A Novel Transport Protocol for Network Telemetry Collection

Authors

1. ABSTRACT

The emergence of programmable switches makes it possible to collect a large amount of fine-grained telemetry data in real time in large networks. However, transferring the data to a telemetry collector and relying on the collector CPUs to process telemetry data is highly inefficient. RDMA is commonly used as an efficient data transfer protocol but it does not work for telemetry data. This is because it is challenging to create RDMA format packets at switches, support lossy networks, and support various memory access patterns such as multiple concurrent writes. In this paper, we introduce a new direct telemetry data access system, which allows fast and efficient telemetry data transfers between switches and the remote collector. Our system introduce telemetry report primitives such as Key-Write, Append, Aggregation, and Key-Increment. To support these primitives and address the limitations of RDMA, we use some programmable switches as translators that generate RDMA packets, handle packet losses, and aggregate diverse memory accesses. Our evaluation demonstrates that we can enhance existing telemetry framework by XX%.

2. INTRODUCTION

ToDo: ***Write an introduction***

3. MOTIVATION

JL: This should build on background and dig into why CPU-based collection is inefficient. JL: Waiting for Gabriele results before planning structure

MY: Give background on classes of telemetry systems, and draw a table on their common requirements/bottlenecks on data collection/query.

Collectors play an important role in network telemetry systems: they receive telemetry reports and store the information in an internal data structure to be used to answer network-wide queries. One key challenge is to ensure that this process is scalable as a datacenter network can comprise hundreds of thousands of switches [3]. For example, a non-sampled INT telemetry system requires the collection of telemetry data from *every single packet*, which would result in an excessive amount of reports. Because of this, event detection is typically implemented at switches in an effort to send reports to a collector only when things change [5]. This helps in reducing the rate of switch-to-collector communication down to a few million telemetry reports per second per switch [12], at the cost of reduced network insight and increased on-switch complexity. Still, telemetry collection costs are high, and the main reason we identified is that the collectors' CPU is the main bottleneck. SR: I think it would be good to break down what steps that goes into collection. For instance, 1) a packet is generated from switch (or multiple switches), 2) packet is then routed to a collector, 3) at collector appropriate fields are read 4) collector then stores 5) some form of bookeeping is done to maintain large amounts of collected data. Then, we should say, we find steps 3,4,5 to be the bottleneck and move onto 3.1, 3.2 and 3.3. Can we also talk about potential race conditions here?

System	**Per-switch Report Rate**
INT (non-sampling)	4.75 Gpps
INT (0.5% sampling)	23.75 Mpps
Marple	950 Kpps
NetSeer	4.29 Mpps

Table 1: Per-reporter data generation rates by various monitoring systems. Assuming 6.4Tbps switches JL: I want something like this in motivation, but maybe not in table format?

3.1 Monitoring Systems are Intense

JL: Show some example numbers on the amount of telemetry repots generated.

3.2 CPU-based collection is inefficient

MY: key bottleneck or the ignored bottleneck in many telemetry systems today

MY: Discuss existing monitoring solutions

JL: Refer to table 1. Put this in comparison to benchmarked performance of Confluo

We show examples of what the CPU spends time on in Figure 1, and discuss how they already attempt to optimize these numbers. DTA would either bypass or offload these steps into hardware.

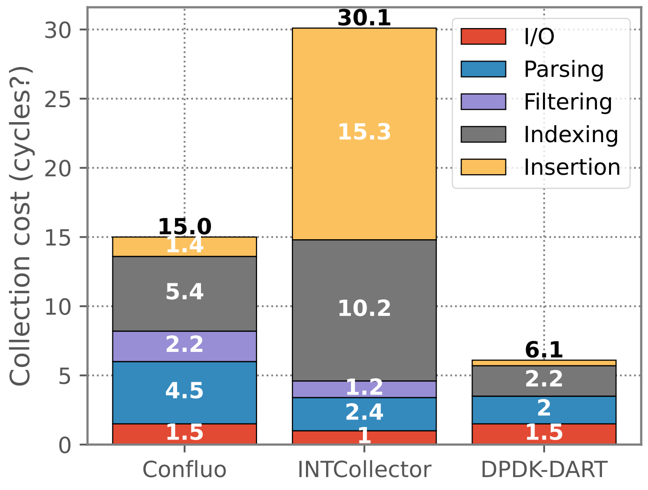

Figure 1: Breakdown of telemetry collection costs in CPU-based solutions. DPDK-DART is functionally equivalent to DTA Multi-Write (This is placeholder data!)

3.3 RDMA enables high-speed memory writes

What if we design telemetry collection around RDMA?

4. RDMA RESTRICTIONS ON COLLECTION

JL: Motivation ends with: RDMA has a huge performance potential! SR: Not too sure if it is needed, but I would find it benefitial to have a small background section that goes through RDMA primitives and requirements (such as consecutive packet identifiers). This would help the reader to understand the problems described in this section. Designing telemetry collection around RDMA has the potential for significant performance improvements. However, RDMA and related protocols impose several restrictions on such a design:

RDMA is limited to basic memory operations Achieving the full performance potential of RDMA requires restricting ourselves to one-way verbs, i.e., pre-defined primitives that execute entirely on the network card without requiring CPU intervention. This limits us to basic memory operations: *Read*, *Write*, *Fetch-and-Add*, and *Compare-and-Swap*. Restricting the design to these fundamental primitives greatly reduces the viability of dynamic data structures such as linked lists and cuckoo hashing. SR: Can we motivate why we need complex data structures and why not a simple store not enought?Such data structures require us to first read memory at the collector before knowing exactly what to write where, leading to significantly more complex on-switch logic. Further, one must ensure that there are no conflicts or race-conditions occurring between different switches that work in parallel to report telemetry information.

A limited number of RDMA connections RDMA network cards have limited memory, which constrains the number of active connections that fit in their local cache. Having more simultaneously active RDMA connections to a collector than can fit in the cache forces the network card to start swapping connection statefulness to server memory, which *significantly* reduces RDMA performance. Potential workarounds such as having several switches share the same connection become very complicated, due to the RDMA requirement of sequential packet identifiers within each connection, which would require sharing switches to synchronize these values on each new telemetry report. JL: Shall I benchmark this to confirm? Can not find source, but heard it in a private presentation

High-speed RDMA assumes a lossless network Another challenge related to the packet identifiers is handling network losses. Because RDMA assumes that incoming packets in a connection have sequentially increasing packet identifiers, even just a single lost RDMA packet would result in the RDMA connection state invalidating due to desynchronized packet identifiers between the switch and collector, which leads to the network card discarding every subsequent RDMA operation until the switch actively manages to resynchronize the connection.

Managing RDMA connections is costly for switches RDMA traffic is not designed for ease-of-generation in network switches. For example, the RoCEv2 protocol, which is the standard for RDMA communication for Ethernet, is especially costly. This protocol requires a relatively high amount of statefulness on a per-connection basic for tracking essential metadata, significant header overhead to calculate and generate, and a large footprint on internal busses handling RoCEv2-imposed checksumming. SR: Can you give an example? Something like, a single connection costs X MB, takes x amount of time to process. This is y% of the entire memory available. Requiring such a footprint on every single telemetry-reporting switch in a network is certainly doable, but very inefficient.

The data must be queryable Allowing efficient centralized access to telemetry data is the main reason for centralized collection. How to make it queryable? Cross-switch collaboration etc..JL: Too similar to first?SR: I am not sure if this is a problem with RDMA. Isnt it a general collection problem?

5. DTA OVERVIEW

DTA achieves scalability, generalizability, and high performance by decoupling telemetry reporting switches from the underlying storage backend. Reporters simply send a DTA packet, containing information about the telemetry data, towards one of the deployed telemetry collectors. These DTA packets are intercepted by the last hop towards the collector, by the top-of-rack switch. This last switch acts as a

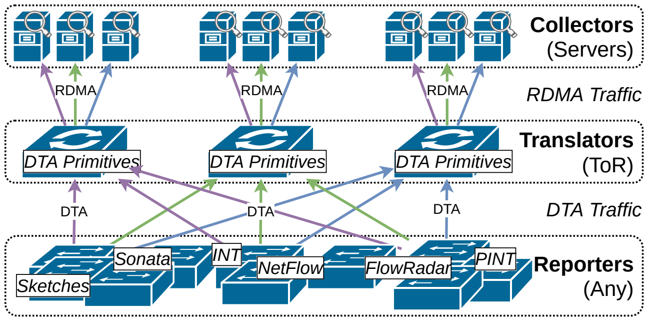

Figure 2: Overview of the DTA report data flowJL: Expand to 2 columns and add more info?

DTA translator, and is responsible for making the telemetry data queryable at the collectors in this rack.

The translator parses incoming DTA packets, and converts these into suitable RDMA operations that are forwarded to the right collector. Generated RDMA traffic is determined by the incoming DTA packet, which specifies the specific DTA primitive that is requested, together with essential DTA metadata according to the needs of the primitive (please see section 6 for a discussion on DTA primitives).

5.1 Translation Benefits

A naive RDMA-based telemetry collection design might generate RDMA traffic immediately at the telemetry generating network switches (i.e., at the reporters), which would require determining a storage server and a memory address inside of that server, and then generating an RDMA call to update the telemetry storage. SR: Suggestion: Maybe we can highlight the translator in telemetry generating switches as a challenge in the previous section? Then the solution below would become our contribution.

Introducing a DTA translator to intercept DTA traffic allows for several benefits compared to directly generating RDMA from individual switches:

Allowing central pre-processing without CPU involvement A translator is guaranteed to be the last processor for all telemetry packets destined to the collectors that it manages. This allows us to expand RDMA with new primitives and functionality that take advantage of a central viewpoint, including the addition of the powerful *Append* 6.2 primitive, centralized data duplicate detection and filtering, and real-time detection of network-wide events. SR: We need to motivate why we want the append feature earlier? Then we need to say why we cannot do this if telemetry generating switches that also do RDMA.

Improved collection speeds The translator is in full control of the RDMA state of each collector, which allows significant performance and stability improvements compared with designs where each reporter would handle their own RDMA communications directly. SR: What is stability? This includes eliminating RDMA queue-pair state swapping in the network card, which has a negative performance impact when several RDMA connections are active simultaneously. SR: Again, I think we need to describe this problem better in the earlier section. What is queue-pair state swapping?

Increased system stability RDMA expects network losslessness to achieve its high-performing potential, and a single RDMA packet lost in transit results in an invalid RDMA queue-pair state and a complete loss of throughput until the two RDMA end-points actively resynchronizes. Having the RDMA-generating switch being directly attached to the RDMA network card significantly reduces the risk of lost traffic, allows for immediate RDMA rate-limitation in case of system congestion, and reduces the RDMA resynchronization delay in the unlikely case that a packet is still lost.

Reduced in-network costs Managing RDMA states and packet generation inside of the switch ASIC is costly. Introducing the translators allows us to strip this footprint from ordinary telemetry-generating switches that exist all across the network, keeping this functionality and footprint inside only a few specialized collector-managing top-of-rack switches. Telemetry reporters would now only have to generate reports through the much more lightweight DTA protocol, designed specifically for ease of in-ASIC generation (see section 8.6 for a cost comparison).

DTA translation future-proofs telemetry systems Several commercial network cards support RDMA communication through the RoCEv2 protocol, which our translator has built-in support for. However, future network cards might very well include native telemetry collection capabilities JL: cite Bluefield latest update showing work towards this. Networks designed around DTA-based telemetry collection might not require network-wide switch overhauls to support new and high-speed telemetry collection techniques, since these network cards might either have native DTA support or allow DTA translation into the new format. SR: This would also be applicable to existing telemetry systems no? We could say, different networks runs different systems, and we can probably support all.

5.2 Translator vs SmartNIC Dilemma

JL: Where to place translator? ToR or NIC?

way 1: switch because xyz (needs to be strong if this approach). no good nic candidates that are not FPGAs?

way 2: we could do either. in this paper we look into what would be needed in a switch? we do in p4, new smartnics also P4, port easily (safer) if this, the discuss pros-and-cons

6. DTA PRIMITIVES AND "GENERALIZEABILITY"

Primitive	Example monitoring	Description
Key-Write	INT (Path Tracing) [2, 7]	INT sinks reporting 5x32b switch IDs using flow 5-tuple keys
	Marple (Host counters) [9]	Reporting 32-bit counters using source IP keys, through non-merging aggregation
	Sonata (Per-query results) [4]	Reporting network query results using queryID keys
	PINT (Per-flow queries) [1]	PINT can leverage the memory redundancies for improved data compression through $n = f(pktID)$
	PacketScope (Flow troubleshooting) [11]	Report per-flow per-switch pipeline traversal information using <switchID,flow 5-tuple> as key
Append	INT (Congestion events) [2, 7]	INT sinks report a period of network congestion to list of network congestion events
	NetSeer (Event aggregation) [12]	Sorting per-switch events into network-wide lists according to category
	Sonata (Raw data transfer) [4]	Sending selected raw data from switches to streaming processors, to be the base for query responses
	PacketScope (Pipeline-loss insight) [11]	On packet drop: send pipeline-traversal information to central list of pipeline-loss events
Sketch-Merge	QPipe (Heavy hitter detection) [6]	Per-epoch report of on-switch counters, merging network-wide sketches JL: let me know
Key-Increment	FlowRadar (Per-flow counters) [8]	Periodic transfer of on-switch counters to central storage using flow 5-tuple keys
	TurboFlow (Per-flow counters) [10]	Sending evicted microflow entries for central aggregation using flowKeys as keys
	Marple (Host counters) [9]	Reporting 32-bit counters using source IP keys, through addition-based aggregation

Table 2: Example telemetry monitoring systems and scenarios, as mapped into the primitives presented by DTA JL: Feel free to rearrange or replace some entries that are in the wrong place

Different monitoring systems each put their own requirements on a potential collection system. Telemetry collection designs therefore have to present several different primitives, to fulfil the collection needs of each system in a heterogeneous monitoring environment. DTA presents four different collection primitives that together enable support for a wide range of different monitoring systems: *Key-Write*, *Append*, *Sketch-Merge*, and *Key-Increment*. Table 2 shows examples of how current state-of-the-art monitoring systems can be integrated with DTA-based collection.

6.1 Key-Write

JL: expand The Key-Write primitive is designed to allow for key-value storage in DTA, where each piece of telemetry data has a unique pre-known key that can be used for retrieval of the reported data. There are several telemetry scenarios that can be expressed in a key-value fashion, and we present a few examples in Table 2.

Telemetry reporters would only have to send a single DTA packet towards one of the collection racks, containing the *key*, *telemetry data*, and a *redundancy* integer which will be further explained in section 6.1.1. The DTA translator will parse this Key-Write request, and generate RDMA traffic to insert this data into a global key-value store in an adjacent server. See section 7.2.1 for details of how DTA implements Key-Write in hardware.

6.1.1 Probabilistic Nature

ToDo: ***Explain the probabilistic nature of DART/Key-Write*** JL: Redundancy, checksumming, plurality voting. Return Empty and Return Error JL: We can likely reuse a lot of Michael's text from HotNets here

6.1.2 Querying JL: no querying here. all primitives together at end, showing how they preent as unified interface

ToDo: ***Explain querying in DART/Key-Write*** JL: Querying does not know level of redundancy N. Pre-define maximum N that reporters can request. Collector will always query up to let's say N=4, even if key did not get written to all 4 possible slots. Reporters can use this feature to set a value "importance"

6.2 Append

Some telemetry scenarios can not easily be collected through a key-value store. One such reason is if there are not obvious keys can can be pre-known and unique for all telemetry data in the system, or if the underlying telemetry data is not fixed-size but can grow over time. An example is if one wants to report high packet loss events in the network.

A more streamlined collection solution is to allow for reporters to append information into global lists containing a pre-defined telemetry category in each list. Network operators could then allocate dedicated lists for the types of telemetry data that they extract from the network. Examples of interesting lists could be *congested links*, *packet loss events*, *latency spikes*, or *suspicious flows*.

Telemetry reporters simply have to craft a single DTA packet to declare what data they want to append to which list, and forward it to a collection rack that hosts this list. The translator would then generate an RDMA packet that inserts this new telemetry data in the correct slot in the pre-allocated list. See section 7.2.2 for details of how DTA implements Append in hardware.

6.2.1 Querying

JL: Describe how an operator can query this data. Can we do this here before explaining ring-buffer design?

6.3 Sketch-Merge JL: Ran?

ToDo: ***High-level from user perspective***

JL: The design is not finalized. High-level is to allow network-wide merging of sketches. For example through

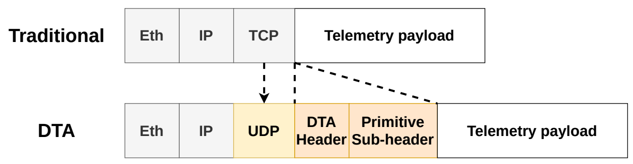

Figure 3: The structure of DTA report packets, compared with legacy TCP-based collection. DTA is designed to be transparent to the underlying network and monitoring system, and these headers will be entirely parsed and processed by the DTA network switches

fetch-and-add. Likely using switch SRAM as cache, periodically updating down into collector. Assuming same sketch dimensions everywhere

6.4 Key-Increment

ToDo: ***High-level from user perspective***

JL: Useful in for example flow counters. E.g., Marple, where we want to merge in-ASIC counters with either old cache-evicted values, or merge network-wide counters.

JL: Design not finalized. Same underlying algo as Key-Write/DART. Incrementing instead of overwriting. Basically building a sketch in server memory :). JL: CMS-like should work. Need to check sign support in the RDMA verb for CS-like support.

JL: We are ignoring potential performance limits of fetch-and-add here (I struggle to find numbers. let me know if you want me to benchmark)

6.5 Collector-Socket Interface

JL: Talk about how the memory handing is abstracted away. This is basically where you ask queries

6.6 DTA Report Structure

We designed a simple packet structure to be used for telemetry data transfer between individual telemetry reporting switches, and DTA translators. Our goal is to allow telemetry data to be forwarded across legacy network, without requiring a complete network overhaul for supporting DTA-based telemetry collection. Therefore, we chose to encapsulate DTA operations within UDP, allowing for legacy network support and even the potential for NAT traversal on its way to collection. One might want to omit the UDP header in cases where pure IP-based forwarding is enough, thereby reducing the bandwidth overheads slightly. DTA is designed to allow ease-of-integration with current telemetry monitoring systems, and requires just a small additional header for specifying DTA primitive specifics as shown in Figure 3.

The *DTA header* is shared for all DTA primitives, and is used to specify the underlying DTA primitive that is requested by including small *Opcode* field. Alongside this, the base header also reserves 8 bits for specifying flags, which could be either global (i.e., for all primitives) or primitive-specific. This includes an *immediate flag*, which is planned to be used to alert the collector of important information that should trigger an interrupt for immediate processing of an incoming report [1].

The structure of the *Primitive sub-header* is dependant on the DTA primitive that is requested by the previous header. This header will, in the case of key-write packets, specify the telemetry key and a requested level of redundancy. The size of the telemetry key is dependant on the telemetry monitoring system. For example, scenarios where the monitoring system reports per-flow information will likely have the size of the telemetry key be equal to the size of the flow 5-tuple (i.e., 13 bytes). Telemetry collection designs where several different data categories are collected simultaneously requires different opcodes to specify the structure of the primitive sub-header, to ensure correct field lengths according to the telemetry requirements. The redundancy value is used as a representation of how aging-resistant this data should be, and is used as the base for selecting a level of redundancy (N) in the key-value primitive as discussed later in section 6.1.1.

7. IMPLEMENTATION

Our DTA implementation has been released open-source[2] and consists of three main components: *reporters*, *translators*, and *collectors*. These three are all implemented, verified, and used as the base for later evaluations. The testbed where DTA was deployed consisted of several servers, RDMA-capable network cards, and one programmable Tofino switch. Reporters and translators are therefore verified in isolation. Reporters process incoming packet traces and generate DTA traffic that is recorded in a Pcap format. A traffic generator can later replay this DTA trace through a translator switch, which was used to verify the correctness of the reporter. The current DTA prototype has full support of the Key-Write and Append primitives in all components.

7.1 DTA Reporters

The reporter pipeline is written in P4_16 for Tofino, and is deployed on a BF2556X-1T switch. Controller functionality is written in Python, running on the switch CPU. DTA is designed to be generic, supporting a wide range of monitoring systems acting as telemetry reporters. The current implementation therefore focuses on just the components required for DTA report generation, while also including simple INT functionality for generating the telemetry data to report.

The Ingress pipeline is responsible for extracting telemetry information, and probabilistically flags packets to generate new telemetry reports. Packets are flagged based on a simple least-recently-used (LRU) cache of data checksums, acting as a change detection system, and additional random sampling for an even higher telemetry report generation rate.

[1]The DTA immediate flag is a proposal, and shall leverage the RDMA immediate functionality as specified by the Infiniband protocol to allow immediate CPU-handling at the collector. It is however not finalized as of the time of writing

[2]The current DTA codebase is publically available at <GIT LINK>

Packets that are flagged for DTA report generation are marked to perform packet mirroring into the Egress pipeline, which inserts two packets into the egress pipeline: the original packet, and a truncated packet clone.

Original packets are forwarded as normal user traffic, while the clones are used as the base for crafting DTA traffic for telemetry collection. The Reporter simply places in the DTA packets what kind of DTA primitive to trigger, together with primitive-specific metadata. This can be for example to write an INT-carried network path under the flowID key using the Key-Write primitive.

7.2 DTA Translators

The translator pipeline is written in P4_16 for Tofino, and is deployed on a BF2556X-1T switch. Controller functionality is written in Python, running on the switch CPU. Our current translator pipeline supports both the Key-Write and Append primitives, and can process these in parallel. There are four different pipeline-traversing paths for different types of network traffic, as shown in Figure 4.

JL: Mention why just keywrite and append

7.2.1 Key-Write JL: and key-increment

DTA Key-Write packets shall generate N RoCEv2 WRITE packets with identical payload, where the addresses point to N different memory locations where redundant copies the data shall be placed. This requires us to inject N packets into the egress pipeline, allowing us to mutate these into N different RoCEv2 packets. This mutation is achieved through multicasting, where the switch controller pre-configures multicast-rules for each $< N, collector >$ tuple. The rules are configured to inject N packet clones into the egress pipeline for the port to that collector NIC, where the packets hold their own value $n \in 0, 1, ..., N-1$ to be used for differentiating the redundant entries.

A lookup table has been pre-populated with metadata regarding the key-value store, including RDMA values, the memory address of the allocated storage buffer, the total size of the storage, and an index in the register that stores the queue-pair PSN tracker. The native CRC extern is used to calculate the key checksum that will be concatenated to the key-value entries, and to map $< key, n >$ into a slot-selecting hash which determines the destination memory addresses. This hash is further mapped into a range equal to the size of the key-value memory buffer through a match-action table, which maps the number of storage entries into a bitmask that can be used to perform modulo-operations by powers of 2[3]. The calculated memory slot is easily translated into a buffer-offset by multiplying it with the size of a single key-value slot (i.e., the sum of the checksum and data value lengths). This is followed by having the packet process the RDMA grace period enforcing block, to verify that RDMA traffic is not currently halted. Please see section 7.2.5 for a description of this block.

Finally, the N injected packets reach the RoCEv2-crafting block, which mutates these into the RoCEv2 standard. The switch then emits the N packets, leading to the N memory write operations to insert the telemetry data into the key-value store.

7.2.2 Append

DTA Append packets insert telemetry data into one of several data lists. These lists are implemented as ring-buffers, where a pointer is kept in the translator to track where in the buffer to append the next piece of incoming data. A naive approach would be to simply generate one RDMA packet for each incoming operation, inserting these values one-by-one. However, values for each list will be written sequentially and contiguously in memory, which allows us to significantly reduce the number of required RDMA calls by batching multiple values together and writing them to memory with a single memory operation.

This is achieved by storing incoming Append-data in SRAM during ingress processing. Our current implementation supports batching of data on a per-list and per-ingress-pipe basis, with batches containing 16 elements each[4]. One might want to design batch-storage in a compressed way by using just a single register to store all 15 cached telemetry entries until a new batch is ready for insertion into collector memory, resulting in a very efficient use of memory logic. However, a design like this would require significant recirculation during RDMA-creation to read back this data, which would degrade the collection performance. Instead, we store Append-data in 15 different registers, each of size 255, to store per-list data that will be used as the base for crafting the next RDMA call. An initial register is used to track the number of data units that have been stored so far for each list, which is used either to decide which of the 15 registers to write into, or to signal that the stored data should be read back and RDMA-creation should be triggered. This design results in egress-traffic and RDMA-creation being triggered just for every $16th$ packet in each list, where these packets have a payload containing 16 units of Append-data, resulting in a significant reduction of RDMA traffic (and therefore improved performance)[5].

Egress contains the logic necessary for issuing an RDMA operation to append the Append operation information into

[3]This implementation choice effectively limits the number of key-value storage slots to powers of 2, and is a simple workaround to the lack of flexible modulo support in the switch architecture. An alternative could be to hard-code a fixed key-value storage size, which allows power-of-2 modulo through compile-time known bitshifting.

[4]The size of the allocated lists should be a multiple of the batch size, to allow resetting the head pointer when the end of the buffer has been reached, without leaving empty gaps in the allocated buffer. Our implementation currently forces both of these to be powers of 2, which fulfills this requirement.

[5]The current batch-design assumes a constant stream of Append-operations to each list. A sudden disruption of incoming data can lead to failure to complete a batch, which would mean that this data will not be written to storage until the telemetry stream continues. In cases where telemetry disruptions are expected, one might want to periodically flush idle batches into storage.

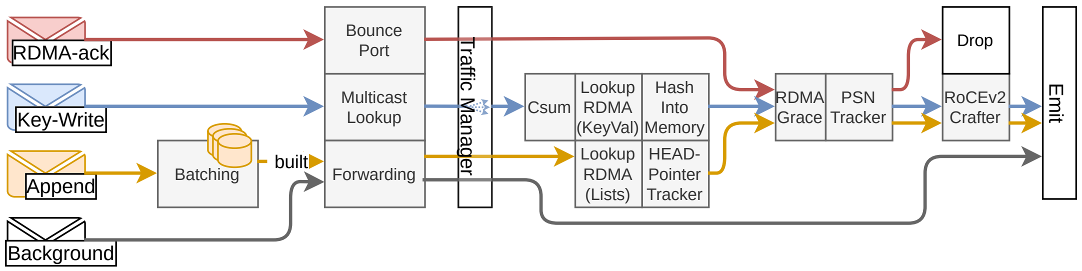

Figure 4: An overview of our translator pipeline implementation, capable of processing Key-Write and Append traffic. Four different pipeline-traversing paths exist for processing different types of network traffic in parallel, which efficiently shares pipeline logic JL: dual or single column?JL: visualize ingress and egressJL: add key-incrementJL: rdma-back -> qp-resyncJL: background->user traffic

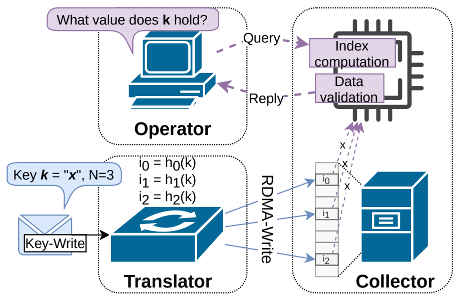

Figure 5: An overview of the DTA Key-Write primitive

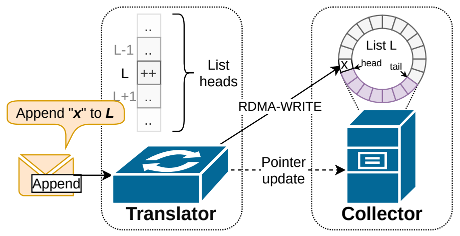

Figure 6: The DTA Append primitive.JL: add querying (somehow)

the specified list. The packet starts by looking up primitive-specific metadata for the chosen list in a pre-populated match-action table, containing essential RDMA information, the size of the allocated buffer, and a register index that holds the head and PSN trackers. This is followed by processing the per-list head-pointer tracking register, which increments the stored value by the size of the batch, which will be used as an offset when calculating the destination memory address for the data [6].

Append traffic then enters the same RDMA-generating pipeline as Key-Write traffic, including RDMA grace enforcement, PSN tracking, and RoCEv2 crafting. The pipeline yields a single RDMA operation being sent for every 16 incoming per-list append packets.

7.2.3 *Sketch-Merge*

JL: It would LIKELY... Reuse components... sketch fragmentation, pipeline caching, batch-writes/increments.

7.2.4 *Initializing RDMA Connections*

Tofino switches do not have native support for processing RDMA traffic. Dedicating limited ASIC resources for setting up RDMA connections is incredibly wasteful, because these processes only occur once during system setup and they are not performance critical. We instead instruct the translator switch CPU to act as a virtual RDMA client during the RDMA initiation phase, sending fake RDMA connection traffic from the Tofino controller service for each. The ASIC will simply forward these to the correct RDMA network card, and send the RDMA replies down to the switch CPU. Initial RDMA traffic is parsed by the switch CPU, where essential metadata regarding the initialized RDMA queue-pair are parsed and inserted into the Tofino ASIC. This phase extracts all RDMA metadata that is required for pushing full control of the RDMA connection into the Tofino ASIC, where all subsequent RDMA communications are generated after the initiation phase is finished.

This initialization process is repeated for every storage

[6]Moving the head-tracking functionality after the Grace block would prevent data gaps in the list during times of collection congestion, if that is preferred.

structure at the collector. The metadata that is extracted for each RDMA queue-pair connection includes:

Queue-Pair Numbers The RDMA network card automatically assigns each connection a unique queue pair identifier, which is included in every RDMA packet from the translator to collector. Per-storage queue-pair numbers are written to an in-ASIC match-action table, and used when crafting RDMA-WRITE packets and processing RDMA acknowledgements from the collector.

Remote Keys Memory buffers that are registered with RDMA get assigned keys that function as passwords to prevent unauthorized remote memory calls. These keys are parsed during the connection phase, and are inserted into an in-ASIC match-action table for use when crafting RDMA-WRITE packets.

Initial Packet Sequence Numbers (PSNs) The RDMA network card gets assigned a random initial PSN, and advertises this value to the translator. The controller parses this value, and updates in-ASIC registers that track per-queue-pair PSN numbers.

Memory information Essential memory information is also extracted during this phase. These are the size of the allocated memory buffer, and the memory address to the buffer. These are both essential for crafting RDMA-WRITE calls for incoming DTA packets, and are written to in-ASIC match-action tables for the primitive that is able to write to that memory buffer.

7.2.5 *Queue-Pair ResynchronizationJL: Move to architectuve/design*

Just a single lost RDMA packet between the translator and collector is enough to desynchronize the RDMA queue-pairs in the two devices, leading to a complete loss of RDMA throughput due to generated packet sequence numbers (PSNs) not being as expected by the NIC.

Queue-pair resynchronization is achieved by implementing in-translator support for processing RoCEv2 acknowledgements that are coming from the collector NIC. In-NIC generation of these packets is triggered when the NIC RDMA engine processes RoCEv2 traffic containing PSNs that are not sequential, which implies that the NIC-side of the system is congested and can not keep up with the current collection load.

These acknowledgements are sent from the collector NIC back to the translator, where they are parsed. The translator ingress-pipeline instructs these packets to bounce back to the same port where they ingressed, which will ensure that the acknowledgements enter the correct egress pipe where the desynchronized RDMA queue-pair state is stored. The PSN-tracking register, used while crafting RDMA packets, is reset according to the value specified by the acknowledgement-packet. This allows for future RDMA traffic in the queue-pair to generate valid PSNs, instead of using the desynchronized counter that resulted in the invalid queue-pair state.

Buffers inside of the collector NIC are filled with invalid RDMA traffic, and require a grace period for clearing these in preparation for new and valid traffic post-resynchronization. Failure to let these buffers clear would result in immediate queue-pair desynchronization once again (please see section 8.5 where we show this phenomena). The acknowledgement-packet updates a per-egress-port register that specifies a number of RDMA packets that should be dropped by the translator during this grace period, and the acknowledgement is then instructed to be dropped as well. Normal DTA traffic, which typically mutates into RoCEv2 packets, decrements this register and reads back the counter. If this counter is greater than zero, then the packet is instructed to drop in the egress deparser without entering the RoCEv2-crafting block. This effectively prevents RDMA traffic and PSN increments during the grace period. As soon as the register has decremented back to zero, indicating that the set number of RDMA packets has been dropped in the grace period, collection proceeds as usual from that point onwards.

7.3 Collector JL: rename. Collector-RDMA Interface? JL: Move/rewrite into design instead?

JL: we implemented socket in X lines of code... c++... basically does conversion between rdma and xx

The DTA collector service is written in C++, using standard Infiniband RDMA libraries and the RDMA Communication Manager (RDMA_CM) for setting up queue pair connections between the collector and translator. Server BIOS has been optimized for high-throughput RDMA, and all RDMA-registered memory is allocated on 1 GB huge tables. We recommend recompiling the Infiniband drivers to allow support for physical memory addresses, since random accesses in large memory spaces might result in a performance degradation due to IOTLB cache misses while translating from virtual to physical addresses.

Each collector service supports registering one key-value region, to be used for the Key-Write and potential Key-Increment primitives, and several parallel lists for the Append primitive. These are all advertised through RDMA_CM using unique ports, allowing the translator controller to request queue-pair connections to each region one-by-one. The collector RDMA service responds back to the translator with an RDMA SEND operation after an RDMA queue-pair connection is established between the collector and translator for one of the storages. This response contains essential metadata about the attached storage, including the total size of allocated memory, and the memory address to the start of the storage buffer.

Initializing RDMA connections is an essential task for the collector CPU. After that is done, the CPU enters an idling phase, since the rest of the collection tasks are entirely CPU-bypassing. The CPU can now dedicate its entire processing capacity to handling telemetry queries by reading the memory buffers that are advertised as DTA storage.

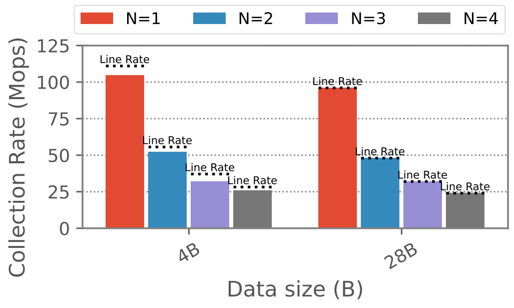

Figure 7: Single-NIC collection-rate of DTA Key-Write operations on a 4GiB key-value store, at various redundancy levels N and payload sizes. The RDMA payloads include a 4B concatenated checksum. Note how the payload size does not impact performance until we reach line-rate at 32B payloads

7.3.1 *Presenting APIS JL: find good name for querying section (interface to controller backends)*

JL: Discuss querying. How do we abstract away the primitive memory mapping?

JL: This is basically like the DTA socket

8. EVALUATION JL: ME

All benchmarks in this section are performed using a single port of the Mellanox Bluefield-2 2x100G DPU as acting RDMA-capable network card, and using a BF2556X-1T Tofino switch as both reporter and translator. The collection speeds that are presented in this section are entirely dependent on the speed of the RDMA hardware, i.e., the RDMA message rate of the network card, which is the current bottleneck in our system. We therefore expect that even higher collection rates should be achievable if one were to use a more powerful network card in the future, and our current translation pipeline to be compatible as long as the network card supports the RoCEv2 protocol. The TReX traffic generator is used to inject DTA packets into the translator switch.

8.1 Key-Write Primitive Performance

DTA Key-Write is a powerful primitive that allows global collection with dynamic keys, without requiring cross-network collaboration or synchronization. It does however impose overheads in terms of built-in checksumming and redundancies. This section will evaluate the high-level performance of the Key-Write primitive, and the effectiveness of Key-Write internals.

8.1.1 *Key-Write Collection Rate*

We have benchmarked the raw performance of the DTA Key-Write primitive. JL: Will continue when new tests are finalized

JL: explain test in Figure 7 JL: Pushing traffic at increasing rates 10 seconds at each rate. Stopping at first packet loss

Parameters: level of redundancy, size of storage. Size of report payloads?

8.1.2 *Redundancy Effectiveness*

JL: This section is heavily based on the hotnets paper

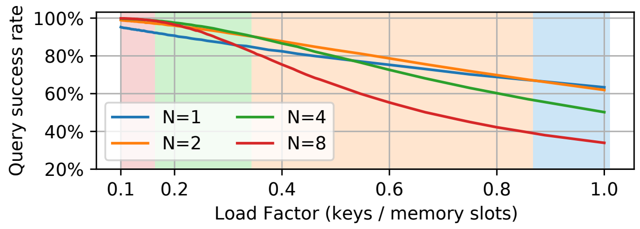

Figure 8: Average query success rates delivered by the Key-Write primitive, depending on the key-value store load factor and the number of addresses per key (N). The background color indicates optimal N in each interval.

The probabilistic nature of the DTA Key-Write primitive cannot guarantee final queryability on a given reported key. We show in Figure 8 how the query success rate depends on the load factor (i.e., the total number of telemetry keys over available memory addresses), and the number of memory addresses that each key can write to. There is a clear efficiency improvement by having keys write to $N > 1$ memory addresses when the storage load factor is in reasonable intervals, and the background color in Figure 8 indicate which number of addresses per key (N) delivered the highest key queryability in each interval.

Recall that RDMA lacks the ability to write into multiple memory addresses with a single packet, and that DTA overcomes this by generating N distinct RDMA packets for writing the redundancy to memory. Requiring N DMA calls per DTA operation results in a decrease of collection throughput, as shown earlier in Figure 7. Determining an optimal redundancy level therefore has to be a balance between an enhanced data queryability and a reduction in collection performance. $N = 2$ appears to be a generally good compromise, showing great queryability improvements over $N = 1$.

8.1.3 *Data Longevity*

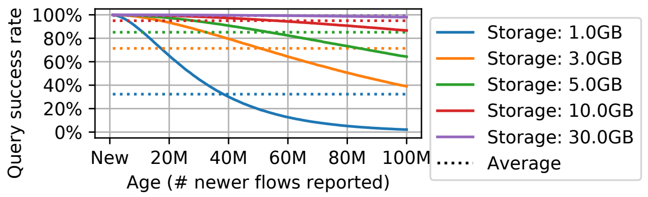

Figure 9: DTA Key-Write ages out stale data. This figure shows INT 5-hop path tracing queryability of 100 million flows at various storage sizes. The results are based on storing 5x32-bit values with 32-bit checksums, where each key writes into $N = 2$ different memory addresses.

JL: Explain figure. Simulation, testing how long until data ages out

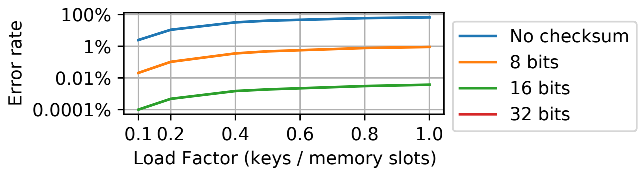

Figure 10: The probability of a query returning the wrong answer in error, due to address and checksum collisions.

8.1.4 *Query Answer Correctness*

JL: Section is heavily based on hotnets paper

There is a theoretical risk of the Key-Write primitive returning incorrect query results if two or more different keys hash into the same memory address, while they at the same time also have the same concatenated checksum. JL: There is so much more than this. Can we fit some theory in the paper? Maybe as appendix? Figure 10 shows results after extensive tests, where multiple simulations of 100M keys have been performed at various storage sizes in an attempt to recreate the theoretically predicted incorrectness. These results clearly show the impact from having key-based checksums included in the DTA Key-Write data structure, with increased lengths greatly reducing the risk of errors. Our simulations with 32-bit key-checksums fail to reproduce these return-error cases, due to the incredibly low probability that these events occur, even while the data structure is under immense load.

Return error cases can be further reduced through data consensus during the query phase, where multiple identical redundancy slots are required to return back a query response. Requiring consensus will reduce the overall query success rates, and is only recommended in scenarios where incorrect query results can not be tolerated regardless of their frequency. See section 6.1.1 for a deeper discussion on Key-Write correctness.

8.1.5 *Query Speed*

JL: How many queries can be answered per second? Just run it iteratively alongside collection JL: Parameters: Level of redundancy, load factor?, data size?

8.2 Append Primitive Performance

JL: explain test in Figure 11

JL: outline test. explain that this is benchmarking the egress performance (since we lack traffic generators powerfull enough to saturate this). It is now more likely that Tofino ASIC Ingress would bottleneck in terms of link speeds or clock

JL: Explain very clearly and emphasize that the 100G LINK is the bottleneck, and that we collect at line-rate. The next step would be to replace RoCEv2 to reduce header overheads.

8.3 Querying rate

JL: How fast can we query?

JL: Both Key-Write at different redundancies, append, Confluo, and INTCollector

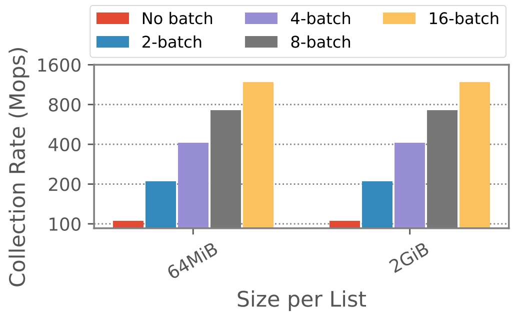

Figure 11: Single-NIC ingestion rate of DTA Append operations, at different batch sizes. The telemetry data entries in the lists are 32-bit integers. Note that these are translator ingress-rates, while egress rates will be reduced by a factor of the batch size.

8.4 DTA Collection Capabilities

JL: Per-switch generation rates of monitoring systems come from their papers. State numbers and cite sources. Present in table format, together with brief description of mapping into DTA? (everything hypothetical)

JL: Marple: evaluating packet counters using properties of non-mergeable aggregations (figure 10 and end of section 5.2). Let's say the DTA epoch is shorter than cache eviction

JL: Possibly not in eval? Since this is potential performance (extrapolating from our primitive tests) and their paper numbers

$$required\_collector\_nics = \left\lceil \frac{num\_switches * switch\_report\_rate}{max\_primitive\_rate} \right\rceil$$

8.5 Queue-Pair Resynchronization

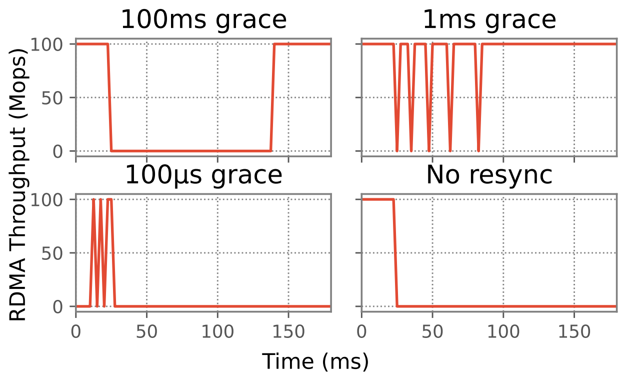

Figure 12: Translator-to-NIC queue-pair resynchronization, at varying grace period lengths. We see how DTA quickly and reliably manages to resynchronize the system after periods of collector-side congestion

The RDMA connections have to be actively resynchronized when a packet loss has occurred. To evaluate the efficiency of our resynchronization mechanism, we sent a constant stream of DTA Key-Write traffic to the translator at a high enough rate, where we start to detect sporadic packet loss on the RDMA-NIC side of the system. In parallel to the Key-Write operations, we run a stream of Append traffic

Resource	Cost w/o batching	Cost w/ batching
SRAM	5.4%	8.5%
VLIW Instruction	6.5%	8.1%
Match Crossbar	4.8%	14.4%
Hash Bits	8.6%	16.0%
Hash Dist Unit	12.5%	34.7%
Meter ALU	8.3%	41.7%

Table 3: Full resource costs in the translator, both while supporting 16-entry Append-batches for 255 lists of 32-bit values, compared with a pipeline that omits batch-building support

with incrementing values into a single list at a fixed inter-packet gap. We run this traffic through the system for a set period of time, and analyze the content of the Append-list. This list can now be seen as a representation of the NIC congestion, since periods of desynchronization and loss will result in gaps in the incrementing values contained in the list.

Figure 12 shows the effectiveness of our queue-pair resynchronization mechanism at various lengths of grace periods, compared with a system without active resynchronization. This figure motivates the need for including a grace period post-resynchronization, to allow the NIC to flush internal buffers of invalid RDMA traffic before accepting new telemetry data. Our tests show that a grace period of $1ms$ is enough to allow for queue-pair resynchronization, while grace periods of $100\mu s$ or shorter fails to keep the system stable post-resynchronization. Very long grace periods, on the order of $100ms$ or longer, manage to stabilize the system but are very wasteful due to unnecessarily long periods of disabled RDMA throughput.

8.6 ASIC Resource Footprint

8.6.1 Translator Resources

DTA translators are responsible for several tailored processes, some of which are resource intensive. Especially the batch-functionality for the Append primitive has a high resource cost in terms of stateful ALU logic (meter ALU), since a non-recirculated batch-reconstructing algorithm requires the ability to read all batch entries from memory during a single pipeline traversal. Table 3 shows the total resource usage of our translator implementation, both with and without built-in Append-batch support. It is clear that there is a significant resource cost for including Append-batching, which is expected. However, batching also has the potential for a tenfold increase in collection throughput, and we therefore argue that this is still a worthwhile cost. A compromise is to reduce the size of these batches, the size of which linearly correlates with the number of additional meter ALU calls.

Building in support for more Append-lists does not require additional logic in the ASIC, it just necessitates more statefulness for keeping per-list information (e.g., head-pointers and per-list batched data). Note how the actual SRAM size requirement has just a 3.1% increase imposed by batching, which shows that the translator can support much more complex list setups than the 255 lists that are included at the time of evaluation. JL: Find out max number of supported lists?

8.6.2 Reporter Resources JL: skip?

JL: Compare pipeline cost of generating RDMA vs generating DTA. Include PSN resync? PHV bits?

9. LIMITATIONS

ToDo: ***Explain limitations of current DTA design***

9.1 Limited Aggregation Capabilities

ToDo: ***Explain how we are limited in data pre-processing and merge-capabilities***

JL: We can't really do RDMA Reads to efficiently merge data

JL: We can definitely utilize collector CPU to perform these, and likely significantly faster than current state-of-the-art.
Example: an Append list containing tuples <address,value,function>.
CPU can just iterate over this list and perform this action.
Main drawback here is that it doesn't fit our paper story

9.2 Real-time Reactivity

ToDo: ***Discuss how CPU easily can immediately react to events depending on complex rules*** JL: We can do this (e.g., RDMA Immediate or CPU constantly polling lists), but not necessarily "real-time" as in part of in-line processing

9.3 Append Data Sizes with Batching

JL: Due to register busses max 32b output

9.4 RDMA rates

ToDo: ***Explain how the translator can grow into the bottleneck, given that we RDMA-side performance is incredibly. Just a dozen RDMA-capable NICs, and the translator can no longer keep up***

JL: This is a good problem to have, but this also means that Tofino ToRs fundamentally requires distributed collection. Not sure if this is a limitation though, since we are designed to do this. Maybe remove this section?

10. DISCUSSION

ToDo: ***Add a discussion. Feel free to modify how you see fit :)***

10.1 Translation vs Programmable SmartNICs

ToDo: ***Discuss the use of SmartNICs for DTA collection***

JL: Translation is cheaper than programmable NICs (legacy RDMA support). DTA COULD support a hybrid of translators and SmartNICs (maybe some primitives require SmartNIC collection, while others do not? No reason why DTA would not be future-proof here for DTA-optimized NICs). Translators work in the meantime

JL: Let's say opcode 0x01 is KeyWrite-1 (designed around translator), while 0x11 is KeyWrite-2 (designed around SmartNIC and Cuckoo). Both from user-perspective used for the same thing, but different implementations. JL: Maybe even the same opcode, just that the translator/smartnic chooses underlying algorithm to process the request? Plug-and-play :)

JL: SmartNICs would allow more complex central processing. Also (likely) easier to design around memory reads

JL: Vision for standardized DTA, allowing vendors to build in support in fixed-function NICs?

10.2 Read-based Writes

ToDo: ***Discuss what we could do if we could read storage before writing. New collision mitigation or aggregation functions***

10.3 Large-scale Deployments

ToDo: ***Discuss how DTA might be deployed in hyperscale networks.***
Requiring several collection clusters. Reporters already hash into one of several collectors.
Possibly segment network to reduce cross-network report tranmissions (one reporter to distant collector)

10.4 Collection Epochs and History

ToDo: ***Discuss the need for epoch-based collection. DRAM is required for high-speed collection. Periodic transfer to disk is required for persistent historical storage.***

11. RELATED WORKS

ToDo: ***Add related works***

11.1 TEA

ToDo: ***Discuss TEA***

12. CONCLUSION

Conclusion goes here

Acknowledgements

Text goes here

13. REFERENCES

[1] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. Pint: probabilistic in-band network telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 662–680, 2020.

[2] T. P. A. W. Group. Telemetry report format specification. `https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report_latest.pdf`. Accessed: 2021-06-23.

[3] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[4] 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*, pages 357–371, 2018.

[5] Intel. In-band network telemetry detects network performance issues. `https://builders.intel.com/docs/networkbuilders/in-band-network-telemetry-detects-network-performance-issues.pdf`. Accessed: 2021-06-04.

[6] N. Ivkin, Z. Yu, V. Braverman, and X. Jin. Qpipe: Quantiles sketch fully in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 285–291, 2019.

[7] 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.

[8] 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)*, pages 311–324, 2016.

[9] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[10] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.

[11] R. Teixeira, R. Harrison, A. Gupta, and J. Rexford. Packetscope: Monitoring the packet lifecycle inside a switch. In *Proceedings of the Symposium on SDN Research*, pages 76–82, 2020.

[12] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.