Direct Telemetry Access

Paper # 224, 12 pages body, 15 pages total

ABSTRACT

The emergence of programmable switches allows operators to collect a vast amount of fine-grained telemetry data in real time. However, consolidating the telemetry reports at centralized collectors to gain a network-wide view poses an immense challenge. The received data has to be transported from the switches, parsed, manipulated, and inserted in queryable data structures. As the network scales, this requires excessive CPU processing. RDMA is a transport protocol that bypasses the CPU and allows extremely high data transfer rates. Yet, RDMA is not designed for telemetry collection: it requires a stateful connection, supports only a small number of concurrent writers, and has limited writing primitives, which restricts its data aggregation applicability.

We introduce Direct Telemetry Access (DTA), a solution that allows fast and efficient telemetry collection, aggregation, and indexing. Our system establishes RDMA connections only from collectors' ToR switches, called *translators*, that process DTA reports from all other switches. DTA features novel and expressive reporting primitives such as Key-Write, Append, Sketch-Merge, and Key-Increment that allow integration of telemetry systems such as INT and others. The translators then aggregate, batch, and write the reports to collectors' memory in queryable form. As a result, our solution can collect over 100M INT reports per second, improving over Confluo, the state-of-the-art CPU-based collector, by 17x.

1 INTRODUCTION

In modern data centers, telemetry is the foundation for many network management tasks such as traffic engineering, performance diagnosis, and attack detection [8, 25, 32, 40, 69, 75, 78, 79]. With the rise of programmable switches [9, 29, 55], telemetry systems can now monitor network traffic in real time and at a fine granularity [8, 19, 45, 52, 74, 81]: a key enabler to support automated network control [1, 23, 46] and detailed troubleshooting [20, 32, 65]. Telemetry systems also aggregate per-switch data to a centralized collector to provide network-wide view and to answer real-time queries for many management tasks [6, 12, 21, 26, 32, 39, 54].

However, fine-grained data increases the telemetry volume and brings significant challenges for both its transport and its processing. A switch can generate up to millions of telemetry data reports per second [52, 79] and a data center network comprises hundreds of thousands of switches [20]. The amount of data keeps growing with larger networks and higher line rates [61]. As a consequence, it is increasingly hard to scale data collection in telemetry systems [39, 70, 79].

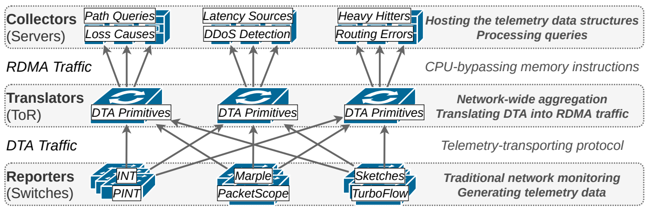


Figure 1: An overview of the DTA telemetry protocol.

Indeed, existing research boosts scalability by improving the collectors' network stacks [39, 70] by aggregating and filtering data at switches [30, 42, 52, 71, 79], or by switch cooperation [44]. However, as we show, these solutions still face the bottleneck of the amount of data processing required at the collector (parsing, wrangling, indexing, and storing incoming reports) which highly limits their performance (§ 2).

We propose *Direct Telemetry Access* (DTA): a telemetry collection protocol (illustrated in Figure 1) optimized for moving and aggregating reports from switches to collectors' memory. DTA completely relieves the collectors' CPU from any sort of processing triggered by incoming reports, so as to minimize their computational overheads. The underlying idea is to have switches generating RDMA (Remote Direct Memory Access) [28] calls to collectors. RDMA is available on many commodity network cards [31, 66, 73] and can perform hundreds of millions of memory writes per second [66], which is significantly faster than the most performant CPU-based telemetry collector [39].

Previous work [41] has implemented the generation of RDMA instructions between a switch and a server for network functions. However, there are several challenges to adopting RDMA between multiple switches and the collector for telemetry systems: (1) RDMA utilizes only basic memory operations, while telemetry systems need aggregated data structure layouts to support diverse queries. (2) To collect lots of reports we need an high-speed RDMA connection, which requires a lossless network. Adopting standard per-hop flow control mechanisms such as PFC would force applications' traffic to be subjected to these rules as well. (3) RDMA performance degrades substantially when multiple clients write to the same server [36].This conflicts with the needs of telemetry collection, where numerous switches send their data to each collector; (4) 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.

To address these challenges, we leverage the top-of-the-rack (ToR) switches in front of collectors, which we refer to as DTA *translators*. We use DTA translators to aggregate and batch reports, and establish an RDMA connection to a collector. All other switches send their telemetry reports to the translator encapsulated by our custom protocol header. The DTA translator then converts the reports into standard RDMA calls. Our approach solves the above challenges: (1) The translator's programmability enables powerful aggregation primitives that are not natively supported by RDMA; (2) The translator is directly connected to the collector, and we provide a reliability protocol allowing it to request retransmissions of lost reports; (3) Only the translator establishes an RDMA connection with the collector, thereby boosting its performance; (4) All switches besides the translators do not use RDMA, freeing resources to other tasks.

We design several switch-level RDMA language extension primitives such as *Key-Write*, *Append*, *Sketch-Merge*, and *Key-Increment* that are converted by the translator into standard RDMA calls. These primitives allow multiple switches to effectively write reports into collectors' memory without conflicts or race conditions. They also allow DTA to support state-of-the-art monitoring systems such as INT [19], PINT [8], Marple [52], and Sonata [21].

We discuss DTA's design (§ 3), and share the challenges in implementing it on programmable switches (§ 4). Our evaluation (§ 5) shows that we are able to write in our custom key-value store over 1 million INT reports per second, without any CPU processing, which is 17x better than Confluo [39], the state-of-the-art CPU-based collector for high-speed networks. Further, when the received data can be recorded sequentially as in the case of event-telemetry reports, we can store up to 1 billion reports per second, ≈194x faster than state-of-the-art.

Our main contributions are:

- We study the bottlenecks of state-of-the-art collectors and show that they are bottlenecked by the CPU ability to process reports and storing them in queryable data structures.
- We propose *Direct Telemetry Access*, a novel telemetry collection protocol generic enough to support major network monitoring systems proposed by the research community.
- We implement DTA using commodity RDMA NICs and programmable switches and will open source code upon publication to foster reproducibility.

2 MOTIVATION

As telemetry systems move to fine-grained, real-time analysis and support network-wide queries, data collection becomes the key bottleneck. In this section, we quantify this bottleneck in existing systems, propose the use of RDMA as the basic transport mechanism between switches and collectors, and discuss key challenges in adopting it.

System	Per-switch Report Rate
INT Postcards (Per-hop latency, 0.5% sampling)	19 Mpps
Marple [52] (TCP out-of-sequence)	6.72 Mpps
Marple [52] (Packet counters)	4.29 Mpps
NetSeer [79] (Flow events)	950 Kpps

Table 1: Per-reporter data generation rates by various monitoring systems, as presented in their individual papers[1]. Numbers are based on 6.4Tbps switches.

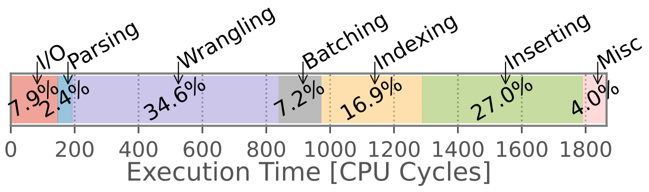


Figure 2: CPU-based collectors are inefficient. Shown above is the collection work breakdown of Confluo, the state-of-the-art collector for high-speed networks.

2.1 Collection overhead of telemetry systems

We investigated a few state-of-the-art telemetry systems and summarize the reporting rate generated by *a single switch* in Table 1, based on the numbers in the corresponding papers[1].

When using INT in postcard mode [30] on a commodity 6.4Tbps switch, and considering a standard load of ≈40% [76], it could generate up to $19M$ reports per second. Other solutions are less demanding on collectors, either because preprocessing and filtering of data at switches reduces the reporting rate, or because they focus on more specific tasks that limit the data to report. For example, NetSeer [79] discusses a system that exports just $950K$ records per second per switch.

To get a network-wide view, we may need to collect data from hundreds of thousands of switches in data center networks [20]. This means a network can easily generate billions of reports per second, even when using a very lightweight system such as NetSeer. For each report from a switch, collectors spend CPU cycles in receiving the data, known as I/O. Even though DPDK greatly reduces the cost of packet I/O, polling traffic at 148Mpps requires as much as 12 dedicated CPU cores [57]. As a consequence, *thousands of CPU cores* have to be dedicated to *just receive* billions of reports per second [5].

Once the data is received, it has to be processed in the user-space to store the information in a queryable data structure. The processing usually involves three key operations that require more CPU cores: *parsing* (extract content from the incoming report packets), *data wrangling* (process the data to make it suitable for insertion, e.g., hashing it into a fixed-size key), and *storing* (determine where to store it in memory).

I/O is costly, and this additional processing can be much more expensive than that. Figure 2 shows the breakdown of Confluo [39], the state-of-the-art collector for high-speed

[1]INT does not advertise a specific telemetry reporting rate, and we chose an arbitrary sampling rate of 0.5% to keep overheads low, as an example.

networks, when receiving and processing $100K$ reports on an Intel Xeon Silver 4114 CPU @ 2.20GHz. Data wrangling in Confluo is the process of organizing the reports based on user criteria (e.g., filtering on event type). Storing involves batching, computing a key (indexing), and insertion, where reports are scheduled to be written in memory. We found that most CPU cycles (86%) are spent in data wrangling and storing, almost 11x the cost of its I/O. As a consequence, optimizing the collector stacks is critical for minimizing the number of cores used for *data collection* [39].

2.2 Challenges for using RDMA

Our goal is to reduce the CPU utilization of collectors to scale with the large number of telemetry reports from many switches. RDMA is a natural solution that has been already used in the past to boost the performance of consensus protocols [38], reads in key-value stores [49, 72], and data replication [64]. Similarly, we could build a strawman solution where switches write their reports directly in collectors' memory with RDMA calls without any CPU involvement. Although this idea appears attractive, and generating RDMA instructions directly from switches is possible [41], it presents several challenges when applied for telemetry collection:

(1) RDMA verbs are limited. 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 lists or cuckoo hash tables can reduce data retrieval times significantly [39]. However, RDMA provides support for a limited set of operations: *Read*, *Write*, *Fetch-and-Add*, and *Compare-and-Swap*. This makes it hard to implement the data structures mentioned above, 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 and increased RDMA traffic. Enabling advanced RDMA offloads on commodity NICs is possible, but at the cost of a reduction of performance that can reach 2 orders of magnitude [59]. Furthermore, it not possible to support multiple senders writing at the same memory location while guaranteeing no conflicts or race-conditions. This is paramount for network telemetry, where multiple switches have to report their data to a collector.

(2) High-speed RDMA assumes a lossless network. An RDMA receiver expects incoming packets to be in sequential order. In case of a packet loss, this expectation fails, leading to queue-pair invalidation which significantly degrades RDMA performance [51]. For this reason, current deployments adopt per-hop flow-control mechanisms such as PFC [27] to rate-limit RDMA traffic in case of congestion [50, 51, 80]. Unfortunately, this practice can introduce deadlocks, a circular buffer dependency between switches that cause network throughput collapse [24]. Even worse, many switches reporting telemetry data at a collector increase the chances of incast, thus pushing PFC to kick in. As telemetry traffic would run on the data plane, it would force applications' traffic to be subjected to PFC rules as well.

(3) A limited number of RDMA connections. RDMA NICs can only handle a limited number of active connections (also known as *queue pairs*) at high speed. Increasing the number of queue pairs degrades RDMA performance by up to 5x [15]. This limits the total number of switches that can generate telemetry RDMA packets to the collector before performance starts degrading. Alternatively, several switches can share the same queue pair, with the assumption (imposed by RDMA) that every packet received at the collector has a strictly sequential ID. This becomes impractical to achieve in a distributed network of switches.

(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 maintaining state for connections that can be expensive to implement. For instance, the switch has to store metadata, and generate appropriate headers and associated checksums. However, current programmable switches are limited in memory, computational logic, and internal bus bandwidth.

3 DIRECT TELEMETRY ACCESS

In this paper, we present Direct Telemetry Access (DTA), an efficient telemetry collection protocol. DTA leverages *translators*, which are the ToR switches adjacent to the collectors. Translators receive telemetry data from *reporters* (i.e., switches exporting telemetry data), encapsulated in our custom protocol. They then aggregate and batch the reports and use standard RDMA calls to write them directly into queryable data structures in the collectors' memory. This way, translators reduce the amount of data written, act as a single RDMA writer to each collector, and eliminate the CPU overheads of ingesting and processing the data. Figure 1 shows an overview of DTA. In the following, we describe how we overcome the challenges discussed in the previous section.

Switch-level RDMA language extension. RDMA verbs cannot support the management of complex data structures while guaranteeing no conflicts or race conditions in the presence of multiple clients accessing the same memory region. Following the spirit of past works that explored ways to extend RDMA [3], the translator can be used as an enabler for custom RDMA operations available at reporters. The translator is in charge of receiving telemetry data, aggregating it, and performing standard RDMA calls to the associated collector.

In Section 3.1, we introduce new primitives that reporters can use when interacting with the translator. As the translator is responsible for managing the memory state at the collector, it resolves conflicts and avoids race conditions.

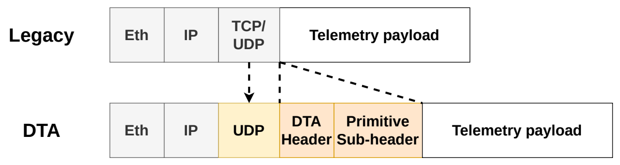


Figure 3: The structure of a DTA packet report, compared with legacy telemetry reports.

Data prioritization and telemetry flow control. RDMA requires a lossless network to provide high-performance data exchange. Here, the translator is the only network component that creates a point-to-point RDMA connection to the collector. As a consequence, we have to avoid packet loss only on that specific link, e.g., using PFC or by applying a rate-limiting scheme [2, 34, 35]. The problem is that, while the collector's capabilities bound the translator's ability to consume data, there is no flow-control between reporters and the translator (i.e., the reporters can still send as much data as they want). In Section 3.2, we introduce our telemetry flow-control mechanism to rate-limit data from reporters to the translator. We could have used PFC, but this would have impacted any packets sharing the path between the two. Instead, our ad-hoc solution regulates only telemetry traffic, and additionally shows how our solution can prioritize specific data to avoid the loss of critical information.

Reduced number of RDMA connections. Standard RDMA NICs suffer performance penalties when an increasing number of connections are created simultaneously [15, 36]. With the proposed architecture, because the translator acts as an aggregator for all the reports coming from the various switches, it is possible to greatly reduce the number of connections handled by the collector RDMA NIC and keep performance high.

Reduced in-network costs. Managing RDMA states and packet generation inside of a switch ASIC is resource-intensive. Our solution limits this cost just to translators as ordinary telemetry-generating switches only generate reports using a more lightweight DTA protocol, as shown in Section 5.5.

3.1 DTA reports and primitives

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

DTA is designed to allow ease-of-integration with state-of-the-art telemetry monitoring systems [8, 19, 21, 52] and does not modify the reported payloads. Instead, the *DTA header* and *primitive sub-header* instruct the translator what and where to write to the collectors' memory. This flexibility is essential as the various monitoring systems require writing telemetry in different ways. To address this need, we provide four different collection primitives that together support a

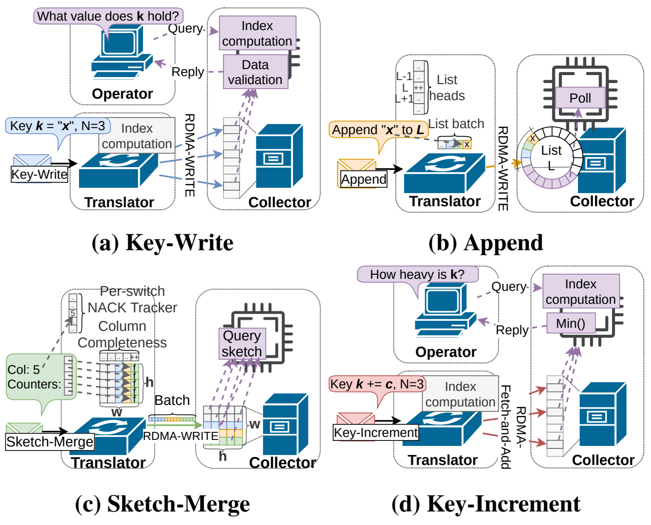


Figure 4: A high-level overview of the DTA primitives

wide range of telemetry solutions: *Key-Write*, *Append*, *Sketch-Merge*, and *Key-Increment*[2]. The DTA header specifies the primitive to use when reporting the information, while the sub-header adds additional information specific to the primitive.

In Table 2, we show that the primitives are generic to support many state-of-the-art telemetry systems. In the following, we discuss our designed primitives (Figure 4).

Key-Write. This primitive (see Figure 4a) is designed for key-value pair collection. Key-value indexing is challenging when the keys come from arbitrary domain (e.g., flow 5-tuples) and we want to map them to a small address space using just write operations. A natural solution would be to use hashing for deciding on the write address for a given key. However, data written to a single location could be overwritten by another key's write. Storing per-flow data for later queries is just one scenario where the Key-Write primitive is useful; we present additional examples in Table 2.

Key-Write provides a probabilistic key-value storage of telemetry data, and is designed for efficient data plane deployments. We achieve this by constructing a central key-value store as a shared hash table for all telemetry-generating network switches. Indexing per-key data in this hash table is performed statelessly without collaboration through global hash functions. This unfortunately allows different keys to hash to the same location. The algorithm, therefore, inserts telemetry data as N identical entries at N memory locations to achieve partial collision-tolerance through built-in data redundancy. In addition, a checksum of the telemetry key is stored alongside each data entry, which allows queries to be verified by validating the checksum.

[2]We design all primitives to work using P4 switch translators while our current hardware implementation only includes the first two.

Primitive	Example monitoring	Description
Key-Write	INT (Path Tracing) [19, 40]	INT sinks reporting *5x4B* switch IDs using *flow 5-tuple* keys
	Marple (Host counters) [52]	Reporting *4B* counters using *source IP* keys, through non-merging aggregation
	Sonata (Per-query results) [21]	Reporting *fixed-size* network query results using *queryID* keys
	PINT (Per-flow queries) [8]	*1B* reports with *5-tuple* keys, using redundancies for data compression through $n = f(pktID)$
	PacketScope (Flow troubleshooting) [67]	Report *fixed-size* per-flow per-switch pipeline traversal information using <*switchID,5-tuple*> as key
Append	INT (Congestion events) [19, 40]	INT sinks append *4B* reports to list of network congestion events
	Marple (Lossy connections) [52]	Report *13B* flows with packet loss rate greater than threshold
	NetSeer (Loss events) [79]	Appending *18B* loss event reports into network-wide list of packet losses
	Sonata (Raw data transfer) [21]	Appending *query-specific* packet tuples from switches to lists at streaming processors
	PacketScope (Pipeline-loss insight) [67]	On packet drop: send *14B* pipeline-traversal information to central list of pipeline-loss events
Sketch-Merge	C [10] and CM [14] sketches	Counter-wise `sum` of the sketches of all switches
	HyperLogLog [11, 16]	Register-wise `max` of the sketches of all switches
	AROMA [7]	Select network-wide uniform packet- and flow samples from switch-level samples
Key-Increment	TurboFlow (Per-flow counters) [62]	Sending *4B* counters from evicted microflow-records for central aggregation using *flow key* as keys
	Marple (Host counters) [52]	Reporting *4B* 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

We further reduce the network and hardware resource overheads of Key-Write 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 (§ 5.5). Isolating Key-Write logic inside collector-managing translators allows us to entirely remove this resource cost from all other switches in the network.

DTA let switches specify the *importance* of per-key telemetry data by including the level of redundancy, or the number of copies to store, as a field in the Key-Write header. Higher redundancy means a longer lifetime before being overwritten, as we discuss in Section 5.2.2. 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 would appear as overwritten entries (collision).

Append. Some telemetry scenarios cannot be easily managed with a key-value store. A classic example is when a switch exports a stream of events (e.g., packet losses [79], congestion events [19], suspicious flows [43], latency spikes [76]). 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 4b)

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 then intercepts the packet and generates 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 data per-list. Append adds reports sequentially and contiguously into memory. This leads to an efficient use of memory and strong query performance. Translation also allows us to *significantly* improve on the collection speeds by batching multiple reports together in a single RDMA operation.

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 [10] and CM [14] sketches), estimating the number of flows (e.g., HyperLogLog [16]), and finding superspreaders (e.g., AROMA [7]).

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., [45]), which merges them together. However, this is a costly operation that our transport protocol can offload into specific RDMA calls (see Figure 4c). 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:

- Programmable switches support merging procedures that RDMA do not such as `max`.
- Aggregating at the translator reduces the translator-collector communication, shrinking the number of writes needed. Moreover, it relieves the collector's CPU from needing to aggregate the data, providing the aggregated data directly to the queryable data structure.

Key-Increment. Our Key-Increment is similar to the Key-Write primitive, but allows for addition-based data aggregation (see Figure 4d). 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 [52, 62]. The Key-Increment primitive can then deliver collection of these evicted counters at RDMA rates. As with Key-Write, the introduction of a translator reduced network overheads compared with a more naive design.

Our Key-Increment memory acts as a Count-Min Sketch [14]. 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 [14]. Note the memory for the counters may have to be cleared periodically, depending on the application.

Extensibility. Our approach is also easily extensible to other primitives, although they would remain constrained by the limitations imposed by commodity programmable switches [48]. These limitations could be overcome by implementing the translator logic into FPGA-based smartNICs (see Section 6).

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 reporters with a level of importance, where higher importance means lower chance of losing a report as explained below.

Telemetry Flow Control & Prioritization. The translator tracks its RDMA packet generation rate so to ensure that it never congests the collector's NIC. This is important as congestion leads 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-routes critical reports to its local CPU for temporary storage[3]. It further informs reporters of the congestion to limit additional 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[4].

[3]The switch has a limited data bandwidth towards its CPU, which limits the amount of reports that can be collected during periods of collector congestion.

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 addresses 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 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. Sketch-merge will aggregate sketches together for a network-wide view, and therefore all go to the same collector. One could improve the aggregation scaling by dividing the network into sections which are first aggregated internally into section-wide sketches, before aggregating these sketches into a single network-wide sketch.

4 IMPLEMENTATION

Our codebase includes approximately $4K$ lines of code divided between the logic for the DTA reporter (§ 4.1), the translator (§ 4.2) and collector RDMA service (§ 4.3). The current implementation has full support for the Key-Write and Append primitives, while Sketch-Merge and Key-Increment are proposed future additions[5].

4.1 DTA Reporter

The reporter pipeline is written in around 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 reports 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.

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

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

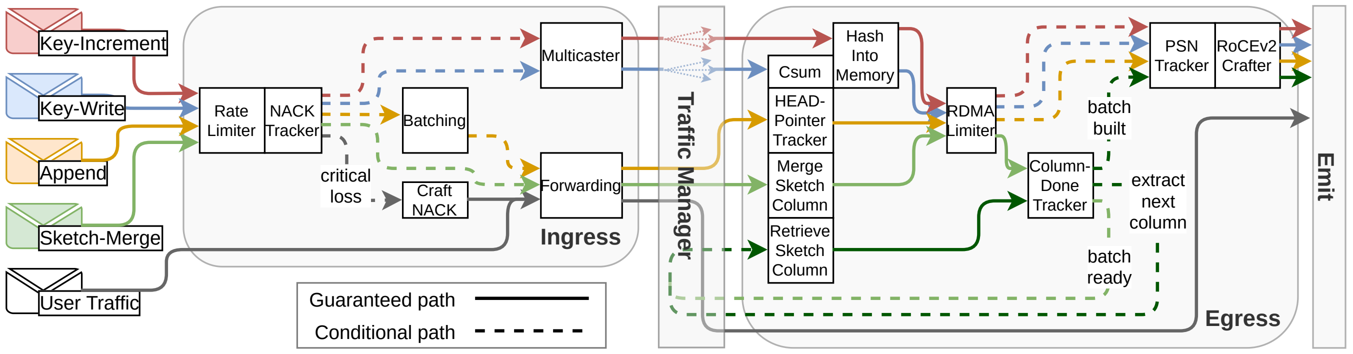


Figure 5: A translator pipeline with support for Key-Write, Key-Increment, Append, and Sketch-Merge. Five paths exist for pipeline traversal, used to process different types of network traffic in parallel while efficiently sharing pipeline logic.

Retransmission can be done in two ways: by storing small reports in SRAM which allows for handling this operation directly in the data plane, or by storing telemetry data in the switch CPU if retransmission is required for large telemetry payloads that can not reasonable fit in switch SRAM[6].

4.2 DTA Translator

The translator pipeline is written in $1.6K$ lines of P4_16 for the Tofino ASIC. The pipeline is designed to efficiently reuse 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 setting up the RDMA connection to the collector by crafting RDMA Communication Manager (RDMA_CM) packets, which are then injected into the ASIC. It also creates several multicast-rules, which are used by Key-Write and Key-Increment packets to trigger multiple simultaneous RDMA operations from a single input DTA operation.

Key-Write and **Key-Increment** both follow the same fundamental logic, with the difference being the RDMA operation that they trigger.Key-Write triggers RDMA Write operations, while a Key-Increment triggers RDMA Fetch-and-Add. Both cause 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 a concatenated $4B$ checksum for Key-Write [7].

Append has its logic split between ingress and egress, as shown in Figure 4b, 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. Assuming each reporter stores one sketch column in each DTA packet [77], the 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 instead is used to craft a NACK that is returned to the source switch. Valid operations continue to egress, where the column counters are merged with the in-translator sketch. The number of merged sketches are tracked per-column, and columns that contain network-wide information are flagged for transfer to the collector. Sketches are written to collector memory through RDMA Write as batches of C columns, which are written contiguously in memory, thus reducing the RDMA message rate of Sketch-Merge by a factor of C. Every Cth completed column triggers 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.

Finally, **flow control** between reporters and translator is achieved by a combination of meters and per-reporter packet sequence trackers. Tofino-native meters gauge the RDMA generation rate of the translator, and conditionally drop or reroute reports to the 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.

[6]Using reporter CPU for retransmission would limit the rate of retransmitable reports, while non-retransmitable reports are unaffected by this limit.
[7]There is further support for bigger checksums if query errors are not acceptable even in very rare cases.

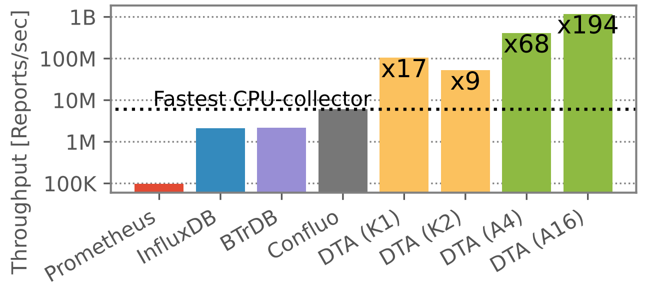


Figure 6: A performance comparison of DTA against state-of-the-art CPU-collectors. CPU-based collectors use 16 cores for data ingestion, while DTA uses none. DTA(K1) collects Key-Write reports at redundancy 1, DTA(K2) at redundancy 2. DTA(A4) collects Append reports at batch size 4, DTA(A16) at batch size 16. DTA speedups vs. the fastest CPU-collector are shown.

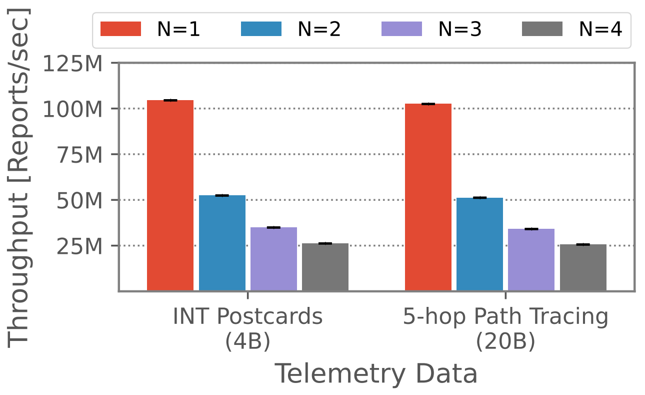


Figure 7: INT per-flow path tracing collection, using the DTA Key-Write primitive, either as per-hop postcards (4B) or full 5-hops path (20B).

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 advertise primitive-specific metadata to the translator using RDMA-Send packets.

5 EVALUATION

We evaluated the performance of our system by connecting two x86 servers through a BF2556X-1T [53] Tofino 1 [29] switch with 100G links. Both servers mount a 2x Intel Xeon Silver 4114 CPUs, 2x32GB DDR4 RAM clocked to 2.6GHz and run Ubuntu 20.04 (kernel 5.4). One of them served as DTA report generator using TRex [13]. The other, equipped with an RDMA-enabled Mellanox Bluefield-2 DPU [56], served as collector. Here, server BIOS has been optimized for high-throughput RDMA[8], and all RDMA-registered memory is allocated on 1 GB huge pages. Across the board, our evaluation results show the tremendous promise of DTA (or RDMA-enabled systems more generally) to improve the data collection rate of network telemetry systems through greater memory transaction speeds.

5.1 CPU-collector vs DTA Performance

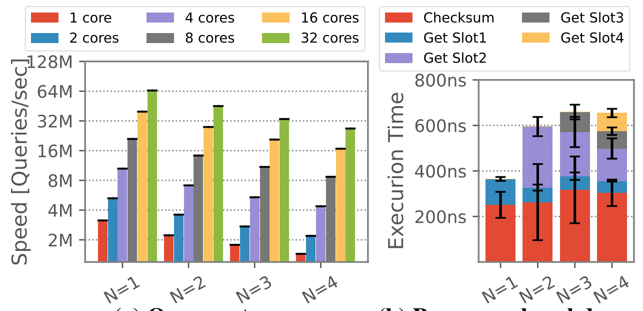
We compare the performance of DTA against several CPU-based collectors when receiving (key, value) reports with 4B values (e.g., switch IDs) and 13B keys (e.g., a flow's 5-tuple). Specifically, we tested INTCollector [70], to the best of our knowledge the only INT collector available in open source, using either Prometheus or InfluxDB for storage. We also studied BTrDB [4], and the state-of-the-art solution for high-speed networks, Confluo. All of them use 16 dedicated CPU cores in the same NUMA-node and with configurations to allow offline key-based queries.

Figure 6 shows their maximum achievable throughput in terms of reports collected per second. We compare these results with the performance of DTA when using different configurations. While the alternative collectors are configured with their default settings, for DTA we evaluate multiple configurations, depicting different primitives and speed to resource tradeoffs. Specifically, we configured DTA to work with the Key-Write primitive without redundancy (e.g., $N = 1$) (DTA (K1)) and with $N = 2$ redundancy (DTA (K2)). This is the case where the incoming 4B reports have to be recorded in our key-value store. We also configured DTA to work with the Append primitive with batch size equal to 4 (DTA (A4)) and 16 (DTA (A16)). This is the case where the incoming 4B reports are treated as events to be appended to a dedicated list. Here, we can see the benefits of batching that makes the Append primitive perform better than the others as it efficiently batches several reports together into fewer RDMA packets.
Takeaway: DTA greatly outperforms all existing solutions. When adopting a key-value store data structure it can ingest up to $17x$ more reports than Confluo. When adopting a list for event analysis, the performance benefit can reach a $194x$ gain.

5.2 Key-Write Primitive Performance

We have benchmarked the collection performance of the DTA Key-Write primitive using INT as a use case. We instantiated a $4GiB$ key-value store at the collector and had the translator receiving either $4B$ or $20B$ INT messages from the reporter (our traffic generator). The former case emulates the scenario of

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



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

Figure 8: Key-Write primitive query performance.

having INT working in postcard mode, while the latter reproduces an INT path tracing configuration on a 5-hops topology where the last hop reports data to a collector. We repeated the test using different levels of redundancy (N) and reported the results we obtained in Figure 7. 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. Furthermore, the collection rate is unaffected by the increase in the telemetry data size, until the $100Gbps$ line rate is reached. In our tests, we saw that this was the case for telemetry payloads $16B$ or larger.

Takeaway: The Key-Write primitive is able to collect over $100M$ INT reports per second. The performance depends on the redundancy level used.

5.2.1 *Key-Write Query Speed.*

Querying for data stored in our key-value store using the Key-Write primitive requires the calculation of several hashes.

Here we evaluate the *worst case* performance scenario, when the collector has to retrieve every redundancy slot before being able to answer a query. Specifically, we queried $100M$ random telemetry keys, with a key-value data structure of size $4GiB$ containing $4B$ INT postcards data alongside $4B$ concatenated checksums for query validation. Figure 8a shows the speed at which the collector can answer incoming telemetry queries using various redundancy levels (N).

Key-Write query processing can be easily parallelized, and we found the query performance to scale near-linearly when we allocated more cores for processing. Figure 8b shows the execution time breakdown for answering queries. Most of the execution time is spent in calculating CRC hashes, for either verifying the concatenated checksum (*Checksum*), or calculating memory addresses of the N redundancy entries (*Get Slot*).

The query performance is therefore highly impacted by the speed of the CRC implementation[9], and more optimized implementations should see a performance increase.

Takeaway: Because of RDMA, our Key-Value store can insert entries faster than the CPU can query. The performance of the CRC implementation plays a key role.

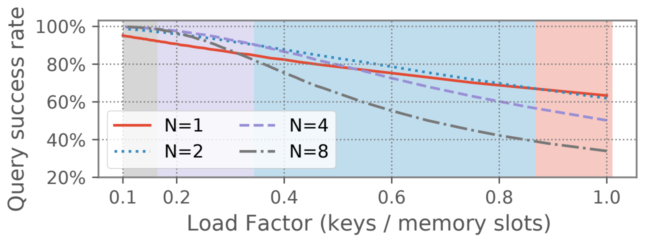


Figure 9: Average query success rates delivered by the Key-Write primitive, depending on the key-value store load factor and the number of addresses per key (N). The background color indicates optimal N in each interval.

5.2.2 *Redundancy Effectiveness.*

The probabilistic nature of Key-Write cannot guarantee final queryability on a given reported key due to hash collisions with newer data entries. We show in Figure 9 how the query success rate[10] depends on the load factor (i.e., the total number of telemetry keys over available memory addresses), and the redundancy level (N). There is a clear data resiliency improvement by having keys write to $N > 1$ memory addresses when the storage load factor is in reasonable intervals. When the load factor increases, adopting more addresses per key does not help because it is harder to reach consensus at query time. The background color in Figure 9 indicate which N delivered the highest key-queryability in each interval.

Higher levels of redundancy improve data longevity, but at the cost of reduced collection and query performance as demonstrated previously in Figures 7 and 8. Determining an optimal redundancy level therefore has to be a balance between an enhanced data queryability and a reduction in primitive performance, and $N = 2$ is a generally good compromise, showing great queryability improvements over $N = 1$.

Takeaway: Increasing the redundancy of all keys does not always improve the query success rate. An optimal redundancy should be set on a case-by-case basis.

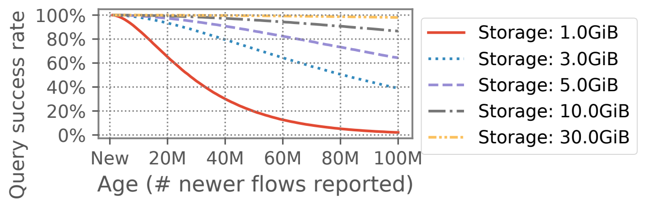


Figure 10: DTA Key-Write ages out eventually. This figure shows INT 5-hop path tracing queryability of 100 million flows at various storage sizes. Note how the data longevity increases linearly when larger key-value storages are allocated at the collector.

[9]Our current collector service calculated CRC using the generic Boost libraries: https://www.boost.org/

[10]The query success rate is defined as the probability at which a previously reported key can still be queried from the key-value store.

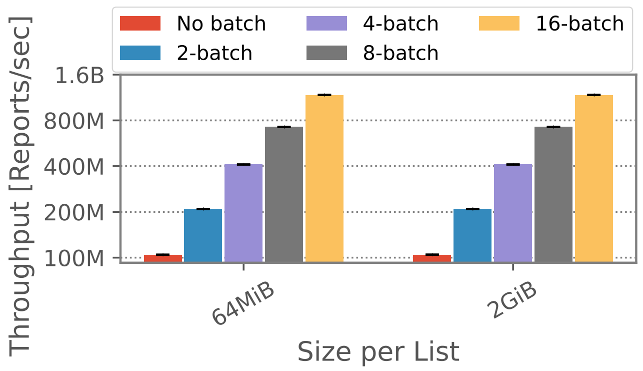


Figure 11: Telemery event-report collection, using DTA Append and different batch sizes. Increased batch sizes yield a linear performance increase until we achieve line-rate with batches of $4x4B$. The collection speed is not impacted by the list sizes.

5.2.3 *Data Longevity.*

Data reported by the Key-Write primitive will age out of memory over time due to hash collisions with subsequent reports, which overwrites the memory slots. Figure 10 shows the queryability of randomly reported INT 5-hop path tracing data at various storage sizes and report ages, with redundancy level $N = 2$ and $4B$ checksums. For example, when 100 million flows share just 3GB (i.e., 30B storage per flow path), we see how the average queryability is 62% across all 100 million flows; with a steep decline to 17.9% for the oldest reports. However, increasing the storage capacity to 30GiB significantly increases the average data queryability to 99.9% across all 100 million flows.

Takeaway: It is possible to record data from around 100M flows in the key-value store while maintaining a 99.9% quryability with just 30GiB of storage.

5.3 Append Primitive Performance

We have benchmarked the performance of the Append primitive for collecting telemetry event-reports, both at different batch sizes and total size of the allocated data list, while reporting data into a single list. The results are shown in Figure 11.

We noticed no performance impact from different report sizes, until we reached the line-rate of $100G$ at large batch sizes after which the performance increased sub-linearly. The results in Figure 11 show this effect for $4B$ queue-depth reports, where we reach line-rate at batches of 4. Our base performance is bounded by the RDMA message rate of the NIC, which is the current collection bottleneck in our system, and the high performance of the Append primitive is due to including several reports in each memory operation. Performing equivalent tests with up to 255 parallel lists showed a negligible performance impact.

Takeaway: The Append primitive is able to collect over 1 billion telemetry event reports per second.

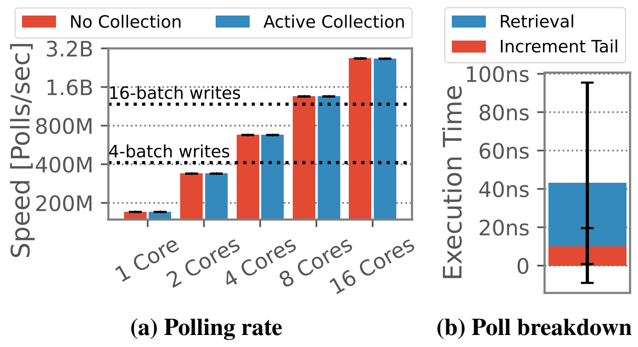


Figure 12: Append primitive query performance. Append-lists are queried either while collecting no reports or at 50% capacity (while collecting *600 million* reports per second). Collection has a negligible impact on data retrieval rate, and processing rate scales near-linearly with the number of cores. The dotted lines show the maximum collection rates at different batch sizes.

5.3.1 *Append List-Polling Rate.*

In Figure 12a, we show the raw list polling rates, which is the speed at which appended data 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 to collecting at 50% maximum RDMA capacity. Simultaneously collecting and processing telemetry data showed no noticeable impact on either collection or processing.

Extracting telemetry data from the lists is a very lightweight process, as shown in Figure 12b, requiring a pointer increment, possibly rolling back to the start of the buffer, and then reading the memory location. We allocated a number of lists equal to the number of CPU cores used during the test to prevent race conditions at the tail pointer[11].

Our tests show that the CPU manages to extract every appended list entry even when using very large batch sizes. Just 8 cores proved capable of extracting every telemetry report even when large batches reported at maximum capacity. This leaves us much processing power for complex real-time telemetry processing.We see that the collector can even retrieve list entries faster than the RAM clock speed, which is likely due to cache prefetching that greatly improves the performance of sequential memory accesses[12].

Takeaway: The CPU can retrieve appended reports faster than they can be collected (Figure 11), with margin left over for further processing.

[11]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.

[12]That also explains the relatively high variation in retrieval latency.

Resource	Base footprint	Batching	Retransmission
SRAM	5.5%	+3.0%	+0.5%
Match Crossbar	5.9%	+9.0%	+0.2%
Table IDs	27.6%	+7.8%	+1.0%
Hash Dist Unit	13.9%	+20.8%	-
Ternary Bus	20.3%	+7.8%	+1.1%
Meter ALU	10.4%	+31.3%	+2.1%

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

5.4 Translator Resource Footprint

In Table 3, we show the resource usage of the translator, alongside the additional costs of including Append batching and per-reporter report-loss detection.

Retransmission-support is achieved by tracking per-reporter sequence numbers and adding NACK-generation. It has a small resource cost, even in the case of large-scale deployments supporting $65K$ reporters in a single translator. We found that batching of Append data has a relatively high cost in terms of memory logic (meter ALU), due to our non-recirculating RDMA-generating pipeline requiring access to all $B-1$ stored entries during a single pipeline traversal. However, batching has also the potential for a tenfold increase in collection throughput, and therefore it is a worthwhile trade-off. A compromise is to reduce the batch sizes, as they linearly correlate with the number of additional meter ALU calls.

Deploying multiple simultaneous 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 that the actual SRAM footprint of the translator is small, which shows that the translator can support much more complex list setups than the 255 lists that are included at the time of evaluation.

From this analysis, we can see that DTA translators are already feasible in first-generation Tofino switches. In next generation programmable switches (i.e., Tofino-2), it will be possible to design even more complex DTA primitives and aggregation functions than those presented here, given their capacity for significantly more memory logic [33].

Takeaway: The translator logic supporting Key-Write and Append primitives is lightweight, requiring up to 10% of the total SRAM. Batching can impose a high toll on Meter ALU.

5.5 DTA vs RDMA Reporter Costs

We compared the cost of generating DTA reports against an equivalent system that produces RDMA packets directly from the reporters. Both pipelines implements the same logic and they only differ in the telemetry generation mechanism (DTA in one case, RDMA in the other).

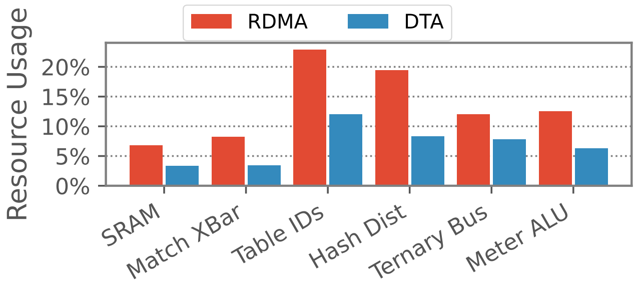


Figure 13: Hardware resource costs of DTA Reporter and a baseline RDMA-generating. Both implementations have same logic except for the reporting mechanism (RDMA in one case, DTA in the other).

The RDMA-generating pipeline supports 1024 collectors, while the DTA-generating pipeline supports 512 translators (we assume 2 collectors per-translator). Figure 13 shows the footprint of both designs, where we see how DTA significantly reduces the overall resource costs in the switching hardware. The RDMA-based reporter costs are in handling RoCEv2 protocol, while for DTA it is just about encapsulating telemetry data into our custom header running over a simple UDP.

Takeaway: DTA halves the resource footprint of reporters compared with RDMA-generating alternatives. The downside is the higher cost in the translator logic (Table 3).

6 DISCUSSION

Translator placement. There are two main approaches we have considered on where to deploy the translator: a SmartNIC located at the collector and a top-of-the-rack programmable switch (which we explored in this work). The SmartNIC approach would allow us to use DTA traffic only within the network, and the on-NIC CPU would process incoming DTA packets and translate them into local DMA calls. However, highly programmable network interface cards [56] potentially suffer from reduced performance when the on-NIC ARM core is in charge of processing most of the incoming traffic [18, 37], as would be the case in this scenario. Using FPGA-based cards would solve the problem and would also allow the design of primitives that are not constrained by the limits of commodity P4 programmable switches [48], but they are also power-hungry, expensive, and difficult to program [18], which limits their potential. Nevertheless, our P4 translator code can be a starting point for P4-capable NICs [63]. Exploring SmartNIC DTA translation is left for future work.

Collector placement. Our framework allows a single translator to write reports to multiple collectors that are physically placed in its rack. Placing multiple collectors on one rack reduces the number of translators, which require more hardware resources than reporter DTA switches (see § 5.4). However, this setup may lead to larger amounts of traffic flow to that rack, possibly affecting collocated servers and applications.

Enhanced data aggregation at switch. If we grant to the translator the ability to *read* the collector's memory via RDMA calls, then more aggressive data aggregation capabilities can be implemented. For example, we could directly manage from the translator a cuckoo hash table located in the collector. Nevertheless, this becomes challenging when a translator has to deal with a growing number of reporters. Alternatively, we could potentially implement partial state-aware aggregation approaches [17]. Trade-offs in terms of performance and hardware resources costs are beyond the scope of this work and a matter for future research.

Push notifications. CPU-based collectors have an advantage over our solution: the CPU can trigger complex analysis tasks as soon as it receives reports. In our case, for key-value store operations, the CPU must first find out if new data has been written into the memory; however, we assume for Append operations the CPU is monitoring the lists continuously, which would allow for equivalent reactivity to CPU-based solutions. Additionally, DTA packets can include an *immediate flag*, which can be used by the translator to immediately inform the CPU that new data has arrived through RDMA immediate interrupts (e.g., a flow is currently experiencing problems). The underlying algorithm that decides which packets include such a flag are beyond the scope of this work.

Trade-offs in batching reports. Batching multiple reports sharing the append primitive greatly increases telemetry collection rates. However, this requires in-ASIC statefulness. Even though the SRAM footprint of batching is relatively small (3% in our Section 5.4 example), we saw a significant impact on the available memory logic. Each memory operation is limited to a 32-bit bus, requiring multiple memory operations to process batch entries larger than $4B$. Complex deployments with large telemetry payloads might therefore have to reduce the batch size to free up switch memory logic (e.g., a batch with $8B$ entries might halve the batch size compared with $4B$ entries to keep a similar footprint). One possible alternative is to reduce the hardware resources by allowing RDMA-crafting packets to traverse the pipeline multiple times while retrieving the batch. For example, batches may use half as much memory logic if allowed to recirculate once, by re-using memory logic between pipeline traversals. This may also allow us to increase the batch sizes further to reduce the load on the collector's NIC, at the cost of increased egress-pipe traffic during RDMA-creation for Append operations.

7 RELATED WORK

Telemetry and Collection. Traditional techniques for monitoring the status of the network have looked into periodically collecting telemetry data [20, 22] or mirroring packets at switches [58, 81]. The former generates coarse-grained data that can be significant given the large scale of todays' networks [68]. The latter has been recognised as viable option only if it is known in advance the specific flow to monitor [81]. The rise in programmable switches has enabled fine-grained telemetry techniques that generate a lot more data [8, 19, 21, 65, 79, 81]. 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 [39, 70], or reducing the load through offloaded pre-processing [44] and in-network filtering [30, 42, 71, 79]. To the best of our knowledge, only one proposal has investigated the possibility to entirely bypass collectors' CPU, but limits the collection process to data that can be represented as a key-value store [5], which is not suitable for telemetry systems based on events or sketches. In this paper, we propose an alternative solution which is generic and works with a number of existing state-of-the-art monitoring systems. An alternative approach is letting end-hosts assist in network-wide telemetry [25, 65], which unfortunately requires significant investments and infrastructure changes.
RDMA in programmable networks. Recent works have shown that programmable switches can perform RDMA operations [41], 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 [47, 60]. As discussed in Section 2, telemetry collection brings new challenges when used in conjunction with the RoCEv2 protocol.

8 CONCLUSION

In this paper, we presented Direct Telemetry Access (DTA), a telemetry collection solution optimized for moving, aggregating, and indexing reports from switches to collectors' memory. We designed a custom protocol that enables reporters (switches reporting telemetry data) to interact with translators (the ToR switch directly connected to a collector). Reporters send telemetry data using our novel and expressive primitives that allows integration with telemetry systems such as INT. Translators aggregate the received data and trigger standard RDMA calls toward their associated collector.

DTA can to write in our key-value store over 100 million INT reports per second, without any CPU processing, which is 17x better than the CPU-based state-of-the-art. Further, when the received data can be recorded sequentially as in the case of event-telemetry reports, we can store up to 1 billion reports per second, ≈194x faster than state-of-the-art.

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] Michael P Andersen and David E Culler. 2016. Btrdb: Optimizing storage system design for timeseries processing. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 39–52.

[5] Anonymous. 2021. Anonymized workshop paper. (2021).

[6] Arista. 2022. Telemetry and Analytics. https://www.arista.com/en/solutions/telemetry-analytics. (2022). Accessed: 2022-02-02.

[7] 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.

[8] 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.

[9] BROADCOM. 2017. Trident Programmable Switch. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series. (2017).

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

[11] 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.

[12] Cisco. 2019. Explore Model-Driven Telemetry. https://blogs.cisco.com/developer/model-driven-telemetry-sandbox. (2019). Accessed: 2021-06-24.

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

[14] 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.

[15] 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.

[16] 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.

[17] Sam Gao, Mark Handley, and Stefano Vissicchio. 2021. Stats 101 in P4: Towards In-Switch Anomaly Detection. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. 84–90.

[18] 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.

[19] The P4.org Applications Working Group. 2020. Telemetry Report Format Specification. https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report_latest.pdf. (2020). Accessed: 2021-06-23.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] Huawei. 2020. Overview of Telemetry. https://support.huawei.com/enterprise/en/doc/EDOC1000173015/165fa2c8/overview-of-telemetry. (2020). Accessed: 2021-06-24.

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

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

[29] Intel. 2016. Intel® Tofino™ Series Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. (2016). Accessed: 2022-01-25.

[30] Intel. 2020. In-band Network Telemetry Detects Network Performance Issues. https://builders.intel.com/docs/networkbuilders/in-band-network-telemetry-detects-network-performance-issues.pdf. (2020). Accessed: 2021-06-04.

[31] Intel. 2020. 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. (2020). Accessed: 2021-06-11.

[32] Intel. 2021. 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. (2021). Accessed: 2021-06-10.

[33] Intel. 2022. Intel Tofino 2: Second-generation P4-programmable Ethernet switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html. (2022).

[34] 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*.

[35] 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*.

[36] 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.

[37] Georgios P. Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostic, and Gerald Q. Jr. Maguire. 2021. What You Need to Know About (Smart) Network Interface Cards. In *Passive and Active Measurement (PAM)*. Springer.

[38] Mikhail Kazhamiaka, Babar Memon, Chathura Kankanamge, Siddhartha Sahu, Sajjad Rizvi, Bernard Wong, and Khuzaima Daudjee. 2019. Sift: Resource-Efficient Consensus with RDMA. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. Association for Computing Machinery, 260–271.

[39] 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.

[40] 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*.

[41] 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.

[42] 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.

[43] 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*. Association for Computing Machinery, 14–26.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

[49] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX Association, 103–114.

[50] 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.

[51] 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.

[52] 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.

[53] APS Networks. 2019. Advanced Programmable Switch. https://www.aps-networks.com/wp-content/uploads/2021/07/210712_APS_BF2556X-1T_V04.pdf. (2019). Accessed: 2022-01-25.

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

[55] NVIDIA. 2017. NVIDIA Mellanox Spectrum Switch. https://www.mellanox.com/files/doc-2020/pb-spectrum-switch.pdf. (2017).

[56] NVIDIA. 2021. NVIDIA BLUEFIELD-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf. (2021). Accessed: 2022-01-25.

[57] NVIDIA. 2021. NVIDIA Mellanox NIC's Performance Report with DPDK 21.08. https://fast.dpdk.org/doc/perf/DPDK_21_08_Mellanox_NIC_performance_report.pdf. (2021). Accessed: 2022-02-01.

[58] 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.

[59] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. 2022. RDMA is Turing complete, we just did not know it yet!. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[60] 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.

[61] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. Association for Computing Machinery, 183–197.

[62] 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.

[63] Pensando Systems. 2021. Pensando DSC-100 Distributed Services Card. https://pensando.io/wp-content/uploads/2020/03/

DSC-100-ProductBrief-v06.pdf. (2021). Accessed: 2022-01-23.

[64] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. 2018. Tailwind: Fast and Atomic RDMA-Based Replication. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 851–863.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. 2021. XStore: Fast RDMA-Based Ordered Key-Value Store Using Remote Learned Cache. *ACM Transactions on Storage* 17, 3 (2021).

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

[74] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 561–575.

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

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.