Direct Telemetry Access

Jonatan Langlet
Queen Mary University of London
j.langlet@qmul.ac.uk

Ran Ben Basat
University College London
r.benbasat@ucl.ac.uk

Gabriele Oliaro
Carnegie Mellon University
goliaro@cs.cmu.edu

Michael Mitzenmacher
Harvard University
michaelm@eecs.harvard.edu

Minlan Yu
Harvard University
minlanyu@g.harvard.edu

Gianni Antichi
Politecnico di Milano
Queen Mary University of London
gianni.antichi@polimi.it

ABSTRACT

Fine-grained network telemetry is becoming a modern datacenter standard and is the basis of essential applications such as congestion control, load balancing, and advanced troubleshooting. As network size increases and telemetry gets more fine-grained, there is a tremendous growth in the amount of data needed to be reported from switches to collectors to enable network-wide view. As a consequence, it is progressively hard to scale data collection systems.

We introduce Direct Telemetry Access (DTA), a solution optimized for aggregating and moving hundreds of millions of reports per second from switches into queryable data structures in collectors' memory. DTA is lightweight and it is able to greatly reduce overheads at collectors. DTA is built on top of RDMA, and we propose novel and expressive reporting primitives to allow easy integration with existing state-of-the-art telemetry mechanisms such as INT or Marple.

We show that DTA significantly improves telemetry collection rates. For example, when used with INT, it can collect and aggregate over 400M reports per second with a single server, improving over the Atomic MultiLog by up to 16x.

CCS CONCEPTS

• **Networks** → **Network monitoring**; *In-network processing*; Programmable networks; Network management;

KEYWORDS

Telemetry Collection,Monitoring,Remote Direct Memory Access

ACM Reference Format:
Jonatan Langlet, Ran Ben Basat, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. 2023. Direct Telemetry Access. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3603269.3604827

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604827

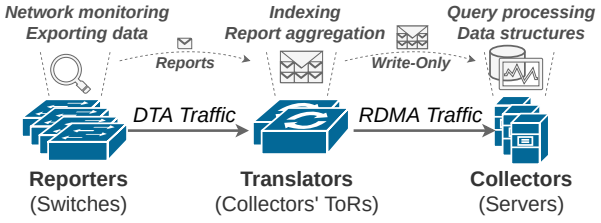


Figure 1: An overview of the telemetry data flow in DTA.

1 INTRODUCTION

In modern data centers, telemetry is the foundation for many network management tasks such as traffic engineering, performance diagnosis, and attack detection [6, 26, 34, 38, 67, 72, 74, 75]. With the rise of programmable switches [8, 31, 54], telemetry systems can now monitor network traffic in real time and at a fine granularity [6, 21, 43, 51, 71, 76]. They are effectively the key enabler to support automated network control [2, 25, 44] and detailed troubleshooting [22, 34, 63]. To provide network-wide views, telemetry systems also aggregate per-switch data into a centralized collector [5, 10, 23, 27, 34, 37, 53], commonly located in a ordinary rack within the datacenter fabric [28, 50].

Unfortunately, as telemetry gets more fine-grained, the amount of data to send to a collector increases and it is progressively harder to scale data collection systems [37, 68, 75]. Indeed, a switch can generate up to millions of telemetry reports per second [51, 75] and a data center network can comprise thousands of them [22]. Also, the amount of data keeps growing with larger networks and higher line rates [60].

Existing research boosts scalability in data collection by improving the collector's network stacks [37, 68], by aggregating and filtering data at switches [32, 40, 51, 69, 75], or by reducing the exported information through switch cooperation [42]. However, as we show, a collector can easily become either CPU- or memory bounded (§2). This is due to the amount of data processing (i.e., I/O, parsing, and data insertion) it is required to perform for every incoming report.

We propose *Direct Telemetry Access* (DTA) — a telemetry collection system (Figure 1) optimized for aggregating and moving hundreds of millions of reports per second from switches into queryable data structures in collectors' memory. In designing DTA, we considered four key goals: (1) relieving a collector's CPU from processing incoming reports while also (2) greatly lowering the number of memory access into it. Those aspects dramatically reduce overheads at the collectors. Furthermore, we wanted (3) to

be compatible with state-of-the-art telemetry reporting solutions (e.g., INT [32], Marple [51]) while (4) imposing minimal hardware resource overheads at switches.

To meet the first goal, we could simply have switches generate RDMA (Remote Direct Memory Access) [30] calls to a collector's memory. RDMA is available on many commodity network cards [33, 64, 70] and can perform hundreds of millions of memory writes per second [64], significantly faster than the most performant CPU-based telemetry collector [37]. Previous work [39] has shown that one can generate RDMA instructions between a switch and a server for network functions. However, it is challenging to adopt RDMA between multiple switches and a collector for telemetry systems as RDMA performance degrades substantially when multiple clients write to the same server [36]. Furthermore, managing RDMA connections at switches is costly in terms of hardware resources and this would conflict with our fourth goal.

We instead developed a solution where the telemetry data exported by switches is encapsulated into our custom and lightweight protocol. This encapsulated data is then intercepted by the last hop switch in front of the collector (generally the Top-of-Rack switch), which we call a DTA *translator*, and converted into standard RDMA calls for the corresponding memory (§3). For the first goal, the CPU avoids processing reports by design as data is inserted into a collector's memory via RDMA. For the second, the translator aggregates and batches reports before invoking RDMA calls and inserts the data in a collector's memory using RDMA-compatible write-only data structures that enable indexing of aggregates without reading from memory, thus reducing memory pressure on a collector's memory. For the third, we designed several switch-level RDMA-extension primitives (*Key-Write*, *Postcarding*, *Append*, and *Key-Increment*), available to reporting switches. These are converted by the translator into standard RDMA calls and allow compatibility with many telemetry systems (§4). Finally, for the fourth, telemetry-reporting switches use our UDP-based protocol to send reports, thus freeing them from the burden of managing RDMA, which is the duty of only the translator.

We implemented DTA using commodity RDMA NICs and programmable switches (§5) and our evaluation (§6) shows that we process and aggregate over 400M INT reports per second, without any CPU involvement, which is 16x faster than the state-of-the-art CPU-based collector for high-speed networks [37]. Further, when the received data can be recorded sequentially, as in the case of temporally ordered event reports, we can ingest up to a billion reports per second, 41x more than state-of-the-art.

Our main contributions are:

- We show that collectors can easily become either CPU- or memory bounded and this greatly limits their ability to process reports and store them in queryable data structures.
- We propose *Direct Telemetry Access*, a novel telemetry collection system generic enough to support major telemetry reporting solutions proposed by the research community (e.g., Marple) or industry (e.g., INT).
- We implement DTA using commodity RDMA NICs and programmable switches.
- We release DTA as open source to foster reproducibility [1].

System	Per-switch Report Rate
INT Postcards (Per-hop latency, 0.5% sampling)	19 Mpps
Marple [51] (Flowlet sizes)	7.2 Mpps
Marple [51] (TCP out-of-sequence)	6.7 Mpps
NetSeer [75] (Loss events)	950 Kpps

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

2 MOTIVATION

Telemetry systems are commonly composed of two main components: (1) switches reporting data and (2) collectors, specialized software installed in dedicated servers located in ordinary racks within the datacenter fabric, that store the reported data [28, 50]. As telemetry systems move to fine-grained real-time analysis with support for network-wide queries, report collection becomes the new key bottleneck [37].

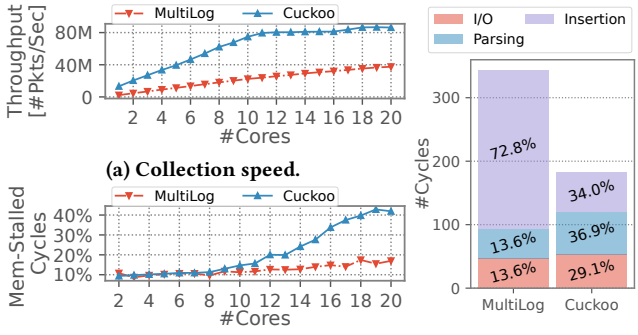
We investigated several state-of-the-art telemetry systems and summarize the reporting rate generated by a single switch in Table 1, based on the numbers available in the corresponding papers.[1] For example, INT Postcard [32] could generate up to $19M$ reports per second when enabled on a commodity 6.4Tbps switch and in the presence of a standard load of ≈40% [73]. Other solutions export less data, either because they pre-process and filter data at switches [23, 75], or because they focus on more specific tasks, thus limiting the data to report [51].

The main takeaway is that state-of-the-art solutions can easily generate *millions of reports per second per switch*. However, to be able to gather a network-wide view at datacenter scale, we may need to collect data from thousands of switches [22] and this requires high-performance collection stacks [37, 68]. For each report from a switch, collectors spend CPU cycles to receive the data (i.e., I/O), parse it (extract content from the report), and to insert it in a queryable data structure for later use (i.e., indexing) [11, 37, 48, 68].

Here, an important trade-off must be taken into account: the more complex the indexing mechanism used for final storage, the more they are suited to efficiently answer different types of queries, but in turn this generally means more CPU cycles spent in inserting data. As an example, consider a simple collector that uses only a hash table to record incoming reports. This solution is good for storing and retrieving counters (e.g., Netflow flow records [13]). However, such a solution might be impractical for essential queries that look, e.g., at a time interval (such as analyzing losses [75], congestion [21], suspicious flows [40] or latency spikes [73] that happens at a certain point in time).

To better understand this trade-off, we have deployed the state-of-the-art DPDK-based telemetry collector allowing storage and diverse queries through an Atomic MultiLog [37] (from now on we refer to it as MultiLog).[2] We used a high-speed server equipped with 2x Intel Xeon Silver 4114 CPUs with 10 cores each clocked to 2.20GHz, and 2x32GB DRAM clocked to 2.67GHz. We compared the performance of this system to a DPDK-based lightweight solution

[1] INT does not advertise a telemetry reporting rate. Thus, we chose an arbitrary sampling rate of 0.5% to keep overheads reasonably low, as an example.
[2] Atomic MultiLog is the basic storage abstraction in Confluo [37], and it is similar in interface to database tables.



(a) Collection speed.

(b) Memory stalling.

(c) Breakdown.

Figure 2: The performance of CPU-based collectors. MultiLog is CPU bounded, while Cuckoo is memory bounded as with 20 cores, 42% of the cycles are spent in waiting for a memory operation to finish.

which employs only a simple cuckoo hash table to store the received information (we refer to it as Cuckoo). We analyzed their behavior when receiving and storing INT reports and found that the MultiLog collector is *CPU bounded*: indeed, its ability to ingest reports grows linearly with its number of cores (Figure 2a). Moreover, the majority of its CPU cycles, around 72.8%, are spent in inserting the data into its internal database (Figure 2c). The main takeaway is that a complex indexing scheme can have a huge toll on the performance of the collector. To put this in perspective, in Figure 3, we show the number of cores that would be needed for a growing size of a datacenter network when employing the MultiLog collector in the presence of switches reporting different information. Here, we can see that for networks comprising around a thousand switches [22], we would need to dedicate nearly $10K$ cores just for collection. For example, in a $K = 28$ fat tree, this would correspond to over 11% of the servers (assuming 16 cores each), and the problem gets worse for smaller networks.

In contrast, the lightweight Cuckoo scheme can ingest more reports per second (Figure 2a) using the same number of cores. However, a new bottleneck arises: in our tests we see that with more than 11 cores it becomes *memory bounded*. For example, with 20 cores, 42% of the cycles are spent waiting for a memory operation to finish (Figure 2b). This is because the high number of reports received impose a huge stress on the memory subsystem, which needs to be read and written to parse the reports, calculate the hashes, and resolve eventual collisions.

Based on the observations, we enumerate our desired goals for a telemetry collection system:

Goals. We wish to have a solution that (1) reduces as much as possible the number of cores required for data collection; (2) lowers the number of memory accesses per report; (3) is compatible with state-of-the-art telemetry reporting systems such as INT and Marple; (4) uses minimal hardware resources to get reports from switches to the collector.

3 DIRECT TELEMETRY ACCESS OVERVIEW

DTA leverages *translators*, which are the last-hop switches adjacent to the collectors. Translators receive telemetry data from *reporters* (i.e., switches exporting telemetry data), encapsulated in our lightweight custom protocol. They then aggregate and batch the reports

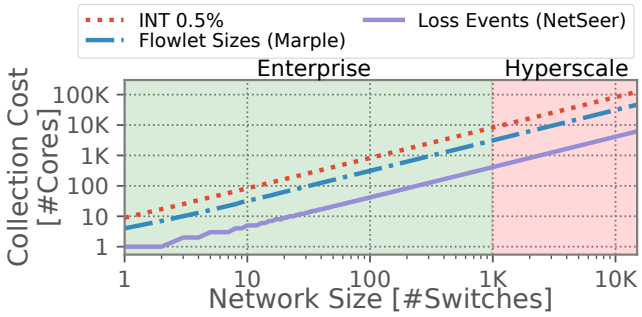


Figure 3: Number of cores needed for single-metric collection with MultiLog at various network sizes.

and use standard RDMA calls to write them directly into queryable data structures in the collectors' memory (Figure 1). In the following, we discuss how, with this architecture, we meet the goals set above.

Meeting goal #1. A strawman solution to meet the first goal could have switches write their reports directly in collectors' memory with RDMA calls. This would zero any CPU requirements at collectors by design. Although this idea appears attractive, and generating RDMA instructions directly from switches is possible [39], it becomes problematic when applied to telemetry collection. Namely, It is inefficient to support multiple RDMA senders writing in the same servers [36]. This is paramount for network telemetry, where multiple switches have to report their data to a collector. Additionally, 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 a collector before performance starts degrading. Alternatively, several switches can share the same queue pair, but RDMA imposes the assumption that every packet received at the collector has a strictly sequential ID, which is impractical for a distributed network of switches. DTA overcomes these challenges by having the translator, which is the last-hop switch before the collector, act as the RDMA writer. Further, by aggregating the reports we can optimize the number of CPU cycles needed for querying as related information is stored contiguously.

Meeting goal #2. We propose two techniques to lower the number of accesses into collectors' memory. First, we aggregate reports at the translator, thereby writing each aggregate using a single write rather than one per report. Second, while telemetry data has to be stored in the collectors' memory in such a way that it is easy to query [37], even simple data structures like hash tables often require an excessive number of memory accesses, e.g., for conflict resolution. Instead, we design RDMA-compatible write-only data structures that enable the indexing of aggregates without reading from memory.

Meeting goal #3. We propose a number of powerful primitives available at the translator that can be used by state-of-the-art telemetry reporting systems (Table 2). The primitives abstract away many common challenges (e.g., deciding where to write data to or how to leverage the small switches' memory) and allow telemetry system designers to seamlessly benefit from our optimizations (e.g., CPU and memory accesses minimization).

Primitive *(Interface)*	**Example monitoring**	**Description**
Key-Write *(key,data)*	INT-MD (Path Tracing) [21, 38]	INT sinks reporting *5x4B* switch IDs using *flow 5-tuple* keys
	Marple (Host counters) [51]	Reporting *4B* counters using *source IP* keys, through non-merging aggregation
	PacketScope (Flow troubleshooting) [65]	Report *fixed-size* per-flow per-switch traversal information using *<switchID,5-tuple>* as key
	PINT (Per-flow queries) [6]	*1B* reports with *5-tuple* keys, using redundancies for data compression through $n = f(pktID)$
	Sonata (Per-query results) [23]	Reporting *fixed-size* network query results using *queryID* keys
Postcarding *(key,hop,data)*	INT-XD/MX (Path Measurements) [21, 38]	Switches report *4B* INT postcards using *(flow 5-tuple, hop)* keys
	Trajectory Sampling (Path Frequencies) [16]	Collection of unique packet labels from all hops for sampled packets
Append *(listID,data)*	dShark (Parser-Grouper transfer) [17]	Parsers append packet summaries to lists hosted by Grouper-servers
	INT (Congestion events) [21, 38]	INT sinks append *4B* reports to a list of network congestion events
	Marple (Lossy connections) [51]	Report *13B* flows to a list with packet loss rate greater than threshold
	NetSeer (Loss events) [75]	Appending *18B* loss event reports into network-wide list of packet losses
	PacketScope (Pipeline-loss insight) [65]	On packet drop: send *14B* pipeline-traversal information to central list of pipeline-loss events
	Sonata (Raw data transfer) [23]	Appending *query-specific* packet tuples from switches to lists at streaming processors
Key-Increment *(key,counter)*	Marple (Host counters) [51]	Reporting *4B* counters using *source IP* keys, through addition-based aggregation
	TurboFlow (Per-flow counters) [61]	Sending *4B* counters from evicted microflow-records for aggregation using *flow key* as keys

Table 2: Existing telemetry monitoring systems, mapped into the primitives proposed by the current iteration of DTA.

Figure 4: DTA supports legacy telemetry systems through encapsulation with new headers.

Meeting goal #4. In DTA, to minimize in-network hardware resources utilization, reporting switches simply use our UDP-based lightweight protocol to send reports to the translator. That way, we alleviate the burden of RDMA generation and aggregation in all switches but the translators. Indeed, the standard RDMA communication protocol (RoCEv2) requires maintaining expensive per-connection metadata and generate appropriate headers and associated checksums.

4 DESIGN

DTA allows easy integration with state-of-the-art telemetry monitoring systems [6, 21, 23, 51] through our four collection primitives that together support a wide range of telemetry solutions: *Key-Write*, *Postcarding*, *Append*, and *Key-Increment*. These primitives provide for placing data in the right place at the collector's memory during reporting time, so as to alleviate as much as possible the cost of query execution.

In Table 2, we show that the primitives are sufficiently generic to support many state-of-the-art telemetry systems. In Figure 4, we show the structure of a DTA report. The telemetry payload exported by a switch, which depends on the specific monitoring system being used, is encapsulated into a UDP packet that carries our custom headers. The *DTA header* (specifying the DTA primitive) and *primitive sub-header* (containing the primitive parameters) are used by the translator to decide what and where to write in the collectors' data structures. This flexibility is essential as the various monitoring systems require writing telemetry in different ways for efficient analysis at the collector. In the following, we discuss our designed primitives. This description assumes that no DTA messages are lost, which could be either through Priority Flow Control (PFC) or a custom flow control solution as discussed later in §7. The primitives themselves would still work even in case of severe in-transit loss of reports, although with degraded probabilistic guarantees which is not accounted for in the following theoretical analysis.

Key-Write (KW). This primitive is designed for key-value pair collection. Storing per-flow data is one scenario where this primitive is useful (additional examples are in Table 2).

Key-value indexing is challenging when the keys come from arbitrary domains (e.g., flow 5-tuples) and we want to map them to a small address space using just write operations. KW provides a probabilistic key-value storage of telemetry data and is designed for resource-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. However, data written to a single memory location is highly susceptible to overwrites due to hash collisions with another key's write. 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 hardware resource overheads of KW by moving the indexing and redundancy generation into the DTA translator. This design choice effectively reduces the 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 (§6.3). Isolating KW logic inside collector-managing translators allows us to entirely remove this resource cost from all other switches.

As analyzed in Appendix A.5, we can derive rigorous bounds on the probability that KW succeeds. There are two possible errors: (i) we fail to output the value for a given key; (ii) we output the wrong value for a given key. Denoting the number of slots by M, the number of pairs written after the queried key by αM, and the checksum length by b bits, we show that the probability of (i) is bounded by:

$$(1 - e^{-\alpha \cdot N})^N \cdot (1 - 2^{-b})^N \tag{1}$$

$$+ (1 - e^{-\alpha \cdot N})^N \cdot (1 - (1 - 2^{-b})^N - N \cdot 2^{-b} \cdot (1 - 2^{-b})^{N-1}) \tag{2}$$

$$+ \Big(\sum_{j=1}^{N-1} \binom{N}{j} \cdot (1 - e^{-\alpha \cdot N})^j \cdot e^{-\alpha \cdot N(N-j)} \cdot (1 - (1 - 2^{-b})^j) \Big). \tag{3}$$

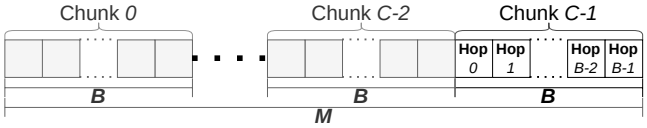


Figure 5: The Postcarding memory structure at the collector.

Here, (1) bounds the probability that all N locations are overwritten with other checksums; (2) bounds the probability that all locations are overwritten and at least two items with our key's checksum write different values; and (3) bounds the probability that not all slots are overwritten, but at least one is overwritten with the query key's checksum. We also bound the probability of giving the wrong output (ii) by

$$(1 - e^{-\alpha \cdot N})^N \cdot N \cdot 2^{-b}. \tag{4}$$

For example, if $N = 2, b = 32, \alpha = 0.1$, the chance of not providing the output is less than 3.3%, while the probability of wrong output is bounded by $1.6 \cdot 10^{-11}$. This aligns with the best effort standard of network telemetry (e.g., INT is often collected using UDP, and packet loss results in missing reports) while having a negligible chance of wrong output. Note that this error is significantly lower than with $N = 1$ (which results in not providing output with probability 9.5%) and higher than for $N = 4$ (probability 1.2%). However, increasing N also has implications to throughput (more RDMA writes) and is not always justified; we elaborate on this tradeoff in §6.5 and show that $N = 2$ is often a good compromise.

DTA also 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 KW header. Higher redundancy means a longer lifetime before being overwritten, as we discuss in §6.5.2. As the level of redundancy used at report-time may not be known while querying, the collector can assume by default a maximum (e.g., 4) redundancy level. If the data was reported using fewer slots, unused slots would appear as overwritten entries (collision).

Postcarding. One of the most popular INT working modes is postcarding (INT-XD/MX [21]), where switches generate *postcards* when processing selected packets and send them to the collector (e.g., for tracing a flow's path.) A report is a collection of one postcard from each hop. Intuitively, while we could use the KW primitive to write all postcards for a given packet, this is potentially inefficient for several reasons. First, each packet can trigger multiple reports that will use multiple RDMA writes even if $N = 1$ (e.g., one per switch ID for path tracing.) In turn, for answering queries with KW (e.g., outputting the switch ID list), the collector will need to make multiple random-access reads, which is slow. Further, adding the KW's checksum to each hop's information is wasteful and degrades the memory-queryability tradeoff. For ease of presentation, we first explain how to reduce the number of writes and later elaborate on how to decrease the width of each slot.

Our observation is that if we know a bound B on the number of hops a packet traverses (e.g., five for fat tree topology), we can improve the above by writing all of a packet's postcards into a consecutive memory block. To that end, we break the M memory locations into chunks of size B, yielding $C = M/B$ chunks. The i'th postcard for a packet/flow ID x is written into $B \cdot h(x) + i$, where h maps identifiers into chunks (i.e., $h(x) \in \{0, \ldots, C-1\}$). This way, the report for all up to B is consecutive in the memory, as shown in Figure 5.

To reduce the number of RDMA writes, we use a mapping from IDs to postcards at the translator. That is, the translator shall cache postcards $0, 1, \ldots, B-1$ before writing the report to the collector's memory using a single RDMA write, once B flow postcards are counted in the translator. Further, answering queries will thus require a single memory random access. As not all packets follow a B hop path, egress switches can provide a packet's path length inside postcards, and translators can use this value to trigger writes before the postcard-counter reaches B. Additionally, reports may be flushed early due to collisions on the switch cache.

Finally, we reduce the number of bits needed for each location, compared with writing the value and checksum to each slot. Intuitively, we leverage the B postcards to amplify the success probability – we output the report only if *all* checksums are valid thereby minimizing the chance of wrong outputs. Intuitively, we use $b > \log_2 |V|$ bits per location to get a collision chance of $\approx (|V| \cdot 2^{-b})$ for each location and $\approx (|V| \cdot 2^{-b})^B$ overall. Here, V is the set of all possible values (e.g., all switch IDs). As noted by PINT [6], $|V|$ is often smaller than 2^{32} (although the INT standard requires that each value is reported using exactly four bytes [21]), allowing us to use small b values.

Let g be a hash function that maps values $v \in V$ into b-bit bitstrings, where b is the desired slot width. We use a "blank" value $\sqcup$ to denote that values for a given hop were not collected (potentially because the path length was shorter than B); this way, each flow always writes all B hops' values, minimizing the chance of false output due to hash collisions. Then, when receiving a postcard value $v_{x,i} \in V$ from the i'th hop of flow/packet ID x, we write $\text{checksum}(x, i) \oplus g(v_{x,i})$ into location $B \cdot h(x) + i$ (here $\text{checksum}(x, i)$ also returns a b-bit result and $\oplus$ is the bitwise-xor operator). When answering queries about x, we check if there exists ℓ such that for all $i \in \{0, \ldots, \ell - 1\}$ there exists a value $v_{x,i} \in V$ for which $\text{checksum}(x, i) \oplus g(v_{x,i})$ is stored in slot $B \cdot h(x) + i$ and for all $i \in \{\ell, \ldots, B-1\}$ $\text{checksum}(x, i) \oplus g(\sqcup)$ is stored. If so, we output that the postcard reports were $v_{x,0}, v_{x,1}, \ldots, v_{x,\ell-1}$. In this case, we say that the chunk contains valid information. We further note that checking the existence of such $v_{x,i}$ can be done in constant time using a pre-populated lookup table that stores all key-value pairs $\{(g(v), v) \mid v \in V \cup \{\sqcup\}\}$.

Our approach generalizes with redundancy $N > 1$: we use N hash functions $h_1, \ldots, h_N$ such that $\text{checksum}(x, i) \oplus g(v_{x,i})$ is written into locations $\{B \cdot h_j(x) + i \mid j \in \{1, \ldots, N\}\}$. For answering queries, we output $v_{x,0}, v_{x,1}, \ldots, v_{x,\ell-1}$ if it appears in a valid subset of the N chunks and all other chunks contain invalid information.

In Appendix A.6, we analyze the primitive and prove that the probability of not providing an output is bounded by:

$$(1 - e^{-\alpha \cdot N})^N \cdot \left(1 - \left((|V| + 1) \cdot 2^{-b}\right)^B\right)^N \tag{5}$$

$$+ (1 - e^{-\alpha \cdot N})^N \cdot \left(1 - \left(1 - \left((|V| + 1) \cdot 2^{-b}\right)^B\right)^N\right.$$

$$\left. - N \cdot \left((|V| + 1) \cdot 2^{-b}\right)^B \cdot \left(1 - \left((|V| + 1) \cdot 2^{-b}\right)^B\right)^{N-1}\right) \tag{6}$$

$$+ \sum_{j=1}^{N-1} \binom{N}{j} \cdot (1 - e^{-\alpha \cdot N})^j \cdot e^{-\alpha \cdot N(N-j)}$$

$$\cdot \left(1 - \left(1 - \left((|V| + 1) \cdot 2^{-b}\right)^B\right)^j\right). \tag{7}$$

We also show that the chance of wrong output is bounded by:

$$(1 - e^{-\alpha \cdot N})^N \cdot N \cdot \left((|V| + 1) \cdot 2^{-b}\right)^B. \quad (8)$$

We consider a numeric example to contrast these results with using KW for each report of a given packet. Specifically, suppose that we are in a large data center ($|V| = 2^{18}$ switches) and want to run path tracing by collecting all (up to $B = 5$) switch IDs using $N = 2$ redundancy. Further, let us set $b = 32$-bit per report and compare it with 64 bits (32 for the key's checksum and 32 bits for the switch ID) used in KW, and that $C \cdot \alpha$ packets' reports were collected after the queried one, for $\alpha = 0.1$. The probability of not outputting a collected report (5-7) is then at most 3.3% and the chance of providing the wrong output (8) is lower than 10^{-22}. In contrast, using KW for postcarding gives a false output probability of $\approx 8 \cdot 10^{-11}$ (in at least one hop) using twice the bit-width per entry!

Append. Some telemetry scenarios are not easily managed with key-value stores. A classic example is when a switch exports a stream of events, where a report would include an event identifier and an associated timestamp (e.g., packet losses [75], congestion events [21], suspicious flows [40], latency spikes [73]). We thus provide a primitive that allows reporters to append information into global lists, with a pre-defined telemetry category in each list.

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.

Key-Increment (KI). This is similar to the KW primitive, but allows for addition-based data aggregation. That is, the KI 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 [51, 61]. The KI primitive can then deliver collection of these evicted counters at RDMA rates. As with KW, the translator reduces network overheads compared with a more naive design.

Our KI memory acts as a Count-Min Sketch [14] and we increment N values using the RDMA Fetch-and-Add primitive. On a query, KI returns the minimum value from these N locations. Hash collisions may lead to an overestimate of the value, with error guarantees matching those of Count-Min Sketches [14]. The counters' memory may be reset periodically, depending on the application.

Extensibility. DTA is easily extensible to other primitives by the addition of new translation paths at translators, although they would remain constrained by the limitations imposed by commodity programmable switches [47]. Some of these limitations could be overcome by implementing the translator logic into FPGA-based smartNICs (see §7). For example, one could extend DTA to support collection of sketch-based measurements. This could allow for either in-network discovery of network-wide heavy hitters, or aggregation of counters at the translator to decrease the collection load at compute servers. Additionally, the translator does not have to be a semi-passive data aggregator as presented here, and primitives could be designed to be more active. For example, one could use techniques similar to the ones presented by Gao et al. [18] to derive the network state directly at the translator based on the intercepted telemetry reports, thereby offloading even parts of analysis from the telemetry collectors.

5 IMPLEMENTATION

Our codebase includes approximately $5K$ lines of code divided between the logic for the DTA reporter (§5.1), the translator (§5.2), and collector RDMA service (§5.3). The hardware resource footprints are presented later in Sections §6.3 and §6.4. DTA is released in open-source [1].

5.1 DTA Reporter

The reporter takes $\approx$ 700 lines of P4_16 for the Tofino ASIC. Controller functionality is written in $\approx$ 100 Python lines, and is responsible for populating forwarding tables and inserting collector IP addresses for the 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.

5.2 DTA Translator

The translator has a control program written in 800 lines of Python that runs on the switch CPU. 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.

The translator pipeline (Figure 6) is written in $2K$ lines of P4_16 for the Tofino ASIC. This pipeline includes support for internal processing of the DTA primitives, RDMA generation, basic user-traffic forwarding, as well as RDMA queue-pair resynchronization and rate limiting to ensure stable RDMA connections in case of congestion events at the collectors' NICs. Rate limiting can be configured to generate a NACK sent back to the reporter in case of a dropped report during these congestion events.

The RDMA logic is shared by all primitives. This includes lookup tables filled with RDMA metadata, SRAM storage for the queue pair packet sequence numbers, and the task of crafting RoCEv2 headers. The DTA packets themselves are used as the base for RDMA generation. This is done by completely substituting the DTA headers with the specific RoCEv2 headers required by the DTA operation. The redundancy in Key-Write, Key-Increment, and Postcarding is generated by the packet replication engine through multicasting (*Multicaster* in Figure 6). The switch CPU crafts specific multicast rules to force the ASIC to emit several packets at the correct egress port as triggered by a single DTA ingress.

Key-Write and **Key-Increment** both follow the same fundamental logic, with the main 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

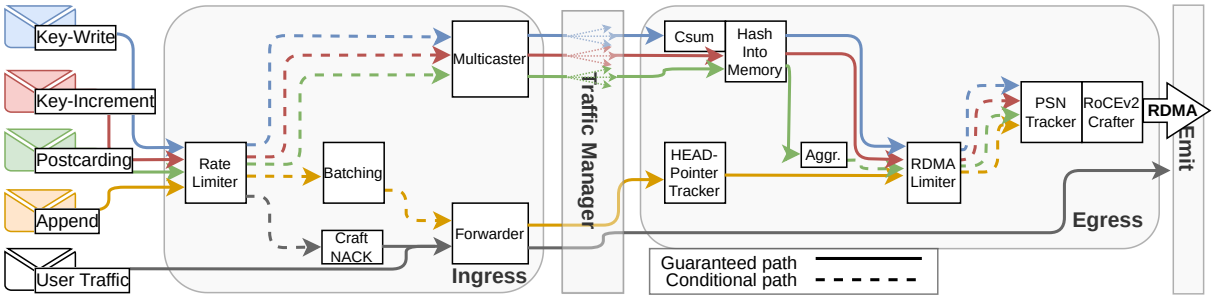


Figure 6: A translator pipeline with support for Key-Write, Key-Increment, Postcarding, and Append. Five paths exist for pipeline traversal, used to process different types of network traffic in parallel while efficiently sharing pipeline logic.

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. Carefully selected CRC polynomials are used to create several independent hash functions using the same underlying CRC engine.

Postcarding uses an SRAM-based hash table with 32K slots storing fixed-size 32-bit payloads. The Tofino-native CRC engine is used for indexing and value encoding. The hop-specific checksums are implemented through custom CRC polynomials instead. Emissions are triggered either by a collision or when a row counter reaches the path length. We note that an efficient implementation requires the RDMA payload sizes to be powers of 2 (due to bitshift-based multiplication during address calculation) and the chunk sizes are therefore padded from $5{*}4B = 20B$ to $32B$, trading storage efficiency for a reduced switch footprint.

Append has its logic split between ingress and egress, 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. Our prototype supports tracking up to $131K$ simultaneous lists.

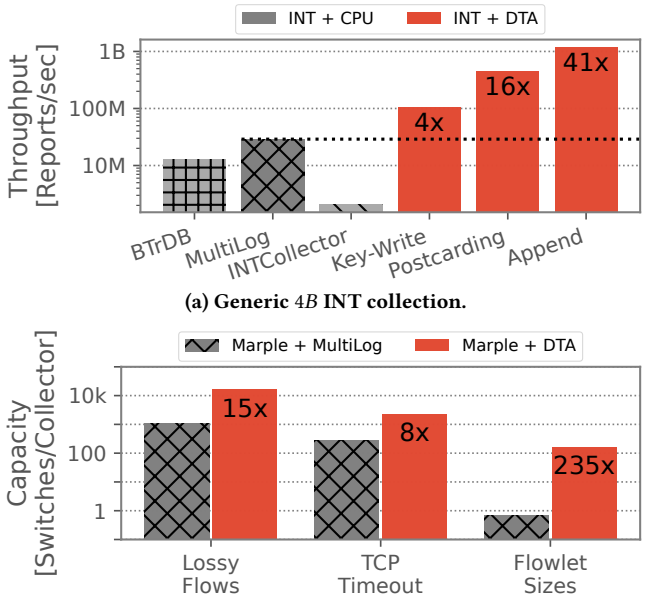
5.3 Collector RDMA Service

The collector is written in $1.3K$ lines of C++ using standard Infiniband RDMA libraries, and has support for 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.

6 EVALUATION

In this section, we show that:

- DTA supports very high collection rates (§6.1).
- DTA imposes a negligible memory pressure at collectors (§6.2).
- DTA is lightweight (§6.3, §6.4).
- DTA's primitives are fast (§6.5, §6.6, §6.7).



(a) Generic $4B$ INT collection.

(b) Diverse Marple queries.

Figure 7: A performance comparison of DTA against state-of-the-art CPU-collectors. These use 16 cores for data ingestion, while DTA essentially bypasses the CPU entirely for data ingestion by using RDMA. (b) MultiLog vs DTA when using Marple as a monitoring system running on switches.

We use two x86 servers connected through a BF2556X-1T [52] Tofino 1 [31] switch with 100G links. Both servers mount 2x Intel Xeon Silver 4114 CPUs, 2x32GB DDR4 RAM @ 2.6GHz, and run Ubuntu 20.04 (kernel 5.4). One of them serves as a DTA report generator using TRex [12]. The other, equipped with an RDMA-enabled Mellanox Bluefield-2 DPU [55], serves as the collector. Here, server BIOS has been optimized for high-throughput RDMA [35], and all RDMA-registered memory is allocated on 1GB huge pages.

6.1 DTA in Action

We first investigate if DTA scales better than CPU-based collectors in the presence of telemetry volumes generated by large-scale networks. To do so, we compare the performance of DTA and state-of-the-art CPU-collectors when coupled with two different monitoring

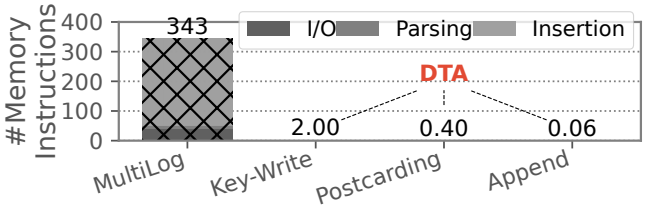


Figure 8: Average number of memory instructions per-report for ingestion of INT postcards.

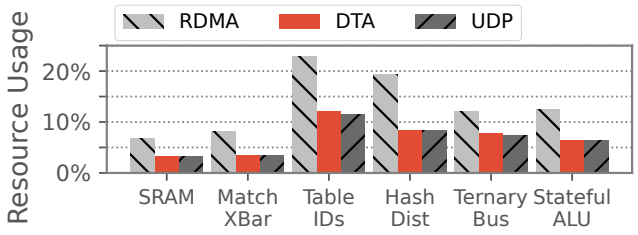


Figure 9: Hardware resource costs of a DTA Reporter compared to an RDMA-generating reporter, and a baseline UDP-based reporter. Note how DTA imposes an almost identical resource footprint to UDP.

	SRAM	Match Crossbar	Table IDs	Ternary Bus	Stateful ALU
Base footprint	13.2%	10.6%	49.0%	30.7%	25.0%
Batching	+3.2%	+7.2%	+7.8%	+7.8%	+31.3%

Table 3: Resource footprint of a translator in Tofino while supporting Key-Write, Postcarding, and Append. Append is batching 16x4B reports.

systems: INT [21, 38] and Marple [51]. Here, we use a DTA configuration with $N = 1$ and batching of size 16, while CPU-collectors use 16 dedicated CPU cores in the same NUMA-node.

The collectors in Figure 7a collect generic 4*B* INT reports that are available for offline queries using the flow 5-tuple as the key. We test INTCollector [68], to the best of our knowledge the only open source INT collector that uses InfluxDB for storage. We also study BTrDB [4], and the state-of-the-art solution for high-speed networks, Confluo, which is based on MultiLog technology.Key-Write inserts each report into its key-value store, and Postcarding assumes 5-hop aggregation with no intermediate reports. Append instead inserts the reports into one of the available data lists. As Figure 7a shows, DTA improves on key-based INT collection by at least 4x, or up to 16x when aggregating the postcards into 5-hop tuples, with even higher performance gains if pre-categorized and chronological storage through Append suffices.

We also integrated Marple with DTA and MultiLog and configured them to support the same queries against the collected data (i.e., Lossy Flows, TCP Timeout, and Flowlet Sizes). Here, *Lossy Flows* reports high loss rates together with their corresponding flow 5-tuples, and DTA uses the Append primitive to store the data chronologically in several lists, allowing operators to retrieve the most recently reported network flows with packet loss rates in one of several ranges. *TCP Timeouts* reports the number of TCP timeouts per-flow in recent time, and DTA uses the Key-Write primitive to allow operators to query the number of timeouts experienced by any arbitrary flow. *Flowlet Sizes* reports flow 5-tuples together with the number of packets in their most recent flowlets, and DTA appends the flow identifiers to one of the available lists to allow the construction of per-flow histograms of flowlet sizes.

We experimented using real data center traffic [7] and found that DTA increases the number of Marple reporters (i.e., network switches) that a collector can support before the rate of data generation overwhelms the collector (Figure 7b). Their queries cost as well as their performances are analyzed in later sections (§6.5, §6.7).

Takeaway: DTA improves on data collection speeds compared with CPU-based collectors by *one to two orders of magnitude* when integrated with state-of-the-art telemetry systems, while supporting the same types of queries.

6.2 Reduced Memory Pressure

In Figure 8, we present the average number of memory instructions required per-report for the DTA primitives, when configured with a redundancy level of 2, path length of 5 hops, and batch size of 16 elements. DTA imposes a low pressure on memory. This is achieved mostly because no accesses are needed for I/O and report parsing, regardless of the indexing scheme used. Some DTA primitives use less than a single memory instruction per report on average, owing to its aggregation and batching techniques, which can intelligently insert several reports simultaneously with a single RDMA operation. For example, Key-Write, the primitive that imposes the heaviest load on memory, needs just 0.58% as many accesses as MultiLog.

Takeaway: DTA *significantly* reduces the number of memory accesses required for report ingestion.

6.3 Reporter Resource Footprint of DTA

We compared the hardware costs associated with generating DTA reports against either directly emitting RDMA calls from switches, or creating UDP-based messages as generally done by CPU-based collectors. We used a switch implementing a simple INT-XD system and, in Figure 9, we show the cost associated with the change of its report-generation mechanism. Here, we see that DTA is as lightweight as UDP, while RDMA generation is much more expensive.

Takeaway: DTA halves the resource footprint of reporters compared with RDMA-generating alternatives, and has a *similar resource footprint to simple UDP generation.*

6.4 Translator Resource Footprint

Table 3 shows the resource usage of the translator, alongside the additional costs of including Append batching.The footprint of the DTA translator is mainly due to its concurrent built-in support for several different primitives. Application-dependent operators might reduce their hardware costs by enabling fewer primitives.

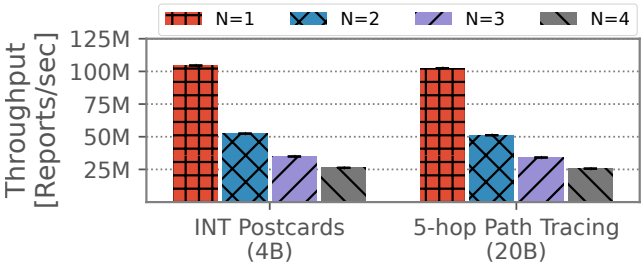
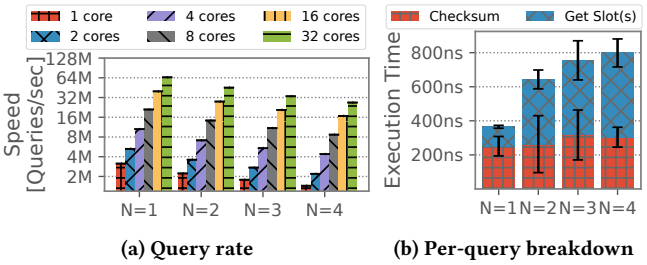


Figure 10: Per-flow path tracing collection rates, using the DTA Key-Write primitive, either as INT-XD/MX postcards (4B) or full 5-hops path as in INT-MD (20B).



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

Figure 11: Key-Write primitive querying performance.

Batching of Append data has a relatively high cost in terms of memory logic (Stateful ALU), due to our non-recirculating RDMA-generating pipeline requiring access to all $B-1$ entries during a single pipeline traversal. It is worth noting that batching also has the potential for a tenfold increase in collection throughput, and we conclude that it is a worthwhile tradeoff. A compromise is to reduce the batch sizes, as they linearly correlate with the number of additional stateful 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, and tests show that the translator can support hundreds of thousands of simultaneous lists for complex setups, which is much more than the 255 lists included at the time of evaluation.

Takeaway: A translator pipeline which simultaneously supports the Key-Write, Postcarding, and Append primitives fits in first-generation programmable switches, while *leaving a majority of resources freed up for other functionality*. Batching can impose a high toll on the Stateful ALUs.

6.5 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 receive either $4B$ or $20B$ encapsulated INT messages from the reporter (our traffic generator). The former case emulates the scenario of having INT working in postcard mode with event detection (so some hops may not generate a postcard), 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 10. Notice the expected linear relationship between the throughput and level of redundancy since each incoming report will generate N RDMA packets towards the collector. However, one might still prefer the performance tradeoff against the increased data robustness in the collector storage, which allows for successful queries against much older telemetry reports. 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 of $16B$ or larger.

Takeaway: Key-Write can collect $100M$ *INT reports per second* and its performance depends on the redundancy level.

6.5.1 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 11a 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. For example, 4 cores could query 7.1 million flow paths per second with $N=2$, while 8 cores manage 14.2 million queries per second.

Figure 11b shows the time breakdown serving queries. Most of the execution time is spent 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[3], 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.

6.5.2 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 12 how the query success rate[4] 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

[3]We use the generic Boost libraries' CRC: https://www.boost.org/.
[4]The query success rate is defined as the probability at which a previously reported key can be queried from the key-value store.

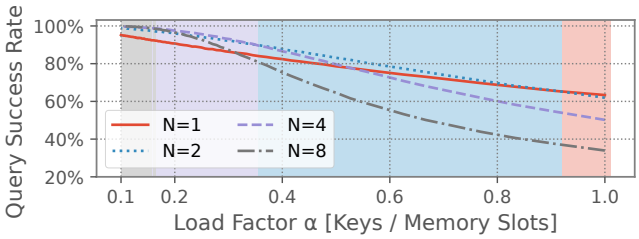


Figure 12: Average query success rates delivered by the Key-Write primitive, depending on the key-value store load factor and the number of addresses per key (N). The background color indicates optimal N in each interval.

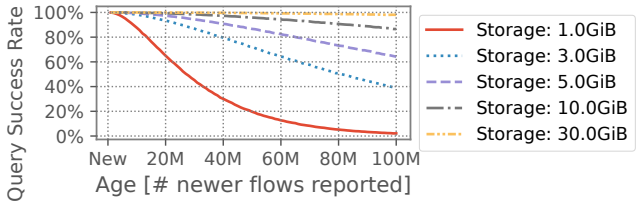


Figure 13: DTA Key-Write ages out eventually. This figure shows INT 5-hop path tracing queryability of 100 million flows at various storage sizes.

because it is harder to reach consensus at query time. The background color in Figure 12 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 10 and 11. 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.

6.5.3 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 13 shows the queryability of randomly reported INT 5-hop path tracing data (i.e., $20B$) at various storage sizes and report ages, with redundancy level $N = 2$ and $4B$ checksums. For example, a key-value storage as small as 3GiB is enough to deliver 99.3% successful queries against flows with as many as 10 million subsequently reported paths, which however falls to 44.5% when 100 million subsequent flow are stored in the structure. However, increasing storage to 30GiB would allow an impressive 99.99% query success rate for paths with 10 million subsequent reports, or 98.2% success even for flows as old as 100 million subsequent reports.

Takeaway: It is possible to record data from around 10M flows in the key-value store while maintaining a 99.99% queryability with just 30GiB of storage.

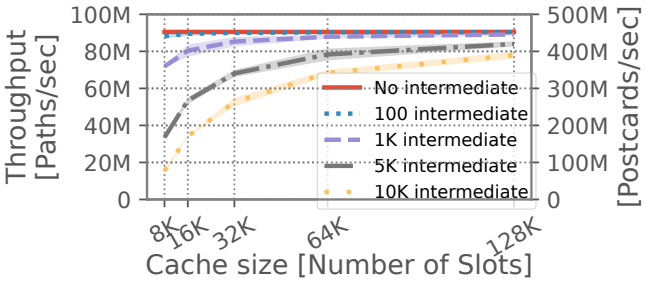


Figure 14: INT-XD/MX postcard collection, using the DTA Postcarding primitive. A report is defined as a successfully aggregated 5-hop path (containing 5 postcards, one per hop).

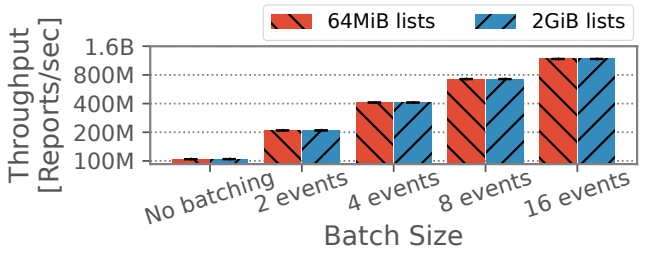


Figure 15: Telemery event-report collection, using DTA Append and different batch sizes. Performance increases linearly with batch sizes until we achieve line-rate with batches of $4x4B$. The collection speed is not impacted by the list sizes.

6.6 Postcarding Primitive Performance

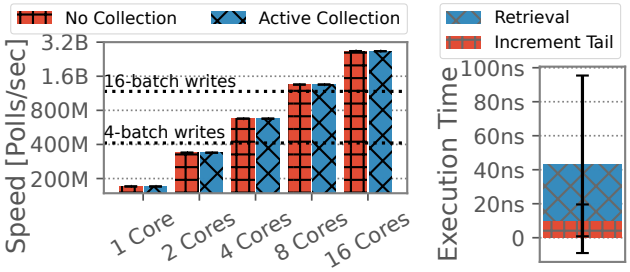
The Postcarding primitive has been benchmarked for aggregating and collecting INT-XD/MX postcards across 5-hop network paths. The number of other flows appearing at the translator while aggregating per-flow postcards increases the risk of premature cache emission. Figure 14 shows us the effect that the number of intermediate flows and the size of the cache has on the aggregation performance, with a maximum achieved collection rate of $90.5MPaths/s$ ($452.5MPostcards/s$).[5] Comparing the performance to Key-Write in Figure 10, where we would need 5 different reports to collect a full path, we see a significant performance gain by the Postcarding primitive.

Takeaway: The performance of Postcarding depends on the rate of cache collisions in the translator during the aggregation-phase, and can *improve upon the best-case Key-Write performance by up to* $4.3x$ for 5-hop collection.

6.7 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 15.

[5]Early emissions (i.e., path-reports with missing postcards) are counted as failures in this test despite being potentially useful (e.g., knowing 4 out of 5 hops in a path), and are not included in the collection throughput.



(a) Polling rate. **(b) Poll breakdown.**

Figure 16: Append primitive querying performance. Append-lists are queried either while collecting no reports or at 50% capacity (while collecting *600M* 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.

We noticed no performance impact from different report sizes, until we reached the line-rate of 100*G* for large batch sizes after which the performance increased sub-linearly. The results in Figure 15 show this effect for 4*B* 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 131*K* parallel lists showed a negligible performance impact.

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

6.7.1 Append List-Polling Rate. Figure 16a shows the raw list polling rates, which is the speed at which appended data can be read into the CPU for processing. We assume that collection runs 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 approximates collection at half capacity. Simultaneously collecting and processing telemetry data show no noticeable impact on either collection or processing, showing that DTA is not memory-bounded even at this speed[6].

Extracting telemetry data from the lists is a very lightweight process, as shown in Figure 16b, 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.Our tests showed that 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.

Takeaway: The CPU retrieves appended reports *faster than they can be collected* (Figure 15), with margin left for further processing.

[6]DTA is neither memory- nor CPU bounded in these tests, regardless of the collection rate, but is instead limited by the message rate of the network card

7 DISCUSSION

The generality and scope of DTA. DTA is not intended to be a competitor of existing data plane assisted monitoring systems [6, 21, 23, 38, 51, 61, 65, 65, 75]. These either focus on extracting new metrics or reducing the costs of telemetry monitoring through intelligent pre-processing and filtering within the switching ASIC. Nevertheless, these systems generate a significant amount of telemetry information, especially with large-scale networks, multiple queries, and/or fine telemetry granularities (Table 1).

DTA can be coupled with existing telemetry systems and serve as an interface between the on-switch monitoring functions and the telemetry analysis back-end in the control plane. To achieve broad compatibility with a variety of monitoring solutions, we have designed several generic and highly flexible primitives to simplify the integration of DTA into both existing and future telemetry environments. As a consequence, with DTA, we replace only the report ingestion mechanism of the telemetry collector (e.g., DPDK along with data structure population), not the rest of the collector (e.g., data analysis and decision-making). For example, it is possible to couple the streaming analysis engine of Sonata [23] with DTA: in this scenario our solution is in charge of transferring data from switches to collector's memory, while the original Sonata's engine performs analysis on the received data. For a more extensive list of examples, we refer to Table 2 that recap how DTA can be integrated into various telemetry systems to enhance their performances.

Implementing the translator in a SmartNIC. There are two main approaches we have considered on where to deploy the translator: a SmartNIC located at the collector and the last-hop programmable switch (which we explored in this work). A SmartNIC would allow us to completely remove RDMA traffic: the NIC data-plane would process incoming DTA packets and translate them into local DMA calls. Exploring DTA translation in SmartNICs is left for future work. Nevertheless, we believe that our P4 implementation can be a starting point for P4-capable NICs [62].

Supporting Multiple Collectors. It is beneficial to enable collection at multiple servers for scalability or resiliency. DTA can be deployed alongside multiple collectors and permit easy partitioning of reports based on the IP and DTA headers.

Flow Control in DTA. Best-effort transport protocols, e.g., UDP, are used by many well-known telemetry systems (e.g., [13, 32]). Similarly, DTA does not assure reliable delivery. However, it can be used in conjunction with flow control mechanisms that allow for lossless delivery of data [20, 29].

Query-Enhancing Extensions. In some cases, queries may be known ahead of time, in which case our translator can aid in their processing.For example, while switches can measure the queuing latency of a flow, we are often interested in knowing the end to end delay [58], which can be expressed as follows:

```
SELECT flowID,path WHERE SUM(latency) > T
```

Knowing the query ahead of time, our translator can wait for postcards from all switches through which the SYN packet of the flow was routed, sum their latency, and report it if it is over the threshold.

Push notifications. An advantage CPU-based collectors have over DTA is that the CPU can trigger 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 inform the CPU that new data has arrived through RDMA immediate interrupts (e.g., a flow is experiencing problems). Deciding which reports should carry such a flag is beyond the scope of this work.

The next telemetry bottleneck. DTA significantly reduces the cost of telemetry ingestion mainly by bypassing any CPU processing. In our experiments the new bottleneck is the message rate of the RDMA NICs at the collectors. To address this message rate limitation, DTA already supports multi-NIC collectors. Future NICs will have better speeds.

A possible future bottleneck is the memory speed where we store the telemetry data structures. However, current-generation DRAM can achieve billions of memory transfers per second and is likely to increase further in the future. Therefore, it is possible that telemetry ingestion itself might no longer be seen as the main bottleneck in telemetry systems going forward, if the CPU is bypassed. Instead, given the increasing sophistication and complexity of data analysis tools, the de-facto bottleneck might instead be the rate at which we can still meaningfully analyze the generated data in real time.

8 RELATED WORK

Telemetry and Collection. Traditional techniques for monitoring the status of the network have looked into periodically collecting telemetry data [22, 24] or mirroring packets at switches [56, 76]. The former generates coarse-grained data that can be significant given the large scale of today's networks [66]. The latter has been recognized as viable option only if it is known in advance the specific flow to monitor [76]. The rise in programmable switches has enabled fine-grained telemetry techniques that generate a lot more data [6, 21, 23, 63, 75, 76]. 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 [37, 68], or reducing the load through offloaded preprocessing [42] and in-network filtering [32, 40, 69, 75]. In an earlier version of our project, we investigated the possibility of entirely bypassing collectors' CPU, but limited the collection process to data that can be represented as a key-value store [41]. DTA expands it significantly by introducing the translator, designing additional primitives, building a prototype, investigating the systems aspects, and showing an end-to-end improvement over state-of-the-art collection systems. In particular, this paper proposes an alternative solution which is generic and works with a number of existing state-of-the-art monitoring systems. We show examples of where these aforementioned systems can integrate DTA earlier in Table 2. A further alternative approach is letting the end-hosts assist in network-wide telemetry [26, 63], which unfortunately requires significant investments and infrastructure changes and still lean on centralized collection to achieve a network-wide view.

RDMA in programmable networks. Recent works have shown that programmable switches can perform RDMA operations to access server DRAM for expanded memory in their stateful network functions [39, 57]. These works are interesting for these scenarios, but are not suited for the queryable aggregation required for telemetry collection. Programmable network cards are also shown capable of expanding upon RDMA with new and customized primitives [3]. Especially FPGA network cards show great promise for high-speed custom RDMA verbs [46, 59]. However, as discussed in Section 2, telemetry collection brings new challenges when used in conjunction with the RoCEv2 protocol. As previously mentioned earlier in discussion, a protocol such as DTA that is tailored for telemetry collection could very well be implemented at the NIC-level.

9 CONCLUSION

We presented Direct Telemetry Access (DTA), a new telemetry collection system optimized for storing reports from switches to collectors' memory. We built DTA on top of RDMA and provided novel and expressive primitives that allow easy integration with existing telemetry solutions.

DTA can write to our key-value store over 400M INT reports per second, without any CPU processing, 16x better than the state-of-the-art collector. When the received data can be recorded sequentially, as in the case of temporally ordered event reports, it can ingest up to a billion reports per second, a 41x improvement over the state-of-the-art.

This work does not raise any ethical issues.

ACKNOWLEDGEMENTS

We thank our shepherd Kate Lin, and the anonymous reviewers, for valuable comments and feedback. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, by the UK EPSRC project EP/T007206/1, by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART"), and by a gift from Facebook/Meta. Michael Mitzenmacher was supported in part by NSF grants CCF-2101140, CNS-2107078, and DMS-2023528. Finally, a big thanks to Sivaram Ramanathan for invaluable input in the early stages of the project.

REFERENCES

[1] 2023. Direct Telemetry Access source code. https://github.com/jonlanglet/DTA. (2023).
[2] 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.
[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] Arista. 2022. Telemetry and Analytics. https://www.arista.com/en/solutions/telemetry-analytics. (2022). Accessed: 2022-02-02.
[6] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-band Network Telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 662–680.
[7] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Conference on Internet Measurement (IMC)*. ACM.

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

[9] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

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

[11] Cisco. 2021. How to scale IOS-XR Telemetry with InfluxDB . https://community.cisco.com/t5/service-providers-knowledge-base/how-to-scale-ios-xr-telemetry-with-influxdb/ta-p/4442024. (2021).

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

[13] Cisco. 2023. Cisco IOS NetFlow. https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html. (2023). Accessed: 2023-02-08.

[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] Nick G Duffield and Matthias Grossglauser. 2001. Trajectory sampling for direct traffic observation. *IEEE/ACM transactions on networking* 9, 3 (2001), 280–292.

[17] Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. 2019. dShark: A general, easy to program and scalable framework for analyzing in-network packet traces. (2019).

[18] 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.

[19] Michael T Goodrich and Michael Mitzenmacher. 2011. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 792–799.

[20] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. 2022. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 779–805.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

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

[28] Huawei. 2021. Telemetry. https://support.huawei.com/enterprise/en/doc/EDOC1100196389. (2021).

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Intel. 2022. Performance Tuning for Mellanox Adapters. https://support.mellanox.com/s/article/performance-tuning-for-mellanox-adapters. (2022).

[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] 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.

[38] 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*.

[39] 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.

[40] 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.

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

[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] Richard J Lipton. 1994. A new approach to information theory. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 699–708.

[46] 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.

[47] 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.

[48] Microsoft. 2013. Cloud Service Fundamentals: Telemetry - Reporting. https://azure.microsoft.com/sv-se/blog/cloud-service-fundamentals-telemetry-reporting/. (2013).

[49] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press.

[50] Tal Mizrahi, Vitaly Vovnoboy, Moti Nisim, Gidi Navon, and Amos Soffer. 2018. Network telemetry solutions for data center and enterprise networks. *Marvell, White Paper* (2018).

[51] 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.

[52] 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.

[53] 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.

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

[55] 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.

[56] 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.

[57] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. 2023. A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.

[58] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2022. Continuous In-Network Round-Trip Time Monitoring. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 473–485.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[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] 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.

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.

[76] 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.

Appendices are supporting material that has not been peer-reviewed.

A APPENDIX

A.1 Key-Write Algorithm

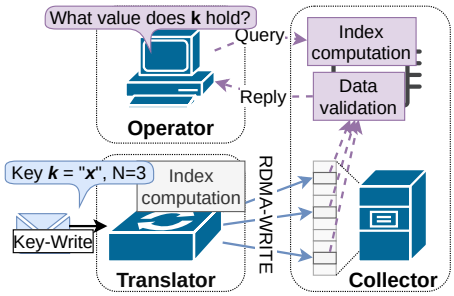


Figure 17: Key-Write Overview.

The Key-Write primitive is an abstraction around a key-value store, allowing read/writes of telemetry data. Figure 17 is a high-level visualization of the primitive, and algorithm pseudo-code is presented in Algorithm 1 and 2.

Algorithm 1: DTA-to-RDMA translation in Key-Write

Input: Redundancy N, Key K, Telemetry data D
$Bufstart \leftarrow$ Address to start of RDMA memory buffer
$Buflen \leftarrow$ Number of allocated KeyVal slots
$Slotlen \leftarrow$ Size of one KeyVal slot
Function `CraftWrite(`n, K, D`)`
  $Slot \leftarrow h_0(n, K) \bmod Buflen$
  $Dest \leftarrow Bufstart + Slot \times Slotlen$
  $Csum \leftarrow h_1(K)$
  Write $(Csum, D)$ to address $Dest$ through RDMA
for $n = 0$ **to** N **do**
  `CraftWrite(`n, K, D`)`

Algorithm 2: Querying the Key-Write storage

Input : Redundancy N, Key K, Consensus threshold T
Output: D_{winner}
$Buflen \leftarrow$ Number of allocated KeyVal slots
$Storage \leftarrow$ Array size $Buflen$ with $\langle Csum, D \rangle$ elements
Function `GetSlot(`n, K`)`
  $Slot \leftarrow h_0(n, K) \bmod Buflen$
  return $Storage[Slot]$
$Csum \leftarrow h_1(K)$
for $n = 0$ **to** N **do**
  $(Csum_{slot}, D) \leftarrow$ `GetSlot(`n, K`)`
  if $Csum == Csum_{slot}$ **then**
    Add D to list of candidates
$D_{winner} \leftarrow$ candidate D if D appears at least T times

A.2 Postcarding Algorithm

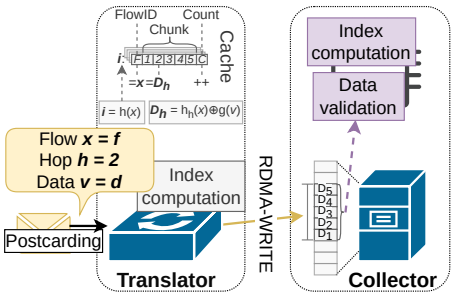


Figure 18: Postcarding Overview.

The Postcarding primitive is an abstraction around a key-value store with per-flow aggregation of INT postcards, allowing read-/writes of telemetry data. Figure 18 is a high-level visualization of the primitive. We refer to Section 4 for details on primitive translation and querying, as well as Appendix A.6 for analysis.

A.3 Append Algorithm

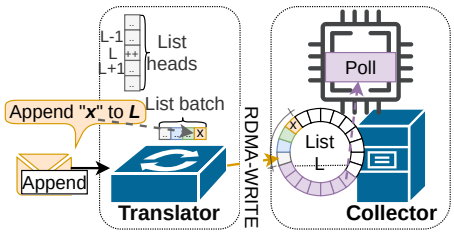


Figure 19: Append Overview.

The Append primitive is an abstraction around data lists, allowing read/insertion of telemetry data. Figure 19 is a high-level visualization of the primitive, and algorithm pseudo-code is presented in Algorithm 3 and 4.

Algorithm 3: DTA-to-RDMA translation in Append

Input : List ID L, Data D
$ListBuffers \leftarrow$ Vector with $|Lists|$ buffer pointers
$BufferLengths \leftarrow$ Vector with $|Lists|$ buffer lengths
$Heads \leftarrow$ Vector with $|Lists|$ head-offsets
$BatchSize \leftarrow$ The global batch size
$BatchPointer \leftarrow$ Vector with $|Lists|$ integers
$Batches \leftarrow$ 2D-vector sized $[|Lists|][BatchSize - 1]$
Function `WriteBatch(L,D)`
  $Batch \leftarrow (Batches[L], D)$
  $Address \leftarrow ListBuffers[L] + Heads[L]$
  Write $Batch$ to address $Address$ through RDMA
  $Heads[L]$ += $BatchSize$
  if $Heads[L] == BatchSize$ **then**
    $Heads[L] \leftarrow 0$
if $BatchPointer[L] == BatchSize$ **then**
  `WriteBatch(L,D)`
  $BatchPointer[L] \leftarrow 0$
else
  $Batches[L][BatchPointer[L]] \leftarrow D$
  $BatchPointer[L]$++

Algorithm 4: Querying the Append storage

Input : List ID L
Output : $data$
$ListBuffers \leftarrow$ Vector with $|Lists|$ buffer pointers
$BufferLengths \leftarrow$ Vector with $|Lists|$ buffer lengths
$Heads \leftarrow$ Vector with $|Lists|$ head-offsets
$data \leftarrow ListBuffers[L] + Heads[L]$
$Heads[L] \leftarrow (Heads[L] + 1) \bmod BufferLengths[L]$

A.4 Key-Increment Algorithm

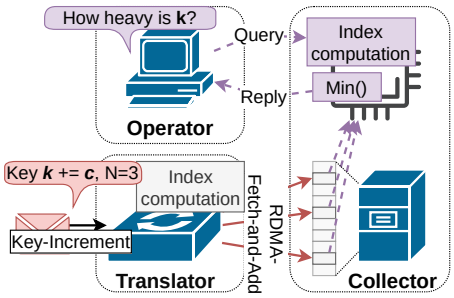


Figure 20: Key-Increment Overview.

The Key-Increment primitive is an abstraction around a key-value store, allowing read/increment of counters. Figure 20 is a high-level visualization of the primitive, and algorithm pseudo-code is presented in Algorithm 5 and 6.

Algorithm 5: DTA-to-RDMA translation in Key-Increment

Input : Redundancy N, Key K, Counter C
$Bufstart \leftarrow$ Address to start of RDMA memory buffer
$Buflen \leftarrow$ Number of allocated KeyVal slots
Function `CraftWrite(n, K, C)`
  $Slot \leftarrow h_0(n, K) \bmod Buflen$
  $Dest \leftarrow Bufstart + Slot * 4$
  Increment $Dest$ by C through RDMA Fetch&Add
for $n = 0 \rightarrow N$ **do**
  `CraftWrite(n, K, C)`

Algorithm 6: Querying the Key-Increment storage

Input : Redundancy N, Key K
Output : C_{winner}
$Buflen \leftarrow$ Number of allocated KeyVal slots
$Storage \leftarrow$ Array size $Buflen$ with $\langle C \rangle$ elements
Function `GetSlot(n, K)`
  $Slot \leftarrow h_0(n, K) \bmod Buflen$
  return $Storage[Slot]$
$Counters \leftarrow []$
for $n = 0 \rightarrow N$ **do**
  $Counters[n] \leftarrow$ `GetSlot(n,K)`
$C_{winner} \leftarrow \min(Counters)$

A.5 Analysis of the Key-Write primitive

Because we treat the RDMA memory as a large key-value hash table where only checksums of keys are stored and values may be overwritten over time, we must consider the possibility that when we make a query, we are unable to return an answer, or we may return an incorrect answer. We call the case where we have no answer to return an *empty return*, and the case where we return an incorrect answer a *return error*. The probability of an empty return or a return error depends on the parameters of the system, and on the method we choose to determine the return value. Below we present some of the possible tradeoffs and some mathematical analysis; we leave further results and discussions for the full paper.

Let us first consider a simple example. When a write occurs for a key-value pair, in the hash table N copies of the b-bit key checksum and the value are stored at random locations. We assume the checksum is uniformly distributed for any given key throughout our analysis. When a read occurs, let us suppose we return a value if there is only a single value amongst the N memory locations matching that checksum. (The value could occur multiple times, of course.)

An empty return can occur, for example, if when we search the N locations for a key, none of them have the right checksum. That is, all N copies of the key have been overwritten, and none of the N locations currently hold another key with the same checksum. To analyze this case, let us consider the following scenario. Suppose that we have M memory cells total, and that there are $K = \alpha M$ updates of *distinct* keys between when our query key q was last written, and when we are making a query for its values. We can use the Poisson approximation for the binomial (as is standard in these types of analyses and accurate for even reasonably large M, N, K; see, for example, [9, 49]). Using such approximations, the probability that any one of the N locations is overwritten is given by $(1 - e^{-KN/M})$, and that all of them are overwritten is $(1 - e^{-KN/M})^N$. The probability that all of them are overwritten and the key checksum is not found is approximated by

$$(1 - e^{-KN/M})^N \cdot (1 - 2^{-b})^N = (1 - e^{-\alpha N})^N \cdot (1 - 2^{-b})^N.$$

We would also get an empty return if the N cells contained two or more distinct values with the same correct checksum.
This probability is lower bounded by

$$\sum_{j=1}^{N-1} \binom{N}{j} (1 - e^{-\alpha N})^j e^{-\alpha N(N-j)} (1 - (1 - 2^{-b})^j) \quad ,$$

and upper bounded by

$$\Big(\sum_{j=1}^{N-1} \binom{N}{j} (1 - e^{-\alpha N})^j e^{-\alpha N(N-j)} (1 - (1 - 2^{-b})^j) \Big) + (1 - e^{-\alpha N})^N (1 - (1 - 2^{-b})^N - N \cdot 2^{-b} (1 - 2^{-b})^{N-1}).$$

The first summation is the probability at least one of the original N locations is not overwritten, but at least one overwritten location gets the same checksum. (We pessimistically assume it obtains a different value.) The second expression adds a term for when all original values are overwritten and two or more obtain the same checksum. Note that we need to give bounds as values in overwritten locations may or may not be the same.

We could have a return error if all N copies of the original key are overwritten and one or more of those cells are overwritten with the same checksum and same (incorrect) value. This probability is lower bounded by

$$(1 - e^{-\alpha N})^N N 2^{-b} (1 - 2^{-b})^{N-1},$$

which is the probability that all of the original locations are overwritten and a single overwriting key obtains the checksum, and upper bounded by

$$(1 - e^{-\alpha N})^N (1 - (1 - 2^{-b})^N),$$

the probability that the original locations are overwritten and at least one overwriting key obtains the checksum.

There are many ways to modify the configuration or return method to lower the empty returns and/or return errors, at the cost of more computation and/or more memory. The most natural is to simply use a larger checksum; we suggest 32 bits should be appropriate for many situations. However, we note that at "Internet scale" rare events will occur, even matching of 32-bit checksums, and so this should be considered when utilizing Key-Write information. One can also use a "plurality vote" if more than one value appears for the queried checksum; additionally one can require that a checksum/value pair occur at least twice among the N values before being returned. (Note that, for example, requiring consensus of two values can be decided on a per query basis without changing anything else; one can decide for specific queries whether to trade off empty returns and return errors this way.) Additional ideas from coding theory [19, 45], including using different checksums for each location or XORing each value with a pseudorandom value, could also be applied. As a default, we suggest a 32-bit checksum and a "plurality vote".

A.6 Analysis of the Postcarding Primitive

We now calculate:

(a) The probability that a flow's values fail to be reported, because the flow has been overwritten.
(b) The probability that a flow is reported with incorrect values.

We assume that the number of *reports* (up to B postcards that belong to the same flow/packet) since the queried ID is $\alpha \cdot C$.

For (a), we consider several reasons (similar to (1)-(3)) for failing to report the values and analyze them separately.

- All of the queried flow's chunks are overwritten by other flows and none of them produce valid information. We have that the probability that a slot is overwritten is bounded by $(1-e^{-\alpha \cdot N})$. Also, the probability of a given overwritten slot to *not* produce valid information is: $1-\left((|V|+1)\cdot 2^{-b}\right)^B$. Therefore, the overall probability of this event is at most

$$(1-e^{-\alpha \cdot N})^N \cdot \left(1-\left((|V|+1)\cdot 2^{-b}\right)^B\right)^N. \quad (9)$$

- All the flow's chunks are overwritten and at least two produce valid information arrays that differ. This probability is bounded by:

$$(1-e^{-\alpha \cdot N})^N \cdot \left(1-\left(1-\left((|V|+1)\cdot 2^{-b}\right)^B\right)^N - N\cdot\left((|V|+1)\cdot 2^{-b}\right)^B \cdot \left(1-\left((|V|+1)\cdot 2^{-b}\right)^B\right)^{N-1}\right). \quad (10)$$

- At least one chunk (but not all) is overwritten and produces valid information. This error probability is at most

$$\sum_{j=1}^{N-1}\binom{N}{j}\cdot(1-e^{-\alpha\cdot N})^j\cdot e^{-\alpha\cdot N(N-j)} \cdot\left(1-\left(1-\left((|V|+1)\cdot 2^{-b}\right)^B\right)^j\right). \quad (11)$$

Next, we analyze the probability of replying incorrectly (b). This happens when all the queried key's chunks are overwritten and all valid chunks are hold the same information. Thus, the probability of such an error is at most:

$$(1-e^{-\alpha\cdot N})^N\cdot N\cdot\left((|V|+1)\cdot 2^{-b}\right)^B. \quad (12)$$

Let us consider a numeric example to contrast these results with using KW for each report of a given packet. Specifically, suppose that we are in a large data center ($|V| = 2^{18}$ switches) and want to run path tracing by collecting all (up to $B = 5$) switch IDs using $N = 2$ redundancy. Further, let us set $b = 32$-bit per report and compare it with 64 bits (32 for the key's checksum and 32 bits for the switch ID) used in KW and that $C \cdot \alpha$ packets' reports were collected after the queried one, for $\alpha = 0.1$. We have that the probability of not outputting a collected report (9-11) is at most 3.3% and the chance of providing the wrong output (12) is lower than 10^{-22}. In contrast, using KW for postcarding gives a false output probability of $\approx 8 \cdot 10^{-11}$ (in at least one hop) using twice the width per entry! This improvement is due to a couple of reasons. First, we leverage the difference between the number of switches (e.g., $|V| = 2^{18}$) and the width of the value field (hardcoded at 32-bits per the INT standard [21]). Second, we leverage the fact that each packet carries multiple (e.g., $B = 5$) reports to amplify the success probability and mitigate the chance of wrong output. Further, for reports for which we are able to cache all postcards at the translator (which depends on the allocated memory and the number of simultaneous postcard reports generated), this approach reduces the number of RDMA writes by a factor of B.