Programmable Event Detection for In-Band Network Telemetry

Jonathan Vestin*, Andreas Kassler*, Deval Bhamare*,
Karl-Johan Grinnemo*, Jan-Olof Andersson*, Gergely Pongracz†
*Karlstad University, Sweden, †Ericsson AB, Hungary
{jonathan.vestin, andreas.kassler, deval.bhamare, karl-johan.grinnemo}@kau.se
janoande02@student.kau.se
gergely.pongracz@ericsson.com

***Abstract*—In-Band Network Telemetry (INT) is a novel framework for collecting telemetry items and switch internal state information from the data plane at line rate. With the support of programmable data planes and programming language P4, switches parse telemetry instruction headers and determine which telemetry items to attach using custom metadata. At the network edge, telemetry information is removed and the original packets are forwarded while telemetry reports are sent to a distributed stream processor for further processing by a network monitoring platform. In order to avoid excessive load on the stream processor, telemetry items should not be sent for each individual packet but rather when certain events are triggered. In this paper, we develop a programmable INT event detection mechanism in P4 that allows customization of which events to report to the monitoring system, on a per-flow basis, from the control plane. At the stream processor, we implement a fast INT report collector using the kernel bypass technique AF_XDP, which parses telemetry reports and streams them to a distributed Kafka cluster, which can apply machine learning, visualization and further monitoring tasks. In our evaluation, we use real-world traces from different data center workloads and show that our approach is highly scalable and significantly reduces the network overhead and stream processor load due to effective event pre-filtering inside the switch data plane. While the INT report collector can process around 3 Mpps telemetry reports per core, using event pre-filtering increases the capacity by 10-15x.**

I. Introduction

Operations, Administration, and Management (OAM) refers to protocols, tools and mechanisms that help network operators in fault indication, performance monitoring, security management, diagnostic functions, accounting, performance monitoring, configuration and service provisioning. In traditional carrier networks, OAM tools such as SNMP and OWAMP-Test are used, however, these tools have been proven inadequate for SDN-NFV data centers. They are not scalable and cannot provide fine-grained, real-time information about the overall performance of the data center infrastructure [1].

In-band Network Telemetry (INT) has gained a lot of momentum over the last few years [1]–[5]. The idea behind the INT framework is that each node along a network path adds telemetry items and network state to in-band, data plane traffic. Telemetry items may include switch ID, ingress timestamps, queue occupancy information, and various other performance-related metadata, which are added at line rate as customized headers to in-band, data plane packets. The telemetry items are forwarded to a distributed network monitoring platform, which uses stream processors to consume the telemetry metadata, compile traffic statistics, and apply machine learning to derive actions and/or recommendations for the operation and management of the data center. Eventually, the original data plane packets are recovered at the network edge and forwarded to the end-user.

The advent of programmable data planes and high-level, platform-independent programming languages such as P4 [6], have enabled fine-granular monitoring of data plane traffic. [7] uses P4-based INT to collect per-packet queue statistics in a metro network, and use that information to enforce QoS; [8] proposes such a telemetry solution for IP-over-optical networks; and Barefoot Networks includes their INT technology in the Smart Programmable Real-time In-band Network Telemetry (SPRINT). A common denominator to these and other previous work is that it lack highly configurable per-flow event detection in the data plane. This results in high load on INT stream processors and limits the scalability or severely limits the event collection possibilities.

In this paper, we make the following contributions. First, we design a programmable event detection framework for INT data. The control plane is responsible to specify high-level event algorithms along with threshold values that trigger the generation of telemetry reports from the programmable data plane. Second, we implement several event detection and filtering algorithms in P4 for INT metadata. Third, we develop an INT monitor using kernel bypass AF_XDP [9], which allows to parse large amounts of INT reports per second on a single core and forward them to a Kafka [10] cluster for further processing. The SDN controller can use the telemetry reports and fine-tune the event detection mechanisms on a per-flow basis according to, e.g., service level agreements and detected events. Our evaluation shows that a single core can process $3.593\,\mathrm{Mpps}$ telemetry reports without P4 event detection. When using event detection, using a filter threshold of 100 μs queue buildup for *web* traffic increases that to $35.22\,\mathrm{Mpps}$, while for *database* traffic it increases to $12.21\,\mathrm{Mbps}$.

The rest of the paper is organized as follows. Section II discusses the INT framework and related work. Section III

978-1-7281-4832-8/19/$31.00 ©2019 IEEE

describes the design and implementation of our solution. Section IV evaluates our P4 design in a testbed and Section V concludes the paper.

II. Background and Related Work

Both industry and academia have explored the area of network monitoring, its overhead on network infrastructure, and how to minimize it [11] However, existing solutions mainly focus on the trade-off between expressiveness, accuracy and speed, and often stress the monitoring platform. For example, systems such as NetQRE [12] can support a wide range of queries using stream processors running on general-purpose CPUs, but they incur substantial bandwidth and processing costs to do so. Telemetry systems such as Chimera [13] and Gigascope [14] are expressive in nature by covering a wide range of telemetry items, however, can only support lower packet rates. This is because these systems process all packets at the stream processor which can become a bottleneck.

Several works try to reduce the amount of INT information that INT monitors and stream processors need to process. Clearly, there is a trade-off between the accuracy of the captured network state and the load imposed on the monitoring framework. Sonata [15] is a framework for performing complex in-network event detection, through an approach which splits event processing between user-space stream processors and programmable switch data planes. Through the use of simple, but powerful well-known operators like *filter*, *map*, *reduce* etc. INTCollector [16] implements a distributed telemetry monitoring system by parsing telemetry reports inside the Linux kernel using eBPF and XDP. It implements threshold- and interval-based event detection at the telemetry collector in the fast path and inserts them into a distributed database (e.g., Prometheus[1] or InfluxDB[2]) in the slow path. Because of its single-core implementation it does not scale adequately and its processing rate is limited to 1.2 Mpps for a single core. Because the events are detected only at the telemetry monitoring system, unnecessary telemetry reports are sent from the data plane resulting in a very high load on the INTCollector.

Joshi et al. [2] present BurstRadar, where the key idea is to first detect a micro burst in the data plane and then capture a snapshot of telemetry information of all the involved packets. This information allows queue composition analysis to identify the culprit flow(s), and burst profiling to know burst characteristics such as duration, queue build-up/drain rates, etc. Authors in [17] also propose an approach called Snappy to to identify the particular flows responsible for a micro burst, and handle them automatically. Snappy maintains multiple snapshots of the occupants of the queue over time, where each snapshot is a compact data structure that makes efficient use of data-plane memory.

Kim, Suh and Pack [4] propose selective INT (sINT) where the ratio of packets to be monitored can be adjusted depending on the frequency of significant changes in network information. Li et al. [18] propose FlowRadar, a novel approach to maintain

[1]https://prometheus.io
[2]https://www.influxdata.com/

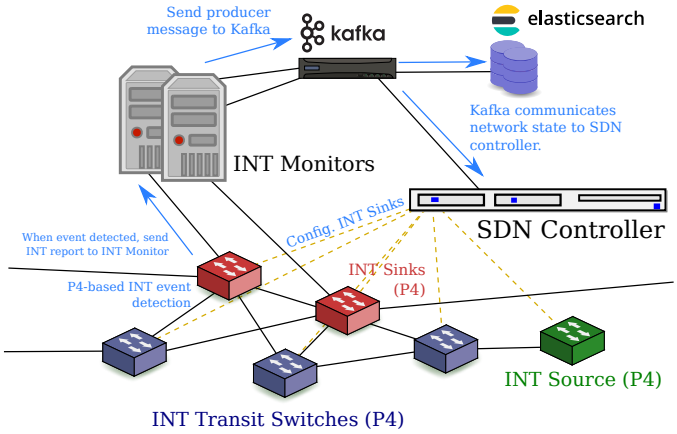

Fig. 1: Overview of the INT monitor application

flows and their counters that scale to a large number of flows with small memory and bandwidth overhead. The key idea of FlowRadar is to encode per-flow counters with a small memory and constant insertion time at switches, and then to leverage the computing power at the remote collector to perform network-wide decoding and analysis of the flow counters. N. Tu et al. [19] have presented an INT architecture for the UDP and discuss its design and integration with Open Network Operating System (ONOS) controller.

In our proposed approach, we have modified the INT Collector to offload the event detection from the stream processor to an in-network P4 application. Not only does this reduce the network overhead, it also reduces the stream processor load, since it enables pre-filtering of telemetry reports inside the data plane.

III. Design and Implementation

Network monitoring in data center networks must balance fine-granular monitoring with expressiveness in order to detect events that are important for network operators to act upon swiftly. Especially, high configurability, scalability and line-rate processing are important design goals. The detection of important events occurring in data center network infrastructure, such as micro bursts, may require correlation of multiple stateful variables. These events are of high interest to network operators, as well as customers, as they may negatively impact important service flows. For example, an event may be defined as when the queue occupancy of any switch in a forwarding path is greater than 50% while the end-to-end latency is higher than $5ms$. On the contrary, micro bursts may not be important for a service flow if end-to-end latency is lower than for example $3ms$. Although complex event processing in the data plane on application layer data has been previously explored [20], [21], this paper applies it to network telemetry data.

A. Overall Design

In our design, we allow specifying per flow event thresholds, thus minimizing the load on the stream processors. This is important as in the INT framework, typically telemetry information is added to each user data packet. Sending telemetry reports for each packet to the network monitoring system

results in excessive load on the stream processor. Some of that telemetry information can be filtered out easily, because it does not provide more information. For example, when two packets of the same service flow are processed back-to-back in a switch, the queue latency may not have changed significantly and thus the second packet's telemetry information for queue latency could be discarded.

In our approach, the control plane, through an SDN controller, configures event detection mechanisms, including switch role (e.g. source, sink, transit), detection algorithm and threshold values on a per-flow basis (see Figure 1). This configuration is communicated to the switch through the controller API exposed by P4 (using table entries and registers). Through this configuration API, the SDN controller specifies the event detection parameters which are executed at the INT sink, allowing for the detection of many different types of events (e.g. queue or latency buildup). Due to the per-flow granularity, this also allows for capturing bi-directional flows.

INT sources insert telemetry headers and an instruction bitmask that determines which telemetry items to collect. When an INT transit switch receives a packet with a telemetry header, it parses the instruction bitmask and pushes INT metadata (e.g., queue occupancy or hop latency) into that header. Finally, when an INT sink receives a packet with a telemetry header, it removes all telemetry headers, and runs the event detection algorithm as configured from the control plane on the appended metadata. Should the criteria for an event be met (e.g., an end-to-end latency spike above the threshold or queue occupancy over the configured threshold), a telemetry report is sent to the INT monitors (stream processor). In case there is no event detected (e.g. latency below configured threshold), the sink discards the telemetry items thus reducing the load on the stream processor.The INT monitor takes incoming telemetry reports and creates Kafka producer messages, which then can be sent to a Kafka topic.

The SDN controller can use this data to further reconfigure and fine-tune the threshold and algorithm settings on a per-flow basis; increasing the report resolution for interesting traffic, while reducing it for, e.g., background traffic. As thresholds and telemetry items can be configured per flow basis, our approach also supports customized per slice INT monitoring. Kafka also communicates the INT event reports to an Elastic Search stack for further analytic processing and visualization.

B. High Performance INT Collection in AF_XDP

For scalable processing of INT telemetry reports, we implemented an INT monitor in C[3], using the AF_XDP [9] socket type for kernel bypass. It also uses the librdkafka[4] library for Kafka producer message generation. When launched, it is associated to a specific queue on a network interface, and uses up a single CPU core from the host machine. Multiple instances of the same program can be run simultaneously to enable multi-core processing. Our implementation follows the telemetry report format specification given in [22], which includes 8 metadata fields: *switch ID*, *hop latency*, *queue occupancy*, *ingress timestamp*, *egress port id*, *queue congestion status*, *egress port utilization*. While some of the metadata items, like *switch ID*, are uninteresting to run event detection on, we still support parsing them in INT monitor, and we also parse the reserved INT fields which may be used in the future, or by non-standard network extensions.

[3]Code available here: https://github.com/jonavest/int-afxdp-kafka

[4]https://github.com/edenhill/librdkafka

C. P4 based Event Pre-filtering using Programmable Data Planes

Our P4 implementation of the INT framework follows the telemetry report format specification for event detection. To configure the switch, it exposes a series of parameters to the SDN controller. Some of these are configured using the P4 register API, while others are configured using flow tables:

INT Mode The mode which the switch is running for the given flow, which can be either: INT sink, INT transit or INT source.

Instruction mask When the switch is in the INT source mode, this configures on a per-flow basis which instruction bitmask should be appended to incoming packets. This is configured through a match-action table matching on a flow match key (e.g. mac src/dst, ip src/dst, port number).

Algorithm It is possible to configure the event-detection algorithm and its parameters (such as threshold, metadata type, although algorithms may differ in the number and type of parameters). Also configured through a match-action table with the *fast_detection* action.

Expression If more complex event detection is required, logic tables installed through P4 registers in conjunctive normal form, adopted from FastReact [21], can be used. This uses the *complex_detection* in the algorithm match-action table. Using complex expression trades performance for allowing more complex expressions.

The processing of incoming packets is divided into parsing, ingress processing and egress processing. The parsing consists of a parsing tree, which parses incoming messages looking for an INT header. If an INT header is found, the switch parses any INT metadata available. In the ingress processing, the *ingress timestamp* is recorded into the packet INT metadata, and a regular IPv4 switch forwarding table is applied. In the egress block, we start by fetching the algorithm ID and parameters from the algorithm match-action table. If the flow is configured with *fast_detection*, the switch runs one out of three algorithms, specified by parameters given to the *fast_detection* action: *per-hop*, *per-flow* or *moving average*. Also, the INT metadata type (e.g., queue occupancy or hop latency) which the algorithm is run on, is also specified as a parameter.

The *per-hop* algorithm keeps a record of the previous INT metadata for each *switch id* in a P4 register table. When a telemetry item is processed by the algorithm, for each *switch id*, it looks at the difference between the incoming INT metadata and the previous metadata. If the difference exceeds the configured threshold, it stores the new value in the register and marks the packet as an event. This is done through an unrolled loop. The *per-flow* algorithm on the other

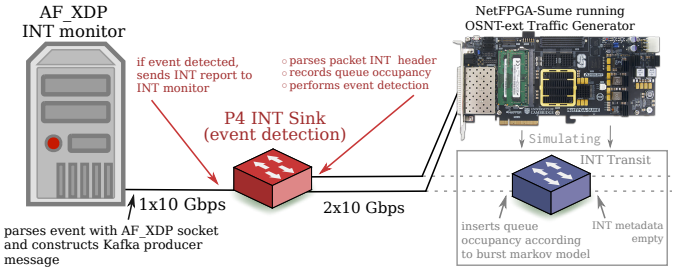

Fig. 2: Testbed setup used for evaluation

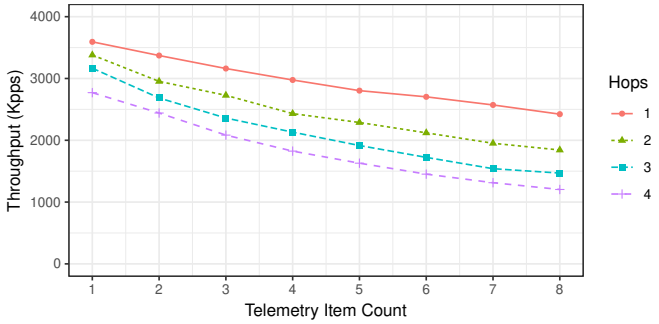

Fig. 3: Event report processing capacity of the AF_XDP INT monitor, per core/interface, varying header and hop count.

hand, cares for the sum of the INT metadata for all hops in a specific flow. It stores a table, for each flow (hashed) and each INT metadata item, what was the previous sum. When a telemetry report arrives, it sums up incoming metadata values, compares them to the previous sums, and if the difference is greater than the threshold, the new sum is stored and the packet is marked to be sent to the INT monitor. Much like for the *per-hop* algorithm, this is done through an unrolled loop. Finally, the moving average works similarly to the flow sum algorithm, however, it also applies a simple exponential moving average between the current and last received value. A configurable α parameter decides how much weight new values will have on the average, i.e., the smoothness of the function. Again, if the difference is greater than the threshold, the new value is stored, and the packet is marked. Finally, there is also a *noop* algorithm, which always detects an event.

If more complex event detection is required, it is possible to use *complex_detection* action for the algorithm table. Complex detection only takes a single parameter, which is a register table index. In this register, an expression according to the FastReact format is stored, and allows specifying more complex, stateful expressions, such as `hop-latency > 10 and queue buildup > 100`. As such complex expression have significantly higher execution time due to complex logical operations, they will increase the per packet latency.

We implemented our approach in P4_16 language on the Netronome Agilio NFP-4000 platform [23], which is a multi-threaded, multicore programmable network interface card. It distributes incoming packets among 48 packet processing cores, each running 8 threads. Memory is divided into a multi-tier structure, where tables and registers are stored in a 2 GiB DRAM, utilizing two blocks of 2 MiB SRAM as a cache. Our P4 implementation allows the SDN controller to dynamically reconfigure the INT behavior of each switch. This includes changing the switch role (sink, source or transit), and configuring which INT fields should by inserted by each switch, as well as changing the threshold and algorithm settings per flow. In our evaluation, we will only look at the *fast_detection*, and leave the *complex_detection* for future work.

IV. Evaluation and Results

In this section we evaluate the effectiveness of our approach by answering the following questions:

1) **INT monitor (Section IV-A):** What is the performance capacity of the INT monitor, and how is this performance capacity affected by the telemetry item count and number of hops traversed by the INT report?
2) **INT sink (Section IV-B):** What is the potential processing capacity for the entire INT deployment, given different traffic patterns and event detection algorithms?

Our testbed consists of an INT sink, an INT monitor and a traffic generator. The INT sink is connected to the traffic generator using 2x10 Gbps SFP+ connections, and is also connected using 1x10 Gbps SFP+ connector to the INT monitor. The INT sink is equipped with a Netronome Agilio 2x40G NFP-4000 SmartNIC, split into eight 10 Gbps ports using a breakout module, running our P4 implementation for INT parsing and event detection. Here, INT reports incoming from the traffic generator (representing an INT-enabled network) are processed, and if an event is detected, an event report is sent to the INT monitor. The INT monitor is connected to the INT sink; processes incoming INT event reports, and parses them into a Kafka message. The constructed Kafka message is, however, not transmitted to a Kafka cluster, but is constructed and then dropped, since evaluating the Kafka message producer is out of scope of this paper. The event report parsing is done using an AF_XDP [24] socket. The machine is equipped with a 20 core Intel® Xeon® Silver 4114 CPU, running at 2.20 GHz, and with 32 GB RAM running Ubuntu 18.04 LTS. This end-host is also equipped with two 2x10G SFP+ Intel X520 network cards. Finally, the traffic generator is a NetFPGA-SUME[5] running the *extmem*[6] variant of the Open Source Network Tester (OSNT) [25]. This platform is equipped with four 10 Gbps SFP+ interfaces, where OSNT-extmem can use two of them to generate synthetic traffic at 20 Gbps. The entire testbed setup is shown in Figure 2.

A. INT Monitor

First, we evaluate the processing capacity of the INT monitor. We connected the INT monitor directly to the NetFPGA running the OSNT traffic generator, and generated 10 Gbps of INT event report traffic for 60 s, while the monitor logged the number of events it was able to process per second. The result is shown in Figure 3, where the y-axis shows the number of packets processed per second and the x-axis shows the

[5]https://github.com/NetFPGA/NetFPGA-SUME-public/wiki
[6]https://github.com/NetFPGA/OSNT-Public/wiki/OSNT-SUME-extmem-project

number of telemetry items included per hop in the event report. Different curves represent different number of hops in the topology, i.e., different number of switches from source to sink that appended telemetry items to the packet INT header. From the figure, we can see that the throughput depends on the size of the INT-event report: With small INT reports, containing only one telemetry item from a single switch, the INT monitor can process 3593 Kpps of event reports. However, when the number of telemetry items increases to four, the processing capacity is reduced to 2976 Kpps and the throughput to 2771 Kpps. Finally, increasing the number of telemetry items to eight and the number of hops to four, which means each INT event report carries 32 values, reduces processing capacity to 1204 Kpps. From these results, it is evident that the performance degradation due to increased number of hops is greater than that of due to increased number of telemetry items. This is because the INT parser needs to parse the entire INT instruction mask for each hop, regardless of how many entries are included.

B. Event Detection

In this section, we evaluate our combined INT framework, including the event pre-filtering at the INT sink and the INT monitor. We vary the event detection algorithm and threshold setting. We connect the OSNT traffic generator to the INT sink, which in turn is connected to the INT monitor, as shown in Figure 2. In our evaluation, we look at the micro-burst event detection of a single INT-enabled switch. The traffic generated follows a burst Markov model [26], which is based on micro-burst measurements taken from a Facebook data center, and provides the duration of and inter-burst time for each micro-burst. We emulate three different traffic patterns: *web*, which is requests to and responses from web servers, *cache*, which consists of traffic from in-memory caching servers used by the web servers and *hadoop*, which is servers used for offline analysis and data mining.

Three traces were generated, each consisting of 200,000 packets, which emulate INT traffic by using the inter-burst times and durations provided in the model for one switch reporting its queue occupancy. Larger traces would have been preferable, however, the OSNT traffic generator, even with the *extmem* extensions limits how many unique packets can be loaded into the NetFPGA memory. The traces are repeated for 60 s, continuously generating 20 Gbps of traffic. From the recorded number of processed packets per second, and knowing the processing capacity of the INT monitor, we calculate the potential INT report processing capacity per processing core of the monitor. For the threshold setting, we use *queue occupancy* specified in microseconds. We have also included the threshold of 0 µs in both experiments, which show the processing capacity if no event detection is done in the data plane. Here, the processing capacity is the same regardless of traffic model as all events are sent to the INT collector.

Results obtained by implementing the *per-hop* or the *per-flow* threshold algorithm are shown in Figure 4: The y-axis shows the potential processing capacity per processing core in the INT monitor in terms of packets per second (pps)

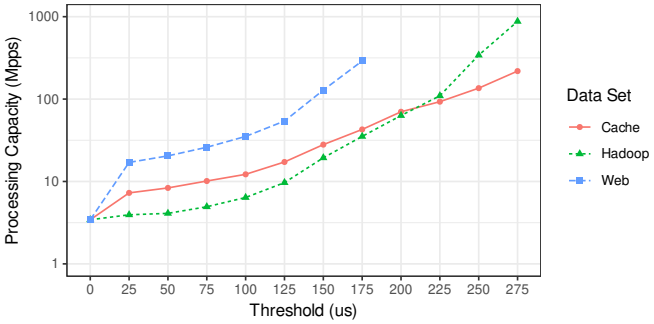

Fig. 4: Processing capacity of the INT monitor using the *per-hop/per-flow* threshold algorithm, with three different traffic types, varying the threshold setting.

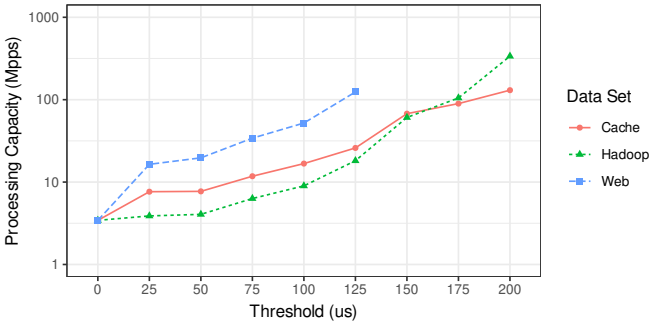

Fig. 5: Processing capacity of the INT monitor using the *moving average* threshold algorithm, with three different traffic types, varying the threshold setting.

including the pre-filtering in the INT sink, while the x-axis shows the configured threshold for the *per-hop/per-flow* algorithm. As we only generate one flow for these experiments, both algorithms have the same behavior. We observe that performing no event filtering gives us a 3.43 Mpps processing rate. With the *web* traffic model, using a threshold of 100 µs, the processing capacity reaches 35.22 Mpps, and with a threshold of 150 µs the processing capacity is 128.59 Mpps. For the *cache* traffic model, which has longer bursts and thus higher queue buildup, a threshold of 100 µs results in 12.21 Mpps processing capacity, while threshold of 150 µs results in the perfornace of 28.01 Mpps. As the *web* traffic has lesser and shorter bursts of traffic, the number of events generated are fewer which gives a higher processing capacity, while more bursty traffic like *cache* and *hadoop* gain comparatively less from in-network event detection. The missing data points from the figure mean that no events were detected with that threshold. This is especially prevalent in the *web* traffic model, as queue occupancy does not reach above 175 µs. Given longer traces such events are, although rare, more likely to be present. It is important to note, however, that the selected settings are based on the Facebook traces and the NFP4000 card characteristics. Real networks with different queue sizes and link latencies, would require different thresold settings.

Finally, results from using the *moving average* threshold algorithm can be seen in Figure 5. As follows, with the *web* traffic model and with a threshold of 100 µs, the processing capacity is 52.07 Mpps, and with a threshold of

150 µs no events are detected. Using the *cache* traffic model and a threshold of 100 µs, the per-core processing capacity reaches 16.76 Mpps, and with a threshold of 150 µs it reaches 67.64 Mpps. That is, we observe similar patterns as those observed for the *per-hop*/*per-flow* algorithm. However, moving average, in most cases, resulted in less data sent to the INT monitor, as shorter bursts are smoothed out, which allows increased total processing capacity of the INT-enabled network. The results demonstrate that our approach is highly scalable, and significantly reduces the network overhead and stream processor load by using effective event pre-filtering inside the switch in the data plane.

V. Conclusions

In this paper, we developed a programmable event detection mechanism for INT metadata. Our solution comprise two parts: First, P4-programmable switches use INT metadata to collect state information from the data plane. Using different filtering mechanisms, the amount of unimportant telemetry information that is sent to the stream processor is significantly reduced, something which reduces both the monitoring overhead and the load on the stream processor. The filtering can be programmed on a per-flow basis from the control plane. Second, our solution comprises a high-performance telemetry report parser in AF_XDP, which streams remaining telemetry items to a distributed Kafka cluster and Elastic Search stack for further processing (e.g., analytics and machine learning) and visualization. In our evaluation, we use a testbed with P4-programmable network interface cards, and use several traces from real datacenter workloads to evaluate the impact of different event detection algorithms and threshold configurations on the event detection ratio and load on the stream processor. The paper shows that our solution scales to process several hundred millions of telemetry headers per second. In our future work, we will extend the event detection algorithms, and evaluate the impact of the more complex event detection mechanisms on switch performance.

Acknowledgement

Parts of this work have been funded by the Knowledge Foundation of Sweden (KKS) through the profile HITS.

References

[1] J. Hyun, N. Van Tu, and J. W. Hong, "Towards knowledge-defined networking using in-band network telemetry," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2018, pp. 1–7. DOI: 10.1109/NOMS.2018.8406169.

[2] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo, "Burstradar: Practical real-time microburst monitoring for datacenter networks," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, ACM, 2018, p. 8.

[3] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[4] Y. Kim, D. Suh, and S. Pack, "Selective in-band network telemetry for overhead reduction," in *Proceedings of the 2018 IEEE 7th International Conference on Cloud Networking, CloudNet*, 2018.

[5] D. Bhamare, A. Kassler, J. Vestin, A. Khoshkholgi, and J. Taheri, "Intopt: In-band network telemetry optimization for nfv service chaining," in *Proceedings of the 2019 IEEE ICC'19 CQRM Symposium*, 2019.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, ISSN: 0146-4833.

[7] F. Cugini, P. Gunning, F. Paolucci, P. Castoldi, and A. Lord, "P4 in-band telemetry (int) for latency-aware vnf in metro networks," in *Optical Fiber Communication Conference*, Optical Society of America, 2019, M3Z–6.

[8] B. Niu, J. Kong, S. Tang, Y. Li, and Z. Zhu, "Visualize your ip-over-optical network in realtime: A p4-based flexible multilayer in-band network telemetry (ml-int) system," *J. Lightw. Technol., submitted*, pp. 1–9, 2019.

[9] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, Heraklion, Greece: ACM, 2018, pp. 54–66, ISBN: 978-1-4503-6080-7.

[10] J. Rao, J. Kreps, and N. Narkhede, "Kafka: A distributed messaging system for log processing," 2011.

[11] D. Bhamare, M. Krishnamoorthy, and A. Gumaste, "Models and algorithms for centralized control planes to optimize control traffic overhead," *Computer Communications*, vol. 70, pp. 68–78, 2015.

[12] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo, "Quantitative network monitoring with netqre," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ACM, 2017, pp. 99–112.

[13] K. Borders, J. Springer, and M. Burnside, "Chimera: A declarative language for streaming network traffic analysis," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 365–379.

[14] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, "Gigascope: A stream database for network applications," Jan. 2003, pp. 647–651.

[15] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '18*, ACM Press, 2018.

[16] N. V. Tu, J. Hyun, G. Y. Kim, J. Yoo, and J. W. Hong, "Intcollector: A high-performance collector for in-band network telemetry," in *2018 14th International Conference on Network and Service Management (CNSM)*, Nov. 2018, pp. 10–18.

[17] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich, "Catching the microburst culprits with snappy," in *Proceedings of the Afternoon Workshop on Self-Driving Networks*, ACM, 2018, pp. 22–28.

[18] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 311–324.

[19] N. Van Tu, J. Hyun, and J. W. Hong, "Towards onos-based sdn monitoring using in-band network telemetry," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Sep. 2017, pp. 76–81. DOI: 10.1109/APNOMS.2017.8094182.

[20] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, and K. Rothermel, "P4cep," in *Proceedings of the 2018 Morning Workshop on In-Network Computing - NetCompute '18*, ACM Press, 2018.

[21] J. Vestin, A. Kassler, and J. Åkerberg, "FastReact: In-network control and caching for industrial control networks using programmable data planes," Sep. 2018.

[22] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwani, D. Daly, M. Hira, and B. Davie, *In-band network telemetry (int)*, Jun. 2016. [Online]. Available: https://p4.org/assets/INT-current-spec.pdf.

[23] I. Netronome Systems, "Nfp-4000 theory of operation," Tech. Rep., 2016. [Online]. Available: https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_TOO.pdf.

[24] J. D. Brouer and T. Høiland-Jørgensen, "Xdp–challenges and future work," in *Proc. Linux Plumbers Conference (November 2018)*, 2018.

[25] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. Mckeown, N. Feamster, B. Felderman, M. Blott, A. Moore, and P. Owezarski, "OSNT: Open source network tester," *IEEE Network*, vol. 28, no. 5, pp. 6–12, Sep. 2014.

[26] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17, London, United Kingdom: ACM, 2017, pp. 78–85, ISBN: 978-1-4503-5118-8.