NETREACT: Distributed Event Detection in Sensor Data Streams with Disaggregated Packet Processing Pipelines

Csaba Györgyi, Károly Kecskeméti, Hiba Mallouhi, Péter Vörös, Sándor Laki
ELTE Eötvös Loránd University, Budapest, Hungary
{*gycsaba96, cgsmef, hibamallouhi, vopraai, lakis*}*@inf.elte.hu*

2022 IEEE 8th International Conference on Network Softwarization (NetSoft) | 978-1-6654-0694-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/NetSoft54395.2022.9844106

***Abstract*—A new phenomenon called in-network computing has recently emerged with the aim of offloading calculations beyond the traditional task of packet forwarding to network switches. One of the most studied in-network computing applications is processing of sensor data streams. Existing works such as FastReact focus on solving this problem using flexible SmartNICs. In this paper, we propose NETREACT: an improved ASIC-oriented design for distributed event detection in sensor data streams to achieve a disaggregated processing pipeline. In contrast to existing approaches, NETREACT distributes the event detection task among a set of switches while leveraging the capabilities of the Intel Tofino platform in terms of boosting throughput and reducing latency. The proposed event-rule disaggregation method has the advantage of overcoming the hardware resource constraints and improving the overall network performance.**

***Index Terms*—P4, in-network computing, pipeline disaggregation, event detection, sensor data processing**

I. INTRODUCTION

Industrial Internet of Things (IoT) scenarios assume a large number of sensors deployed, which sense the environment and transmit a vast volume of data towards the cloud in order to analyze and use it for feedback control. With recent advances in edge computing, the latency of such a control loop can significantly be reduced by deploying computational power close to the industrial site since data packets do not need to be shipped all the way to the cloud. However, the virtualization techniques used by edge cloud platforms still introduce an extra delay that is too high for real-time applications (e.g., emergency stop). Today's setups often consist of very simple sensors periodically sharing their current state (sensor value) with a central controller (e.g., deployed in an edge cloud). Ensuring a fast reaction to changes in the environment can be achieved by sharing sensor values very frequently, e.g., in every 1-2ms. Flooding the network with packets might overload both the network and the controller. Keeping only the important packets (e.g., a value above a certain threshold) alleviates this issue by filtering out the unnecessary network traffic as early as possible.

Programmable packet processing pipelines have opened up new opportunities for use-cases requiring ultra-low latency and/or high-throughput by performing computations in the data plane [1]. In 2014, a new programming language called P4 [2] was released, making data plane programming much more convenient. P4 became a widely-used programming language supported by many different networking targets, including software switches [3], SmartNICs, and even performance-centric programmable switches such as the ones based on Intel Tofino ASIC [4].

Prior work [5]–[7] demonstrated that programmable data planes could offload real-time processing tasks and can provide much faster response times than their server-based counterparts. In [6], Vestin et al. propose FastReact that moves sensor data processing and event detection closer to the IoT devices by moving parts of the data processing logic from the industrial controller to the network data plane itself. FastReact is an event-based publish/subscribe Industrial IoT processing framework in P4 language, which can be flexibly customized from the control plane in run-time. Together with stateful processing, it supports windowed time series analysis as well as complex event detection and processing based on Boolean logic directly in the data plane of newly emerging programmable networking devices. The logic can be adjusted dynamically from the control plane without the need for recompilation. FastReact achieves significantly lower control latency compared to end-host processing in software while also capable of handling publish-subscribe scenarios which are common to Industrial-IoT protocol stacks.

Though FastReact was also implemented in P4, it was originally designed for the architecture of a specific smartNIC where register operations can be used without any restrictions, making the portability of this design to a programmable ASIC simply impossible. In addition, FastReact was proposed to be used on a single node, solely solving the event detection by a single P4-programmable device. However, different P4 targets have various limitations on the complexity (e.g., SRAM and TCAM usage, number of stages, processing latency, etc.) they are able to handle.

In this paper, we propose NETREACT that addresses the above limitations of FastReact while keeping its event-detection capabilities. Our main contributions are the following:

- Keeping the idea of FastReact that an event is represented as a logical expression in conjunctive normal form, we have fully redesigned the data plane algorithm so that it could be executed on Intel Tofino ASIC. The proposed ASIC-oriented pipeline exploits the advantage of match-action tables for storing event-detection rules and only uses stateful registers for storing sensor data and partial evaluation results.
- We also provide an algorithm that can disaggregate logical rules and distribute the event-detection task among

multiple P4 devices. In NETREACT, event detection is solved by the network instead of a single node.
- We show the performance limits of NETREACT on a single node and investigate how rule disaggregation can increase performance and help in overcoming potential resource limitations.
- As a contribution to the community, the source code is available at https://github.com/p4elte/netreact.

II. RELATED WORK

Yu et al. [8] provide a summary on the exhaustive literature of event detection in general.

Singh et al. [9] investigate machine learning-based solutions for distributed forest fire detection in wireless sensor networks (WSN) where computations of their machine learning algorithm are distributed among selected WSN nodes called cluster heads. Though this method has not been implemented in the data plane, its general idea is similar to NETREACT.

In addition to FastReact [6] mentioned in the previous section, implementing event detection in P4 data planes has already been investigated in several papers. Though their goals are similar, the different proposals have their own advantages and limitations.

Kohler et al. [5] moves the traditional CEP (Complex Event Processing) implementation on servers to the in-network computing paradigm. They propose timed sequenced events using a sliding window restriction, enabling events detection by state-machine logic using stateful packet processing. They also introduce a compiler that translates the P4CEP rules language to the adjacent P4 code deploying it on SmartNIC.

In [10], the authors provide two methods to enable attribute/value pairs encoding flexibility in a dynamic environment placing the focus on values of specific events in the content-based subscription model.

Combining controlled latency methods of distributed stream processing applications into a heterogeneous system has been addressed in [11]. The authors partition the latency frame restriction over control units and compose an ILP, in addition, to a heuristic as solutions for an optimal low latency- minimal cost problem distribution.

Mai et al. in [12] design an in-network computing-based architecture to complement mobile edge computing, identifying the tasks to migrate to programmable switches. The switches perform matching of the rules by integrating the CEP tool to do the conversion of the tasks and leaving the learning and updating of the rules as the responsibility of mobile edge computing layer. The authors implemented in-network fire detection and robotic motion control to demonstrate their architecture.

COMUS method [7] is a new network architecture design for an efficient pub/sub communication model. The authors introduce a packet subscription language and a compiler to produce the data and control plane configuration utilizing a BDDs algorithm, which saves up the limited resources of the programmable switch ASIC. They also describe a controller to decide on the routing policy depending on filters put in a packet subscription language provided by applications.

III. FILTERING ON TOFINO

A. Introduction to FastReact

The FastReact system, which provides the base idea for NETREACT, was initially designed to run on Netronome NFP. Since our Tofino-based implementation is built on its operational principle, we briefly introduce FastReact.

As NETREACT, FastReact treats the filtering rules as logical expressions in conjunctive normal form (CNF). These expressions consist of atomic predicates concatenated with disjunction operators, forming clauses connected with conjunctions. FastReact stores CNF rules to be evaluated in a hierarchy of register arrays. It also provides in-network caching in the form of storing historical and aggregated values. Finally, it can handle arbitrary packet structure with two restrictions: 1) the structure has to be known before the compilation of the P4 code and 2) each packet can only carry a single (sensor) value to be used in the filtering expressions.

B. Differences Between P4 Targets

Although many devices can execute P4 programs, they support various externs (built-in functionalities) and have their own constraints, making the portability of P4 code difficult or even impossible without deep modifications. The initial FastReact implementation was designed for a Netronome SmartNIC (NFP), while NETREACT targets Intel Tofino-based switches (TNA). We briefly summarize the key differences between the two platforms in the following paragraphs.

NFP has a hard limit of 256 for the number of match-action tables and actions, while the use of tables is encouraged on the Tofino for more efficient implementations and workarounds.

The registers of the NFP act as global variables allowing multiple reads and writes during the processing of a single packet. However, the TNA poses more strict requirements. One may access a register only once during the packet processing. (Although recirculation and resubmit could grant multiple opportunities, they also introduce concurrency and consistency issues.)

The Tofino switch has stage-based processing acting like a one-way assembly line where the maximum number of stages is limited. In contrast, the Netronome SmartNIC has a more flexible island-based internal operation.

Due to the previously mentioned differences and some additional constraints, implementing the FastReact on Tofino requires substantial design efforts in order to take advantage of its line-rate high throughput. In particular, FastReact has a heavy register usage and avoids match-action tables. In contrast, NETREACT primarily relies on tables, and when it uses registers, it can access them only once and in the same order during packet processing.

C. System Design

The available operators in the atomic predicates are: greater than, smaller than, equal, not equal, and in-range. We use the

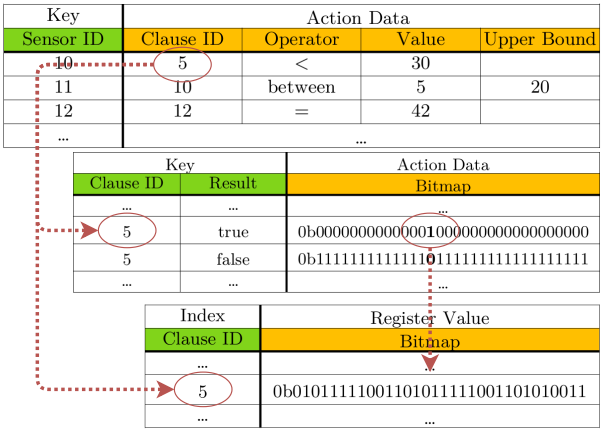

Fig. 1. Expression representation in NETREACT's data plane.

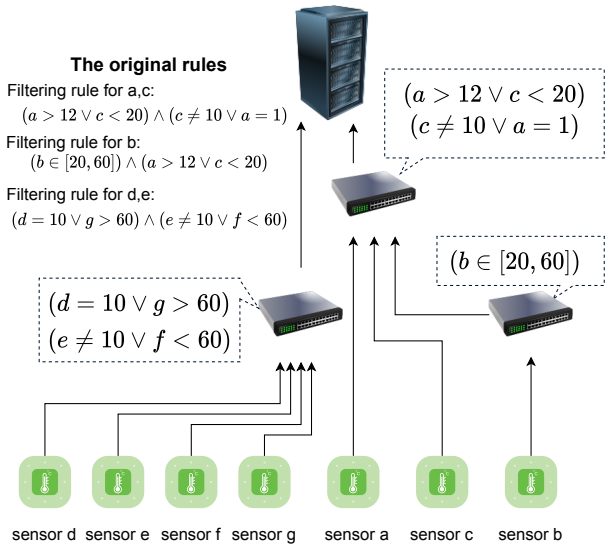

Fig. 2. Example placement of following sensors and their filtering rules: $a, c : (a > 12 \vee c < 20) \wedge (c \neq 10 \vee a = 1); b : (b \in [20, 60]) \wedge (a > 12 \vee c < 20); d, e : (d = 10 \vee g > 60) \wedge (e \neq 10 \vee f < 60)$

same sensor data format as FastReact, assuming a UDP packet containing the sensor ID and the sensor value.

The initial step of the pipeline (as can be seen in Figure 3) is to decide whether the incoming packet is a sensor packet or other traffic. In the latter case, we can perform some other tasks, e.g. L2 forwarding.

If it is a sensor packet, the adequate rule's id (Clause ID), the needed operators and numerical values are read from the match-action tables. Later, this Clause ID is used as a register index to store partial results of the evaluation. The number of tables and registers involved equals the maximum number of clauses in a rule. Thus if the maximal number of clauses in a rule is n, we need n conjunction tables, each providing one or zero clauses for each sensor.

The data plane representation of a single condition can be seen in Figure 1. Each clause is stored as a discrete entry in the conjunction tables containing a Clause ID, the operation to be performed and the optional operands. (For technical reasons, it is possible not to perform any operation, just use the stored results). The clauses are accessed via the Sensor ID, using exact matching.

A preparation step precedes the evaluation of the clauses. We subtract every sensor value from the value we would like to compare it. This way, we can get a Boolean value simply by checking the sign of the result, which fits the Tofino's match-action table structure much better than the direct comparison of two variables. When the required operation is in-range, two subtractions are needed, leading to two results that must be evaluated (essentially, we reduce it to a greater than and a less than operation).

Since most use cases require the value of multiple sensor values, we need to store the results of all the clauses processed on the device. As mentioned before, this is done by storing 32 long bitmaps in registers. Keeping only the Boolean result of each clause means that a clause using some sensor value a can only appear in rules that do not filter a if the same clause appears in a rule that filters a.

When a new Boolean value is computed, the result is stored in a bitmap variable where each bit represents the current state of a given clause. To this end, the method applies a lookup table matching the computed Boolean value, sensor ID and the Clause ID. The latter identifies the bit position carrying the result in a temporary bitmap. Then it is merged with the stored values by applying logical AND/OR on it, and the appropriate map is stored in the register (indexed with the Clause ID). The new maps are both stored in registers and used to determine whether the CNF, as a whole, is true or false. Because of this kind of merging, we can use each sensor only once in a clause.

Besides its primary filtering function, our implementation maintains a configurable sized history. It uses multiple registers similarly to a queue. Moreover, tracking a moving average for each sensor value is also possible. These are not depicted in Figure 3 since they are not part of the main functionality and would clutter the flowchart.

IV. DISAGGREGATION OF THE FILTERING EXPRESSIONS

A. Motivation

Being able to disaggregate the filtering expressions across multiple switches allows us to circumvent the resource limitations inherent in all network devices. A well-thought-out rule placement can help filter out packets that do not carry interesting information early, thus reducing the network's overall traffic. To simplify our work, in the following, we assume a tree-topology.

We slice the expressions into clauses to spread the filtering across multiple devices. These can be potentially evaluated on different nodes. However, we must ensure that the distributed and centralized setup yields the same overall results.

Let us consider the example provided by Figure 2 that presents a correct disaggregation of the rules. However, if we move the $(a > 12 \vee c < 20)$ to the root node, the filtering would become inconsistent. Supposing that we drop an a sensor packet, because of the $c \neq 10 \vee a = 1$, the value

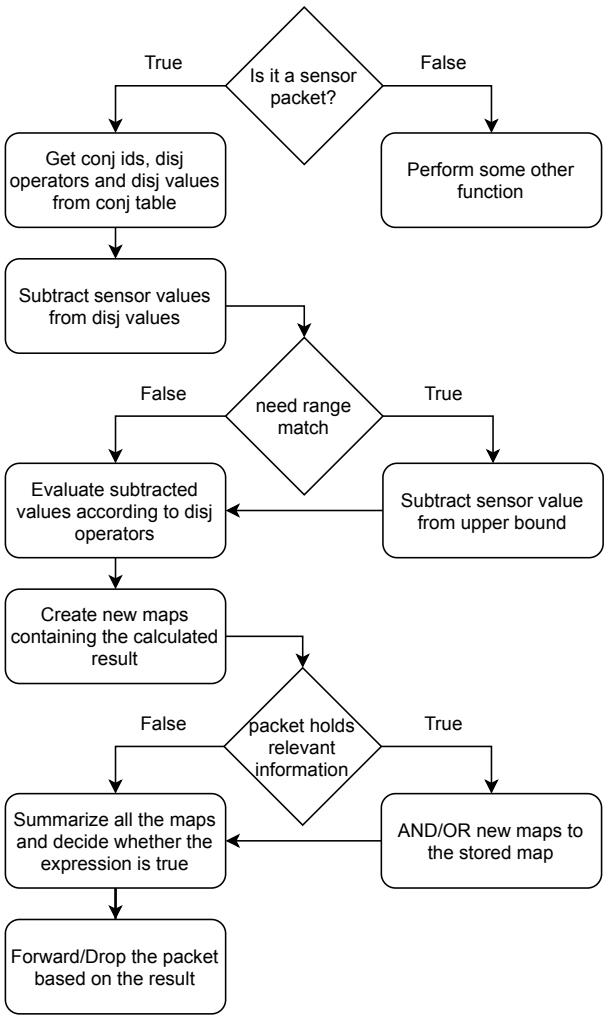

Fig. 3. Simplified NETREACT flowchart

of logical expression $a > 12$ in the root can not be updated in this case.

B. Formal requirement

In order to ensure the consistency of our system, we define the "do not drop packets too early" requirement for a disaggregation as follows:

$$\forall c \in Conj, \forall x \in M(c), \forall y \in C(c) : x \nearrow y \qquad (1)$$

Where $M(c)$ is the sensor value variables the conjunction containing the clause filters (matches), $Conj$ is the set of conjunctive clauses and $C(c)$ is the set of sensor value variables the clause contains. $x \nearrow y$ means that the evaluation of clauses containing x has to precede the evaluation of clauses containing y, or it has to be at the same node as the clauses of y. Figure 4 illustrates the notation.

To give a formal solution to this problem, we treat the system of conjunctions as a directed graph. The graph's vertices are the sets of clauses. The edges between the vertices define the order in which the clauses are meant to be evaluated. Finding the strongly connected components yields us the parts to be moved together and the dependencies between them. Figure 5 depicts how clauses in a graph could form strong components.

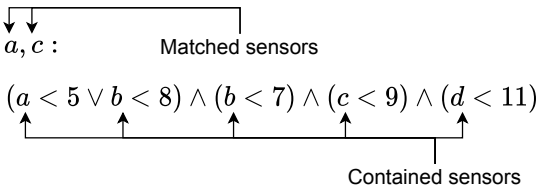

Fig. 4. An example rule. Conjunction k filters the a and b sensor values. We indicate the elements of $M(k)$ and $C(k)$.

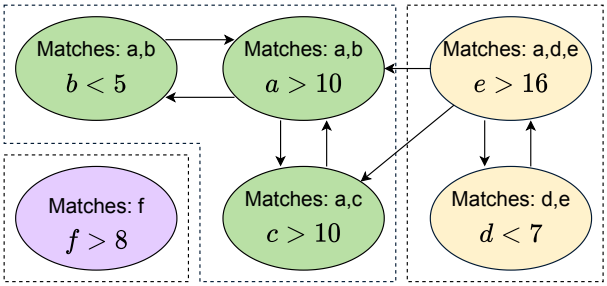

Fig. 5. Strong components and filtering expressions for sensors: $b : (b < 5) \wedge (a > 10)$; $a, c : (a > 10) \wedge (e > 16) \wedge (c > 10)$; $d, e : (d < 7) \wedge (e > 16)$

C. Limitations and workarounds

The way our pipeline is structured comes with some limitations regarding the disaggregation of filtering expressions. Namely, each clause is evaluated only when the switch running our code gets a packet containing sensor data which is both used in the clause and also matched on it ($x \in M(c)$). For example, if the clause ($a > 10$) is only used for matching packets generated by sensor b, this clause's value will never be calculated since it relies on values of sensor a. This limitation, in most cases, can be circumvented by adding extra clauses to rules that match the target sensor value.

V. EVALUATION AND DISCUSSION

A. Data Plane

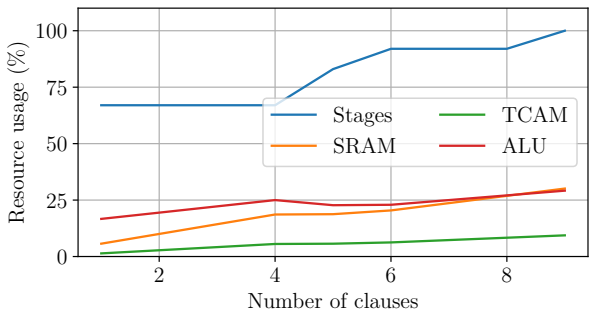

Fig. 6. Resource usage based on the complexity of expressions

We implemented NETREACT in P4 using the Intel Tofino's architecture. Instantiating NETREACT with the highest possible complexity of the CNFs (at most 9 clauses) occupies all the available stages, 30.1% of the available SRAM, 9.4% of the available TCAM and 29.2% of the available ALU resources on average. Figure 6 shows how resource usage changes with the increased complexity of filtering rules. Since our implementation does not resubmit or recirculate packets, we claim that low-latency and high throughput are guaranteed by the high-speed Tofino hardware.

We have also compared NETREACT to COMUS [7] since their designs show similarities despite the fact that COMUS has been designed to deal with packet subscriptions while our goal is the filtering of sensor packets. However, in contrast the CNF used by our NETREACT, COMUS relies on disjunctive normal form (DNF) to represent its rules. Another important difference is that NETREACT heavily relies on the usage of SRAM, while the default algorithm of COMUS needs more TCAM entries. Because of this, COMUS might need $O(n)$ table entries to represent a single comparison where n is the bit length of sensor value fields, while NETREACT requires only one or two (for in range conditions). For the sake of a fair comparison, we have to note that COMUS provides more functionality than NETREACT in exchange for higher resource usage, and it also introduces additional mechanisms to reduce the required TCAM space.

Finally, another advantage of NETREACT is that the majority of its pipeline can be executed at the egress side, and thus it does not affect the deployment with co-locating other network functions at ingress.

B. Disaggregation

NETREACT can handle CNF expressions with at most 9 clauses on a single device. Since the expressions can be disaggregated, each additional device increments the total number of deployable clauses by 9 clauses if the rules are properly sliceable.

Since this disaggregation method does not require any changes in the initial FastReact protocol, it is also possible to mix devices running either FastReact or NETREACT, allowing the creation of systems with heterogeneous data plane solutions. The graph representation of the disaggregation provides a basis for future work. One can build different tree-like network topology to maximize the positive effects of disaggregation, also considering the case of dynamically updating the network structure [13]. In this paper, we have only focused on providing a disaggregation of logical rules that correctly filters out traffic in case of an ideal topology. As part of our future work, we aim to extend the proposed method to consider additional constraints such as given topology, event occasions with different probabilities and hardware limitations, and find the optimal placement/distribution of rules in the presence of such constraints.

VI. CONCLUSION

We have presented NETREACT, a Tofino-friendly implementation of event detection pipiline inspired by FastReact. It leverages the line-rate high throughput of programmable ASIC in exchange for certain limitations.Moreover, we provided a method to disaggregate filtering rules based on a graph representation and strongly connected components. This model provides a strong basis for our future work.

ACKNOWLEDGMENT

This research is in part supported by the project no. FK_21 138949, provided by the National Research, Development and Innovation Fund of Hungary. P. Vörös and S. Laki also thank the support of "Application Domain Specific Highly Reliable IT Solutions" project that has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

REFERENCES

[1] A. Sapio *et al.*, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017, pp. 150–156.

[2] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[3] P. Vörös *et al.*, "T4p4s: A target-independent compiler for protocol-independent packet processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8.

[4] A. Agrawal and C. Kim, "Intel tofino2-a 12.9 tbps p4-programmable ethernet switch." in *Hot Chips Symposium*, 2020, pp. 1–32.

[5] T. Kohler *et al.*, "P4cep: Towards in-network complex event processing," in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018, pp. 33–38.

[6] J. Vestin *et al.*, "Fastreact: In-network control and caching for industrial control networks using programmable data planes," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1. IEEE, 2018, pp. 219–226.

[7] T. Jepsen *et al.*, "Forwarding and routing with packet subscriptions," in *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 282–294. [Online]. Available: https://doi.org/10.1145/3386367.3431315

[8] M. Yu *et al.*, "Spatiotemporal event detection: A review," *International Journal of Digital Earth*, vol. 13, no. 12, pp. 1339–1365, 2020.

[9] Y. Singh *et al.*, "Distributed event detection in wireless sensor networks for forest fires," in *2013 UKSim 15th International Conference on Computer Modelling and Simulation*. IEEE, 2013, pp. 634–639.

[10] R. Kundel *et al.*, "Flexible content-based publish/subscribe over programmable data planes," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–5.

[11] H. Röger *et al.*, "Combining it all: Cost minimal and low-latency stream processing across distributed heterogeneous infrastructures," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 255–267.

[12] T. Mai *et al.*, "In-network computing powered mobile edge: Toward high performance industrial iot," *IEEE Network*, vol. 35, no. 1, pp. 289–295, 2020.

[13] M. N. Hall *et al.*, "A survey of reconfigurable optical networks," *Optical Switching and Networking*, vol. 41, p. 100621, 2021.