INTFusion: Unifying Network and Host Telemetry in Data Center Networks

Leonardo Alberro
Universidad de la República
Montevideo, Uruguay
lalberro@fing.edu.uy

Matias Richart
Universidad de la República
Montevideo, Uruguay
mrichart@fing.edu.uy

Eduardo Grampin
Universidad de la República
Montevideo, Uruguay
grampin@fing.edu.uy

***Abstract*—Modern data center networks require fine-grained, low-latency telemetry to support congestion control, performance diagnosis, and learning-driven transport adaptation. While programmable data-plane telemetry and host-level tracing have independently advanced network observability, existing monitoring systems remain fragmented, limiting their effectiveness for closed-loop control and cross-layer optimization.**

In this paper, we present INTFusion, a unified monitoring architecture that integrates in-network telemetry with host-level observability at per-flow granularity. Our design offloads both INT source and sink functionality to smartNICs at end hosts, enabling line-rate telemetry collection while tightly coupling network measurements with application context. Hosts additionally collect system- and application-level traces, including kernel state, packet-level statistics, and syscall events via eBPF. Telemetry is aggregated per flow and exported using a two-tier mechanism: latency-critical events are delivered in real time, while non-critical measurements are rate-regulated and deferred to off-peak periods to minimize overhead. At a central collector, we propose an approach to fuse network and host traces into unified per-flow records, providing end-to-end visibility. Our results demonstrate that edge-terminated INT, combined with host-level tracing, provides scalable, low-overhead, and application-aware telemetry, establishing a practical foundation for next-generation congestion-control telemetry in data center environments.

***Index Terms*—Telemetry, INT, Data Center Networks**

I. Introduction

Modern data centers increasingly rely on fine-grained telemetry to drive traffic engineering, anomaly detection, congestion control, and application-level performance optimization. While programmable data planes and host-based instrumentation have independently enabled unprecedented visibility into network and system behavior, existing monitoring architectures remain fragmented: network telemetry is often collected in isolation from host- and application-level signals, whereas host-based tracing lacks visibility into in-network dynamics such as queuing, path changes, and microbursts. This separation, for example, limits the effectiveness of closed-loop congestion control and learning-based transport adaptation, both of which require synchronized, per-flow observability across the network stack. As interactions between applications and the network grow increasingly complex, diagnosing application performance anomalies becomes progressively more challenging [1]. For instance, identifying TCP incast [2] requires not only visibility into the individual flows affected, but also precise timing information regarding when these flows are generated at hosts, when they arrive at switches, and which concurrent packets contend with them along the path. Emerging congestion control requirements further motivate advances in network telemetry, particularly to monitor and support the growing set of transport protocols recently proposed for data center environments [3].

Current network-centric approaches, including INT-based fabrics [4], [5] and switch-level telemetry pipelines, provide detailed visibility into in-network conditions but lack application context and end-host semantics. Conversely, host-based tracing systems offer deep insight into application behavior and kernel execution paths but are blind to in-network phenomena such as queue buildup, path changes, and transient congestion. As a result, existing congestion control frameworks typically operate on partial observability, relying on coarse end-to-end signals (e.g., ECN) or heuristics (e.g., RTO or duplicated ACKs in TCP) rather than coordinated, per-flow cross-layer measurements. Moreover, most telemetry systems either stream raw measurements continuously—incurring prohibitive bandwidth and processing overhead—or rely on aggressive sampling [6], top-k counting [7], or sketches [8], sacrificing precision exactly when congestion dynamics are most volatile [9].

In this paper, we present INTFusion, a unified monitoring architecture for data centers and private clouds that integrates in-network telemetry with host-level observability. INTFusion enables fine-grained telemetry signals and trace collection that are instrumental for congestion-control mechanisms. Our design leverages In-band Network Telemetry (INT) [10] while offloading both the INT source and sink functions to smartNICs (sNICs) at end hosts, eliminating the need for switch-resident sinks and enabling direct coupling between network measurements and application context. SmartNICs are widely deployed in data center networks (DCNs) and typically feature architectures composed of many low-power processing cores, making them well-suited to the highly parallel workloads characteristic of streaming analytics frameworks. INTFusion takes advantage of this power to further collect and aggregate on a per-flow basis INT information across different packets in a hash table and track custom events for anomaly detection at line rate, combining P4 [11] and C-based functions.

The hosts simultaneously collect system- and application-level telemetry, including kernel state from /proc, packet-level

ISBN 978-3-903176-82-9 © 2026 IFIP

traces, and syscall events via eBPF. All telemetry is aggregated at per-flow granularity prior to export, ensuring scalability and enabling consistent correlation across layers.

Unlike prior INT deployments that either mirror every telemetry-carrying packet or rely on centralized switch sinks, our architecture introduces a two-tier export model. First, the smartNIC-resident INT sink performs real-time event detection and exports congestion signals immediately upon detecting anomalies. Second, non-critical telemetry is rate-regulated and deferred to off-peak periods, preventing monitoring traffic from interfering with production workloads. At a central collector, we fuse network and host traces into unified per-flow records, producing end-to-end flow profiles that capture both path-level dynamics and application-level behavior. These datasets directly support learning-based congestion control, flow-size prediction, and transport adaptation without requiring intrusive application instrumentation.

We implemented INTFusion on a local cluster using P4-INT [12] and servers equipped with Netronome Agilio CX dual-port 10-GbE sNICs [13]. The sNIC packet-processing pipeline integrates a P4 program with dedicated C-based modules that implement complex INT processing tasks.

Our work demonstrates that edge-terminated INT, combined with host-level tracing, enables scalable, low-latency, and application-aware network monitoring, overcoming the limitations of existing network-only or host-only approaches. Furthermore, we show how this combination enables new use cases that could assist congestion-control mechanisms. In summary, this paper makes the following contributions:

- The design of a monitoring architecture that offloads INT source, sink, and metric collections functionality to sNICs while preserving line-rate, per-hop telemetry collection.
- The integration of in-network telemetry with host-level system and application traces at per-flow granularity. We demonstrate that unified per-flow traces can create new telemetry signals and support learning-based congestion control.
- An event-driven and rate-regulated export mechanism that balances real-time responsiveness with bandwidth efficiency.
- An open-to-the-community implementation of the hosts and hardware artifacts.

The rest of this paper is organized as follows. Section II introduce the background and related work. Then, in Section III we present the design and implementation of INTFusion. Finally, Section IV and Section V present the evaluation and main conclusions.

II. Background and Related Work

A. Programmable Telemetry

The emergence of programmable data planes [14], [15] has enabled a new class of network telemetry systems that perform end-to-end monitoring directly within the data plane. These systems rely on in-band measurement, in which telemetry information is embedded in packets as they traverse the network. This approach extends network observability beyond conventional polling-based mechanisms in Software-Defined Networks (SDNs), such as OpenFlow, which—although simple to deploy—can impose substantial monitoring overhead on both controllers and switches. More broadly, traditional telemetry mechanisms, including SNMP polling [16], flow export (NetFlow/IPFIX) [17], and packet sampling (sFlow) [18], operate at coarse temporal and spatial granularity, limiting their effectiveness in diagnosing transient congestion, microbursts, and path-dependent anomalies [19].

Programmable switches based on languages such as P4 [11] enable packet headers to be parsed, modified, and augmented at line rate, allowing telemetry data to be embedded directly in transit packets. This capability underpins In-band Network Telemetry (INT), whose specification [12] defines three header types—MD-type, Destination-type, and MX-type—with MD-type commonly used to collect detailed hop-by-hop measurements. As illustrated in Figure 1, an INT source inserts telemetry instructions into selected packets, transit switches append local metadata (e.g., latency or queue occupancy), and an INT sink removes the telemetry header and exports the collected information to monitoring systems before forwarding the original packet to its destination.

INT supports configurable telemetry formats through instruction masks and metadata templates, enabling a trade-off between measurement granularity and overhead. Similar functionality is standardized by the IETF's IOAM framework [20], which likewise enables data-plane visibility into transient network dynamics.

In contrast to SDN-based approaches, INT operates directly in the data plane without requiring control-plane intervention. However, the current specification exhibits some limitations. First, transit nodes append local metadata to packets, introducing non-negligible bandwidth overhead. To address this issue, prior work has proposed optimizations to packet formats and metadata encoding [21], [22].

A second challenge arises from the processing overhead at the sink: at high link rates, per-packet telemetry reports can overwhelm analytics and storage systems. Beyond packet-format optimizations, prior work has leveraged specialized co-processors, such as sNICs, to accelerate telemetry parsing and export [23]–[25].

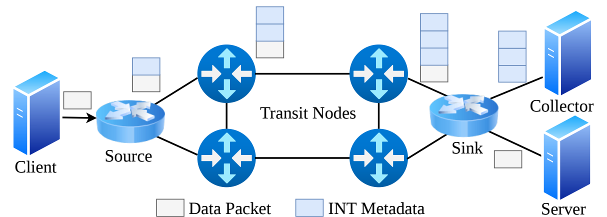


Fig. 1: INT architecture.

B. INTFusion within the Monitoring Systems Landscape

Although network monitoring has been extensively studied, rapid advances in the field continually create new needs

for network telemetry and for approaches to address them. INTFusion draws inspiration from prior work, filling new gaps and consolidating efforts to present an architecture that tracks the entire lifecycle of a flow to assist congestion control using efficient end-to-end telemetry.

INT offloading. SmartNICs' architectural properties enable compute-intensive and latency-sensitive networking functions to be offloaded from general-purpose processors, improving performance isolation and reducing host overhead. These capabilities have been leveraged to accelerate packet processing with P4 [26] and, more closely related to our work, to offload INT packet processing [25] [23] [27]. For instance, [27] processes INT packets on a sNIC and reports simple threshold-based telemetry events; however, reliance on the P4 pipeline constrains per-flow state to basic registers and counters. In contrast, [25] proposes partitioning INT processing between a P4 pipeline and C-based programs on a Netronome Agilio sNIC, enabling more complex analytics. While we adopt a similar architectural partitioning, we redesign the flow pipeline and metric collection mechanisms to achieve more precise flow identification, as detailed in Section III-A.

Combining INT telemetry and host telemetry. Existing programmable telemetry systems typically terminate telemetry within the network fabric, thereby decoupling in-network measurements from host and application contexts and limiting accurate end-to-end flow visibility. Several prior efforts have explored combining INT and eBPF to achieve end-to-end telemetry, but these approaches remain constrained to specific scenarios or exhibit notable limitations. In [28] and [29], INT collects network-level information while eBPF processes this data at the end host; however, these designs primarily offload analysis to reduce processing overhead and do not incorporate additional host-level context. In [30], telemetry is collected from all devices, but each device independently reports its state to the INT engine via dedicated packets, increasing overhead and complicating correlation. Finally, [31] leverages eBPF-XDP and TC hooks on hosts to intercept incoming packets, while switches rely on sampling and discretization of monitoring values, which can reduce telemetry precision.

Our design employs eBPF to intercept and monitor outgoing packets at hosts, while collecting INT metadata and events from incoming packets at the network edge. The rationale behind this asymmetric strategy is detailed in Section III-A.

Finally, the use of network telemetry to support congestion control is not new. For instance, INT has been leveraged by data center transport protocols such as [32] to improve performance metrics, including flow completion time. Moreover, prior work such as [28] and [29] combines INT with TCP metrics to enable TCP optimizations. However, none of them have developed a telemetry architecture capable of supporting a wide range of use cases.

III. INTFusion Overview

A. Architecture Design and Implementation

Monitoring data center networks requires balancing fine-grained telemetry with expressive analytics to enable the timely detection of operationally significant events. Such events are of interest to both operators and tenants, as they can adversely affect latency-sensitive flows. For example, an event may be defined as queue occupancy exceeding 50% at any switch along a path or end-to-end latency exceeding a predefined threshold. In conventional INT-based monitoring systems, these events are typically generated by a centralized stream processor that analyzes per-packet INT metadata to derive higher-level insights. However, exporting telemetry for every packet becomes impractical in modern data center fabrics operating at 100/400/800 Gbps.

To address this problem, our architecture distributes telemetry processing to reduce the load on the central analytics infrastructure while preserving real-time event-detection capabilities. Specifically, we detect INT events at the first tier of the architecture and aggregate per-packet telemetry into per-flow summaries prior to export to a central collector. Figure 2 illustrates the high-level architecture of INTFusion. In Step (1), the source host inserts INT instructions into outgoing packets, acting as an INT source. In Step (2), transit nodes append metadata according to these instructions (e.g., queue occupancy, node ID). In Step (3), the destination host functions as an INT sink, aggregating telemetry and detecting events. Upon flow completion, in Steps (4) and (5), hosts export aggregated per-flow telemetry to a central analyzer, while detected events are reported immediately. Finally, in Step (6), host- and network-level telemetry are combined to form an end-to-end view of each flow.

This architecture is end-host-oriented, with core functionalities—including INT source and sink operations, per-flow aggregation, and telemetry export—implemented on hosts. The network fabric is only required to support INT transit functionality, which can be readily realized using P4-programmable switches managed via the controller P4Runtime.

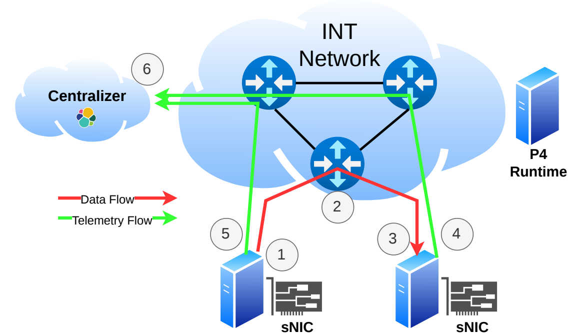


Fig. 2: INTFusion architecture.

B. Host Workflow and Implementation

The INTFusion architecture distributes monitoring tasks across multiple network locations. A centralized analyzer is responsible for collecting exported metadata and executing computationally intensive algorithms to correlate information

and detect anomalies, while network devices report requested measurements via INT. End hosts, however, constitute a fundamental component of this multi-tier architecture. Although end hosts must primarily deliver high application performance, their available computational resources can be leveraged efficiently to support advanced monitoring tasks. Modern data center servers increasingly incorporate dedicated co-processors—such as sNICs or DPUs for networking functions and GPUs for parallel, data-intensive workloads—that complement the main CPU and enable task specialization.

INTFusion exploits this heterogeneous architecture by distributing monitoring functions across three execution domains within the host: user space, kernel space, and sNIC space.

Figure 3 illustrates the placement of the different components inside a host. At the sNIC level, the INT source and INT sink modules implement edge INT functionality. The INT sink processes INT headers and performs per-flow aggregation, event detection, and notification. At the kernel level, a Spooler module retrieves per-flow traces produced by the INT sink. In addition, a Trace Collector module collects conventional packet metadata, high-level host statistics (e.g., memory utilization), and low-level function-call traces (e.g., `send()` invocations) for outgoing flows. Finally, at the user-space level, an Exporter Agent coordinates with the Centralizer to export aggregated traces and events at appropriate transmission intervals, ensuring controlled and efficient reporting.

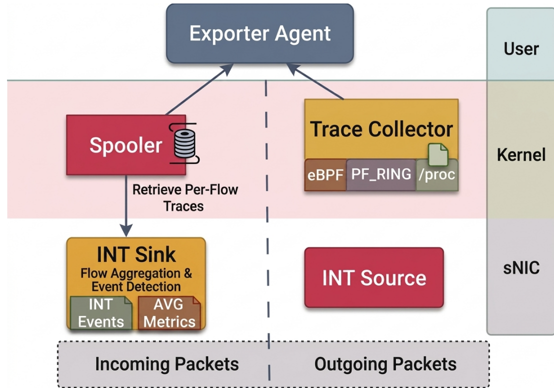


Fig. 3: Host design: modules are partitioned across User space, Kernel space, and the sNIC.

End-to-end Flow Traces. A first observation from Figure 3 is that the set of modules differs for incoming and outgoing flows. With respect to INT functionality, the INT Sink naturally operates on incoming packets, whereas the INT Source instruments outgoing packets. At the kernel level, the Trace Collector is intentionally restricted to outgoing flows. This design rationale is motivated by two considerations.

First, host-level trace collection consumes valuable CPU cycles and storage resources. To avoid unnecessary overhead, we limit kernel-level tracing to information strictly relevant to monitoring objectives, thereby minimizing the collection of nonessential data. Second, the primary objective of INTFusion is to monitor flows in support of congestion control. In most congestion control algorithms, the sender regulates the transmission rate based on network feedback or transport-layer control signals (e.g., TCP ACKs). Consequently, we define an end-to-end flow view as follows: i) host-level traces—comprising packet metadata, high-level host statistics, and low-level function-call traces—captured at the sender when the flow is initiated, combined with ii) network-level INT telemetry collected along the flow's path through the fabric. A detailed description of these traces is provided later in this section.

SmartNIC Space. The sNIC serves as the execution domain for INT edge functionality. Beyond implementing the standard INT packet-processing specifications, we offload fine-grained INT traffic analysis to the sNIC and execute it at line rate. Such functionality is typically infeasible on conventional P4 switches, where programmable logic is constrained to match–action pipelines with limited support for complex stateful processing. In contrast, sNICs can integrate P4-based packet parsing and match–action rules with more expressive programming models, such as Micro-C, enabling advanced stateful analytics directly in the data plane. This capability is supported by Netronome sNICs [13]. In this work, we use Netronome Agilio 4000 CX Dual-Port 10 Gigabit sNICs as the hardware platform for our implementation.

Beyond conventional sNIC offloads (e.g., checksum computation, segmentation, and reassembly), the Netronome sNIC integrates 60 programmable flow-processing cores, or Micro-engines (MEs), enabling advanced data-plane functions such as traffic shaping, match–action processing, and analytics. Furthermore, it supports P4-based programmable parsing and header extraction through a subset of MEs dedicated to packet processing. In our design, 54 MEs are assigned to the packet-processing pipeline, including INT header handling, while the remaining MEs execute independent MicroC programs to support asynchronous, stateful operations within the sNIC.

INT Source Module. The INT Source is the entity responsible for initiating INT for selected packets or flows. Its primary function is to instrument packets with the required headers and instructions so that transit devices along the forwarding path can append telemetry metadata, which is subsequently processed by the INT Sink. In addition, the INT Source defines the telemetry scope, i.e., the specific metadata fields that transit nodes must collect (e.g., queue occupancy, ingress and egress timestamps, per-hop latency, or switch identifiers).

To enable INT in INT-MD mode, the module modifies outgoing packets by inserting an INT Shim Header and an INT Metadata Header, as illustrated in Figure 4. This functionality is implemented entirely within the P4 pipeline and does not require interaction with Micro-C programs.

Our INT Source implementation operates as follows.

i) The INT headers are inserted immediately after the transport header (e.g., TCP or UDP) and before the application payload.
ii) The INT Shim header identifies the presence of INT, encodes the INT type (INT-MD), and specifies the total length of the telemetry block.

iii) The INT Metadata header (Instruction Header) is inserted to define the telemetry instructions, including the set of metadata fields that transit nodes must collect, the maximum hop count, the current hop count (initialized to zero), and the metadata length.

iv) Packet header fields are updated accordingly. After header insertion, the module adjusts the IP total length, transport-layer length (if applicable), and recalculates the relevant checksums to ensure packet consistency.

In our prototype implementation, we do not incorporate mechanisms to prevent MTU violations. In production deployments, this issue can be addressed through the use of jumbo frames, selective instrumentation or sampling policies, or encapsulation of INT packets within tunneling protocols.

INT Sink Module. The fundamental operation of an INT Sink consists of extracting telemetry metadata from the INT-MD metadata header, removing the INT headers, and forwarding both the restored original packet and the extracted metadata for further processing. This functionality can be implemented entirely within the P4 pipeline by combining programmable parsing with match–action processing.

Building upon these baseline functionalities, we extend the INT Sink with fine-grained traffic analysis capabilities implemented in Micro-C. These extensions support per-flow aggregation and programmable event detection directly on the sNIC. A similar offloading approach was proposed in [25], where the authors implement the INT Sink on a Netronome sNIC. Our sink module is inspired by that architecture but introduces a key distinction: the explicit incorporation of the flowlet abstraction, which enables more precise state management and event detection granularity.

This distinction is important because, to avoid the overhead of connection setup and TCP slow start, many data centers employ persistent, long-lived TCP connections that multiplex multiple application messages over time. Since our monitoring granularity is per-flow, relying solely on the conventional 5-tuple identifier would be insufficient in such environments, as it would aggregate heterogeneous traffic phases into a single, long-lived connection.

In these settings, a more appropriate abstraction is the *flowlet*, defined as a burst of packets exchanged between two hosts that is separated from other bursts by an inter-packet time gap. Accordingly, we identify a flowlet using the standard 5-tuple augmented with a unique identifier assigned by a mechanism that verifies whether the inter-arrival time between consecutive packets exceeds a predefined, configurable threshold. When this time gap is surpassed, a new flowlet identifier is generated, enabling finer-grained monitoring.

Algorithm 1 describes the main processing steps applied to a flowlet (hereafter referred to simply as a flow) upon arrival at the sNIC. When a packet is received, it traverses the P4 processing pipeline, which incorporates hooks to custom Micro-C functions executed on dedicated MEs within the sNIC. The P4 program defines standard Layer 2–4 headers (Ethernet, IPv4, UDP, and TCP), together with INT-specific headers: shim, metadata header, and a stack used to store per-hop metadata blocks collected along the INT path. The parser is designed to sequentially recognize these headers, determine the number of instrumented nodes, and extract the corresponding telemetry data. After step 5 (Algorithm 1), the number of participating INT nodes is determined, and the parser can iteratively extract per-node metadata blocks. Once it interprets the instructions encoded in the INT header, it extracts the requested telemetry fields and stores them in an internal metadata structure H that will later be consumed by Micro-C firmware. In the instruction bitmap, each bit enables the extraction of a specific metric (node ID, ingress and egress interfaces, per-hop latency, queue occupancy, ingress and egress timestamps, and buffer occupancy). Depending on the activated bits, dedicated actions (e.g., populate node-id or hop-latency metadata) transfer values from the raw, per-node, parsed data into typed metadata structures. After all requested fields have been extracted, in step 13, the program invokes the external function `save_in_hash()`, implemented in Micro-C. This function provides the interface between the P4 data plane and Micro-C firmware, forwarding the structured metadata to the routine responsible for inserting and updating entries in a hash table.

Algorithm 1 INT Sink SmartNIC Pipeline

Input: Packet
Output: Packet with INT removed
1: Load packet P
2: Extract Ethernet header
3: Extract IP header
4: Extract Layer 4 header
5: Extract INT Shrim header
6: Extract INT Metadata header MH
7: From MH, interpret the Instruction Bitmap
8: **for** each INT Metadata Block $h \in MH$ **do**
9: Extract h
10: Interpret metadata fields
11: Store h in H
12: **end for**
13: SAVE_IN_HASH(H, P)
14: Rebuild P without INT headers
15: Enqueue P

The logic implemented in Micro-C constitutes the core of the INT Sink. At this level, incoming flows are classified, per-flow aggregation buckets are generated and updated on a per-packet basis, and events are detected. Algorithm 2 presents the corresponding pseudocode. The program maintains two primary data structures: i) a hash table with 2^{17} entries and 12 buckets per entry to mitigate collisions, and ii) a set of circular buffers with 2^{17} entries used to export information to the host kernel, where it is consumed by the Spooler module (see Figure 3). Prior work [25] has shown that, on this hardware platform, 2^{16} entries are sufficient to prevent queue overflows and packet loss; we conservatively provision a larger table to accommodate higher concurrency.

The main function, `save_in_hash()`, is invoked by the P4 pipeline after INT metadata has been extracted from a packet. At this stage (line 3), the flow identifier is computed from the packet headers, the hash key is constructed, and

the corresponding position in the hash table is determined. Depending on the configured time gap, the flow state is either initialized or updated. If a matching bucket is found but must be replaced due to flowlet expiration, its contents are copied into the circular buffers for subsequent export by the Spooler. The hash table entry is then initialized or updated with the newly extracted metadata fields (e.g., Node ID, Ingress timestamp, Queue occupancy).

In addition, per-node INT metrics are calculated. For each node contributing metadata to the packet, the aggregated values (i.e., averages and last-observed values) are updated. Current measurements are compared against previous values to detect deviations or out-of-range conditions. Additionally, it also computes global flow-level metrics, such as end-to-end latency, when sufficient metadata is available. Detected events are immediately exported through dedicated circular buffers.

An additional function converts hardware timestamps from CPU ticks to nanoseconds. Ticks increase every 16 ME clock cycles at 633 MHz and are transformed into nanoseconds for consistency. These values are relative to firmware load time; therefore, the host exports its wall-clock time at initialization. During preprocessing, the offset between this reference and the stored timestamps is computed, enabling alignment with external logs and measurements.

In addition to the operations triggered by `save_in_hash()`, the function `export_flowlets()` (line 14 in Algorithm 2) executes asynchronously. This routine periodically scans the hash table to identify expired flowlets, exports their aggregated state, and releases the corresponding entries. The complete implementation is available at [33].

Algorithm 2 INT Sink Micro-C Pipeline

Input: Packet
Output: Packet with INT removed

1: **function** SAVE_IN_HASH(h, P)
2: Get Hash Table HT
3: Get HT entry with key P $(IP_s, IP_d, s_port, d_port, protocol)$, e
4: Locate flow bucket or free bucket b
5: **if** Bucket existed AND $e_{last_packet_ts} - timenow$ >time_gap **then**
6: Offload flow to buffers and clean entry
7: **end if**
8: CHECK_AND_NOTIFY_EVENTS(e, P)
9: Update: timestamp_first_packet, timestamp_last_packet, metrics per-node (last and averages), count_packets
10: **end function**

11: **function** CHECK_AND_NOTIFY_EVENTS(e, P)
12: Offload to event to buffers
13: **end function**

14: **function** EXPORT_FLOWLETS(b) (async)
15: Get Hash Table HT
16: **for** each $h \in HT$ **do**
17: **for** each Bucket $b \in h$ **do**
18: Extract h
19: **if** $e_{last_packet_ts}$ - timenow>time_gap **then**
20: Offload flow and clean entry
21: **end if**
22: **end for**
23: **end for**
24: **end function**

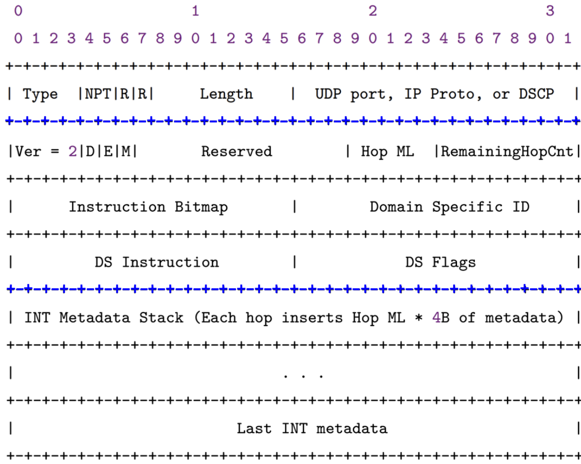


Fig. 4: INT-MD shim and metadata headers followed by the metadata stack.

Kernel Space. At the kernel level, two modules are active (see Figure 3). The Spooler module reads the circular buffers containing aggregated metadata on completed flows and events that must be reported immediately. The Trace Collector module gathers host-level metrics for outgoing flows and aggregates this information using the same flowlet definition employed by the INT Sink.

Spooler. It is responsible for interacting with the sNIC and draining the circular buffers exported by the sNIC firmware. It implements a classical bounded producer–consumer pattern in shared memory.

For terminated flowlets, the design employs a single global queue served by a dedicated consumer thread. Producer threads continuously read the sNIC circular buffers and enqueue valid buckets into the Spooler queue for processing. In contrast, event handling follows a slightly different design: one Spooler instance is created per event buffer (as event buffers are exclusive). This approach enables parallel event processing, avoids contention, and supports low-latency, real-time notification. The Spooler's organization into dedicated or general-purpose threads is modular, allowing scheduling strategies to be modified without requiring changes to the INT Sink logic running on the sNIC.

Toward the user space, the Spooler interacts with the Exporter Agent. To guarantee continuous, non-blocking writes, the consumer thread does not append to a single growing file. Instead, it organizes the output into dynamically managed on-disk segments. This design ensures that the Exporter Agent processes only fully written files, thereby avoiding race conditions and partially generated records. The implementation further incorporates timestamp management and temporal alignment mechanisms, as previously described, along with orderly

shutdown procedures and robustness mechanisms required for reliable operation in production environments.

Trace Collector. It is a background module that captures execution traces without requiring modifications to application code. It records three complementary categories of traces:

1) Stream Traces. Rather than capturing full packets, this module uses PF_RING to extract selected header metadata per packet, including source and destination addresses, packet size, and arrival timestamp, reducing memory overhead while preserving the information needed for flow-level analysis.
2) State Traces. These traces provide operating system–level statistics for a given process, such as virtual memory consumption and CPU utilization. Such metrics are available through the Linux `/proc` filesystem. The Trace Collector is configured with the target application or workload and periodically polls the relevant */proc* entries to obtain the required statistics. To ensure consistency with the flowlet abstraction adopted in our system, the polling interval is configured to be shorter than the defined flow time gap.
3) Micro-level Traces. These traces correspond to fine-grained function-call events occurring during the execution of a monitored process. They are captured using eBPF, which supports event-driven instrumentation within the Linux kernel. Through eBPF hooks attached to selected system calls and user-space library functions (e.g., `read`, `malloc`), the collector extracts relevant event information at runtime. The specific set of monitored events is described in detail in Section III-C.

All these traces are aggregated per flow and made available to the Exporter Agent at the user space.

C. Telemetry Signals Supporting Congestion Control

Here, we present two functional use cases developed to support congestion control in DCNs. With INTFusion, the final aggregation of network and host telemetry is performed at the Centralizer (see Figure 2). At this stage, pre-processed INT sink outputs and host-level traces are fused into unified per-flow records, enabling end-to-end visibility across the network and the application stack.

INTFusion provides two complementary mechanisms to extract information relevant to congestion control: direct signaling and inference through correlation. Direct signaling is implemented by generating explicit events at the sNIC level, allowing latency-critical conditions to be reported immediately. In contrast, inference-based insights are obtained by correlating network and host traces at the Centralizer, enabling the derivation of higher-level metrics that are not explicitly reported by any single telemetry source.

TCP Incast Detection. It is implemented through the direct signaling mechanism for an accurate and timely detection. To this end, the INT Source configures the instruction bitmap to request the collection of egress queue occupancy at transit nodes. Consequently, each INT-capable switch allocates a 4-byte metadata field in the INT stack to record the instantaneous queue size at the time the packet traverses the device.

Upon reception at the destination host, the INT Sink extracts and aggregates the reported queue occupancy values on a per-flow and per-node-ID basis. This aggregation allows tracking queue dynamics at specific bottlenecks associated with individual flows or groups of flows. Additionally, whenever the reported queue occupancy for a given node ID exceeds 60% of the corresponding switch buffer capacity, the Sink generates an event and pushes it into the dedicated circular buffers for immediate export to the host. This mechanism enables prompt detection and generates a telemetry signal that can be used by congestion-control mechanisms.

Although not in real time, TCP incast can also be identified at the Centralizer by correlating host-level traces that reveal multiple concurrent flows to the same receiver within a short time window. Combined with network-level telemetry, this enables differentiation of transient incast episodes from other congestion events.

Flow Size Estimation. Flow size estimation is relevant for congestion-control and scheduling algorithms [34], as it enables systems to anticipate the resource demands of flows. The objective is to predict flow size early enough to adapt congestion control or scheduling accordingly.

INTFusion leverages correlated host- and network-level traces to construct feature-rich per-flow datasets suitable for training ML models, extending existing approaches [34], [35]. Beyond providing an automated pipeline for dataset construction, our architecture enriches the feature space, surpassing the limited feature sets employed in prior work.

Table I summarizes the traces collected at each host. The Trace Collector aggregates Stream, State, and Micro-level traces for outgoing flows, whereas the INT Sink aggregates network telemetry for incoming flows. To unify network and host telemetry, the Centralizer collects these traces from all hosts and performs flow-level matching between outgoing traces at the source and corresponding incoming traces at the destination. Because our flow definition incorporates the packet gap time, unification is non-trivial. First, reliable timestamp alignment requires clock synchronization across hosts (e.g., via NTP), a common practice in production data centers.

To perform cross-host matching, we rely on the timestamps of the first packet in each flow. Specifically, we align the timestamp of the first packet in the outgoing trace with that of the first packet in the corresponding incoming trace, adjusted by subtracting the cumulative hop latencies reported by INT for that flow. This latency-compensated alignment enables accurate pairing of host- and network-level records into a unified end-to-end flow trace.

IV. Evaluation

Setup. We implemented a prototype of the INTFusion architecture on a local cluster environment. The end hosts were deployed on two commodity PCs, each equipped with an AMD Ryzen 7 7700 8/16 processor, 64 GB of RAM, and a Netronome Agilio CX 4000 sNIC with dual 10 GbE ports. The INT transit nodes were emulated using Mininet P4 switches. The Centralizer was implemented on top of Elasticsearch

TABLE I: Features table

Trace Type	Direction	Features
Stream	Outgoing	IP src, IP dst, port src, port dst, packet size
State	Outgoing	disk read and write bytes, vmemory size and data bytes, available and resident memory
Micro-level	Outgoing	sendmsg, cpu and gpu allocations, tcp sendmsg, write, sendto, kcache
Network	Incoming	Egress and ingress timestamps, hop latency, queue occupancy (Per NodeID)
Aggregation	-	Flow ID, first and last packet timestamps, averages, min, and max values

and deployed on a production-grade server. For workload generation, we used a publicly available large-scale real-world Internet traffic dataset [36] and a distributed machine-learning cluster to generate real traffic.

Objective. In addition to validating all architectural components, our evaluation focuses on the performance of the hosts on which the core components of INTFusion are deployed.

For the Centralizer and its communication with the hosts, we relied on a production-grade data ingestion and storage stack capable of handling large data volumes. Consequently, our evaluation of this component was limited to data integrity and correctness tests rather than scalability stress testing. To this end, we employed a synthetic traffic generator capable of producing high-rate traffic with predetermined metrics. The telemetry collected and centralized by the system was then compared against the ground-truth values generated by the traffic source, allowing us to verify the correctness and completeness of the exported traces.

Throughput and Latency. We focus on demonstrating INTFusion ability to process information at line rate without adding substantial overhead. In this regard, we can observe in Figure 5b that the latency added to the sNIC for each INTFusion pipeline operation is limited to a few microseconds, with the Update operation being the most expensive due to the calculation of aggregated metrics (i.e., averages using subtraction, addition, multiplication, and division).

On the other hand, Figure 5a shows that the number of nodes appending INT metadata has a significant impact on the sNIC's packet-processing capacity. In fact, we considered up to 5 INT nodes in the architecture of Figure 1, given that the radius of a typical DCN topology does not exceed this number. It can be seen that processing capacity decreases with both the number of INT hops and the number of added instructions, indicating that the management process should carefully select the monitoring parameters at the INT Source.

Host CPU overhead. The host-side implementation comprises three CPU-consuming components: the Trace Collector, the Spooler, and the Exporter. As described in Section III-A, the Spooler employs dedicated threads to drain the flow traces exported by the INT sink. In our implementation, the number of such threads is configurable in the range 1 to 8, directly affecting the CPU utilization of this module.

To quantify CPU overhead, we collected discrete-time samples while the host simultaneously received packet traces from a reference dataset and transmitted traffic generated by a real distributed machine learning training workload. As shown in Figure 5c, the Spooler actively utilizes its dedicated threads to drain the sNIC circular buffers. To enable real-time event notification, threads assigned to event buffers operate in polling mode, each consuming approximately one full CPU core (i.e., close to 100% utilization).

In contrast, the Trace Collector exhibits moderate CPU consumption, as it relies on efficient packet capture and kernel instrumentation mechanisms. The Exporter's CPU usage depends on configurable parameters (e.g., batching and flushing intervals), which are designed to prevent sustained high CPU utilization. Figure 5c reports the measured CPU overhead of the three components under these conditions.

The performance evaluation of INTFusion indicates a non-negligible yet bounded CPU overhead on end hosts, representing a reasonable trade-off for enabling the advanced telemetry-driven use cases for which the system is designed.

We acknowledge the limitations of the current testbed, particularly regarding the capacity and bandwidth of the sNICs, and are actively working to address them. Specifically, we are developing a larger-scale deployment aimed at demonstrating the benefits of fine-grained telemetry for congestion control.

V. Conclusion

In this paper, we presented INTFusion, a unified monitoring architecture that integrates in-network telemetry with host-level observability at per-flow granularity. By offloading the INT source and sink functions to programmable sNICs at end hosts, our design achieves line-rate telemetry processing while preserving host CPU resources.

INTFusion correlates INT-derived network measurements with host-level traces, producing unified per-flow records that enable explicit detection of congestion-related phenomena and support higher-level analytics at a central aggregator. Moreover, the resulting feature-rich datasets provide a foundation for predictive mechanisms, such as flow-size estimation, supporting proactive congestion control.

Finally, INTFusion demonstrates that edge-terminated INT, combined with host-level tracing, provides a practical foundation for next-generation telemetry in DCNs.

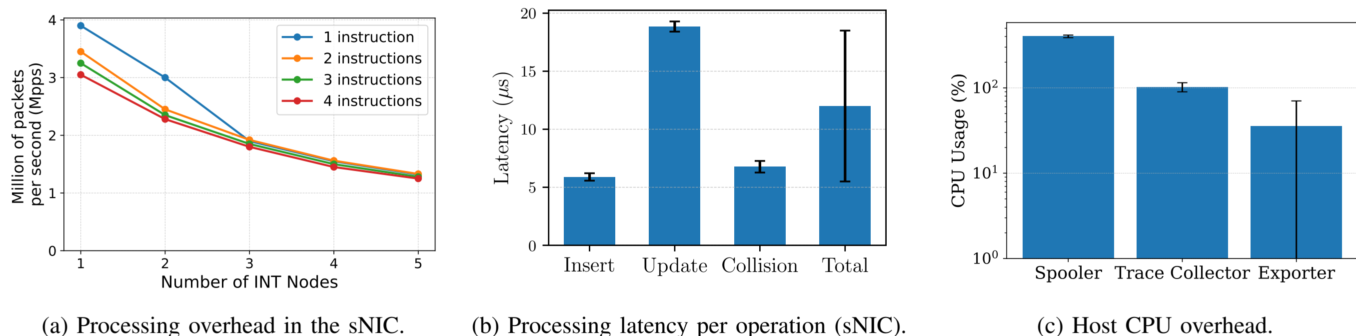
References

[1] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim, "Profiling network performance for multi-tier data center applications," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USA: USENIX Association, 2011, p. 57–70.

[2] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, ser. SIGCOMM '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 303–314.

[3] L. Alberro and E. Grampin, "Transport protocols in the data center: Understanding proposals and research challenges," *Computer Networks*, vol. 274, p. 111793, 2026.

[4] J. A. Marques and L. P. Gaspary, "Advancing Network Monitoring and Operation with In-band Network Telemetry and Data Plane Programmability," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–6.



(a) Processing overhead in the sNIC. (b) Processing latency per operation (sNIC). (c) Host CPU overhead.

Fig. 5: Overhead in the host stack.

[5] X. Xiong *et al.*, "An Integrated Solution for High-efficiency In-band Network Telemetry," in *Proceedings of the 8th Asia-Pacific Workshop on Networking*. New York, NY, USA: Association for Computing Machinery, 2024, p. 115–121.

[6] V. Sekar *et al.*, "Revisiting the case for a minimalist approach for network flow monitoring," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 328–341.

[7] R. Ben Basat *et al.*, "Constant Time Updates in Hierarchical Heavy Hitters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 127–140.

[8] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: a better NetFlow for data centers," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. USA: USENIX Association, 2016, p. 311–324.

[9] M. Yu, "Network telemetry: towards a top-down approach," *SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 1, p. 11–17, Feb. 2019.

[10] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, L. J. Wobker *et al.*, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, vol. 15, 2015, pp. 1–2.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014.

[12] P4 Language Consortium, "P4 Applications Working Group repository," https://github.com/p4lang/p4-applications/tree/master/docs, 2025, [Online; accessed 2026 Feb].

[13] Netronome, "Agilio smartnics," https://netronome.com/agilio-smartnics/, 2026, [Online; accessed 2026 Feb].

[14] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications," *ACM Comput. Surv.*, vol. 54, no. 4, May 2021.

[15] S. Kianpisheh and T. Taleb, "A Survey on In-Network Computing: Programmable Data Plane and Technology Specific Applications," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 701–761, 2023.

[16] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple network management protocol," Internet Requests for Comments, RFC Editor, RFC 1067, August 1988.

[17] B. Claise, "Cisco Systems NetFlow Services Export Version 9," Internet Requests for Comments, RFC Editor, RFC 3954, October 2004.

[18] P. Phaal *et al.*, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks," Internet Requests for Comments, RFC Editor, RFC 3176, September 2001.

[19] T. Zseby, T. Hirsch, and B. Claise, "Packet sampling for flow accounting: Challenges and limitations," in *International conference on passive and active network measurement*. Springer, 2008, pp. 61–71.

[20] F. Brockners, S. Bhandari, D. Bernier, and T. Mizrahi, "In Situ Operations, Administration, and Maintenance (IOAM) Deployment," Internet Requests for Comments, RFC Editor, RFC 9378, April 2023.

[21] J. A. Marques and L. P. Gaspary, "Advancing Network Monitoring and Operation with In-band Network Telemetry and Data Plane Programmability," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–6.

[22] E. Song *et al.*, "INT-label: Lightweight In-band Network-Wide Telemetry via Interval-based Distributed Labelling," in *INFOCOM 2021 IEEE Conference on Computer Communications*. IEEE Press, 2021, p. 1–10.

[23] X. o. Xiong, "An integrated solution for high-efficiency in-band network telemetry," in *Proceedings of the 8th Asia-Pacific Workshop on Networking*, ser. APNet '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 115–121.

[24] A. Kumar *et al.*, "Feasibility of Application Layer Header Parsing in eBPF and P4," in *2024 IFIP Networking Conference (IFIP Networking)*, 2024, pp. 475–481.

[25] Y. Feng, S. Panda, S. G. Kulkarni, K. K. Ramakrishnan, and N. Duffield, "A smartnic-accelerated monitoring platform for in-band network telemetry," in *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN*, 2020, pp. 1–6.

[26] J. Xing *et al.*, "Unleashing SmartNIC Packet Processing Performance in P4," in *Proceedings of the ACM SIGCOMM 2023 Conference*, ser. ACM SIGCOMM '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1028–1042.

[27] J. Vestin, A. Kassler, D. Bhamare, K.-J. Grinnemo, J.-O. Andersson, and G. Pongracz, "Programmable Event Detection for In-Band Network Telemetry," in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, 2019, pp. 1–6.

[28] J.-T. Hinz *et al.*, "TCP's Third Eye: Leveraging eBPF for Telemetry-Powered Congestion Control," in *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, ser. eBPF '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–7.

[29] G. Jereczek, T. Jepsen, S. Wass, B. Pujari, J. Zhen, and J. Lee, "TCP-INT: lightweight network telemetry with TCP transport," in *Proceedings of the SIGCOMM '22 Poster and Demo Sessions*. New York, NY, USA: Association for Computing Machinery, 2022, p. 58–60.

[30] T. Osiński and C. Cascone, "Achieving End-to-End Network Visibility with Host-INT," in *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '21. New York, NY, USA: Association for Computing Machinery, 2022, p. 140–143.

[31] C. Puttlitz *et al.*, "P4NetIntel: End-to-End Network Telemetry with eBPF and XDP," in *2024 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2024, pp. 1–6.

[32] Y. Li *et al.*, "Hpcc: high precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 44–58.

[33] L. Alberro *et al.*, "INTFusion Repository," https://github.com/leoalb/INTFusion, 2026.

[34] V. undefinedukić *et al.*, "Is advance knowledge of flow sizes a plausible assumption?" in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. USA: USENIX Association, 2019, p. 565–580.

[35] S. M. Hosseini, S. Sadrhaghighi, and M. Ghaderi, "Flow Size Prediction with Short Time Gaps," in *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops*, 2024, pp. 01–07.

[36] W. Jiang *et al.*, "A real network environment dataset for traffic analysis," *Scientific Data*, vol. 12, no. 1, p. 756, 2025.