Direct Telemetry Access

Paper # 224, XXX pages

ABSTRACT

The emergence of programmable switches allow 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. Our evaluation demonstrates that we can enhance the existing telemetry framework by XX%.

1 INTRODUCTION

In modern data centers, network telemetry is the foundation for many network management tasks such as traffic engineering, performance diagnosis, and attack detection [7, 24, 29, 39, 67, 73, 76, 77]. With the rise of programmable switches [8, 30, 54], telemetry systems can now monitor network traffic in real time and at a fine granularity [7, 18, 43, 49, 72, 79]: a key enabler to support automated network control [1, 22, 44] and detailed troubleshooting [19, 29, 62]. 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 [5, 11, 20, 25, 29, 38, 51].

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 [49, 77] and a data center network comprises hundreds of thousands of switches [19]. The amount of data keeps growing with larger networks and higher line rates [58]. As a consequence, it is increasingly hard to scale data collection in telemetry systems [38, 68, 77].

Indeed, existing research boosts scalability by improving the collectors' network stacks [38, 68] by aggregating and filtering data at switches [28, 41, 49, 69, 77], or by switch cooperation [42]. 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 limit their performance (§2).

We propose *Direct Telemetry Access* (DTA), a telemetry collection protocol 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) [27] calls to collectors. RDMA is available on many commodity network cards [31, 63, 71] and can perform hundreds of millions of memory writes per second [63], which is significantly faster than the most performant CPU-based telemetry collector [38].

Previous work [40] 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 [35].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 propose to 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 designed a number of switch-level RDMA language extension primitives such as *Key-Write*, *Append*, *Sketch-Merge*, and *Key-Increment* that can be converted by the translator into standard RDMA calls. These primitives allow multiples 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 [18], PINT [7], Marple [49] and Sonata [20].

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 limited 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 proposed by the research community.
- We implement DTA using commodity programmable switches and RDMA NICs. (We 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.

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 configuring INT in postcard mode [28] on a commodity 6.4Tbps switch, and considering a standard load of approximately 40% [74], it could generate up to $19M$ reports per second. Recent solutions greatly reduce the pressure on collectors by adopting smart pre-processing and filtering of data at switches bringing down the reporting rate to just $950K$ reports per second [77].

[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.

System	Per-switch Report Rate
INT Postcards (Per-hop latency, 0.5% sampling)	19 Mpps
Marple [49] (TCP out-of-sequence)	6.72 Mpps
Marple [49] (Packet counters)	4.29 Mpps
NetSeer [77] (Flow events)	950 Kpps

Table 1: Per-reporter data generation rates by various monitoring systems, as presented in their individual papers (INT is assuming an arbitrary sampling rate of 0.5%)). Numbers are based on 6.4Tbps switches.

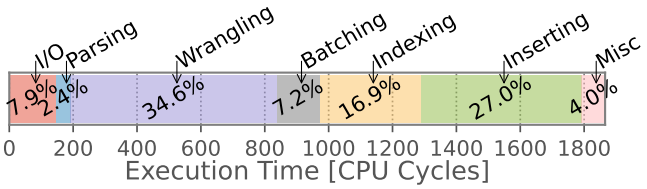


Figure 1: CPU-based collectors are inherently inefficient. Shown above is the collection work breakdown of Confluo as a demonstration, representing a highly optimized collector.

To get a network-wide view, we may need to collect data from hundreds of thousands of switches in data center networks [19]. 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 the 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 [53]. As a consequence, *thousands of CPU cores* have to be dedicated to *just receive* billions of reports per second. Once the data is received, it has to be processed at the user-space to store the information in a queryable data structure. The processing usually involves three key operations that further require more CPU cores: *parsing* (extract content from the incoming report packets), *data wrangling* (process the data to make it suitable for insertion, which is dependent on the specific data structure being used), and *storing* (determine where to store it in memory).

This adds up on the already costly I/O and we show that it is even much more expensive than that. Figure 1 shows the breakdown of Confluo [38], the state-of-the-art collector for high-speed networks, when receiving and processing $100K$ reports on an Intel Xeon Silver 4114 CPU @ 2.20GHz. Data wrangling in Confluo is the process of applying user-based criteria to input reports. Storing involves batching, computing a key (indexing), and insertion, where reports are scheduled to be written in memory. We found that most of the CPU cycles (93%) are spent in data wrangling and storing, almost 90x of the cost of its I/O. As a consequence, optimizing the collector stacks is becoming essential to minimize the number of cores used for *data collection* [38].

2.2 Challenges for using RDMA

Our goal is to reduce the CPU utilization at 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 [37], reads in key-value stores [46, 70] and data replication [61]. 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 [40], 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 [38]. 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. Enabling advanced RDMA offloads on commodity NICs is possible, but at the cost of a reduction of performance that can hit even 2 orders of magnitude [56]. Furthermore, it not possible to support multiple senders writing at the same memory location while guaranteeing no conflicts or race-conditions. This is of paramount importance 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 loss this assumption does not hold anymore, leading to queue-pair invalidation which significantly degrades RDMA performance [48]. For this reason, current deployments adopt per-hop flow-control mechanisms such as PFC [26] to rate-limit RDMA traffic in case of congestion [47, 48, 78]. Unfortunately, this practice can introduce deadlocks, a circular buffer dependency between switches that cause network throughput collapse [23]. 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 [14]. 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

MY: Rather than having four bullets one for each challenge in sec 2, I think this preamble can present the key design points first: 1) Introduce the translator 2) introduce a few primitives that enable basic data structures 3) Data prioritization and telemetry flow control MY: We can have one paragraph for each design decision and how they address the RDMA challenges (not necessarily one-to-one mapping). MY: We then present Figure 2 the design architecture.

SR: My attempt to address Minlan's comments: In this paper, we present Direct Telemetry Access (DTA), a collection protocol that addresses the outlined challenges. The goal of DTA is to allow *Reporters* (i.e., switches exporting telemetry data) to directly insert telemetry information into the collector's memory to increase throughput at the collector. The key ideas of DTA are as follows:

- DTA uses a translator at the last-hop to intercept telemetry packets. The translator maintains an active connection to the RDMA NIC at the collector and generates the intercepted telemetry packets into suitable RDMA calls. This reduces the number of connections handled by the collectors RDMA NIC and also reduces the chance of generating out-of-order packets to the RDMA NIC.
- To enable faster queries, DTA introduces four primitives that makes RDMA compatible with many existing telemetry solutions.
- To avoid packet loss between the translator and the collector, DTA introduces a flow control mechanism that rate-limits data from reporters to translator that prioritizes specific telemetry data (e.g., pipeline-failure reports can be prioritized above path tracing).

Figure 2 shows an overview of DTA. Reporters running network monitoring systems generate telemetry data to translators that sit one hop from the collectors. The translators,

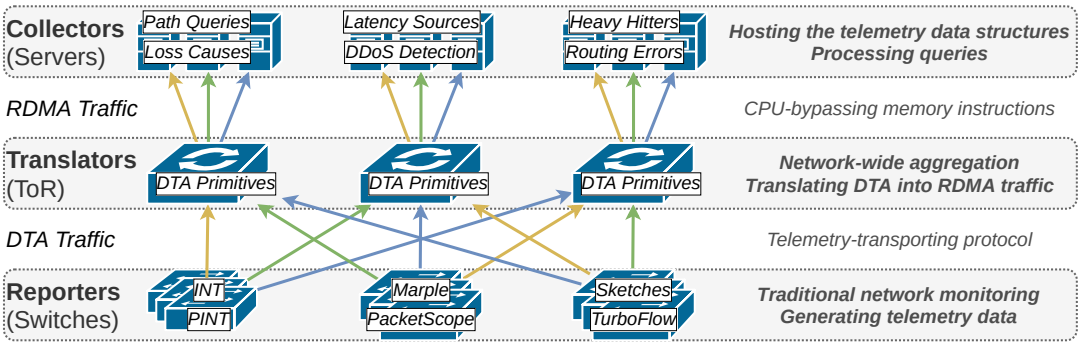


Figure 2: Overview of the DTA telemetry reporting flow

generate DTA specific packets that include RDMA calls to insert into custom DTA primitives in the collectors memory. Finally, a network operator can query on the stored data to support various applications. SR: End of attempt to address Minlan's comments:

Figure 2 shows an overview of DTA. *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), or *translator*, 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 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.

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, 33, 34]. The problem is that, while

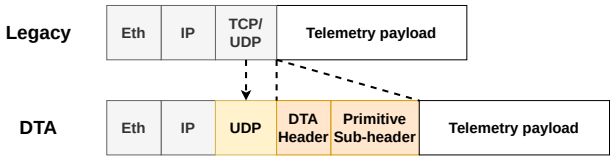


Figure 3: The structure of a DTA packet report, compared with a legacy TCP-based one.

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-limits 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, we design something ad-hoc that regulates only telemetry traffic and show 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 [14, 35]. 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 packet. The telemetry payload exported by a switch is encapsulated into a UDP packet that carries our custom header.

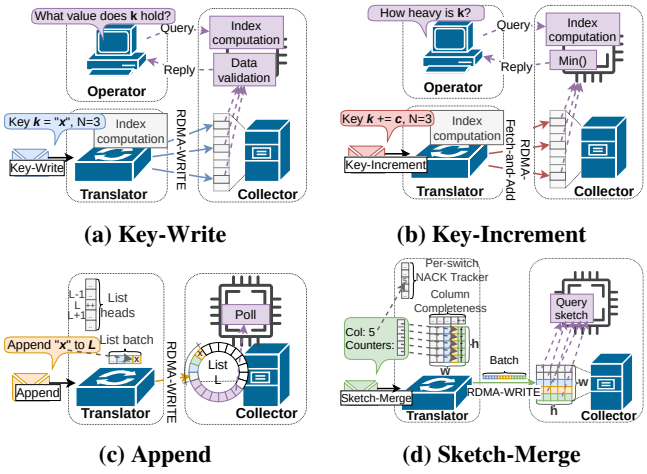


Figure 4: A high-level overview of the DTA primitives

DTA is designed to allow ease-of-integration with state-of-the-art telemetry monitoring systems [7, 18, 20, 49] 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 enable support for a 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 enough to support many state-of-the-art monitoring systems. SR: I think it would be good to have a discussion on what primitives we cannot support and give general solution. Then we can point to that part of the paper here. Our approach is also easily extensible to allow other useful primitives to be added in the future. In the following, we discuss our designed primitives, illustrated in Figure 4.

SR: Suggestion: Maybe we can start off by giving how the primitive can be used, that is, just describe a few examples from the table. Then highlight the challenge, then provide the solution. **Key-Write.** This primitive (see Figure 4a) is designed for key-value pairs 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.

JL: I changed the text below to remove references to DART. 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, which 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.

We further reduce the network and 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 lets 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 5.3.2. 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). SR: Can we give an example of how we can prioritise telemetry reports? For example, you can say somthing like an elephant flow is detected in a switch. Prioritising that report is import for congestion control algorithms like HPCC to adjust the sending rate. Therefore deliveiry of that report is very important. This way this contribution comes as necessity.

Append. Some telemetry scenarios cannot be easily managed with just a key-value store. A classic example is when a switch is expected to export events. SR: What does export mean? It seems like any report from a switch is exported from the switch to the collector. Maybe we can say a stream of events that needs to be tracked?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.

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

Primitive	Example monitoring	Description
Key-Write	INT (Path Tracing) [18, 39]	INT sinks reporting *5x4B* switch IDs using *flow 5-tuple* keys
	Marple (Host counters) [49]	Reporting *4B* counters using *source IP* keys, through non-merging aggregation
	Sonata (Per-query results) [20]	Reporting *fixed-size* network query results using *queryID* keys
	PINT (Per-flow queries) [7]	*1B* reports with *flow 5-tuple* keys, using redundancies for improved data compression through $n = f(pktID)$
	PacketScope (Flow troubleshooting) [64]	Report *fixed-size* per-flow per-switch pipeline traversal information using <*switchID*,*flow 5-tuple*> as key
Append	INT (Congestion events) [18, 39]	INT sinks append *4B* reports to list of network congestion events
	Marple (Lossy connections) [49]	Report *13B* flows with packet loss rate greater than threshold
	NetSeer (Loss events) [77]	Appending *18B* loss event reports into network-wide list of packet losses
	Sonata (Raw data transfer) [20]	Appending *query-specific* packet tuples from switches to lists at streaming processors
	PacketScope (Pipeline-loss insight) [64]	On packet drop: send *14B* pipeline-traversal information to central list of pipeline-loss events
Sketch-Merge	C [9] and CM [13] sketches	Counter-wise `sum` of the sketches of all switches
	HyperLogLog [10, 15]	Register-wise `max` of the sketches of all switches
	AROMA [6]	Select network-wide uniform packet- and flow samples from switch-level samples
Key-Increment	TurboFlow (Per-flow counters) [59]	Sending selected *4B* counters from evicted microflow-records for central aggregation using *flow key* as keys
	Marple (Host counters) [49]	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 DTAJL: make keys and sizes EVEN MORE explicit? it's own column? Fix width

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)SR: Nit:Can we maitain the same order in Figure 4 as it is described in text? That is make (c) as (b). 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 in a single RDMA operation. SR: I think points about translator should be in the translator section. We can just stick to implementation of primitives here.

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 [9] and CM [13] sketches), estimating the number of flows (e.g., HyperLogLog [15]), and finding superspreaders (e.g.,SR: What is this? [6]).

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., [43]), 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.
- Because it is the first point to observe all data, we can use aggregation at the translator to reduce network overheads. SR: I think in Arch figure, we show many translators. So a single translator need not necessary see all the data.

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 [75] 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.SR: The content of this section has implementation mixed with the idea. Maybe we can push all these implementation details to Section 4?

Key-Increment. SR: I think we should put the motivation for the primitive first. 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 [49, 59]. 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 [13]. 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 [13]. 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* SR: Earlier, we set N=4, how does four values of redundancy match to the level of importance?. This impact the behavior of the translator as discussed below. SR: I think we should have a section before 3.1, that details DTA reports. We need to give a bit more details on the headers (including priority)

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-routes critical reports to its local CPU for temporary storage[3]. 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[4].

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. All columns for sketch-merge will be aggregated together for a network-wide view, and therefore all go to the same collector. One could improve 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. JL: Does this make sense?

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

[3]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

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

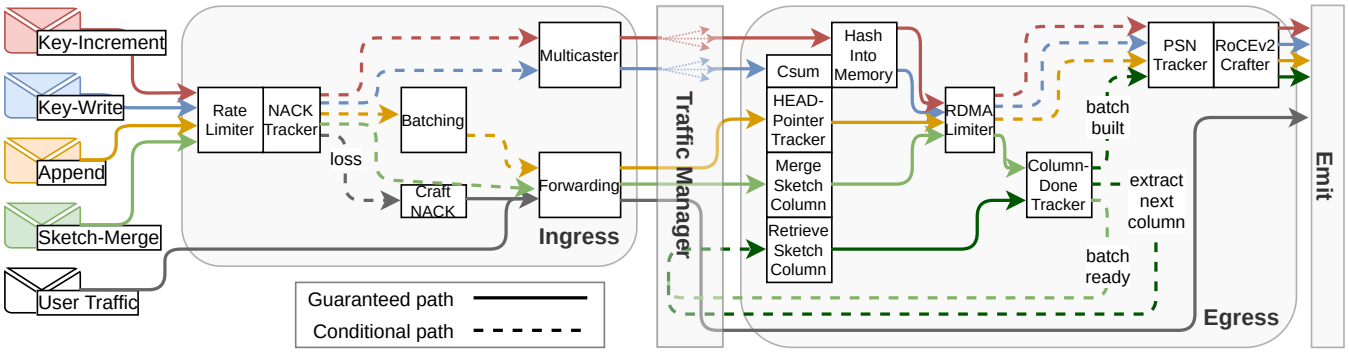


Figure 5: A translator pipeline with support for Key-Write, Key-Increment, Append, and Sketch-Merge. Five paths exist for pipeline traversal, used to process different types of network traffic in parallel while efficiently sharing pipeline logic.

Append primitives, while Sketch-Merge and Key-Increment are proposed future additions[5].

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

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 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

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

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 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

We evaluated the performance of our system by connecting two x86 servers connected by a BF2556X-1T [50] Tofino 1 [32] switch, which acts as either a reporter or translator depending on the test. All collection benchmarks are performed using *a single port* of the Mellanox Bluefield-2 2x100G DPU [52] without ARM processing as acting RDMA-capable network card. This network card has an optimal RDMA operation rate over $100Mops$ [56], which we later show is the base speed of our collection. The TRex traffic generator [12] is used to inject traffic into the switch. DTA packet traces, as emitted by the reporter, are written to storage for analysis and later replay through the translator, while RDMA traffic generated by the translator is sent directly to the collector RDMA-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[6], and all RDMA-registered memory is allocated on 1 GB huge pages.

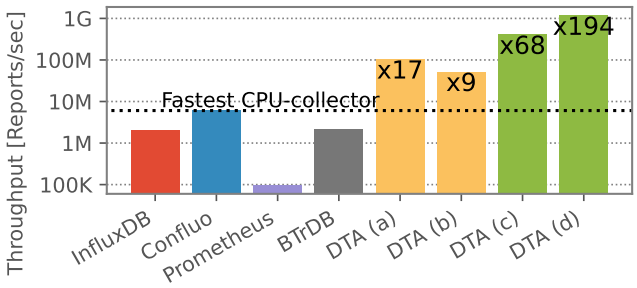


Figure 6: A performance comparison of DTA collection and current state-of-the-art CPU-collectors. CPU-based collectors dedicate 16 cores for data ingestion, while DTA uses 0 cores. (a,b): Collecting Key-Write reports at redundancy 1(a) and 2(b) e.g., for Marple. (c,d): Collecting telemetry event reports through Append at batch size 4(c) and 16(d) e.g., for NetSeer. CPU-based collectors omit the cost of packet I/O in this figure. JL: confluo is placeholder (single-core extrapolated following the curve in their paper)

5.1 CPU-collector vs DTA Performance

We have benchmarked several CPU-based collectors in our system testbed, where we report $4B$ telemetry payloads using collector-specific data headers (e.g., INTCollector [68] collected data with INT v0.5 headers [18] using either *Prometheus* or *InfluxDB* for storage; Confluo [38] processes a custom packet structure). The collectors are all configured to allow offline flow 5-tuple queries against the stored data to ensure equivalent collector functionality (Confluo uses 5 indexes combined with 5 pre-defined filters for simple real-time processing)[7]. 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 system did not include multi-core support [8]. Confluo [38] proved to be the fastest CPU-based collector, which it achieved by designing a highly efficient telemetry data structure. 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

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

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

[8]The InfluxDB and Prometheus results are linearly extrapolated from single-core results, and are therefore possibly over-estimating their performance.

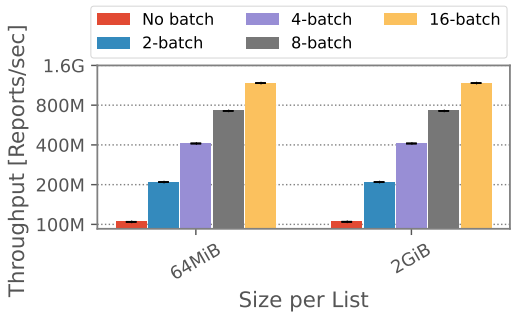
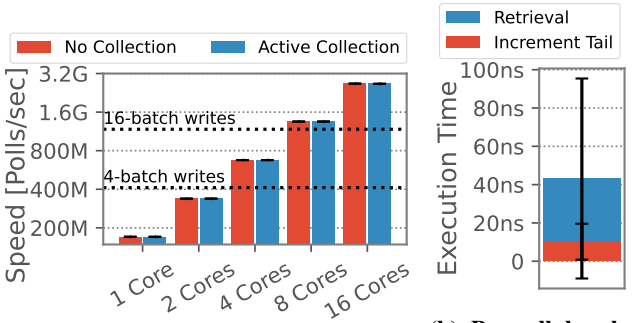


Figure 7: Telemery event-report collection, using DTA Append at different batch sizes. Increased batch sizes yield a linear performance increase until we achieve line-rate with batches of $4x4B$ or greater. Note how the collection speed is not impacted by the list sizes.



(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?

primitives). Our collection primitives outperformed all CPU-based collectors by an order of magnitude. JL: write specifics when we get the data

5.2 Append Primitive Performance

Our traffic generator fails to generate traffic at the incredible collection rates of this primitive ($> 1Gpps$ with large batches), and we therefore had every generated report trigger several append operations at once in the translator. This test therefore does not evaluate the switch ingress rates[9], but results in a valid DTA-translation and NIC-side stress test with verified functionality.

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 presented in Figure 7. We noticed no performance impact from different report sizes, until we reached the line-rate of $100G$ at batch sizes of $16B$. The results in Figure 7 show this effect for $4B$ queue-depth reports, where we reach line-rate at batches of 4 or greater. 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.

We ran tests that appended data into several lists (up to 255 lists) in parallel, which showed a small ($< 1\%$) performance *improvement* when the number of lists increased. Multi-list tests otherwise followed the same performance patterns as single-list tests.

5.2.1 Append List-Polling Rate. In Figure 8a, 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 collection 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 straightforward process, as shown in Figure 8b, requiring just 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[10].

Ours tests show that the CPU manages to extract every appended list entry even when we use very large batch sizes, given that we dedicate enough cores. We see that the collector can even retrieve list entries faster than the RAM clock speed, which is likely due to cache prefetching greatly improving

[9]The Tofino ASIC is per-design guaranteed to deliver 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. Egress processing is not affected by this choice.

[10]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.

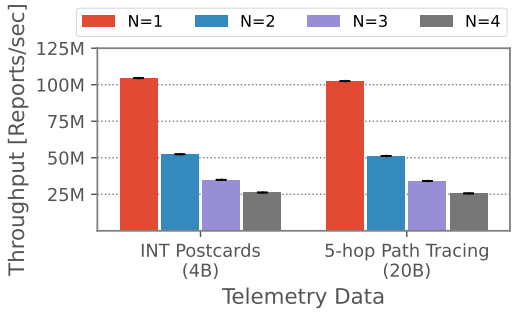
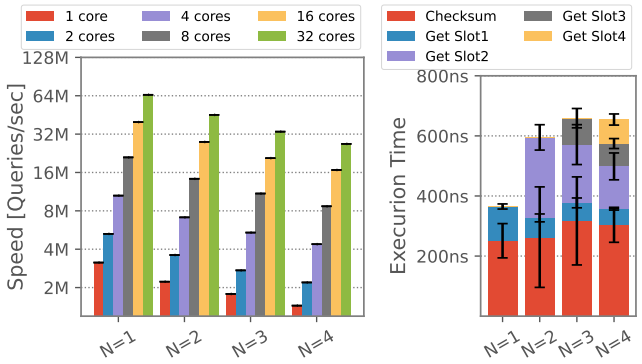


Figure 9: Per-flow path tracing collection, using the DTA Key-Write primitive, either as per-hop postcards (4B) or full 5-hop paths (20B). Note how the telemetry data size does *not* degrade DTA collection performance until we reach $100G$ line-rate at $24B$ payloads



(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.

the performance of sequential memory accesses. (That might also explain the relatively high variation in retrieval latency.)

5.3 Key-Write Primitive Performance

We have benchmarked the collection performance of the DTA Key-Write primitive for path tracing, with a $4GiB$ memory structure using $4B$ concatenated checksums, as presented in Figure 9. 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 $20B$ ($24B$ including the checksum). This, combined with a negligible performance impact from intense background memory operations at the collector, suggests that the RDMA message rate of the NIC is the current bottleneck for Key-Write.

We conclude that a reduction in N would be preferred for optimal performance, and suggest a redundancy level $N = 2$, which we demonstrate to have a high data robustness in Section 5.3.2.

5.3.1 Key-Write Query Speed. The DTA Key-Write primitive requires calculation of several hashes for querying reported data. Here we evaluate the *worst case* performance, when the collector has to retrieve every redundancy slot before a query answer is compiled, even though non-consensus queries often do not require retrieval of every redundancy entry.

We query $100M$ random telemetry keys, 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 at various redundancy levels (N). The returned query answer was either *empty* or *valid*, depending on if an entry with a valid checksum was found in one of the N locations. No query errors were detected during these tests.

Key-Write query processing can be easily parallelized, and we found the query performance to scale near-linearly when we allocated more cores for query processing.

Figure 10b shows a breakdown of the execution time for answering Key-Write queries. Most of the query execution time is spent calculating CRC hashes, either for verifying the concatenated checksum, or to calculate memory addresses of the N redundancy entries. The query performance is therefore highly impacted by the speed of the CRC implementation[11], and more optimized implementations should see a performance increase.

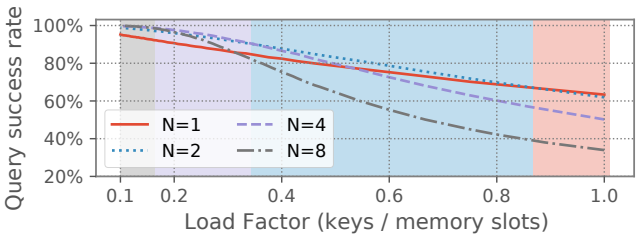


Figure 11: Average query success rates delivered by the Key-Write primitive, depending on the key-value store load factor and the number of addresses per key (N). The background color indicates optimal N in each interval.

5.3.2 Redundancy Effectiveness. JL: remove DART reference The probabilistic nature of DART [4], which is the base for Key-Write, cannot guarantee final queryability on

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

a given reported key. We show in Figure 11 how the query success rate[12] 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, and the background color in Figure 11 indicate which N delivered the highest key-queryability in each interval.

MM: Above seems a bit redundant with 5.3.1 (Query speed) – make sure the two sections work together on Figure 11 discussion. JL: Let me know if this is still an issue

Higher levels of redundancy does indeed improve data longevity, but at the cost of reduced collection and query performance as demonstrated previously in Figures 9 and 10. Determining an optimal redundancy level therefore has to be a balance between an enhanced data queryability and a reduction in primitive performance, and $N = 2$ appears to be a generally good compromise, showing great queryability improvements over $N = 1$.

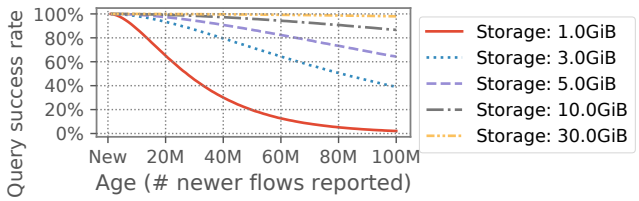


Figure 12: DTA Key-Write ages out stale data. This figure shows INT 5-hop path tracing queryability of 100 million flows at various storage sizes. Note how the data longevity increases linearly when larger key-value storages are allocated at the collector.

5.3.3 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 12 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 raising the storage capacity to 30GB significantly increases the average data queryability to 99.9% across all 100 million flows.

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 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.

5.4 Translator Resource Footprint

DTA translators are responsible for several tailored processes, some of which have a high footprint on Tofino resources. Table 3 shows the total resource usage of our base translator implementation, and the additional costs of including Append batching and per-reporter report-loss detection.

Retransmission-support by tracking per-reporter sequence numbers and adding NACK-generation had 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 had 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 also has the potential for a tenfold increase in collection throughput, and we therefore argue batching is a worthwhile tradeoff. A compromise is to reduce the size of these batches, the size of which linearly correlates with the number of additional meter ALU calls.JL: connect to discussion?

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.

We demonstrate here that DTA translators are already feasible in first-generation Tofino switches, even while including memory-intensive batch-building functionality. It will likely be possible to design even more complex DTA primitives and aggregation functions than those presented here, given that the

[12]The query success rate is defined at the probability at which a previously reported key can still be queried from the key-value storage. Failure to query a key results in an *empty* response, and does not return an incorrect answer in all but incredibly rare cases. Increased data reliability can be achieved through strict query consensus or larger checksums (64-bit and beyond).

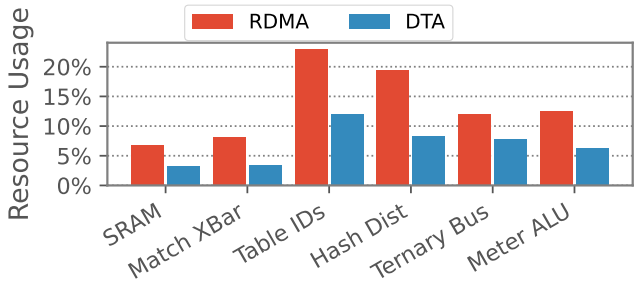


Figure 13: Resource costs of a non-translating Key-Write RDMA reporter, and the resource savings by instead reporting through DTA. The RDMA and DTA pipelines contain idential monitoring backends and change detection algorithms, and only differ in the telemetry reporting logic. These costs omit retransmission-support, where telemetry data is temporarily stored in-ASIC.

current-generation Tofino-2 has the capacity for significantly more memory logic than its predecessor.

5.5 DTA vs RDMA Reporter Costs

Aside from the aforementioned benefits, DTA also reduces the cost of report-generation compared to directly generating RDMA at the reporters. This is due to several reasons, including simplified headers, not tracking RDMA metadata, and offloading primitive processing. Recall how DTA Key-Write also reduces the network telemetry traffic by a factor of N, since DTA only sends a single Key-Write instruction instead of N redundancy packets.

We compare the cost of generating DTA Key-Write reports with DART [4], which is an equivalent system that generates RDMA packets directly from the reporters. Both pipelines employ identical monitoring and change detection components, and only differ in the way that the telemetry reports are generated. The RDMA-generating pipeline supports 1024 collectors, while the DTA-generating pipeline supports 512 translators (we assume 2 collectors per-translator for a fair comparison). Figure 13 shows the footprint of both designs, where we see how DTA significantly reduces the overall resource costs in the switching hardware.

Here we only demonstrate the gain with a single primitive, but note further that DTA-based reporters would only need to modify a few DTA header fields to add support for new primitives, while RDMA-based reporters would need to implement the entire primitive algorithm inside *every reporter* that supports it. The telemetry footprint of several ASIC resources are *halved* compared with RoCEv2 generation, and is resource-wise comparable to generating UDP-based telemetry reports. Removing this cost from all but a few specialized translators significantly increases the scalability of our system compared with non-translating RDMA designs.

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 [52] potentially suffer from reduced performance when the on-NIC ARM core is in charge of processing most of the incoming traffic [17, 36] as would be the case in this scenario. Using FPGA-based cards would solve the problem, but they are also known to be power-hungry, expensive, and difficult to program [17] which limits their potential. Nevertheless, our P4 translator code can be a starting point for P4-capable NICs [60]. 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, we note this may mean that larger amounts of traffic flow to that rack, possibly affecting collocated servers and applications.

Enhance data aggregation at switch. More aggressive data aggregation capabilities could be implemented at the translator provided it is possible to read collector's memory fast enough. This is a challenge when a translator has to deal with a growing number of reporters. Alternatively, the use of partial state-aware aggregation approaches [16] can also be a possibility. Trade-offs in terms of performance and hardware resources costs are beyond the scope of this work and matter of future research. MM: This paragraph above is strange – not clear what the point is. What does implemented at the translator provided it is possible to read collector's memory mean? We haven't discusssed the translator reading collector memory, needs more context.

Data post-processing. As our approach only leverages RDMA writes, the proposed data structures are simple. This simplicity means that reading the data is fast (e.g., comparable to in-memory lookup tables), but the data is only indexed on a single dimension (e.g., the flow ID, or flow+switch pair). As a result, more complex queries may require further processing. Further, primitives like key-write naturally age out, and must be backed up if long-term accountability is desired. We envision that our approach can be supplemented

by post-processing services such as data backup and data warehousing.

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, the CPU must first find out if new data has been written into the memory. To overcome this limitation, 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. JL: I do not agree with this. We assume real-time CPU processing of data with the Append, and therefore don't have to "overcome a limitation". That one is just doing raw pre-categorized transfer to CPU basically

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 have to 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.

Dynamic redundancies. JL: Translators could dynamically adjust key-write redundancy levels. This would allow for increased data queryability in case of storage saturation, and would function as a sort of RDMA-rate controlling mechanism to throttle traffic when nearing capacity. This would be a very light pipeline addition.

7 RELATED WORKS

Telemetry and Collection. Traditional techniques for monitoring the status of the network have looked into periodically collecting telemetry data [19, 21] or mirroring packets at switches [55, 79]. The former generate coarse-grained data that can be significant given the large scale of todays' networks [65]. The latter has been recognised as viable option only if it is known in advance the specific flow to monitor [79]. The rise in programmable switches has enabled fine-grained telemetry techniques that generate a lot more data [7, 18, 20, 62, 77, 79]. 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 [38, 68], or reducing the load through offloaded pre-processing [42] and in-network filtering [28, 41, 69, 77]. 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 [4], which is 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 [24, 62], which unfortunately requires significant investments and infrastructure changes.

RDMA in programmable networks. Recent works have shown that programmable switches can perform RDMA calls [40], 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 [45, 57]. 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 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 [66] or ???.

JL: Might have to fix [n.d.] in references (no date provided)

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] Anonymous. [n. d.]. Anonymized workshop paper. ([n. d.]).

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

[6] 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.

[7] 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.

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

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

[10] 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.

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

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

[13] 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.

[14] 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.

[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] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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*.

[34] 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*.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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*.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

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

[55] 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.

[56] 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)*.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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).

[71] 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.

[72] 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.

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

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.