Direct Telemetry Access

Paper # 224, XXX pages

ABSTRACT

The emergence of programmable switches enables collecting a vast 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 introduces 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 the existing telemetry framework by XX%.

1 INTRODUCTION

Network telemetry is the foundation for managing modern data centers [6, 23, 28, 35, 36, 60, 64, 67, 68]. With the rise of programmable switches [7, 29, 50], network telemetry systems today can monitor network traffic in real time and at high granularity [41**? ? ?**]. There have also been many proposed telemetry systems that enable both automated network control [1, 21, 42] and advanced troubleshooting [18, 28, 55]. many telemetry systems are built around centralized collection of network-wide reports [4, 10, 24, 28, 48].

MY: still editting... will get back

However, the communication channel between programamble switches and query systems. A key problem is that a production datacenter network can comprise hundreds of thousands of switches [18], each generating up to millions of telemetry data reports per second [68], thus requiring thousands of CPU cores just for real-time data collection [39]. As a consequence, it is increasingly hard to build a suitably scalable collection system [35, 61]. Existing research boosts collectors' scalability by improving their network stacks [35, 61] or by preprocessing [40] and filtering data at the switches [27, 38, 46, 62, 68]. Our insight, however, is that the current main bottleneck of collectors is the cost of inserting incoming reports in queryable data structures (§2).

We propose *Direct Telemetry Access* (DTA), a telemetry collection protocol specifically designed for moving and aggregating reports from switches to collectors' memory. The aim is to architect a solution that completely relieve collectors' CPU from any sort of processing triggered by incoming reports, so to minimize as much as possible their computational needs. The underlying idea is to have switches generating RDMA (Remote Direct Memory Access) [26] calls towards collectors. RDMA is a technology available on many commodity network cards [30, 56, 63] that can perform hundreds of millions of memory writes per second [56], significantly faster than the most performant CPU-based telemetry collector [35]. Generating RDMA instructions directly from switches is possible [37] but it also raises several challenges when used for telemetry collection: (1) RDMA performance degrades substantially when multiple clients write to the same server [34]. This conflicts with the needs of telemetry collection, where multiple switches need to send their data to few collectors; (2) High-speed RDMA requires a lossless network: if a switch and a collector desynchronize, all subsequent reports are lost. (3) Managing RDMA connections at switches is costly in terms of hardware resources, thus limiting their ability to perform other tasks such as monitor data plane traffic; (4) RDMA utilizes only basic memory operations, limiting the ability of RDMA-based schemes to aggregate and store data in complex queryable data structures.

To address these challenges, switches send their reports encapsulated by our custom protocol header, and the last-hop switch takes responsibility for report aggregation, batching, and establishing an RDMA connection to a collector. This reduces the cost by introducing RDMA to fewer switches, and avoids performance degradation as now the RDMA NICs at the collectors have to manage only a small number of connections. The last-hop switch is also in charge of translating DTA traffic into standard RDMA calls. This choice allows us to design a number of telemetry-collecting primitives such as *Key-Write*, *Append*, *Sketch-Merge*, and *Key-Increment*, that permit DTA to support state-of-the-art monitoring systems such as INT [17], PINT [6], Marple [46] or Sonata [19], to name a few.

We discuss DTA's design (§3), and share the challenges in implementing it on commodity programmable switches (§4). We show..(evaluation)...

Our main contributions are:

- We study the bottlenecks of state-of-the-art network collectors and show that their performance is bottlenecked by

the CPU ability to process incoming data and store it in queryable data structures.
- We propose *Direct Telemetry Access*, a novel telemetry collection protocol generic enough to support major network monitoring systems recently proposed by the research community.
- We implemented our idea using commodity programmable switches and RDMA NICs and will open source code upon publication to foster reproducibility.

2 MOTIVATION

Networks today are growing in size to support large amounts of traffic [18]. To enable advance control operations and improve performance, operators are seeking for solutions capable of getting increasingly fine-grained insights into data plane traffic [17, 46, 70]. The problem is that data collected at switches has to be exported to collectors to gain a network-wide view [35]. As a result, monitoring all simultaneous events can be challenging [68].

In Table 1, we show the number of telemetry reports generated by *a single switch*, when using different state-of-the-art telemetry systems as reported on their papers.Ran: Does INT report any such numbers or is this based on calculations? At the worst case, reports can be generated at the rate of GA: XX, when adopting INT working in postcard mode [27] on a commodity 6.4Tbps switch and when considering a load of approximately 40% which is standard in production data center networks [65]. GA: I think this is a bit too much probably. INT has change detection, not sure reviewer will buy it and it might sounds like we are trying to exagerate here.Ran: I agree that this sounds like an exaggeration; no need to be so bold :). Recognizing this problem, latest solutions adopt smart pre-processing and filtering of data at switches so to reduce as much as possible the pressure on collectors. For example, with NetSeer every switch generates approximately 950K reports every second [68].Ran: Let's add Barefoot's change detection citation here.

In practice, data center networks are big, potentially comprising hundreds of thousands of switches [18]. As a consequence, optimizing the collector stacks is becoming essential to minimize the number of cores used for *data collection* [35].

In current systems, collectors spend CPU cycles in receiving a report (known as I/O) and processing it in the the user-space to store the information in a queryable data structure. This last step usually involve three key operations: *parsing*, *data wrangling* and *storing*. Indeed, collectors must parse the content of the report, process the data to make it suitable for insertion and determine where to stored it in memory. Using DPDK as the I/O framework can improve performance over the common socket interface [39], but the rest of the user-space processing often remain the bottleneck. To better understand this, we studied the behavior of Confluo [35], the state-of-the-art collector designed for high-speed networks. We found that the cost of user-space processing is dominant (SR: X%) when compared to the I/O (SR: Y%). We then analyzed the cost of every operation performed at user-space as reported in Figure 1. Data wrangling in Confluo involves filtering, where certain user based criteria are applied to extracted telemetry reports. Storing involves batching and insertion, where reports are grouped into batches scheduled to be written in memory. We found that most of the CPU cycles (88%) are spent in data wrangling and storing.

Ideally, if we are able to reduce the CPU utilization required by collectors, they can better scale with the size of a network. RDMA can potentially assist in this as it allows to write to memory without any CPU involvement. We could then build a strawman solution where switches write their reports in collectors' memory with RDMA calls. Although this idea is attractive, and generating RDMA instructions directly from switches is possible [37], it presents several challenges when applied for telemetry collection:

(1) RDMA is limited to basic memory operations. Ran: This seems to contradict with https://mcanini.github.io/papers/redn.nsdi22.pdf. Should we say that it is possible to extend RDMA but the cost is high? Telemetry data has to be stored in the collectors' memory in such a way that it is then easy to query. For example, using data structures such as linked list or cuckoo

System	Per-switch Report Rate
INT Postcards (Per-hop latency, 0.5% sampling)	23.75 Mpps
Marple [46] (TCP out-of-sequence)	6.72 Mpps
Marple [46] (Packet counters)	4.29 Mpps
NetSeer [68] (Flow events)	950 Kpps

Table 1: Per-reporter data generation rates by various monitoring systems, as presented in their individual papers. We are assuming 6.4Tbps switches. JL: Show in relation to Confluo performance, when we have throughput numbers

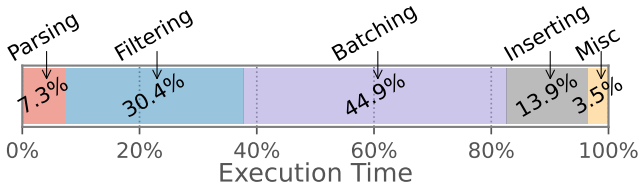


Figure 1: CPU-based collectors are inherently inefficient. Shown above is the collection work breakdown of Confluo as a demonstrationJL: todo: show clock cycles on y-axis. update with bare-metal results (waiting for Garbriele results)

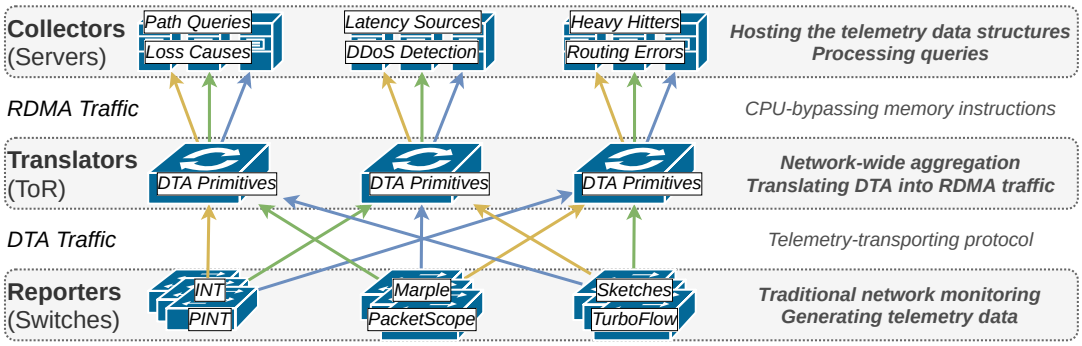


Figure 2: Overview of the DTA telemetry reporting flow JL: add flow control packets?

hashing can reduce data retrieval times significantlyGA: REF. However, RDMA provides support for very limited operations: *Read*, *Write*, *Fetch-and-Add*, and *Compare-and-Swap*. This makes it hard to implement the mentioned data structures as they require us to first read memory at the collector before knowing exactly what to write and 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.

(2) High-speed RDMA assumes a lossless network. An RDMA receiver expects incoming packets to be in sequential order. A lossy network can lead to queue-pair invalidation which significantly degrades RDMA performance [45]. For this reason, current deployments adopt per-hop flow-control mechanisms such as PFC [25] to rate-limit RDMA traffic in case of congestion [44, 45, 69]. Unfortunately, this practice can introduce deadlocks, a circular buffer dependency between switches that cause network throughput collapse [22]. Furthermore, as telemetry traffic would run on the data plane, it would force applications' traffic to be subjected to PFC rules as well. It is be possible to put the two traffic classes on different PFC priorities but this would mean strictly favouring one over the other.

(3) A limited number of RDMA connections. RDMA NICs have limited memory, which constrains the number of active connections (also known as *queue pairs*) that fit in their local cache. This limits the total number of switches that can generate telemetry RDMA packets to the collector. To support more connections, the collector can swap the connection states from the RDMA NICs cache to its own memory. However, this can degrade RDMA performance by as much as a factor of 5 [13]. Alternatively, several switches can share the same queue pair, but are limited by RDMA's requirement of sequential packet identifiers within each queue pair, which becomes impossible to achieve in a distributed network of switches. Therefore, to benefit from RDMA, it would be ideal to keep the number of active queue pairs low to ensure maximum performance [34].

(4) Managing RDMA connections is costly for switches. Programmable switches are not designed to generate RDMA telemetry packets. The standard protocol used for RDMA communication (RoCEv2) requires to maintain states for connection that can be expensive to implement. For instance, the switch will be required to store metadata, generate appropriate headers, and its associated checksums. However, current programmable switches are limited in memory, computational logic, and internal bus bandwidth.

3 DIRECT TELEMETRY ACCESS

In Figure 2, we show an overview of our solution. Reporters (i.e., switches exporting telemetry data) send their information to the deployed collectors using a special DTA packet format (§ 3.1). The last-hop (i.e., top-of-the-rack switch) is then responsible of intercepting those packets and converting them into suitable RDMA calls to collectors' memory (§ 3.2). This architecture brings several benefits compared to a naive solution where all switches in the network report their data to collectors through standard RDMA calls. In the following, we provide a one-to-one mapping on how we overcome the challenges discussed in the previous Section.

RDMA calls customization. The translator is located at the last hop before the report reaches the targeted collector. Following the spirit of past works that explored ways to extend RDMA [3], this component can be used to customize 1-sided RDMA operations and design them specifically for telemetry reporting (i.e., write data in queryable data structures). In Section 3.1, we introduce new primitives that reporters can use when interacting with the translator which in turn is the only one responsible for managing the memory state at the collector and thus in the perfect place to perform data aggregation.

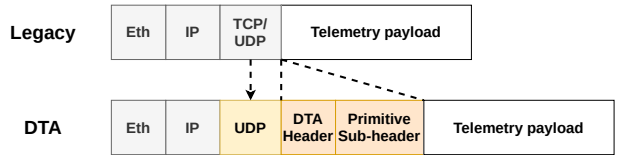
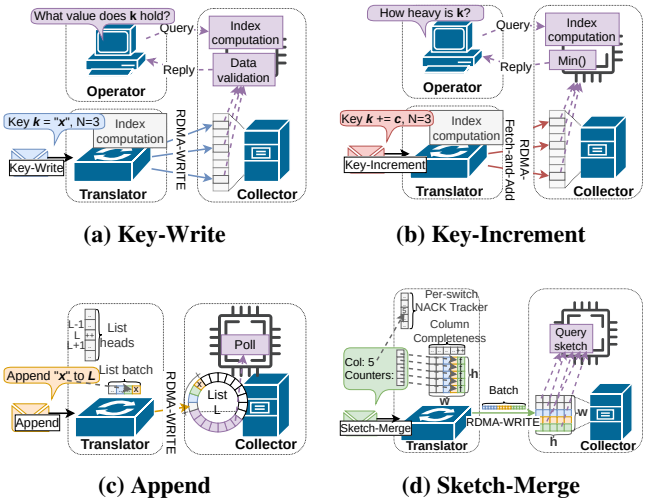


Figure 3: The structure of a DTA packet report, compared with a legacy TCP-based one.



(a) Key-Write (b) Key-Increment

(c) Append (d) Sketch-Merge

Figure 4: A high-level overview of the proposed DTA primitives

Data prioritization and telemetry-flow control. The translator is the only network component that has to create a point-to-point RDMA connection to the collector. This means that applications' traffic is not impacted in any way by specific flow-control mechanisms adopted. Moreover, by having the translator as the only source of RDMA traffic to the collector's NIC, even apply a rate-limiting scheme would help in avoiding packet loss [2, 32, 33]. The problem is that, while the ability of the translator to consume data is bounded by the capabilities of the collector, there is no flow-control between reporters and translator. In Section3.2, we introduce our ad-hoc telemetry flow-control mechanism and show also how our solution can prioritize specific data so to avoid the loss of important information.

Reduced number of RDMA connections. Standard RDMA NICs suffer performance penalties when an increasing number of connections are created simultaneously [13, 34]. With the proposed architecture, thanks to a translator that acts as an aggregator for all the reports coming from the various switches, it is possible to greatly reduce the number of connections to be handled by the collector RDMA NIC and hence keep performance high.

Reduced in-network costs. Managing RDMA states and packet generation inside of a switch ASIC is resource intensive. Our solution, by design, limits this cost only to translators as ordinary telemetry-generating switches only have to generate reports using a much more lightweight DTA protocol as we demonstrate in the Evaluation section.

3.1 DTA reports and primitives

In Figure 3, we show the structure of a DTA packet. Basically, the telemetry payload to be exported by a switch is encapsulated into a standard UDP packet that carries our custom header.

DTA is designed to allow ease-of-integration with state-of-the-art telemetry monitoring systems [6, 17, 19, 46] and does not modify in any way the payload to be reported. Instead information needed by the translator to understand how the data has to be written in the collectors' memory is encoded in the *DTA header* and *primitive sub-header* fields. Indeed, each monitoring system has its own requirements in terms to how the telemetry information has to be presented to the collectors. Because of this, we provide four different collection primitives that together enable support for a wide range of telemetry solutions: *Key-Write*, *Append*, *Sketch-Merge*, and *Key-Increment*. The DTA header specifies the primitive to use when reporting the information, while the sub-header adds additional information that are specific to the primitive. In Table 2, we show that the primitives are generic enough to support many of state-of-the-art monitoring systems and effectively allow DTA to act as transport protocol for telemetry collection. Additionally, our approach is easily extensible to allow other useful primitives to be added in the future. In the following we discuss in details our designed primitives, which are shown in Figure 4.

Key-Write. This primitive (see Figure 4a) is designed to allow for storage of key-value pairs. Key-value storage is especially useful in cases where dynamic identifiers are preferred, such as when we want to report information on a per-flow basis. Identifiers for this information would include some sort of flow identification such as the 5-tuple, and would therefore be difficult to efficiently map into fixed, pre-allocated, and unique identifiers. Storing per-flow data for later queries is just one scenario where the Key-Write primitive is useful; we present several examples in Table 2.

Our low-level realization of the Key-Write primitive is fundamentally based around DART [39], which has been extensively evaluated in terms of data robustness, reliability, and memory efficiency. This algorithm is built to deliver probabilistic key-value storage of telemetry data, and is designed for efficient data plane deployments. It achieves this by constructing a central key-value store that acts as a shared hash

Primitive	Example monitoring	Description
Key-Write	INT (Path Tracing) [17, 36]	INT sinks reporting 5x32b switch IDs using flow 5-tuple keys
	Marple (Host counters) [46]	Reporting 32-bit counters using source IP keys, through non-merging aggregation
	Sonata (Per-query results) [19]	Reporting network query results using queryID keys
	PINT (Per-flow queries) [6]	PINT leverages memory redundancies for improved data compression through $n = f(pktID)$
	PacketScope (Flow troubleshooting) [57]	Report per-flow per-switch pipeline traversal information using <switchID,flow 5-tuple> as key
Append	INT (Congestion events) [17, 36]	INT sinks report a period of network congestion to list of network congestion events
	Marple (Lossy connections) [46]	Report flows with packet loss rate greater than threshold
	NetSeer (Event aggregation) [68]	Sorting per-switch events into network-wide lists according to category
	Sonata (Raw data transfer) [19]	Appending selected raw data from switches to lists at streaming processors
	PacketScope (Pipeline-loss insight) [57]	On packet drop: send pipeline-traversal information to central list of pipeline-loss events
Sketch-Merge	C [8] and CM [12] sketches	Counter-wise `sum` of the sketches of all switches
	HyperLogLog [9, 15]	Register-wise `max` of the sketches of all switches
	AROMA [5]	Select network-wide uniform packet- and flow samples from switch-level samples
Key-Increment	TurboFlow (Per-flow counters) [53]	Sending evicted microflow entries for central aggregation using flowKeys as keys
	Marple (Host counters) [46]	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 DTAJL: make keys and payload sizes explicit

table for all telemetry-generating network switches. Indexing of per-key data in this hash table is performed statelessly without collaboration through global hash functions, which unfortunately allows different keys to hash to the same location. The algorithm therefore inserts telemetry data as N identical entries at N different memory locations to achieve partial collision-tolerance through built-in data redundancy. A checksum of the telemetry key is stored alongside each data entry, which allows queries to be verified by validating the checksum.

DTA allows switches to specify the *importance* of per-key telemetry data by including the level of redundancy, or number of copies to store, as a field in the Key-Write header. Higher redundancy means a likely longer lifetime before being overwritten, as we discuss in Section 5.4.3. As the level of redundancy used at report-time may not be known while querying, the collector can assume by default a maximum redundancy level (we use $N = 4$). If the data was reported using fewer slots, unused slots will appear as overwritten entries (collision).

We significantly reduced the network overheads of DART by moving the indexing and redundancy generation into the DTA translator. This design choice effectively reduces the network telemetry traffic by a factor of the level of redundancy, and further reduces the telemetry report costs in the individual switches by replacing costly RDMA generation with the much more lightweight DTA protocol.

Append. Some telemetry scenarios cannot be easily managed with just a key-value store. A classic example is when a switch is requested to export events. In this case, a report could include an event identifier and an associated timestamp. A key-value store is inappropriate for this scenario; rather, a list or queue is a better abstraction.

We therefore provide a collection solution that allows for reporters to append information into global lists containing a pre-defined telemetry category in each list (see Figure 4c). Network operators can 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 declaring what data they want to append to which list, and forward it to the appropriate collector. The translator will then intercept the packet and generate an RDMA call to insert the data in the correct slot in the pre-allocated list. The translator utilizes a pointer to keep track of the current write location for each list, allowing it to insert incoming per-list Append reports sequentially and contiguously into memory. This leads to a very efficient use of memory and strong query performance. Translation also allows us to *significantly* improve on the collection speeds by batching multiple reports together.

Sketch-Merge. Sketches are a popular mechanism to summarize the traffic going through a switch using a small amount of memory and with provable guarantees. Common problems that are solved with sketches include flow-size estimation (e.g., C [8] and CM [12] sketches), estimating the number of flows (e.g., HyperLogLog [15]), and finding superspreaders (e.g., [5]).

An essential property of many of these sketches is *mergability*. Broadly speaking, one can take the sketches of individual switches and merge them to obtain a network-wide

view. Merging procedures differ between sketches; for example, the Count and Count-Min sketches require counter-wise summation while Hyperloglog needs to do register-wise `max`. Existing solutions commonly report the sketches to the controller at the end of each epoch (e.g., [41]), which merges them together. However, this is a costly operation that our transport protocol can offload into specific RDMA calls (see Figure 4d).

While some sketches are directly mergable using existing RDMA primitives (e.g., Fetch & Add), we argue that it is better to do the aggregation at the translator for several reasons:

- It can support merging procedures that RDMA cannot such as `max`.
- It can RDMA the aggregated result using a small number of writes, thus reducing the NIC and memory overheads at the collector. MM: This last point doesn't make sense to me, what does this mean, what are we reducing?
- It is the first point to observe all data, thus translator-aggregation would reduce the network overheads.

Accordingly, we designed a translator-aggregation solution that is efficient and robust to packet loss. First, we need the switches to send their sketch; this problem is not specific to our solution, and we can use a similar approach to LightGuardian [66] where we send one or more columns in each packet. Unlike LightGuardian, every switch sends the columns *sequentially*, and the translator keeps track of the last column received from each switch. If columns arrive out-of-order, the translator will notify the originating switch with a NACK packet and the trasmission will start again from the last received in-order column. Next, the translator tracks the number of sketches it has aggregated *per column*. Once a column has received counters from all switches, it is ready to be written to collector memory. For efficiency, our solution batches several columns together into a single RDMA write, thereby minimizing the number of communicated packets.

Key-Increment. Our Key-Increment is similar to the Key-Write primitive, but allows for addition-based data aggregation (see Figure 4b). That is, the Key-Increment primitive does not instruct the collector to set a key to a specific value, but it instead *increments* the value of a key. For example, switches might only store a few counters in a local cache, and evict old counters from the cache periodically when new counters take their place [46, 53]. The Key-Increment primitive can then deliver collection of these evicted counters at RDMA rates. As with Key-Writes, the introduction of a translator reduced network overheads compared with a more naive design,

Our Key-Increment memory acts as a Count-Min Sketch [12]. A Key-Write increments N value locations using the RDMA Fetch-and-Add primitive. On a query, Key-Increment returns the minimum value from these N locations. Data collisions may lead to an overestimate of the counter value, but the theoretical guarantees would match those for Count-Min Sketches [12]. Note the memory for the counters may have to be cleared periodically, depending on the application.

3.2 DTA Translator

The translator is the last-hop switch in charge of converting DTA traffic into standard RDMA calls for storing data directly in collectors' memory. DTA traffic is marked by the reporters with a level of importance: *low*, *medium*, and *high*. This impact the behavior of the translator as discussed below.

Telemetry Flow Control & Prioritization. The translator tracks its RDMA packet generation rate so to ensure that it never congest the collector's NIC. This is important as congestion lead to packet loss, RDMA queue pair desynchronization and consequent telemetry insertion throughput drop. In a non-congested scenario, the translator generates appropriate RDMA calls upon the reception of DTA traffic. In case of congestion, it either drops low-priority data or re-route critical reports to its local CPU for temporary storage[1]. Furthermore, it informs reporters about the congestion so to prevent the reception of non-critical telemetry data.

Reporters are configured to store a small number of high-priority reports in their local memory, so to allow their retransmission in case of loss. Indeed, high-priority reports include an incrementing packet sequence number that is compared with an in-translator tracker to detect transit-loss. A detected loss will trigger NACK-generation at the translator, which informs the reporter that the data should be re-sent[2].

Supporting Multiple Collectors. Large-scale telemetry environments cannot rely on a single server for processing telemetry reports, regardless of the collection technology. DTA is therefore designed to easily scale horizontally by deploying additional collectors, and relies on reporter-based load balancing of telemetry data among collectors. However, we need to ensure that the load balancing is stateless and can be centrally recalculated, to ensure scalability and efficiency in finding the storage locations for queries.

The destination IP addresss of DTA reports are decided on a per-primitive basis. The Key-Write and Key-Increment primitives are designed as distributed key-value stores, where a hash of the telemetry key is used by reporters as the basis for deciding the destination collector that will store this information. This design choice allows for horizontal scaling of

[1]The switch has a limited data bandwidth towards the local CPU, which limits the amount of telemetry data that can be collected during periods of collector congestion

[2]This design assumes that essential telemetry reports are not reordered in the network, which allowed for a resource-efficient design.

collection capacity by deploying more servers, and updating the collector-mapping lookup tables hosted in each reporter. Append traffic selects a collector based on the chosen list ID, as decided by pre-loaded lookup tables. This ensures that all per-category telemetry data is efficiently aggregated in a single location, and telemetry scaling can be achieved by deploying additional lists to host data for the Append primitive. All columns for sketch-merge will me aggregated together, and therefore all go to the same collector. JL: We COULD introduce an additional step. E.g., tree-like aggregation. Layer 3: all switcher. Layer 2: some aggregators that each merge a subset of network sketches. Layer 1: Translator, merge the pre-merged aggregator sketches. Would help scaling.

4 IMPLEMENTATION

Our codebase includes approximately $3.2K$ lines of code divided between the logic for the DTA reporter (§ 4.1), the translator (§ 4.2) and collector RDMA service (§ 4.3).JL: Update number The current implementation has full support for the Key-Write and Append primitives, while Sketch-Merge and Key-Increment are proposed future additions[3].

4.1 DTA Reporters

The reporter pipeline is written in approximately 600 lines of P4_16 for the Tofino ASIC. Controller functionality is written in about 100 lines of Python, and is responsible for populating forwarding tables and to insert collector IP addresses for the different DTA primitives.

DTA report packets are generated entirely in the data plane and the logic is in charge of encapsulating the telemetry report into a UDP packet followed by the two DTA specific headers where the primitive and its configuration parameters are included.

Report retransmission can be achieved in two ways: either by storing small reports in SRAM which allows for a pure data plane retransmission design, or temporarily storing critical telemetry data in the switch CPU if retransmission is required for large telemetry payloads that can not reasonable fit in switch SRAM.

4.2 DTA Translators

JL: we now support NACK generation and detection of lost reports The translator pipeline is written in $1.6K$ lines of P4_16 for the Tofino ASIC. The pipeline is designed to efficiently re-use logic between system primitives. There are five main pipeline-traversing paths for different types of network traffic, as shown in Figure 5.

A translator controller is written in 800 lines of Python. It is in charge of crafting RDMA Communication Manager (RDMA_CM) packets, which are then injected into the ASIC. These will initiate RDMA connections with the collector, and RDMA response packets are parsed by the controller to extract essential RDMA metadata. This metadata is then used to populate various lookup tables and registers in the translator pipeline. The controller also creates several multicast-rules, which are used by Key-Write and Key-Increment packets to trigger multiple simultaneous RDMA operations from a single ingressed DTA operation.

Key-Write and **Key-Increment** both follow the same fundamental logic, with the main difference being the RDMA operation that they trigger, as shown in Figures 4a and 4b. Key-Write triggers RDMA Write operations, while a Key-Increment implementation would trigger RDMA Fetch-and-Add. Both of these trigger N packet injections into the egress pipeline, using the multicast technique. The Tofino-native CRC engine is used to calculate the N memory locations, and is also used to calculate the concatenated checksum for Key-Write.

Append has its logic split between ingress and egress, as shown in Figure 4c, where ingress is responsible for building batches, and egress tracks per-list memory pointers. Batching of size B is achieved by storing $B-1$ incoming list entries into SRAM using per-list registers. Every Bth packet in a list will read all stored items, and bring these to the egress pipeline where they are sent as a single RDMA Write packet. Lists are implemented as ring-buffers, and the translator keeps a per-list head pointer to track where in server memory the next batch should be written.

Sketch-Merge primitive implementation uses ingress to verify that per-switch columns are reported in-order. This is done by storing the last in-order column index received by each switch. An incoming Sketch-Merge operation with a non-sequential column index will not be merged into the sketch, but is instead used to craft the NACK that is sent back to the source switch. Valid operations continue to egress, where the column counters are merged with the in-translator sketch. Sketches are written to server memory through RDMA Write as batches of C columns, which will be written contiguously in memory. Merging every Cth column will trigger RDMA-creation, which recirculates the packet $C-1$ times to allow access to all columns that will be included in the batch.

The **RDMA** logic is shared by all primitives, and includes controller-populated lookup tables containing RDMA metadata, SRAM storage of the queue pair packet sequence numbers (PSNs), and RoCEv2-header crafting. The translator parses RDMA NACKs, which are used to resynchronize PSNs in case of NIC-side RDMA congestion or failures, including updating RDMA rate limiting to force a pause on telemetry collection to allow clearing internal NIC buffers.

Finally, **flow control** is achieved by a combination of meters and per-reporter packet sequence trackers. Tofino-native

[3] All DTA primitives are designed in-depth to ensure ease-of-implementation in the Tofino architecture

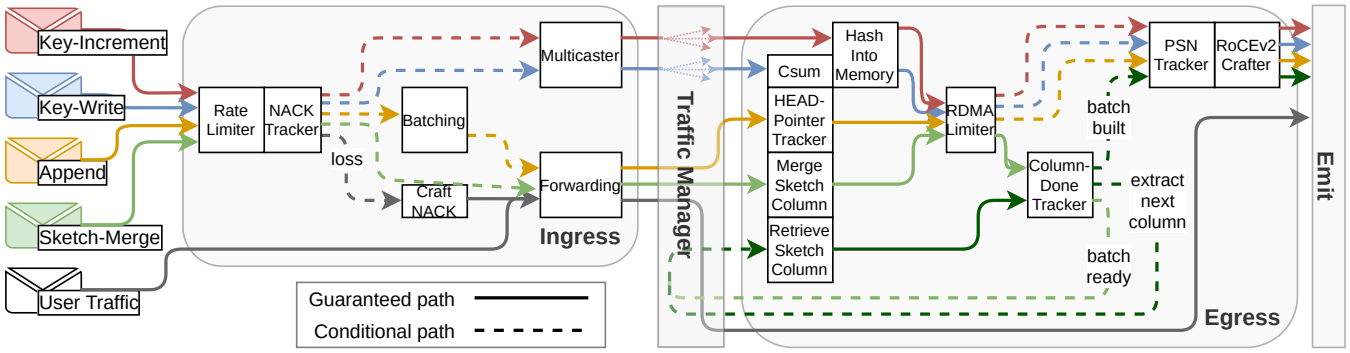


Figure 5: A translator pipeline with support for Key-Write, Key-Increment, Append, and Sketch-Merge. Five different paths exist for pipeline traversal, used to process different types of network traffic in parallel while efficiently sharing pipeline logic.

meters guage the RDMA generation rate of the translator, and conditionally drop or reroute reports to switch CPU depending on in-header priorities. The CPU can simply re-inject these packets into the pipeline when the RDMA generation rate falls below a threshold. Lost reports are detected through per-reporter registers, detection of which will abort report processing and instead generate a DTA NACK which is bounced back to the reporter.

4.3 Collector RDMA Service

The DTA collector service is written in 1K lines of C++ using standard Infiniband RDMA libraries, and includes support for hosting per-primitive memory structures and querying the reported telemetry data. The collector can host several primitives in parallel using unique RDMA_CM ports, and advertises primitive-specific metadata to the translator using RDMA-Send packets.

5 EVALUATION

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.

5.1 Experimental Setup

All benchmarks in this section are performed using *a single port* of the Mellanox Bluefield-2 2x100G DPU [49] without ARM processing as acting RDMA-capable network card, and using a BF2556X-1T [47] Tofino 1 [31] switch as both

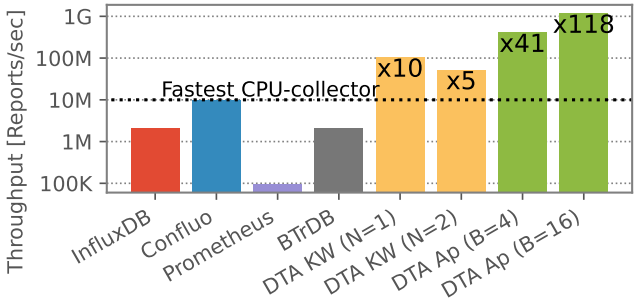


Figure 6: A performance comparison of DTA-based collection with current state of the art solutions. Key-Write with different levels of redundancy N, Append with different batch sizes B. CPU-based collectors dedicate 16 cores for data ingestion, while DTA uses 0.JL: confluo is placeholder

reporter and translator. JL: Does single-port equal halved performance? (probably, needs source) The TRex traffic generator [11] is used to inject DTA traffic into the translator switch, which further translates these into RDMA packets that are sent to the collector NIC. The collector server, running Ubuntu 20.04 and kernel 5.4.0, is hosting 2x Intel Xeon Silver 4114 CPUs, and 2x32GB DDR4 RAM clocked to 2.6GHz. Server BIOS has been optimized for high-throughput RDMA[4], and all RDMA-registered memory is allocated on 1 GB huge pages.

5.2 CPU-collector vs DTA Performance

We have benchmarked several CPU-based collectors in our system testbed, where we report 4B telemetry payloads using

[4]https://community.mellanox.com/s/article/performance-tuning-for-mellanox-adapters

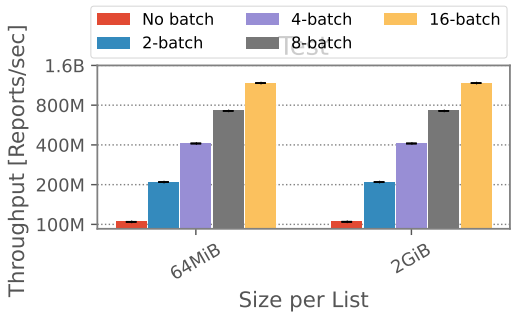
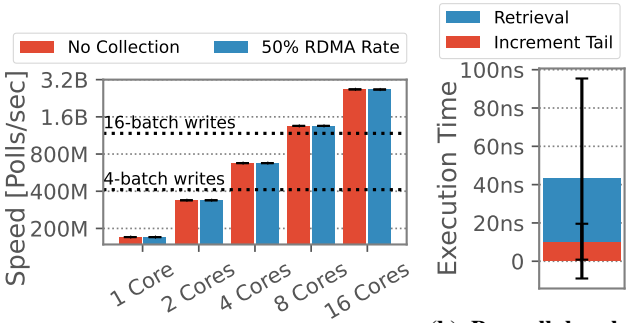


Figure 7: Packet loss collection rates, using DTA Append, at different batch sizes. The translator is inserting packet loss reports into a single list. Increased batch sizes yield a linear performance increase until we achieve line-rate with batches of 4 or greater. Note how the collection speed is not impacted by the list sizes. JL: Compare with INT-Collector/Confluo as baseline



(a) Polling rate **(b) Per-poll breakdown**

Figure 8: Packet loss processing rates. Append-lists are polled either while collecting no reports or at 50% capacity (i.e. while collecting *600 million* loss reports per second). Note how simultaneous collection has a negligible impact on the data retrieval rate, and how the processing rate scales near-linearly with the number of cores. The dotted lines show the maximum collection rates at different batch sizes. JL: Do this with different processing functions? e.g., Max, moving average, thresholding+counting, etc?

collector-specific data headers (e.g., INTCollector [61] collected data with INT v0.5 headers [17]). The collectors are all configured to allow offline flow 5-tuple queries against the stored data to ensure equivalent collector functionality (Confluo uses 4 filters and 4 indexes for very simple real-time reactivity)[5]. All system performances are either while dedicating 16 CPU cores in the same NUMA-node for collection, or extrapolating from single-core performance if the evaluated system did not include multi-core support. Confluo [35] proved to be the fastest CPU-based collector, which it achieved by writing data into a highly efficient Atomic MultiLog. However, we see that the performance of Confluo significantly degrades when we introduce more complex data organization through data indexing (which allows for queries against header attributes, similarly to the DTA Key-Write and Key-Increment primitives). Our collection primitives outperformed all CPU-based collectors by JL: XX.

JL: INTCollector is performing XDP-enhanced key-value collection of per-hop latencies, using InfluxDB as key-value backend. Unique flows, so event detection is useless

5.3 Append Primitive Performance

We have benchmarked the performance of the Append primitive in terms of maximum data collection rate, both at different batch sizes and size of the data list. The results are presented in Figure 7. We allocate a single data list which collects $4B$ data entries containing a shortened switch ID ($2B$) together with a queue ID ($1B$) and a queue occupancy approximator ($1B$). Our traffic generator fails to generate traffic at the incredible collection rates of this primitive ($> 1Gpps$), and we therefore had every generated report trigger several append operations at once. This test therefore does not evaluate the switch ingress rates[6], but results in a valid DTA-translation and NIC-side stress test.

The size of the underlying data lists are not expected to have an impact on the collection performance, which is what we see in our results. We also see that increased batch sizes has a linear increase in collection rate until we reach the port line-rate of $100Gbps$ at batch sizes $B \geq 4$. This suggests that the system bottleneck is the RDMA message rate of the collector NIC, given that we saw a negligible impact on collection rates when the collector CPU was simultaneously polling the underlying data list.

5.3.1 Append List-Polling Rate. The Append primitive assumes that appended data will be somehow further processed by the collector CPU, for example to trigger traffic engineering in case of network congestion. It is therefore important that the collected data can be retrieved and processed in an efficient way by the collector CPU. In Figure 8a, we show the raw list polling rates, which is the speed at which

[5]Confluo has the ability to run online queries against data in real-time through Filters, at a big performance cost. DTA can achieve the same functionality without a performance penalty by leveraging match-action tables in the translator to conditionally trigger the Append primitive.

[6]The Tofino ASIC is per-design guaranteed to handle line-rate processing in cases where incoming packets do a single pipeline traversal each, and our choice to not benchmark the translator ingress-rates does therefore not ignore a potential system bottleneck under the assumption that the switch ingress line-rates are not all surpassed.

reported information can be read into the CPU for continued processing. We assume that collection is running simultaneously to the CPU reading data from the lists, by having the translator process 600 million Append operations per second in batches of size 16, which equals collection at 50% maximum RDMA capacity. Simultaneously collecting and processing telemetry data showed no measureable impact on either collection or processing.

Extracting telemetry data from the lists is a very straightforward process, as shown in Figure 8b. First, we increment the tail pointer, and conditionally reset the pointer back to the start of the buffer when we have reached the end. Second we read the data where the pointer is pointing, and replace reset the value to 0. JL: redo Data that is valued 0 is assumed to be empty, preventing the tail pointer from moving beyond where data is written.

We allocated a number of lists equal to the number of CPU cores used during the test to prevent race conditions where they use the same tail pointer[7]. Ours tests show that the CPU manages to extract every list entry even when we use very large batch sizes, given that we dedicate enough cores.

Our results show that the collector can even retrieve list entries faster than the RAM clock speeds, which is due to cache prefetching greatly improving the performance of sequential memory accesses.

5.4 Key-Write Primitive Performance

This section will evaluate the raw report-collection performance of the Key-Write primitive, query speed, and the effectiveness of design internals.

5.4.1 Key-Write Collection Rate. We have benchmarked the performance of the DTA Key-Write primitive, with a $4GiB$ memory structure using $4B$ concatenated checksums, in terms of collected reports per second, as presented in Figure 9. Telemetry traffic is generated with sequential keys and data, at steadily increasing packet rates. The presented speeds are when the system does not experience *any* packet loss in a several second interval, meaning that the system does not yet require active resynchronization. This test was repeated at different levels of redundancy (N), and we notice the expected linear relationship between the throughput and level of redundancy since each incoming report will generate N RDMA packets towards the collector. The collection rate was unaffected by increases in the telemetry data size, until the translator reached egress line-rate of $100Gbps$ at data sizes of $28B$. This, combined with a negligible performance impact

[7]DTA does not limit the number of polling cores per list, and several cores can poll a single list by for example assigning them a set of non-overlapping indexes in the list and using per-core tail pointers.

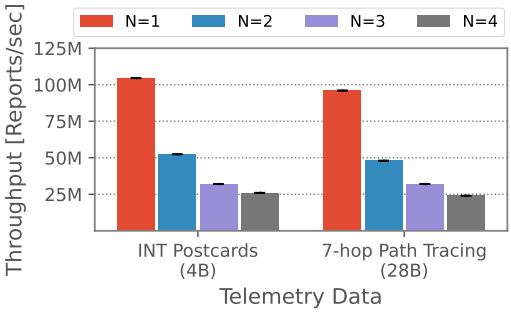
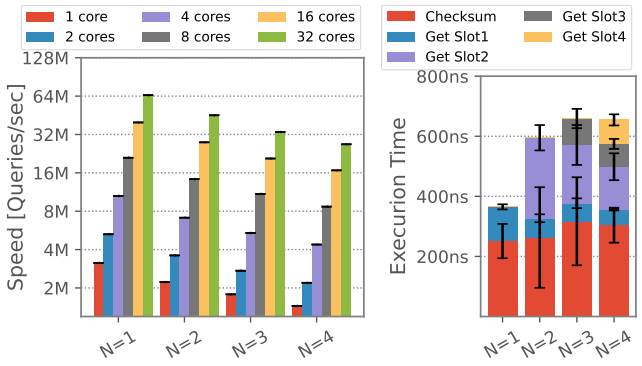


Figure 9: Per-flow path tracing collection, using the DTA Key-Write primitive, either as per-hop postcards (4B) or full 7-hop paths (28B). Note how the telemetry data size does *not* degrade DTA collection performance until we reach $100G$ line-rate at 32B payloads, which in practice increases path tracing collection rates by a factor 7 in the above test.



(a) Query rate **(b) Per-query breakdown**

Figure 10: Querying performance of the DTA Key-Write primitive, overlaying the maximum speed at which Key-Write can collect reports.JL: Run more queries to shrink stddev

from intense background memory operations at the collector, suggests that the RDMA message rate of the NIC is the current bottleneck in the design.

We conclude that a reduction in N would be preferred for optimal performance, and reiterate the suggested redundancy level $N = 2$ from Section 5.4.3 as a good compromise.

5.4.2 Key-Write Query Speed. The DTA Key-Write primitive requires calculation of several hashes for querying reported data. The number of hash functions to calculate will be $N + 1$, one for each of the redundancy slots to locate them in memory, and an additional one to calculate the concatenated checksum that is used to validate the query answer. A query validation algorithm without consensus can however

immediately answer a query when the first slot with a valid checksum was found, since a valid checksum would be the only criteria for validation. In this case, we would expect a reduction in the average query processing speeds since the processor would often not require iterating over all N redundancy slots to return back a query answer. However, here we evaluate the *worst case* performance, when the collector has to process every redundancy slot before a query answer is compiled.

We instructed the collector to query a set of sequential telemetry keys ranging from 0 to $100M$ in-order, with a key-value data structure of size $4GiB$ containing $4B$ INT postcards with $4B$ concatenated checksums for query validation. Figure 10a shows the speed at which the collector can answer incoming telemetry queries through the Key-Write primitive. The overall query success probabilities correspond to the previously presented results from Figure 11. We see the expected performance impact by the level of redundancy. Key-Write query processing can be easily parallelized, and we see a near-linear performance improvement when we allocate more cores for query processing.

Figure 10b shows a breakdown of the execution time for querying information reported through the Key-Write primitive. We can here see that most of the query execution time is spent either calculating the per-key concatenated checksum, or to calculate the memory address for each redundancy entry. Both of these are highly impacted by the speed of the in-CPU CRC implementation[8], and more optimized implementations should see a performance increase here.

The results show that even this sub-optimal query implementation, using generic CRC libraries, deliver impressive query response rates. Key-Write is not expected to receive queries regarding every single key that is collected, we do however deliver processing speeds that nearly allow for this. We show how just a single DTA collector has the ability to process tens of millions of key-based queries every second, for example to validate network flow paths[17, 36] or troubleshoot invalid pipeline traversals [57] across the network.

5.4.3 Redundancy Effectiveness. The probabilistic nature of the DTA Key-Write primitive cannot guarantee final queryability on a given reported key. We show in Figure 11 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 11 indicate which number of addresses per key (N) delivered the highest key queryability in each interval.

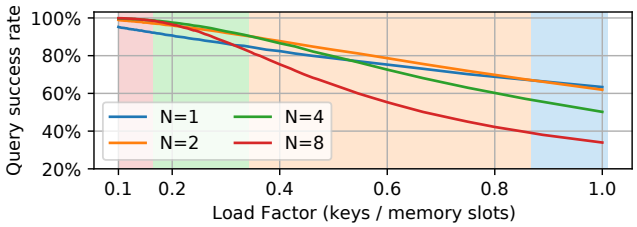


Figure 11: Average query success rates delivered by the Key-Write primitive, depending on the key-value store load factor and the number of addresses per key (N). The background color indicates optimal N in each interval.

Recall that RDMA lacks the ability to write into multiple memory addresses with a single packet, and that the Key-Write primitive 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 later in Figure 9. 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 $N = 1$.

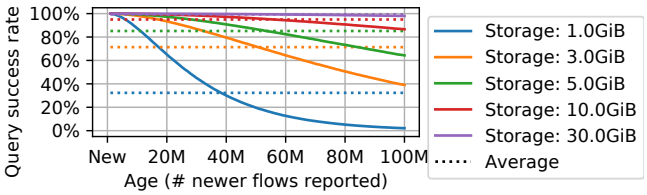


Figure 12: DTA Key-Write ages out stale data. This figure shows INT 5-hop path tracing queryability of 100 million flows at various storage sizes. The results are based on storing 5x4B switch IDs with 4B checksums, where each key writes into $N = 2$ different memory addresses.

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

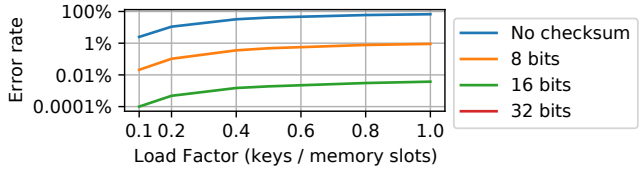


Figure 13: The probability of a query returning the wrong answer in error, due to address and checksum collisions.

5.4.5 Query Answer Correctness. JL: Section is heavily based on hotnets paper

[8] Our current collector service calculated CRC using the Boost C++ libraries: https://www.boost.org/

Resource	Base footprint	Batching	Retransmission
SRAM	5.5%	+3 %	+0.5%
VLIW Instruction	6.8%	+1.8 %	+0.5%
Match Crossbar	5.9%	+9 %	+0.2%
Table IDs	27.6%	+7.8 %	+1.0%
Hash Bits	9.0%	+7 %	-
Ternary Bus	20.3%	+7.8%	+1.1%
Meter ALU	10.4%	+31.3 %	+2.1%

Table 3: Resource costs of the implemented translator. Append batching creates batches of 16x4B data payloads, and retransmission supports tracking 65K reporter sequence numbers with 256 in-transit retransmitable reports each to trigger NACK creation.

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. Figure 13 shows results after extensive tests, where multiple simulations[9] 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 **??** for a deeper discussion on Key-Write correctness.

5.5 Translator Resource Footprint

DTA translators are responsible for several tailored processes, some of which have a high footprint on Tofino resources. 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 current translator pipeline, 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 due to the high amount of memory logic. 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.

We demonstrate here that DTA translators are already feasible in first-generation Tofino switches, even while including memory-intensive batch-building functionality. It is likely possible to design even more complex DTA primitives and aggregation functions than is presented here, given that the current-generation Tofino-2 has the capacity for significantly more memory logic than its predecessor.

JL: Add numbers on retransmission and NACK counters

6 LIMITATIONS

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

6.1 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 of zero-cpu

6.2 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]Our Key-Write simulator is verified to be algorithmically identical to our hardware implementation. Simulations were only used for benchmarking the probabilistic aspects of the Key-Write primitive, and performance benchmarks are exclusively performed in hardware

6.3 Efficiency of Append-Batching with Larger Data Slots

JL: Due to register busses max 32b output

INTEL REVIEWERS: we want to discuss how the register 32-bit bus width limits the data sizes that we can batch. Working around this limitation (e.g., splitting data across several register) is costly in terms of SALU calls. Conclusion being that we likely reduce the batch size (e.g., 16B slots can do batches of 4, instead of 16 as we can do with 4B data). This seems reasonable

7 DISCUSSION

7.1 ToR vs SmartNIC Translation

There are two main approaches we have considered in how to deploy the translator framework. One could leverage programmable network cards to build in DTA support to the actual network card that is installed at the collector. The NIC could then process incoming DTA operations, and translate these not into RDMA, but directly into *DMA* operations that are immediately sent through the PCI bus to memory. This approach would allow for faster memory operations compared with a switch-based approach, and might make stateful solutions that leverage server DRAM more feasible. For example, one might be able to include collision-mitigating techniques such as Cuckoo Hashing for the Key-Write primitive, or the addition of more powerful aggregation functions alongside Key-Increment.

However, highly programmable network cards often suffer from reduced performance compared with fixed-function network cards [16], and our evaluations show that the ToR-as-translator approach already results in the NIC message rate being the collection bottleneck (§5.4.1 and §5.3), even while using top-of-the-line fixed-function RDMA NICs. Not even upcoming P4-programmable ASIC-based NICs deliver packet processing rates close to what current-generation fixed-function RDMA-capable NICs deliver [54]. The exception is FPGA-based network cards, which deliver near fixed-function ASIC performance, and are already deployed in hyperscale data centers to a limited extent [14]. FPGA-based NICs are however known to be power hungry, expensive, and difficult to program [16] which limits their potential.

We acknowledge that using programmable NICs could result in smaller infrastructure changes than programmable ToRs, since modifications would only have to take place at the server-level. The DTA design presented in this paper is designed around highly pipelined architectures, and we therefore expect that the design choices of our system would translate well into most programmable packet processing architectures that might exist in future network cards.

7.2 Non-Telemetry Usecases

JL: Database acceleration

7.3 Read-based Writes

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

7.4 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.***

8 RELATED WORKS

Telemetry and Collection. Traditional techniques for monitoring the status of the network have looked into periodically collecting telemetry data [18, 20] or mirroring packets at switches [51, 70]. The former generate coarse-grained data that can be significant given the large scale of todays' networks [58]. The latter has been recognised as viable option only if it is known in advance the specific flow to monitor [70]. The rise in programmable switches has enabled fine-grained telemetry techniques that generate a lot more data [6, 17, 19, 55, 68, 70]. Irrespective of the techniques, collection is identified to be the main bottleneck in network-wide telemetry, and previous works focus on either optimizing the collector stack performance [35, 61], or reducing the load through offloaded pre-processing [40] and in-network filtering [27, 38, 62, 68]. To the best of our knowledge, only one proposal has investigated the possibility to entirely bypass collectors' CPU but limits the collection process to only data that can be represented as a key-value store [39], thus not suitable for telemetry systems based on events or sketches. In this paper, we propose an alternative solution which is generic and work with a number of existing state-of-the-art monitoring systems. An alternative approach is letting end-hosts assist in network-wide telemetry [23, 55], which unfortunately requires significant investments and infrastructure changes.

RDMA in programmable networks. Recent works have shown that programmable switches can perform RDMA calls [37], and that programmable network cards are capable of expanding upon RDMA with new and customized primitives [3]. Especially FPGA network cards show great promise for high-speed custom RDMA verbs [43, 52]. As discussed in Section 2, telemetry collection brings new challenges when used in conjunction with the RoCEv2 protocol. Challenges that have not be tackled by existing solution, designed to leverage RDMA to augment a switch storage capability [37].

9 CONCLUSION

In this paper, we presented DTA, an RDMA-based approach to telemetry collection. DTA leverages the collector's ToR as a translator that both aggregates network-wide telemetry and is responsible for writing the data in a queryable form directly into memory. The benefit of our work is twofold, where we save CPU on both storing the data and its aggregation and is readily deployable with commodity RDMA NICs and programmable switches. We envision that similar principles can also aid many non-telemetry applications such as database acceleration [59] or ???.

REFERENCES

[1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 503–514.

[2] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*.

[3] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Remote Memory Calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 38–44.

[4] Arista. [n. d.]. Telemetry and Analytics. https://www.arista.com/en/solutions/telemetry-analytics. ([n. d.]). Accessed: 2021-06-24.

[5] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. 2020. Routing Oblivious Measurement Analytics. In *2020 IFIP Networking Conference (Networking)*. IEEE, 449–457.

[6] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. 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*. 662–680.

[7] BROADCOM. [n. d.]. Trident Programmable Switch. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series. ([n. d.]).

[8] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2004. Finding frequent items in data streams. *Theoretical Computer Science* 312, 1 (2004), 3–15.

[9] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. 2020. Beaucoup: Answering many network traffic queries, one memory update at a time. 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*. 226–239.

[10] Cisco. [n. d.]. Explore Model-Driven Telemetry. https://blogs.cisco.com/developer/model-driven-telemetry-sandbox. ([n. d.]). Accessed: 2021-06-24.

[11] Cisco. [n. d.]. TRex. https://trex-tgn.cisco.com/. ([n. d.]). Accessed: 2022-01-25.

[12] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 401–414.

[14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 51–66.

[15] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 137–156.

[16] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. 2020. SmartNIC performance isolation with FairNIC: Programmable networking for the cloud. 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*. 681–693.

[17] The P4.org Applications Working Group. [n. d.]. Telemetry Report Format Specification. https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report_latest.pdf. ([n. d.]). Accessed: 2021-06-23.

[18] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. 2015. 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*. 139–152.

[19] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 357–371.

[20] Chris Hare. 2011. Simple Network Management Protocol (SNMP). (2011).

[21] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. 2010. Elastictree: Saving energy in data center networks.. In *Nsdi*, Vol. 10. 249–264.

[22] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. 2017. Tagger: Practical PFC Deadlock Prevention in Data Center Networks. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*. Association for Computing Machinery, 451–463.

[23] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. 2020. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. 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*. 404–421.

[24] Huawei. [n. d.]. Overview of Telemetry. https://support.huawei.com/enterprise/en/doc/EDOC1000173015/165fa2c8/overview-of-telemetry. ([n. d.]). Accessed: 2021-06-24.

[25] IEEE 802.11Qbb. 2011. Priority Based Flow Control.

[26] Infiniband Trade Association. 2015. InfiniBandTM Architecture Specification. (2015). Volume 1 Release 1.3.

[27] Intel. [n. d.]. In-band Network Telemetry Detects Network Performance Issues. https://builders.intel.com/docs/networkbuilders/in-band-network-telemetry-detects-network-performance-issues.pdf. ([n. d.]). Accessed: 2021-06-04.

[28] Intel. [n. d.]. Intel Deep Insight Network Analytics Software. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/network-analytics/deep-insight.html.

([n. d.]). Accessed: 2021-06-10.

[29] Intel. [n. d.]. Intel Tofino Series Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html. ([n. d.]). Accessed: 2021-05-12.

[30] Intel. [n. d.]. Intel® Ethernet Network Adapter E810-CQDA1/CQDA2. https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/network-adapters/ethernet-800-series-network-adapters/e810-cqda1-cqda2-100gbe-brief.html. ([n. d.]). Accessed: 2021-06-11.

[31] Intel. [n. d.]. Intel® Tofino™ Series Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. ([n. d.]). Accessed: 2022-01-25.

[32] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable Message Latency in the Cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*.

[33] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*.

[34] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance {RDMA} systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 437–450.

[35] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2019. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 421–436.

[36] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*.

[37] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. Tea: Enabling state-intensive network functions on programmable switches. 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*. 90–106.

[38] Jan Kučera, Diana Andreea Popescu, Han Wang, Andrew Moore, Jan Kořenek, and Gianni Antichi. 2020. Enabling event-triggered data plane monitoring. In *Proceedings of the Symposium on SDN Research*. 14–26.

[39] Jonatan Langlet, Ran Ben-Basat, Sivaramakrishnan Ramanathan, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. 2021. Zero-CPU Collection with Direct Telemetry Access. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. 108–115.

[40] Yiran Li, Kevin Gao, Xin Jin, and Wei Xu. 2020. Concerto: cooperative network-wide telemetry with controllable error rate. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 114–121.

[41] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. Flowradar: A better netflow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 311–324.

[42] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.

[43] Wassim Mansour, Nicolas Janvier, and Pablo Fajardo. 2019. FPGA implementation of RDMA-based data acquisition system over 100-Gb ethernet. *IEEE Transactions on Nuclear Science* 66, 7 (2019), 1138–1143.

[44] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-Based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. Association for Computing Machinery, 14.

[45] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 313–326.

[46] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 85–98.

[47] APS Networks. [n. d.]. Advanced Programmable Switch. https://www.aps-networks.com/wp-content/uploads/2021/07/210712_APS_BF2556X-1T_V04.pdf. ([n. d.]). Accessed: 2022-01-25.

[48] Juniper Networks. [n. d.]. Overview of the Junos Telemetry Interface. https://www.juniper.net/documentation/us/en/software/junos/interfaces-telemetry/topics/concept/junos-telemetry-interface-oveview.html. ([n. d.]). Accessed: 2021-06-24.

[49] NVIDIA. [n. d.]. NVIDIA BLUEFIELD-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf. ([n. d.]). Accessed: 2022-01-25.

[50] NVIDIA. [n. d.]. NVIDIA Mellanox Spectrum Switch. https://www.mellanox.com/files/doc-2020/pb-spectrum-switch.pdf. ([n. d.]).

[51] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-Scale Monitoring and Control for Commodity Networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. Association for Computing Machinery, 407–418.

[52] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[53] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*. 1–16.

[54] Pensando Systems. [n. d.]. Pensando DSC-100 Distributed Services Card. https://pensando.io/wp-content/uploads/2020/03/DSC-100-ProductBrief-v06.pdf. ([n. d.]). Accessed: 2022-01-23.

[55] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2018. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 453–456.

[56] Mellanox Technologies. [n. d.]. ConnectX®-6 VPI Card. https://www.mellanox.com/files/doc-2020/pb-connectx-6-vpi-card.pdf. ([n. d.]). Accessed: 2021-05-12.

[57] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. 2020. Packetscope: Monitoring the packet lifecycle inside a switch. In *Proceedings of the Symposium on SDN Research*. 76–82.

[58] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. 2018. Stroboscope: Declarative Network Monitoring on a Budget. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 467–482.

[59] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2407–2422.

[60] Nguyen Van Tu, Jonghwan Hyun, and James Won-Ki Hong. 2017. Towards onos-based sdn monitoring using in-band network telemetry. In *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 76–81.

[61] Nguyen Van Tu, Jonghwan Hyun, Ga Yeon Kim, Jae-Hyoung Yoo, and James Won-Ki Hong. 2018. Intcollector: A high-performance collector for in-band network telemetry. In *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE, 10–18.

[62] Jonathan Vestin, Andreas Kassler, Deval Bhamare, Karl-Johan Grinnemo, Jan-Olof Andersson, and Gergely Pongracz. 2019. Programmable event detection for in-band network telemetry. In *2019 IEEE 8th international conference on cloud networking (CloudNet)*. IEEE, 1–6.

[63] Xilinx. [n. d.]. Xilinx Embedded RDMA Enabled NIC. https://www.xilinx.com/support/documentation/ip_documentation/ernic/v3_0/pg332-ernic.pdf. ([n. d.]). Accessed: 2021-06-11.

[64] Minlan Yu. 2019. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review* 49, 1 (2019), 11–17.

[65] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference*. Association for Computing Machinery, 78–85.

[66] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. 2021. LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets.. In *NSDI*. 991–1010.

[67] Yu Zhou, Jun Bi, Tong Yang, Kai Gao, Jiamin Cao, Dai Zhang, Yangyang Wang, and Cheng Zhang. 2020. Hypersight: Towards scalable, high-coverage, and dynamic network monitoring queries. *IEEE Journal on Selected Areas in Communications* 38, 6 (2020), 1147–1160.

[68] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. 2020. 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*. 76–89.

[69] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. Association for Computing Machinery, 523–536.

[70] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 479–491.