A Novel Transport Protocol for Network Telemetry Collection

Authors

1. ABSTRACT

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

2. INTRODUCTION

ToDo: ***(Please feel free to change the intro structure!)***

ToDo: ***What is telemetry? Why is it important?***

ToDo: ***State-of-the-art systems are designed around centralized collection. (I have several sources in hotnets paper for this, pointing to proprietary solutions)***

ToDo: ***But collection is very costly! This requires operators to reduce network insight through various methods (sampling, selection, only super-specific queries)***

ToDo: ***RDMA has huge potential!***

ToDo: ***RDMA is limited (verbs, reliability, number of QPs, costly generation)***

ToDo: ***Introduce DTA. Achieving RDMA-performance and support of standard NICs.***

3. MOTIVATION

JL: This should build on background and dig into why

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

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

CPU-based collection is inefficient. JL: Waiting for Gabriele results before planning structure

Collectors play an important role in network telemetry systems: they receive telemetry reports and store the information in an internal data structure to be used to answer network-wide queries. One key challenge is to ensure that this process is scalable as a datacenter network can comprise hundreds of thousands of switches [3]. For example, a non-sampled INT telemetry system requires the collection of telemetry data from *every single packet*, which would result in an excessive amount of reports. Because of this, event detection is typically implemented at switches in an effort to send reports to a collector only when things change [5]. This helps in reducing the rate of switch-to-collector communication down to a few million telemetry reports per second per switch [12], at the cost of reduced network insight and increased on-switch complexity. Still, telemetry collection costs are high, and the main reason we identified is that the collectors' CPU is the main bottleneck.

3.1 Monitoring Systems are Intense

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

3.2 CPU-based collection is inefficient

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

MY: Discuss existing monitoring solutions

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

We show examples of what the CPU spends time on in Figure 1, and discuss how they already attempt to optimize

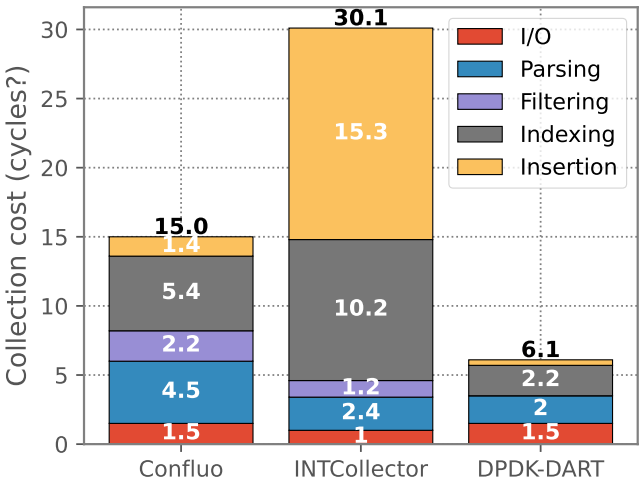

Figure 1: Breakdown of telemetry collection costs in CPU-based solutions. DPDK-DART is functionally equivalent to DTA Multi-Write (This is placeholder data!)

these numbers. DTA would either bypass or offload these steps into hardware.

3.3 RDMA enables high-speed memory writes

What if we design telemetry collection around RDMA?

4. RDMA RESTRICTIONS ON COLLECTION

JL: Motivation ends with: RDMA has a huge performance potential!

Designing telemetry collection around RDMA has the potential for significant performance improvements. However, RDMA and related protocols impose several restrictions on such a design:

RDMA is limited to basic memory operations Achieving the full performance potential of RDMA requires restricting ourselves to one-way verbs, i.e., pre-defined primitives that execute entirely on the network card without requiring CPU intervention. This limits us to basic memory operations: *Read*, *Write*, *Fetch-and-Add*, and *Compare-and-Swap*. Restricting the design to these fundamental primitives greatly reduces the viability of dynamic data structures such as linked lists and cuckoo hashing. Such data structures require us to first read memory at the collector before knowing exactly what to write where, leading to significantly more complex on-switch logic. Further, one must ensure that there are no conflicts or race-conditions occurring between different switches that work in parallel to report telemetry information.

A limited number of RDMA connections RDMA network cards have limited memory, which constrains the number of active connections that fit in their local cache. Having more simultaneously active RDMA connections to a collector than can fit in the cache forces the network card to start swapping connection statefulness to server memory, which *significantly* reduces RDMA performance. Potential workarounds such as having several switches share the same connection become very complicated, due to the RDMA requirement of sequential packet identifiers within each connection, which would require sharing switches to synchronize these values on each new telemetry report.

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

Managing RDMA connections is costly for switches RDMA traffic is not designed for ease-of-generation in network switches. For example, the RoCEv2 protocol, which is the standard for RDMA communication for Ethernet, is especially costly. This protocol requires a relatively high amount of statefulness on a per-connection basic for tracking essential metadata, significant header overhead to calculate and generate, and a large footprint on internal busses handling RoCEv2-imposed checksumming. Requiring such a footprint on every single telemetry-reporting switch is very inefficient.

The data must be queryable Allowing efficient centralized access to telemetry data is the main reason for centralized collection. How to make it queryable? Cross-switch collaboration etc..JL: Too similar to first?

5. DTA OVERVIEW

ToDo: ***The plan is to give high-level overview. Explain DTA split into reporters, translators, and collectors. We are designed to be generic, and are aiming to support a wide range of monitoring systems.***

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

The translator parses incoming DTA packets, and converts these into suitable RDMA operations that are forwarded to the right collector. Generated RDMA traffic is determined

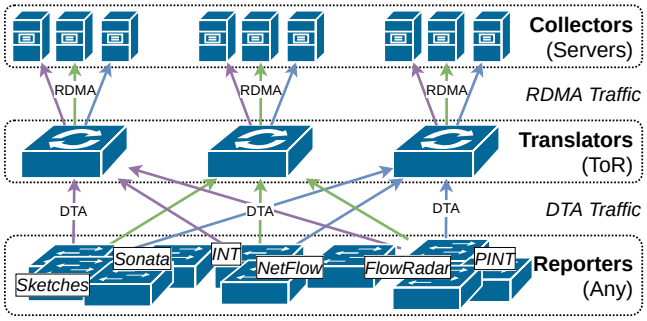

Figure 2: Overview of the DTA report data flowJL: Expand to 2 columns and add more info?

by the incoming DTA packet, which specifies the specific DTA primitive that is requested, together with essential DTA metadata according to the needs of the primitive (please see section 6 for a discussion on DTA primitives).

5.1 Translation Benefits

A naive RDMA-based telemetry collection design might generate RDMA traffic immediately at the telemetry generating network switches (i.e., at the reporters), which would require determining a storage server and a memory address inside of that server, and then generating an RDMA call to update the telemetry storage.

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

Allowing central pre-processing without CPU involvement A translator is guaranteed to be the last processor for all telemetry packets destined to the collectors that it manages. This allows us to expand RDMA with new primitives and functionality that take advantage of a central viewpoint, including the addition of the powerful *Append* 6.2 primitive, centralized data duplicate detection and filtering, and real-time detection of network-wide events.

Improved collection speeds The translator is in full control of the RDMA state of each collector, which allows significant performance and stability improvements compared with designs where each reporter would handle their own RDMA communications directly. This includes eliminating RDMA queue-pair state swapping in the network card, which has a negative performance impact when several RDMA connections are active simultaneously.

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

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

DTA translation future-proofs telemetry systems Several commercial network cards support RDMA communication through the RoCEv2 protocol, which our translator has built-in support for. However, future network cards might very well include native telemetry collection capabilities JL: cite Bluefield latest update showing work towards this. Networks designed around DTA-based telemetry collection might not require network-wide switch overhauls to support new and high-speed telemetry collection techniques, since these network cards might either have native DTA support or allow DTA translation into the new format.

5.2 Translator vs SmartNIC Dilemma

JL: Where to place translator? ToR or NIC?

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

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

6. DTA PRIMITIVES AND "GENERALIZEABILITY"

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

6.1 Key-Write

The Key-Write primitive is designed to allow for key-value storage in DTA, where each piece of telemetry data has a unique pre-known key that can be used for retrieval of the reported data. There are several telemetry scenarios that can

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

Table 2: Example telemetry monitoring systems and scenarios, as mapped into the primitives presented by DTA

be expressed in a key-value fashion, and we present a few examples in Table 2.

Telemetry reporters would only have to send a single DTA packet towards one of the collection racks, containing the *key*, *telemetry data*, and a *redundancy* integer which will be further explained in section 6.1.1. The DTA translator will parse this Key-Write request, and generate RDMA traffic to insert this data into a global key-value store in an adjacent server. See section 7.3.1 for details of how DTA implements Key-Write in hardware.

6.1.1 *Probabilistic Nature*

ToDo: ***Explain the probabilistic nature of Key-Write*** JL: Redundancy, checksumming, plurality voting

6.1.2 *Querying*

JL: How is it queried?

6.2 Append

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

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

Telemetry reporters simply have to craft a single DTA packet to declare what data they want to append to which list, and forward it to a collection rack that hosts this list. The translator would then generate an RDMA packet that inserts this new telemetry data in the correct slot in the pre-allocated list. See section 7.3.2 for details of how DTA implements Append in hardware.

6.2.1 *Querying*

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

6.3 Sketch-Merge JL: Ran?

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

JL: The design is not finalized. High-level is to allow network-wide merging of sketches. For example through fetch-and-add. Likely using switch SRAM as cache, periodically updating down into collector. Assuming same sketch dimensions everywhere

6.4 Key-Increment

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

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

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

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

7. IMPLEMENTATION JL: ME

JL: Implemented through a combination of P4, Python, and C++ for Tofino switches and x86 servers

7.1 DTA Reporters

JL: Ignoring monitoring-specific details. Assuming we already have data to report. JL: DTA reports is designed for ease of generation. They simply state opcode and primitive-specific metadata

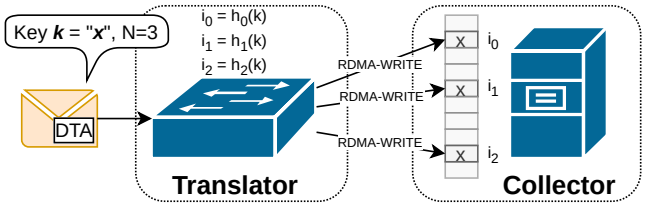

Figure 3: The DTA Key-Write primitive.JL: add querying

The reporter pipeline is written in P4_16 for Tofino, and deployed on a BF2556X-1T switch. Controller functionality is written in Python, running on the switch CPU. Generated DTA output that the reporter emits is recorded in a Pcap format, allowing us to replay this trace through a translator at a later time.

7.2 DTA Report Structure

JL: Visualize and explain DTA headers and encapsulation

JL: Draw headers similarly to HPCC fig 7. Show a couple of examples (keywrite and append)

7.3 Translator

JL: Visualize pipeline? Ingress/egress split (very short and wide, keeping figure compact)

JL: The translator has three main code paths: background traffic, DTA, and RDMA (acks) (visualize these paths in figure as colored lines/arrows)

JL: RDMA generation logic is re-used for all primitives. Significantly reducing footprint (show overlap in figure) JL: Resynchronization through CNP parsing

JL: Push hard on the slim design making our system viable! (maybe in eval under asic costs)

The translator pipeline is written in P4_16 for Tofino, and deployed on a BF2556X-1T switch. Controller functionality is written in Python, running on the switch CPU. The current translator pipeline has support for both the Key-Write and Append primitive, and can process these in parallel.

7.3.1 Key-Write

JL: fix text. just infodumped Multicasting is triggered in the Ingress pipeline, with multicast groups pre-configured by the controller. Each <N,collector> tuple has corresponding multicast rule, ensuring N packets are injected into the correct egress pipeline. These N packets each hold a separate integer $n \in 0, 1, ..., N-1$, declaring which redundancy entry they should generate an RDMA packet for. The native CRC extern is used to map <key,n> into a memory-slot hash. This hash is further bounded into the size of the key-value memory buffer through a M/A table, which maps the number of storage entries into a bitmask. A bitwise AND operation is performed on the hash using the fetched bitmask, thereby allowing modulo of powers of 2[1]. The calculated memory slot is easily translated into a buffer-offset by multiplying it with the size of a single key-value slot (i.e., the combined size of the checksum and data value).

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

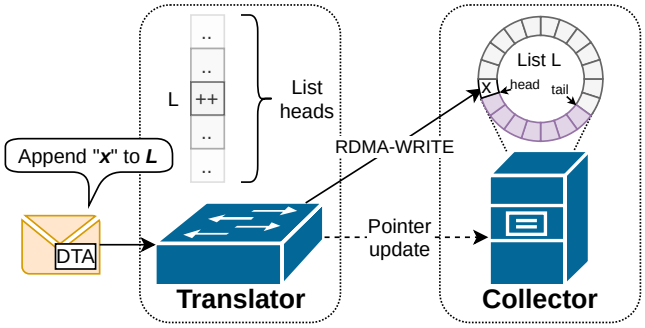

Figure 4: The DTA Append primitive.JL: add querying

7.3.2 Append

JL: ring-buffer. HEAD counter in register. Periodically sent to collector (every X append).

7.3.3 Sketch-Merge

JL: POTENTIAL implementation solution. Not yet verified

7.3.4 Key-Increment

JL: POTENTIAL implementation solution. Not yet verified

7.3.5 RDMA Connection

Initiator script. PSN counter. Desynchronization. PktID. Metadata-population. etc...

7.4 Collector

JL: Explain C++ RDMA service, registering allocated buffer in Bluefield RDMA engine

JL: Allocated advertised storage as 1g hugetables. Recompiled Mellanox driver to support physical addresses, removing IOTLB performance degradataion for psuedo-random accesses in large memory buffers

8. EVALUATION JL: ME

All benchmarks in this section are performed using the Mellanox Bluefield-2 DPU as acting RDMA-capable network card, and using a BF2556X-1T Tofino switch as both reporter and translator. The TReX traffic generator is used to inject DTA packets into the translator switch.

8.1 Key-Write Primitive Performance

DTA Key-Write is a powerful primitive that allows global collection with dynamic keys, without requiring cross-network

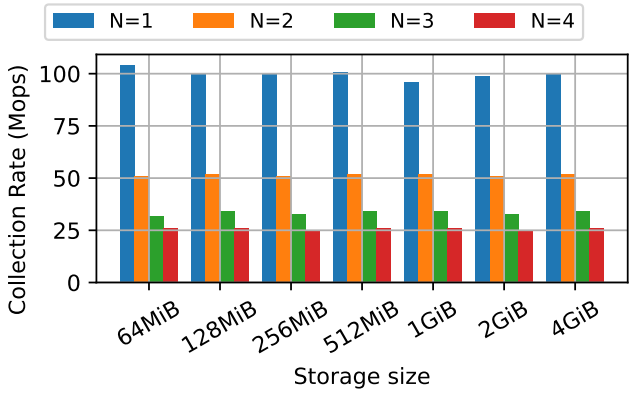

Figure 5: Single-NIC ingestion rate of DTA Key-Write operations, at various redundancy levels N. The telemetry data entries are 32-bit integers, with a 32-bit concatenated checksum.

collaboration or synchronization. It does however impose overheads in terms of built-in checksumming and redundancies.

8.1.1 *Key-Write Collection Rate*

JL: explain test in Figure 5 JL: Pushing traffic at increasing rates 10 seconds at each rate. Stopping at first packet loss

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

8.1.2 *Redundancy Effectiveness*

JL: This section is heavily based on the hotnets paper

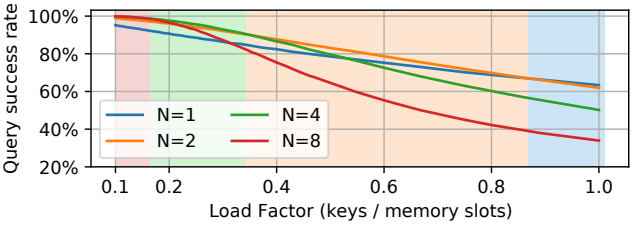

Figure 6: Average query success rates delivered by the Key-Write primitive, depending on the key-value store load factor and the number of addresses per key (N). The background color indicates optimal N in each interval.

The probabilistic nature of the DTA Key-Write primitive cannot guarantee final queryability on a given reported key. We show in Figure 6 how the query success rate depends on the load factor (i.e., the total number of telemetry keys over available memory addresses), and the number of memory addresses that each key can write to. There is a clear efficiency improvement by having keys write to $N > 1$ memory addresses when the storage load factor is in reasonable intervals, and the background color in Figure 6 indicate which number of addresses per key (N) delivered the highest key queryability in each interval.

Recall that RDMA lacks the ability to write into multiple memory addresses with a single packet, and that DTA overcomes this by generating N distinct RDMA packets for writing the redundancy to memory. Requiring N DMA calls per DTA operation results in a decrease of collection throughput, as shown earlier in Figure 5. Determining an optimal redundancy level therefore has to be a balance between an enhanced data queryability and a reduction in collection performance. $N = 2$ appears to be a generally good compromise, showing great queryability improvements over $N = 1$.

8.1.3 *Data Longevity*

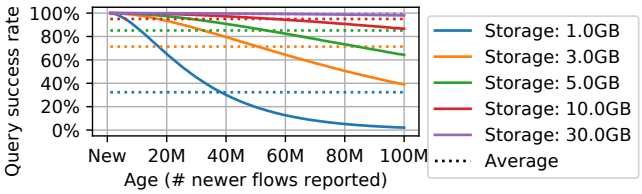

Figure 7: DTA Key-Write ages out stale data. This figure shows INT 5-hop path tracing queryability of 100 million flows at various storage sizes. The results are based on storing 160-bit values with 32-bit checksums, where each key writes into $N = 4$ different memory addresses.

Text

8.1.4 *Query Answer Correctness*

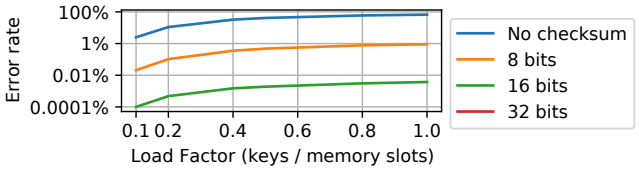

Figure 8: The probability of a query returning the wrong answer in error, due to address and checksum collisions.

There is a theoretical risk of the Key-Write primitive returning incorrect query results if two or more different keys hash into the same memory address, while they at the same time also have the same concatenated checksum. JL: There is so much more than this. Can we fit some theory in the paper? Maybe as appendix? Figure 8 shows results after extensive tests, where multiple simulations of 100M keys have been performed at various storage sizes in an attempt to recreate the theoretically predicted incorrectness. These results clearly show the impact from having key-based checksums included in the DTA Key-Write data structure, with increased lengths greatly reducing the risk of errors. Our simulations with 32-bit key-checksums fail to reproduce these return-error cases, due to the incredibly low probability that these events occur, even while the data structure is under immense load.

8.1.5 *Query Speed*

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

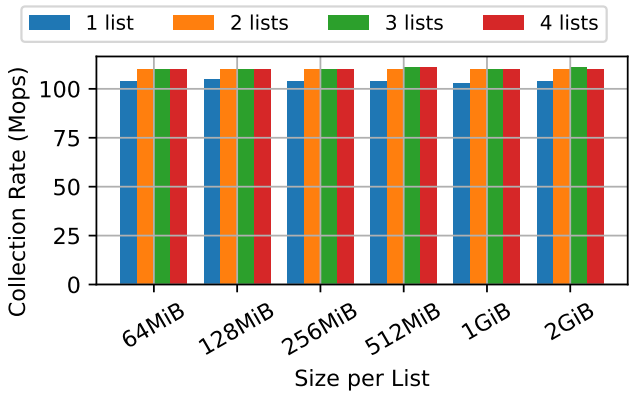

Figure 9: Single-NIC ingestion rate of DTA Append operations, at various numbers of parallel lists and sizes of individual lists. The telemetry data entries in the lists are 32-bit integers.

8.2 Append Primitive Performance

JL: explain test in Figure 9

8.3 Querying rate

JL: How fast can we query?

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

8.4 DTA Collection Capabilities

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

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

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

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

8.5 Queue-Pair Resynchronization?

JL: Show the responsiveness of RDMA PSN resynchronization.

8.6 ASIC Resource Footprint

JL: Show the cost of our system in terms of Tofino resources.

JL: Also show cost of generating RDMA vs generating DTA, to back up our claim that DTA reduces in-reporter costs. Show with full iCRC calculation and PSN resync.

8.6.1 RDMA vs DTA Generation

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

8.7 Network Overhead?

JL: Quick bar-plot showing header costs of DTA vs RoCEv2? Showing benefit from translation

9. LIMITATIONS

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

9.1 Limited Aggregation Capabilities

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

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

JL: We can definitely utilize collector CPU to perform these, and likely significantly faster than current state-of-the-art.

Example: an Append list containing tuples <address,value,function>. CPU can just iterate over this list and perform this action.

Main drawback here is that it doesn't fit our paper story

9.2 RDMA rates

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

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

10. DISCUSSION

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

10.1 Translation vs Programmable SmartNICs

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

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

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

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

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

10.2 Read-based Writes

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

10.3 Large-scale Deployments

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

10.4 Collection Epochs and History

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

11. RELATED WORKS

ToDo: ***Add related works***

11.1 TEA

ToDo: ***Discuss TEA***

12. ACKNOWLEDGEMENTS

Text goes here

13. CONCLUSION

Conclusion goes here

14. REFERENCES

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

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

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

[4] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 357–371, 2018.

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

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

[7] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[8] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better netflow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 311–324, 2016.

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

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

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

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