator instances, P4SS uses a Round Robin (RR) mechanism. Note that other scheduling techniques such as Weighted Round Robin (WRR) are also possible with P4.

Listing 1: Event Header

```
typedef bit<16>  type_t;
typedef bit<16>  seq_num_t;
typedef bit<64>  timestamp_t;
header event_hdr_t {
    bit<16> type;
    bit<16> seqNum;
    bit<64> timestamp;
}
```

3.2 P4SS Data Plane Pipeline

Upon the reception of an event packet, the event type is extracted from the event header then mapped to a stream ID through a lookup table, where also the window size and the window shift are set in the action data. Next in the pipeline, we retrieve the number of operator instances assigned to this particular stream. Thereby, we can change the window specification and the number of operators at run-time and independently. Implementing the windowing semantics requires stateful processing which is limited in P4 to the use of registers. With registers we can keep the state beyond the
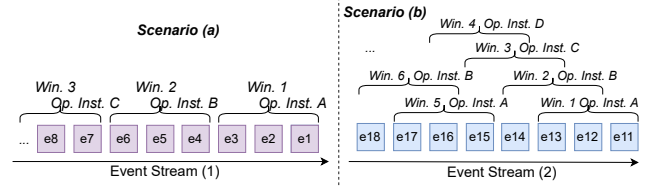

Figure 2: Count-based non-overlapping (scenario a) and overlapping (scenario b) Windows. Event Stream 1 has a window specification with $n = 3, \delta = 3$ and 3 operator instances A, B, C. Event Stream 2 has a window specification with $n = 3, \delta = 1$ and 4 operator instances A, B, C, D.

lifetime of a packet. We define three main register arrays where the Stream ID is used as an index : (1) *position_reg*[i] for maintaining the event current position within the latest active window, (2) *operator_reg*[i] for keeping track of the current operator instance receiving the events, and (3) *overlap_reg*[i] is for the overlap degree of how many windows are currently active in the case of overlapping windows.

P4SS starts the processing at the initial state where all registers and metadata values are set to zeros and the match-action tables are pre-configured with a set of match-action entries added by the controller via the control plane interface. For overlapping windows, multicast groups are created through the Traffic Manager (TM) API (this is outside of P4). The objective of P4SS is to determine for each received event to which operator or set of operators it must be sent. To resolve this decision, the main program (Figure 3) first retrieves the window type that will be applied. In the case of *tumbling windows* ($n \leq \delta$) (Figure 2.a), the event goes through the unicast path in the P4 pipeline. An Operator ID is identified then a lookup in the operators table allows to map an operator ID to an Operator IP. Next, IP routing is resolved to reach the operator instance which can be located anywhere in the data center. Note that this is a design choice to bring more flexibility into the operator placement and network topology. By decoupling the operator instances from their location, P4SS does not require a specific network topology. However, for simplification in a controlled environment, the Operator ID can be mapped directly to an egress port, where operator instances are servers directly connected to P4SS. In the case of *sliding windows* ($n > \delta$) (Figure 2.b), based on the values of the registers *position_reg*, *operator_reg* and *overlap_reg*, a multicast group ID is identified and the event goes through the Packet Replication Engine (PRE) where more copies of the event are created and sent each to the operator instances involved with the currently active windows.

3.3 P4SS Windowing Semantics

Algorithm 1: For count-based *tumbling* windows, we maintain the event position within a window in *position_reg* register. The current position gets incremented for each received event of the stream until it reaches the window size. Then, the event position gets re-initialized. Additionally, we keep the current operator identifier within the *operator_reg* register. As long as events are received within an ongoing window, the operator ID i remains the same and
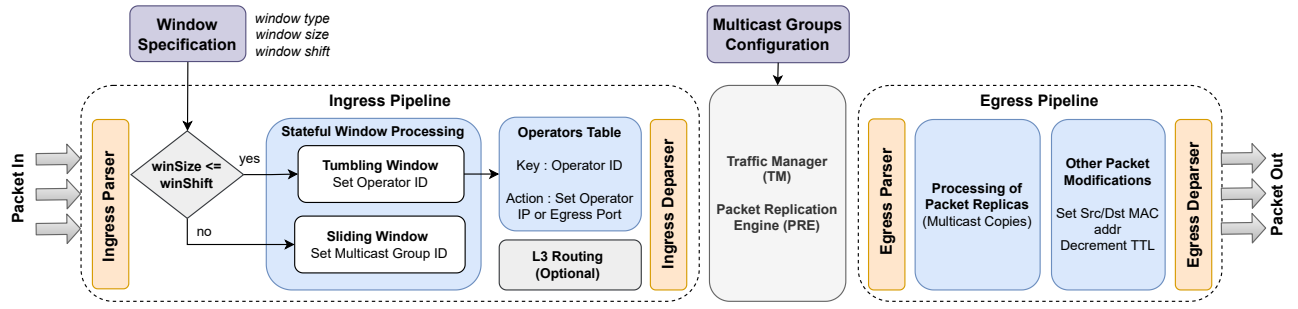
Figure 3: P4SS Data Plane Pipeline Design.

these events are forwarded to the same operator instance. When the window size is reached, the operator ID is incremented in a way that the new window will be sent to operator ID $i + 1$. When the maximum number of operator instances is reached, the operator ID is re-initialized to zero. Thereby, P4SS assigns identified windows in a RR fashion to the operator instances.

Algorithm 1 Count-based Tumbling Windows

Input: Stream ID, window size, window shift, maximum number of operators.
Output: Operator ID.
register_read(position_reg[stream_id], position)
register_read(operator_reg[stream_id], operator_id)
if $n \leq \delta$ **then**
    if $position > \delta - 1$ **then**
        $position \leftarrow 0$ ▷ Start of new window
        $operator\_id \leftarrow operator\_id + 1$ ▷ Next operator
        if $operator\_id \geq max\_operators\_num$ **then**
            $operator\_id \leftarrow 0$
        end if
    end if
    $position \leftarrow position + 1$
    register_write(position_reg[stream_id], position)
    register_write(operator_reg[stream_id], operator_id)
end if

Algorithm 2: With count-based *sliding* windows, we have overlapping windows resulting in events being forwarded to multiple operator instances at the same time with the guarantee for each active window to contain consistently a number of events according to its size. For example, with a window size $n = 3$ with a shift $\delta = 1$, a new window is created for each received event and each active window must end after 3 events (Figure 2.b). We define a parameter of the maximum overlap equal to $n - \delta$ and a current overlap counter which is increased iteratively during a ramp up phase until it reaches the maximum overlap degree. Consider if we have four Operators A, B, C, D. At the start, the current overlap is zero and the first packet is sent only to Operator A. With the second packet the overlap is 1 and the packet is sent to Operators A and B. Then the overlap is 2 and the third packet is sent to Operators A, B and C. Next packet is sent to Operator B, C and D and so on. To express this behaviour in P4, we maintain a truth table of the overlapping windows to multicast groups mapping, and we populate it with a set of entries to program all the possible combinations using as

a key the current overlap degree, the latest operator ID and the stream ID. The multicast groups corresponding to each case must be configured in the TM beforehand.

Algorithm 2 Count-based Sliding Windows

Input: Stream ID, window size, window shift, maximum number of operators.
Output: Operator ID or Multicast group ID.
if $n > \delta$ **then**
    $max\_overlap \leftarrow n - \delta$
    register_read(overlap_reg[stream_id], curr_overlap)
    register_read(operator_reg[stream_id], operator_id)
    overlapping_windows_to_mcast_grps_table($stream\_id$, $operator\_id, curr\_overlap$)
    if $\delta == 1$ **then**
        $operator\_id \leftarrow operator\_id + 1$
        if $operator\_id \geq max\_operators\_num$ **then**
            $operator\_id \leftarrow 0$
        end if
    end if
    register_write(operator_reg[stream_id], operator_id)
    if $curr\_overlap < max\_overlap$ **then**
        $curr\_overlap \leftarrow curr\_overlap + 1$
    end if
    if $curr\_overlap \geq max\_operators\_num$ **then**
        $curr\_overlap \leftarrow max\_operators\_num - 1$
    end if
    register_write(overlap_reg[stream_id], curr_overlap)
    register_write(operator_reg[stream_id], operator_id)
end if

---

4 EVALUATION

We developed a prototypical implementation of P4SS using the P4$_{16}$ language [6] and the behavioral model *bmv2*, a software switch as a P4 target. We tested our prototype on Mininet [19], a python-based network emulator for experimenting with SDN. Within Mininet, we create a topology as depicted in Figure 4. The splitter is connected to $k + n$ hosts : k hosts used as data sources generating independent event streams each with a different event type, and the other n hosts represent the operator instances. Note that for each input stream the packets contain a unique and continuously increasing sequence number. Moreover, in our testing environment the packets are delivered in a lossless and in-ordered fashion.
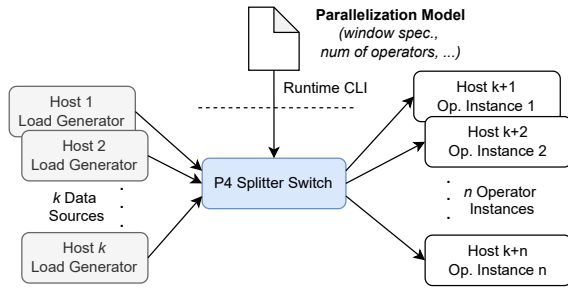
Figure 4: Testing P4SS with a Mininet-based Virtual Testbed.
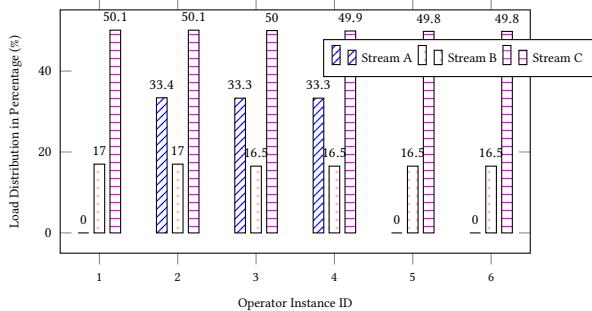
4.1 P4SS Properties

Figure 5: P4SS Load Distribution.

As a preliminary evaluation of P4SS, we experiment with 3 data sources, each with a different window specification and a specific assignment of the operator instances. The goal is to evaluate the key properties of our solution (i) *correctness*, (ii) *adaptability* and (iii) *even load distribution* as explained in Section 2.2. First, we want to validate the *correctness* of the splitting function. In our testing scenario, Data Source 1 generates Stream A with a window specification of $n = 3$ and $\delta = 3$ assigned to 3 operator instances 2, 3 and 4. Data Source 2 generates Stream B with $n = 5$ and $\delta = 5$ assigned to 6 operator instances 1 to 6. Data Source 3 generates Stream C with $n = 3$ and $\delta = 1$ assigned to the same 6 operator instances as Stream B. We validate the *correctness* of the splitting behaviour by verifying for each independent stream the sequence numbers in the event header of the subset of events within each window received at each operator. P4SS performs the event partitioning of the data streams correctly and the windows are delivered to the operator instances according to the window specification and the RR scheduling towards the configured operator instances is correct. Second, P4SS *adaptability* to the dynamic configurations was proven to work as we were able to install the parallelization models effectively at run-time. Finally, as presented in Figure 5, P4SS delivers an even load distribution among the operator instances both in the case of non-overlapping (Stream A and Stream B) and overlapping (Stream C) windows. Hence, through the software emulation of P4SS we verified its key properties and essentially its feasibility as a P4 function.

4.2 Resource Usage

While increasing the parallelization degree, one critical aspect is the switch resource consumption in terms of the number of entries, the registers and the multicast groups that need to be provisioned. The relevant parameters which influence the resource consumption are mainly the number of stream types, the number of operator instances, the window type, i.e., overlapping or non-overlapping windows, and the overlapping degree for the overlapping windows. The size of the register arrays depends on the number of stream types. For both overlapping and non-overlapping windows, the window size does not affect the number of entries which is rather dependent on the number of operator instances in the system. An entry per operator is needed to map an operator ID to an egress port and two more entries are required per stream (i) one for setting the maximum number of operators and (ii) another for setting the window specification of this particular stream. As presented in Figure 6, P4SS provides linear scalability with respect to the resource usage.
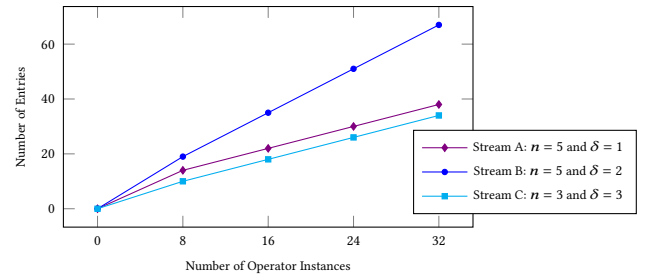

Figure 6: Resource usage of three independent data streams with different window specifications.

4.3 Performance Analysis

By mapping the splitter semantics to the P4 pipeline and having it executed on hardware we can obtain the line-rate performance of the switch and inherently benefit from the properties of the match-action pipeline such as the guarantees of bounded latency and very high-speed throughput. For our prototype, the software emulation on *bmv2* does not provide the time accuracy required for fine-grained performance measurements but there are practical instances of P4 switches [13] on which we plan to perform concrete performance evaluation of our solution. For instance with Tofino 3, it is possible to achieve a throughput of up to 10 billion events per second [12]. Although, we expect potential challenges for the planned data plane implementation as we adapt our prototype to Tofino and deal with its hardware constraints. To perform fine-grained measurements at the accuracy level of the Tofino switch, the P4sta framework represents an interesting tool for traffic load generation and hardware timestamping [18].

5 RELATED WORK

In-network load balancers using programmable data planes have been proposed in Hula [15], Silkroad [24] and iLoad [10]. However, these works focus on connection-oriented Internet Protocol (IP) traffic and they are not suitable for window-based operators. In [27],

the authors propose a dynamic key-based partitioning and load balancing solution for Distributed Stream Processing Systems (DSPS) that is adaptive to the changing workload volume. They propose an algorithm for the splitter to re-balance the workload by moving selected keys from an overloaded operator to an underutilized one. They recognize the splitter as being a bottleneck and leave it as future work. SCTXPF [14] is a CEP rule-aware load balancer where both the queries and the input data are dispatched so that the relevant events are sent to the operators where the concerned rules are being executed. In SCTXPF, the concept of windowing is not used and the events are considered individually. The authors argue that conventional load balancing with a RR scheduling does not work in the context of CEP because of lack of stateful processing. We demonstrated that with programmable data planes and the P4 language we can support stateful windowing semantics. P4CEP [16] presents the general idea of using programmable data planes for CEP. In P4CEP, high-level CEP queries are translated to the P4 language and executed by a P4 switch. To our knowledge, the particular problem of having the splitter as an in-network function is not addressed in the literature yet. Proposed solutions for window-based parallel operators either assume a multi-thread multi-core single machine [22] or a cluster of distributed general-purpose machines [1] but regardless of the system architecture, data parallelization framework using in-network computing has not been investigated. However, in [20], the authors examined less conventional parallel hardware architectures using Graphics Processing Unit (GPU)s for enabling high-performance publish/subscribe matching. Our work demonstrates for the first time how to utilize programmable data planes to support the splitter functionality.

6 CONCLUSION AND FUTURE WORK

In this work, we explored a network-centric approach to propose a P4-based data parallelization framework supporting parallel operator execution using in-network computing and software-defined networking. We focused on having the splitter as an in-network function by leveraging programmable data planes and the P4 language. Given a customizable window specification, our solution P4SS parallelizes and load balances the incoming data streams to a dynamically configurable set of operators. It supports multiple independently configurable parallelization models for multiple stream types using overlapping and non-overlapping *count-based* windows. For future work, we plan to expand our splitter capabilities with additional windowing semantics such as *time-based* and *marker-based* windows. We will adapt our implementation to hardware to perform a performance evaluation of the splitter with more advanced scenarios. Moreover, we aim for an evaluation of the control overhead to understand the time to change the splitter configuration.

REFERENCES

[1] Pramod Bhatotia, Umut A Acar, Flavio P Junqueira, and Rodrigo Rodrigues. 2014. Slider: Incremental sliding window analytics. In *Proceedings of the 15th international middleware conference*. 61–72.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM*

SIGCOMM Computer Communication Review 43, 4 (2013), 99–110.

[4] Bochra Boughzala and Boris Koldehofe. 2021. Accelerating the performance of data analytics using network-centric processing. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. 192–195.

[5] Alejandro Buchmann and Boris Koldehofe. 2009. Complex event processing. (2009).

[6] The P4 Language Consortium. 2021. *P4₁₆ Language Specification*. https://p4.org/p4-spec/docs/P4-16-v1.2.2.pdf

[7] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–62.

[8] Tiziano De Matteis and Gabriele Mencagli. 2017. Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. *International Journal of Parallel Programming* 45, 2 (2017), 382–401.

[9] Sebastian Frischbier, Mario Paic, Alexander Echler, and Christian Roth. 2019. Managing the Complexity of Processing Financial Data at Scale-An Experience Report. In *International Conference on Complex Systems Design & Management*. Springer, 14–26.

[10] Garegin Grigoryan, Yaoqing Liu, and Minseok Kwon. 2019. iload: In-network load balancing with programmable data plane. In *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*. 17–19.

[11] Martin Hirzel. 2012. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. 191–200.

[12] Intel. 2022. *Tofino 3 Intelligent Fabric Processor*. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/product-brief-final-version-pdf.pdf

[13] Intel. 2022. *Tofino Product Family*. https://www.intel.cn/content/dam/www/central-libraries/us/en/documents/tofino-product-family-brochure.pdf

[14] Kazuhiko Isoyama, Yuji Kobayashi, Tadashi Sato, Koji Kida, Makiko Yoshida, and Hiroki Tagato. 2012. A scalable complex event processing system and evaluations of its performance. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. 123–126.

[15] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*. 1–12.

[16] Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. 2018. P4CEP: Towards in-network complex event processing. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*. 33–38.

[17] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2014. Software-defined networking: A comprehensive survey. *Proc. IEEE* 103, 1 (2014), 14–76.

[18] Ralf Kundel, Fridolin Siegmund, Jeremias Blendin, Amr Rizk, and Boris Koldehofe. 2020. P4STA: High performance packet timestamping with programmable packet processors. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–9.

[19] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. 1–6.

[20] Alessandro Margara and Gianpaolo Cugola. 2013. High-performance publish-subscribe matching using parallel hardware. *IEEE Transactions on Parallel and Distributed Systems* 25, 1 (2013), 126–135.

[21] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. 2015. Predictable low-latency event detection with parallel complex event processing. *IEEE Internet of Things Journal* 2, 4 (2015), 274–286.

[22] 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*. 161–173.

[23] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. 2017. Minimizing communication overhead in window-based parallel complex event processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. 54–65.

[24] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 15–28.

[25] P4.org. 2022. *P4Runtime Specification*. https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html

[26] S Pramod and OP Vyas. 2012. Data stream mining: A review on windowing approach. *Global Journal of Computer Science and Technology Software & Data Engineering* 12, 11 (2012), 26–30.

[27] Nikos Zacheilas, Nikolas Zygouras, Nikolaos Panagiotou, Vana Kalogeraki, and Dimitrios Gunopulos. 2016. Dynamic load balancing techniques for distributed complex event processing systems. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 174–188.