DeltaINT: Toward General In-band Network Telemetry with Extremely Low Bandwidth Overhead

Siyuan Sheng[1,3], Qun Huang[2], and Patrick P. C. Lee[3]
[1]*University of Chinese Academy and Sciences* [2]*Peking University* [3]*The Chinese University of Hong Kong*

***Abstract*—In-band network telemetry (INT) enriches network management at scale through the embedding of complete device-internal states into each packet along its forwarding path, yet such embedding of INT information also incurs significant bandwidth overhead in the data plane. We propose DeltaINT, a general INT framework that achieves extremely low bandwidth overhead and supports various packet-level and flow-level applications in network management. DeltaINT builds on the insight that state changes are often negligible at most time, so it embeds a state into a packet only when the state change is deemed significant. We theoretically derive the time/space complexities and the bounds of bandwidth mitigation for DeltaINT. We implement DeltaINT in both software and P4. Our evaluation shows that DeltaINT reduces up to 93% of INT bandwidth, and its deployment in a Barefoot Tofino switch incurs limited hardware resource usage.**

I. Introduction

In-band Network Telemetry (INT) [32] has become a critical network management paradigm for real-time, fine-grained, and network-wide monitoring of flows and network devices. At a high level, INT enables network devices (called *nodes*) to collect various device-internal states (e.g., device ID, link utilization, queue occupancy) from the data plane, and embed the collected INT information as packet headers into packets (e.g., normal data packets or special probe packets) in each node along the packet forwarding path [32]. Such INT information can be collected by the destinations of the forwarding path for aggregate analysis, at packet-level [12], [26], [36] or flow-level [8], [14], [23] granularities. In particular, INT aims for *generality* and supports various applications in network management, such as load balancing [6], latency profiling [21], microburst detection [19], congestion control [8], [24], and network troubleshooting [12], [17]; it is also deployed in production data centers [24].

A drawback of the original INT framework [32] is that it embeds complete states into each packet along its forwarding path on a per-node basis. The packet size, and hence the bandwidth overhead, grow linearly with the path length. Such bandwidth overhead degrades network performance, as it not only reduces the effective bandwidth for network flows, but also increases the likelihood of IP-level fragmentation if the packet size exceeds the Maximum Transmission Unit (MTU).

Recent studies (e.g., [8], [15], [19], [22], [26], [29], [31], [33], [35]) have proposed different INT bandwidth mitigation approaches. A straightforward approach is *sampling* [8], [22], [29], [31], which embeds INT information to only a subset of sampled packets. However, sampling inherently has *slow convergence*, since it needs to collect sufficient packets for accurate measurement decisions (e.g., PINT [8] needs to sample enough packets to collect all device IDs in path tracing). Also, sampling fails to support fine-grained per-packet measurement, thereby compromising the generality of the original INT design for various applications. As we argue in §VI, existing studies often sacrifice generality for INT bandwidth mitigation.

We design DeltaINT, a novel INT framework that aims to support general applications with extremely low bandwidth overhead. DeltaINT builds on two main features: (i) the tracking of per-node state changes across adjacent packets and (ii) the stateful programmability in the data plane. Our insight is that adjacent packets traversing the same device often observe negligible (or even non-existent) state changes at most time (while the abnormal events are relatively rare albeit possible). By recording stateful information and tracking state changes in the programmable data plane, DeltaINT embeds INT information into packets only when there exist significant state changes. This allows DeltaINT to save significant INT bandwidth, without compromising the measurement accuracy. DeltaINT aims for three design goals: (i) *generality*, in which it supports various packet-level and flow-level applications in network management; (ii) *convergence*, in which it monitors every traversed packet in the programmable data plane and collects real-time telemetry data; and (iii) *compatibility*, in which it is compatible with existing INT techniques, such as path planning [26] and value approximation [8], for further bandwidth savings. We summarize our contributions as follows.

- We design DeltaINT, a general INT framework that achieves extremely low bandwidth overhead and aims for generality, convergence, and compatibility.
- We derive the theoretical properties of DeltaINT on its bandwidth mitigation guarantees.
- We conduct software simulations on DeltaINT for various applications and compare it with INT-Path [26] and PINT [8]. DeltaINT significantly reduces the average per-packet bit overhead (e.g., by up to 93% in gray failure detection compared to INT-Path [26]), while showing fast convergence and being compatible with existing INT techniques.
- We provide P4 hardware implementation for DeltaINT and show that it has limited resource usage when it is deployed in a Barefoot Tofino switch [1].

We open-source our DeltaINT prototype in both software and P4 at **http://adslab.cse.cuhk.edu.hk/software/deltaint**.

II. Background and Motivation

A. Basics of INT

Figure 1 shows the INT framework. In the data plane, each node not only supports basic network functions such

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

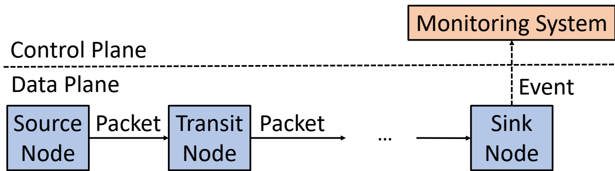


Figure 1: INT framework.

as packet forwarding, but is also programmable to support network management applications. Specifically, each node can be programmed to collect multiple device-internal *states*, each of which corresponds to a type of statistics, and embeds the states into a traversed packet. Each packet carries INT information along its forwarding path, including an 8-byte *metadata header* that specifies the INT request details, as well as a variable-length *metadata stack* that embeds multiple states; each state is encoded in 4 bytes by default.

Consider a packet p that traverses a path of k nodes, denoted by $s_1, s_2, \cdots, s_k$. Let $v(p, s_i)$ be the latest state in s_i when p traverses s_i. Each node along the forwarding path of p can belong to one of the three roles: *source*, *transit*, and *sink* nodes. The source is the first node s_1 in the forwarding path of p. It inserts an INT header into p, including the INT instructions that specify which states are to be collected (e.g., device ID, queue occupancy, and access latency). It then pushes $v(p, s_1)$ into the metadata stack of p and forwards p to the next node. The transit nodes are the intermediate nodes $s_2, s_3, \cdots, s_{k-1}$ in the forwarding path of p. Upon receiving p from the previous node, each transit node s_i $(2 \leq i \leq k-1)$ parses the INT header of p, pushes $v(p, s_i)$ into the metadata stack of p, and forwards p to the next node. The sink is the last node s_k in the forwarding path of p. After receiving p, it first pushes $v(p, s_k)$ into the metadata stack of p. It then extracts the metadata header and metadata stack from p. It finally reports the extracted information, as an *event*, to the control plane.

In the control plane, a logically centralized monitoring system is deployed to collect events reported by all sinks. It extracts the statistics from the events for further collective analysis. It is also responsible for configuring the role of each node in the data plane.

The original INT framework incurs significant bandwidth overhead. Each packet is inserted with the complete device-internal states from each node along its forwarding path, so its packet size grows linearly with the path length. For example, consider a 5-node fat-tree data center topology that uses INT to track three states, including the device ID, ingress port, and egress port. Thus, each packet carries a maximum of 68 bytes of INT information (i.e., 8 bytes from the metadata header, and 12 bytes for the three states from each of the five nodes), accounting for at least 4.53% of the packet size under the MTU of 1,500 bytes in the Ethernet. If the packet size grows beyond the MTU, IP-level fragmentation will occur.

B. Applications

In this work, we consider four families of applications in network management, in which the first one is based on the original INT framework, while the last three are based on the aggregation of statistics [8]. In the following, we provide the definitions, as well as a representative application that is used by existing INT solutions, for each family of applications.

Per-packet-per-node monitoring. It corresponds to the original INT framework, by collecting all per-node states into each packet along the packet forwarding path. Specifically, for each packet p and its traversed path of nodes $s_1, s_2, \cdots, s_k$, the collected telemetry data is $\{v(p, s_1), v(p, s_2), \cdots, v(p, s_k)\}$.

One application is the detection of *gray failures* [13], which refer to the components that remain functional (i.e., bypassing failure detection) but suffer from performance anomalies. To detect gray failures, INT-Detect [18] broadcasts probes to collect real-time statistics of all nodes in a network, such that the probes cover all possible routing paths for identifying any unavailable path.

Per-packet aggregation. It summarizes the per-node states for each packet with some *aggregation function* (e.g., min, max, sum, or product). Specifically, for each packet p, its traversed path of nodes $s_1, s_2, \cdots, s_k$ and an aggregation function $G(.)$, the collected telemetry data is $G(v(p, s_1), v(p, s_2), \cdots, v(p, s_k))$.

One application is *congestion control*. Packet queueing, and hence network congestion, become prevalent in data center networks due to their high-bandwidth nature [24]. To support fine-grained congestion control, HPCC [24] adopts INT to collect aggregate statistics (e.g., maximum link utilization) from each node for sending-rate adaptation.

Static per-flow aggregation. It collects the per-node states for each flow, assuming that the forwarding path and the states of each node for a given flow are static. Let $V(x, s) = v(p, s)$ be the state at node s for any packet p of flow x. Then the collected telemetry data is $\{v(x, s_1), v(x, s_2), \cdots, v(x, s_k)\}$.

One application is *path tracing* [17], which discovers the forwarding path traversed by a flow for path-level performance control (e.g., load balancing [23] and network debugging [30]). To trace the path of a flow, PathDump [30] embeds the device ID into each traversed packet.

Dynamic per-flow aggregation. It summarizes the per-node states for each flow with the aggregation of statistics (e.g., median, quantile, or frequency). Given an aggregation function $G(.)$, let $V(x, s) = G(v(p_1, s), v(p_2, s), \cdots)$ be the aggregate statistics of flow x for all its packets $p_1, p_2, \cdots$ at node s. If flow x traverses a path of nodes $s_1, s_2, \cdots, s_k$, the collected telemetry data is $\{V(x, s_1), V(x, s_2), \cdots, V(x, s_k)\}$.

One application is *latency measurement*, which collects latency statistics (e.g., average, median, or 99th percentile) for flow-level performance monitoring, such as anomaly detection [16] and delay monitoring [10]. For efficient statistics collection, PINT [8] collects per-node latency statistics carried by a packet into a compact data structure based on the quantile sketch [20].

C. Our Solution

DeltaINT is an INT framework that achieves extremely low bandwidth overhead and aims for generality, convergence, and compatibility (§I). Recall that a node collects multiple

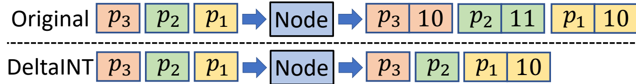


Figure 2: Motivating example.

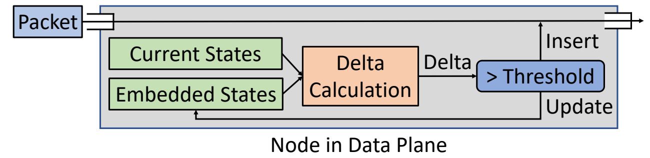


Figure 3: DeltaINT per-node architecture.

device-internal states (§II-A). DeltaINT monitors the *delta* of each state in a node, defined as the change between the state that has been provided by the node for each traversed packet (called the *current state*) and the state that has been most recently embedded into a packet (called the *embedded state*). Our key observation is that the delta is often “negligible” (i.e., within some pre-specified threshold) at most time in typical applications. For example, the queue occupancy and access latency remain relatively stable unless a microburst occurs [19], [34]; the link utilization remains stable unless under network congestion [24]; some static states, such as device ID and ingress/egress ports, are well-defined and remain unchanged. Under DeltaINT, each node embeds a state to only a subset of traversed packets, thereby reducing the INT bandwidth usage.

Figure 2 gives an example on how DeltaINT reduces the INT bandwidth overhead. Suppose that there are three packets, denoted by p_1, p_2, and p_3, entering a node in order, and let the current states of the node upon receiving the three packets are 10, 11, and 10, respectively. The original INT framework always embeds the current state into each packet. In contrast, DeltaINT embeds the state into a packet only if the delta exceeds a pre-specified threshold, say one. In this case, DeltaINT first embeds the current state 10 into p_1, and updates the embedded state as 10. Since the deltas for the current states of p_2 and p_3 are one and zero, respectively, DeltaINT does not embed the current states, and hence reduces the INT bandwidth usage by roughly two-thirds (note that DeltaINT additionally adds a bitmap into each traversed packet to track which states are carried; see §III for details).

III. DeltaINT Design

A. Architecture

Figure 3 shows the per-node architecture in the data plane in DeltaINT. For each traversed packet, DeltaINT first identifies all current states provided by the node. For each state, it calculates the delta between the current state and the embedded state. Only if the delta exceeds a predefined threshold, DeltaINT embeds the current state into the packet and updates the embedded state with the current state. Note that even though the delta of the current state is small in two adjacent packets, if the variations accumulated across a series of packets are significant and cause the delta of the current state to exceed the threshold, DeltaINT can still embed the current state. If the sink receives a packet, it extracts the INT information and sends an event to the monitoring system in the control plane, as in the original INT framework.

The current states are directly provided by the node without using its stateful memory. Specifically, upon receiving a packet, the node retrieves its device-internal states as the current states. For example, the node continuously updates its local timer, and loads the current time as the ingress timestamp when a packet arrives. Note that the current states are only used for processing the currently traversed packet, and they do not need to be recorded in the stateful memory of the node.

In contrast, the embedded states are used for stateful tracking for all traversed packets. DeltaINT now records the embedded states on a per-flow basis in the stateful memory of each node, so as to support both packet-level and flow-level INT. However, the data-plane resources are limited. For example, commodity switches only have 10-20 MB SRAM [25] and restrain the number of operations for line-rate processing. In this work, we explore *sketch-based techniques*, which store approximate information in a *sketch* data structure with fixed memory and limited computations, at the expense of bounded errors. A key challenge is to design a general sketch that supports various INT applications. We detail our solution in §III-B.

DeltaINT deploys a monitoring system in the control plane. It in essence has the same design as the original INT framework, except for dynamic flow-level aggregation (§III-D).

B. Framework Design

Data structure. Recall that each node in DeltaINT maintains a sketch for tracking the embedded states on a per-flow basis with limited stateful memory (§III-A). Figure 4 shows the sketch layout. The sketch A is a two-dimensional array composed of d rows with w *buckets* each. Each bucket records the flow identifier *flowkey* and the embedded states *embstates*. Let $A[i][j]$ be the j-th bucket in the i-th row, where $1 \leq i \leq d$ and $1 \leq j \leq w$. When a node receives a packet with flowkey x, it maps x into a bucket $A[i][h_i(x)]$ of each i-th row with an independent hash function $h_i(.)$, where $1 \leq i \leq d$.

For each packet that carries INT information, it comprises multiple *entries*, each of which includes a *bitmap* and the states being embedded. The bitmap tracks which state is being carried. Specifically, the bitmap is initialized with a number of bits equal to the number of states of interest (pre-specified by the control plane). Each bit is equal to one if the corresponding state is embedded, or zero otherwise.

Primitives. DeltaINT builds on four primitives for its framework design to support various applications; see Algorithm 1.

- StateLoad (Lines 1-7): It loads the embedded states for flowkey x from the sketch. DeltaINT locates the mapped bucket $A[i][h_i(x)]$ from each row, for $1 \leq i \leq d$. If x is matched, DeltaINT returns the embedded states from the bucket. Note that more than one bucket in different rows may match x. Nevertheless, each matched bucket must have the same embedded state based on our Update algorithm (see details below), so we simply choose the first matched bucket. If no bucket matches x, nothing is returned (i.e., the embedded states of x are not recorded).

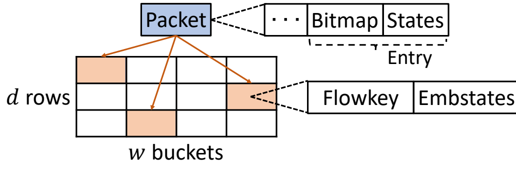


Figure 4: Sketch data structure for tracking embedded states.

- DELTACALC (Lines 8-14): It performs delta calculation for each state based on the absolute difference between the current state *curstate* and the embedded state *embstate* (Line 9). It returns true if the delta does not exceed the pre-specified threshold ϕ, or false otherwise.
- STATEUPDATE (Lines 15-20): It updates the flowkey x and the relevant embedded states of the sketch.
- METADATAINSERT (Lines 21-28): It inserts INT information, including the bitmap and the states with non-negligible deltas, into packet p.

Update algorithm. Algorithm 1 shows how the DeltaINT framework works in a node. DeltaINT calls the UPDATE procedure for each received packet p (Lines 29-52). Initially, the UPDATE procedure first extracts a flowkey x from p (Line 30) and loads the current states *curstates* from the node (Line 31). It also calls STATELOAD to load the embedded states into *embstates* from the stateful memory (Line 32). It further initializes all bits in *bitmap* as ones, meaning that each state is to be embedded by default (Line 33).

DeltaINT proceeds to check if each current state needs to be embedded. If *embstates* exists, DeltaINT calls DELTACALC to calculate the delta for each state (Lines 36-38). If the delta is within the threshold (i.e., DELTACALC returns true), DeltaINT resets the corresponding bit in *bitmap* to zero. It then updates the embedded states in all d hashed buckets in the sketch (Lines 41-50) if the bit in *bitmap* is one (i.e., the delta is non-negligible and the current state needs to be embedded) or if the hashed bucket does not store the current flowkey x (i.e., overwriting the embedded states with x's when a hash collision happens). Finally, DeltaINT calls METADATAINSERT to embed INT information into packet p (Line 51).

Update example. Figure 5 provides an example of the UPDATE procedure. Suppose that the sketch A has $d=2$ rows with $w=3$ buckets each. It tracks two states, with thresholds 0 and 2, respectively. Consider two flows x_1 and x_2 with two packets each. The UPDATE procedure works as follows.

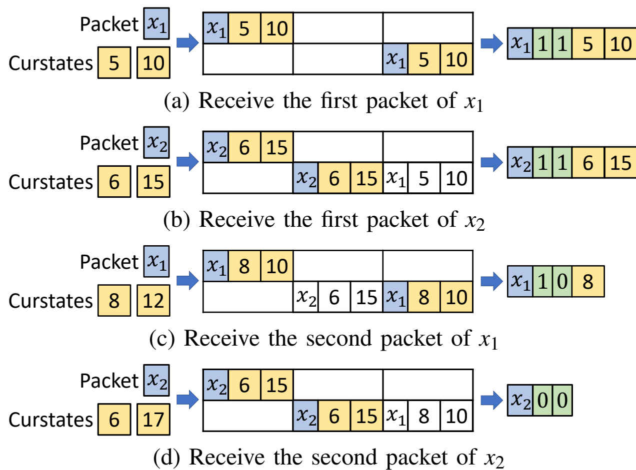
- First, DeltaINT receives the first packet of x_1 with the current states of 5 and 10 (Figure 5a). It locates two buckets, say $A[1][1]$ and $A[2][3]$. Since no bucket matches x_1, DeltaINT directly stores $\langle x_1, 5, 10\rangle$ in each of the hashed buckets. It also embeds the bitmap with bits $\langle 1,1\rangle$ and both current states $\langle 5,10\rangle$ into the packet.
- Second, DeltaINT receives the first packet of x_2 with the current states of 6 and 15 (Figure 5b). Suppose that the hashed buckets are $A[1][1]$ and $A[2][2]$. It updates the buckets by $\langle x_2, 6, 15\rangle$ (note that $A[1][1]$ is overwritten with x_2) and embeds the bitmap with bits $\langle 1,1\rangle$ and both current states $\langle 6,15\rangle$ into the packet.

Algorithm 1 Update Algorithm of DeltaINT Framework

Input: Packet p; Thresholds Φ

1: **function** STATELOAD(x)
2:     **for** $i = 1, 2, ..., d$ **do**
3:         **if** $A[i][h_i(x)].flowkey = x$ **then**
4:             **return** $A[i][h_i(x)].embstates$
5:         **end if**
6:     **end for**
7: **end function**
8: **function** DELTACALC($curstate, embstate, \phi$)
9:     $delta \leftarrow |curstate - embstate|$
10:     **if** $delta \leq \phi$ **then**
11:         **return** true
12:     **end if**
13:     **return** false
14: **end function**
15: **function** STATEUPDATE($row, x, idx, state$)
16:     $A[row][h_i(x)].embstates[idx] \leftarrow state$
17:     **if** $A[row][h_i(x)].flowkey \neq x$ **then**
18:         $A[row][h_i(x)].flowkey \leftarrow x$
19:     **end if**
20: **end function**
21: **function** METADATAINSERT($bitmap, curstates, p$)
22:     Insert *bitmap* into p
23:     **for all** $1 \leq idx \leq \#curstates$ **do**
24:         **if** $bitmap[idx] = 1$ **then**
25:             Insert $curstates[idx]$ into p
26:         **end if**
27:     **end for**
28: **end function**
29: **procedure** UPDATE(p)
30:     Extract a flowkey x from p
31:     Load *curstates* from the deployed node
32:     $embstates \leftarrow$ STATELOAD(x)
33:     Initialize *bitmap* with one
34:     **if** *embstates* exists **then**
35:         **for all** $1 \leq i \leq \#curstates$ **do**
36:             **if** DELTACALC($curstates[i], embstates[i], \Phi[i]$) **then**
37:                 $bitmap[i] \leftarrow 0$
38:             **end if**
39:         **end for**
40:     **end if**
41:     **for** $i = 1, 2, ..., d$ **do**
42:         $notrecorded \leftarrow A[i][h_i(x)].flowkey \neq x$
43:         **for all** $1 \leq idx \leq \#curstates$ **do**
44:             **if** $bitmap[idx] = 1$ **then**
45:                 STATEUPDATE($i, x, idx, curstates[idx]$)
46:             **else if** *notrecorded* **then**
47:                 STATEUPDATE($i, x, idx, embstates[idx]$)
48:             **end if**
49:         **end for**
50:     **end for**
51:     METADATAINSERT($bitmap, curstates, p$)
52: **end procedure**

- Third, DeltaINT receives the second packet of x_1 with the current states of 8 and 12 (Figure 5c). It loads the embedded states of 5 and 10 from bucket $A[2][3]$, so it calculates the deltas as 3 and 2, respectively. Note that the second state is within the threshold. Thus, DeltaINT updates the hashed buckets by $\langle x_1, 8, 10\rangle$ and only inserts the first current state 8 with a bitmap with bits $\langle 1,0\rangle$ into the packet.
- Finally, DeltaINT receives the second packet of x_2 with the



(a) Receive the first packet of x_1

(b) Receive the first packet of x_2

(c) Receive the second packet of x_1

(d) Receive the second packet of x_2

Figure 5: Update example of DeltaINT framework.

current states of 6 and 17 (Figure 5d). It loads the embedded states of 6 and 15 from bucket $A[2][2]$ and calculates the deltas as 0 and 2. Since both the current states are within the thresholds, DeltaINT updates the hashed buckets by $\langle x_2, 6, 15 \rangle$ and only inserts a bitmap $\langle 0, 0 \rangle$ into the packet.

C. Discussion of Design Issues

We discuss the potential design issues in DeltaINT, and argue that they have limited impact on DeltaINT.

Hash collisions. Recall that DeltaINT may overwrite a bucket of the sketch if the flowkey of the received packet does not match the recorded flowkey in the bucket, yet the impact of such hash collisions on DeltaINT is limited. If the packets of a flow do not match the recorded flowkey in any bucket, they cannot load the embedded states for delta calculation. In this case, DeltaINT must embed all current states (i.e., the default) and cannot mitigate the INT bandwidth overhead. To mitigate the impact of hash collisions, DeltaINT can allocate more rows (i.e., a larger d) in the sketch, which we can show that the likelihood that at least one bucket holds the embedded states for a flowkey increases exponentially with d (§IV). In fact, DeltaINT works well even with $d = 1$ in our evaluation (§V), as the number of conflicting flows with overlapping life cycles is small in practice.

Sensitivity to delta threshold. We argue that DeltaINT is insensitive to the choice of the delta threshold. For static states (e.g., device ID), they remain unchanged, so a zero threshold can suppress the embedding of static states. For dynamic states (e.g., per-node latency), since they have limited changes at most time, a reasonably small threshold suffices for avoiding the embedding of the states into packets. Our evaluation shows that DeltaINT works well even with a threshold of one for the integer states (32 bits) in practical applications.

Bitmap size. DeltaINT currently includes a bitmap into the packet at each node in the forwarding path, yet the extra INT bandwidth overhead from the bitmap is limited. Recall that the number of bits in the bitmap is equal to the number of states of interest. The original INT framework is designed for at most 16

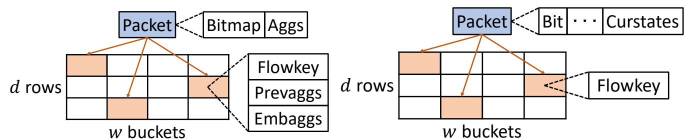


Figure 6: Sketch in per-packet aggregation.

Figure 7: Sketch in static per-flow aggregation.

states [32]. Also, practical applications only have at most three states [8]. Furthermore, as we show in §III-D, for per-packet aggregation, DeltaINT only maintains one bitmap in the packet for all nodes in the forwarding path, while for static per-flow aggregation, only the source inserts a one-bit bitmap into each packet without compromising the measurement accuracy.

Packet fragmentation. Since DeltaINT reduces the INT bandwidth in the data plane, it is less likely to incur IP-level fragmentation than the original INT. To avoid IP-level fragmentation, DeltaINT can also configure a larger maximum transmission unit (MTU) size or limit the maximum number of per-path hops for recording INT information as in the original INT [32], but we do not make this assumption in our design.

D. Fitting DeltaINT into Applications

We explain how we apply the DeltaINT framework into each family of applications described in §II-B.

Per-packet-per-node monitoring. The DeltaINT framework can be directly applied, such that each packet embeds the states with non-negligible deltas at each node in the forwarding path.

Per-packet aggregation. Recall that this family tracks the aggregation function on each state of all nodes traversed by a packet (§II-B). Here, we assume that the aggregation can be incrementally updated by each node in the forwarding path of a packet, such that the aggregation of a state in the current node (or *current aggregate state* in short) can be obtained by aggregating both the state in the current node and the aggregation of the state in the previous node of the forwarding path (or *previous aggregate state* in short). Our assumption works for common aggregation functions, such as min, max, and sum. For other complicated aggregation functions, we place them in the control plane as in dynamic flow-level aggregation (see details below).

DeltaINT shares a similar data structure as in the framework, except with two changes (Figure 6). First, each packet only carries one entry with a bitmap and the aggregate states *aggs* being tracked. The bitmap is used to track which aggregate states are carried by the packet. Note that each node only overwrites the single entry with the current aggregate states instead of inserting a new entry. Second, in addition to the flowkey and the embedded aggregate states *embaggs*, each bucket in the sketch also stores the previous aggregate states *prevaggs*, since they may not be embedded in the received packet if the deltas do not exceed the thresholds in the previous node. For each packet, DeltaINT computes the current aggregate states and performs delta calculation between the current and embedded aggregate states.

DeltaINT slightly changes the UPDATE procedure. First, in addition to the flowkey, it extracts the aggregate states from each packet. Second, it compares each current aggregate state with the corresponding embedded aggregate state for delta calculation. Specifically, for each state, DeltaINT uses the aggregate state extracted from the packet as the previous aggregate state; if it is not carried by the packet, DeltaINT uses the previous aggregate state recorded in the sketch. It then performs aggregation on the current state and the previous aggregate state to obtain the current aggregate state. Finally, DeltaINT updates the sketch with the flowkey, the previous aggregate states (as carried by the received packet), and the embedded aggregate states. It also updates the bitmap in the packet depending on which aggregate states are carried.

Static per-flow aggregation. Figure 7 depicts the sketch layout for this family. Note that DeltaINT only maintains a sketch in the source of each path. The data structure is in essence the same as that of the framework, with two exceptions. First, each bucket only records a flowkey without the embedded states. Second, each packet only carries a one-bit bitmap, inserted into an entry by the source along with the current states, for all nodes in the forwarding path. Each transit/sink node in the forwarding path only embeds the current states into an entry guided by the bit in the received packet.

DeltaINT works differently in the source and the transit/sink nodes. For the source, DeltaINT slightly changes the UPDATE procedure. If the flowkey of the received packet is not recorded in the sketch, the source embeds a bit of one and all current states into the packet; otherwise, it only embeds a bit of zero. For the transit/sink nodes, if the bit carried by the received packet is one (i.e., the static flow-level aggregation has not been collected), the transit/sink nodes embed the current states into the packet; otherwise, they do not embed any state.

Note that the transit/sink nodes do not need to insert a bitmap. To know the number of entries carried by a packet, DeltaINT exploits the time-to-live (TTL) field to infer the path length (i.e., the number of nodes that a packet has traversed) [8]. Specifically, if the one-bit bitmap has a bit one (i.e., all current states of each node are embedded), the control plane uses the TTL field to infer the path length and hence extract the same number of entries.

Dynamic per-flow aggregation. We keep the same data plane as in the framework and slightly change the monitoring system in the control plane. Recall that this family tracks the aggregation function performed on each state over all packets of a flow in each node (§II-B). Since the aggregation is complicated (e.g., median or 99th percentile latencies) and cannot be readily supported by the data plane, the control plane is responsible for performing flow-level aggregation. To avoid enumerating all collected states, DeltaINT maintains the critical states in a *quantile sketch* [20] in the control plane as in PINT [8]. The control plane extracts all states from each event and updates them into the quantile sketch, which stores the states in fixed capacity. If the number of stored states exceeds the capacity, the quantile sketch discards some states. For flow-level aggregation, DeltaINT queries the quantile sketch for each flow, which provides approximate quantile information by enumerating a limited number of the stored states.

IV. THEORETICAL ANALYSIS

In this section, we present theoretical analysis of DeltaINT framework. Our analysis considers a DeltaINT instance with a sketch of d rows and w buckets per-row. We analyze the time complexity, space complexity, error probability, and bit overhead. We denote the logarithm with the base of the Euler number e by ln, and that with the base of 2 by log.

Theorem 1 gives the time and space complexities of the sketch algorithm in the data plane of DeltaINT.

Theorem 1. *For a DeltaINT sketch, the time complexity is $O(d)$ and the space complexity is $O(wd\log N)$, where N is the size of flowkey space.*

Proof. For each packet, it calculates d hash functions and performs the updates for d buckets. Note that the complexity of updating one bucket is O(1). Therefore, the time complexity of the sketch is $O(d)$. The two-dimensional sketch has wd buckets. Each bucket contains a flowkey of $\log N$ bits and an embedded state of constant bits. Thus, the space complexity of the sketch is $O(wd\log N)$. □

Theorem 2 quantifies the extent of hash collisions in the DeltaINT sketch because the hash collisions are the only cause of errors in DeltaINT. We characterize the errors with two user-specified parameters: the maximum accepted error probability σ, and the approximation coefficient ε that is the ratio of error flows to the total number of flows. Theorem 2 shows that if we appropriately configure d and w, the user-specified errors can be satisfied.

Theorem 2. *With the configuration that $d = \log\frac{1}{\varepsilon\sigma}$ and $w = \frac{n}{\ln 2}$, the number of error flows is less than εn with a probability of at least $1-\sigma$.*

Proof. The proof is similar to that of §5.2.5 and §5.3.1 in [11] and we omit the details here. □

Theorem 3 gives the theoretical bounds of the bit overhead (i.e., the number of bits for INT) of DeltaINT based on Lemma 1. Since the theoretical analysis of each state in each node is the same, we present the result of one state in one node.

Lemma 1. *For each packet of a flow, the probability that DeltaINT does not embed the current state satisfies $P_s = (1-\varepsilon\sigma)(1-\alpha)$, where α is the probability that the delta exceeds the predefined threshold of delta calculation.*

Proof. Please refer to the technical report [28]. □

Theorem 3. *We define the* bandwidth ratio *as the ratio between the bit overhead of DeltaINT and that of the original INT framework. The lower bound of the bandwidth ratio is $O(\frac{1}{b}+\alpha)$, while the upper bound is $O(\frac{1}{b}+1-P_s)$, where b is the number of bits of the state.*

Proof. Please refer to the technical report [28]. □

Remark. We use an example to show the bounds of bandwidth ratio. We consider 32-bit latency and assume that it holds relatively stable in 90% of the time. It means that $b = 32$ and $\alpha = 0.1$. Then, the lower bound of the bandwidth ratio is $\frac{1}{b} + \alpha = 13.1\%$. Given $\varepsilon = 0.1$ and $\sigma = 0.1$, the upper bound of the bandwidth ratio equals $\frac{1}{b} + 1 - P_s = 14\%$. Note that α can be smaller in practice, which means more bandwidth mitigation. For instance, α is always zero for all static states.

V. Evaluation

We evaluate DeltaINT on four representative applications described in §II-B. We summarize the results as follows:

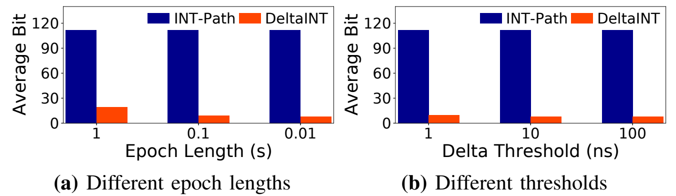
- In gray failure detection, DeltaINT incurs 93% less INT bandwidth than the probing-based INT framework INT-path [26] (Exp#1), with similar detection time (Exp#2).
- In congestion control, DeltaINT consumes only one bit per packet on average compared to the sampling-based INT framework PINT [8], which requires 8 bits (Exp#3). DeltaINT also has smaller flow completion time for large flows (Exp#4).
- In path tracing, DeltaINT consumes nearly one bit per packet on average, while PINT requires 8 bits (Exp#5). DeltaINT also has better convergence than PINT (Exp#6).
- In latency measurement, DeltaINT uses 2.4 bits per packet on average, while PINT requires at least 9.9 bits (Exp#7). DeltaINT also has higher accuracy than PINT (Exp#8).
- For INT bandwidth usage and the performance of each representative application (§II-B), DeltaINT is insensitive to sketching parameters (Exp#9).
- DeltaINT consumes limited resources when being deployed in a Barefoot Tofino switch (Exp#10).

Methodology. We evaluate DeltaINT in both software and P4 hardware [9]. For the former, we conduct simulations using both bmv2 [5] and NS3 [4]; for the latter, we compile DeltaINT in a Barefoot Tofino switch [1] connected with 40Gbps NICs of 16 servers. We configure DeltaINT with 1 MB memory and one hash function in each node. We also evaluate different choices of the two parameters in §V-E. DeltaINT uses 5-tuple flowkeys by default.

A. Gray Failure Detection

We first evaluate DeltaINT in gray failure detection. We compare it with the state-of-the-art probing-based INT framework INT-path [26]. INT-path sends probes along data-plane paths, in which the probe packets use the original INT framework to record path status. To reduce the number of probe packets, INT-path performs path planning and finds the non-overlapping paths in the control plane.

We follow the open-sourced framework of INT-Detect [18] to implement the data planes of both INT-path and DeltaINT. For fair comparisons, both data planes select the same non-overlapping paths based on the path planning scheme in INT-Path. In the control plane, we implement a gray failure detector that supports both data planes. The control plane examines all received packets. If the packets of some path are not received or the latency of some node exceeds a *heavy threshold* for a



(a) Different epoch lengths **(b)** Different thresholds

Figure 8: (Exp#1) Bandwidth usage in gray failure detection.

period of time (called the *aging time*), the control plane reports a potential gray failure.

We configure both INT-path and DeltaINT as follows, based on the original INT-path paper [26]. For both systems, we set the epoch length as 0.01 s to keep the probe sending rate at 100 packets per second. We set the heavy threshold as 1 μs for heavy latency detection. To mimic gray failures, we inject link down and heavy latency events. Both systems collect the 8-bit device ID, 8-bit ingress port, 8-bit egress port, and 32-bit latency as the states. For DeltaINT, we fix the delta thresholds of the device ID, ingress port, and egress port as zero, and that of the latency as 10 ns. We run the experiment for 10 s.

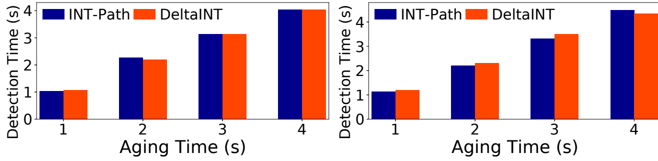
(Exp#1) Bandwidth usage in gray failure detection. We compare the bandwidth usage of INT-path and DeltaINT. We consider the *average bit cost* in the packets, defined as the average number of bits per packet for carrying the INT information (i.e., device ID, ingress port, egress port, and latency). For DeltaINT, we also count the number of bits in the bitmap. Note that the average bit cost excludes the metadata header (§II-A) in the calculation, as the control plane can directly configure which states are tracked [8].

Figure 8a shows the average bit cost versus the epoch length, with a fixed delta threshold of 10 ns. The average bit cost of DeltaINT decreases as the epoch length decreases, mainly because DeltaINT only needs to embed all states for the first probe packet of each path (which has no embedded states at the beginning), and no longer needs to embed the states with negligible deltas in the subsequent probe packets. Such bandwidth overhead is amortized as the number of epochs (i.e., the number of probe packets of each path) increases.

Figure 8b shows the average bit cost versus the delta threshold, with a fixed epoch length of 0.01 s. The average bit cost of DeltaINT decreases as delta threshold increases, as a larger delta threshold implies fewer latency states being embedded after delta calculation.

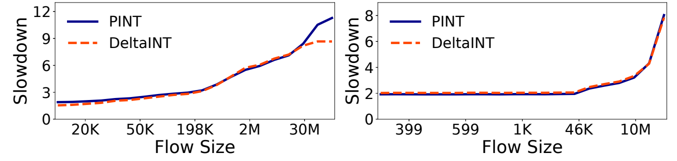
Overall, the average bit cost of INT-path is always 112 bits[1], while that of DeltaINT is as low as 8.1 bits when the delta threshold is 10 ns and the epoch length is 0.01 s (i.e., at most 93% bandwidth mitigation). The reason is that INT-path always inserts all the four states into each packet at each node, while DeltaINT only embeds those with non-negligible deltas.

[1]Each path in the topology has three nodes, and each node embeds 56 bits of INT information into a packet (i.e., 8-bit device ID, 8-bit ingress port, 8-bit egress port, and 32-bit latency). Thus, each of the three nodes generates a packet with INT information of size 56 bits, 112 bits and 168 bits, so the average bit cost is 112 bits.



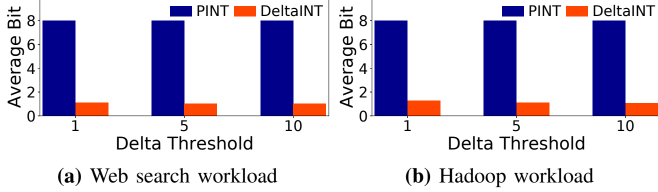
(a) Link down **(b)** Heavy latency

Figure 9: (Exp#2) Detection time in gray failure detection.



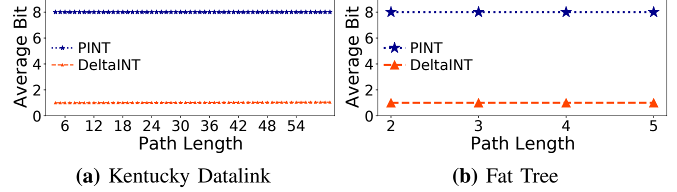
(a) Web search workload **(b)** Hadoop workload

Figure 11: (Exp#4) 95th percentile slowdown in congestion control.



(a) Web search workload **(b)** Hadoop workload

Figure 10: (Exp#3) Bandwidth usage in congestion control.



(a) Kentucky Datalink **(b)** Fat Tree

Figure 12: (Exp#5) Bandwidth usage in path tracing.

(Exp#2) Detection time in gray failure detection. We follow INT-Detect [18] to evaluate the detection time of gray failures (including link down and heavy latency) versus the aging time.

Figures 9a and 9b show that DeltaINT has almost the same detection time as INT-path, as DeltaINT collects all critical states with non-negligible deltas that could imply the presence of gray failures. Note that both approaches have fast convergence to report gray failures, as they process probe packets without sampling.

B. Congestion Control

We evaluate DeltaINT in congestion control. We compare it with the state-of-the-art sampling-based INT framework PINT [8]. We follow PINT to implement DeltaINT in NS3 [4] with the same Fat Tree topology. For fair comparisons, both PINT and DeltaINT use the same approach of value approximation to get link utilization. We fix both schemes to report 8-bit values of link utilization, while the results hold for other numbers of bits for value approximation. After collecting each link utilization, both systems use HPCC [24] for congestion control.

We use the following configuration. We follow the same flow size (i.e., number of packets in a flow) distribution to generate traffic, including web search workload [7] and Hadoop workload [27], as in the PINT paper [8]. For DeltaINT, we set the delta threshold of the 8-bit link utilization as one.

(Exp#3) Bandwidth usage in congestion control. Figure 10 compares PINT and DeltaINT on INT bandwidth usage. It shows that the average bit cost of DeltaINT decreases to nearly one bit in both web search and Hadoop workloads, as the link utilization remains stable at most time and DeltaINT only needs to embed a limited number of states.

(Exp#4) Flow completion time in congestion control. We evaluate the flow completion times of both PINT and DeltaINT based on the *slowdown* of a flow [8], defined as the ratio between the flow completion time of the flow when all other flows exist and when only the flow itself exists; a smaller slowdown means better congestion control. We consider the 95th percentile *slowdown* for each flow size.

Figures 11a and 11b show that DeltaINT has a smaller slowdown for large flows than PINT, albeit a similar slowdown for small flows. Specifically, for the largest 5% of flows, the slowdown of DeltaINT is smaller than that of PINT by 2.6 and 0.2 in web search and Hadoop workloads, respectively. The reason is that DeltaINT has a smaller average bit cost, which reduces the likelihood of IP-level fragmentation particularly for large flows and hence reduces the flow completion time.

C. Path Tracing

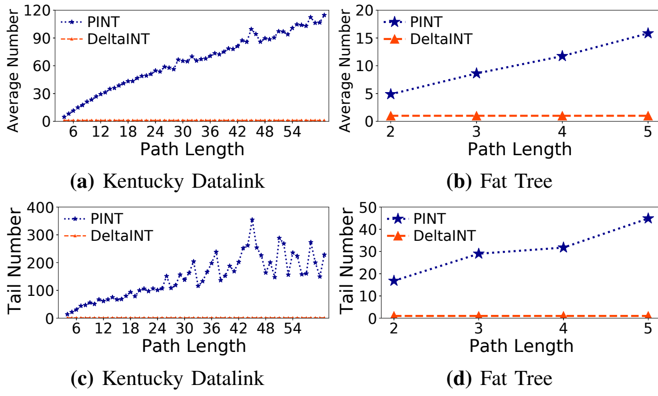
We evaluate DeltaINT in path tracing. We compare it with PINT [8]. For each flow, PINT amortizes all device IDs of the path into sampled packets by distributed encoding. We follow PINT [8] to implement DeltaINT in bmv2 [5] and build both PINT and DeltaINT in Mininet [3]. Both systems use two topologies: (i) Kentucky Datalink, an ISP topology with 753 switches; and (ii) Fat Tree, a data center topology with 80 switches. We compute the number of traversed nodes for each packet from the TTL field. For PINT, we use the maximum number of bits supported (i.e., 8 bits) for distributed encoding to achieve high convergence, while other configurations for PINT are the same as stated in its original paper [8]. For DeltaINT, we fix the threshold of device ID as zero.

(Exp#5) Bandwidth usage in path tracing. We compare the INT bandwidth usage of PINT and DeltaINT. To see the impact of the path length, we calculate the average bit cost on the packets traversing a path for each given path length.

Figures 12a and 12b show that the average bit cost of PINT is always 8 bits, as it always maintains an 8-bit state for distributed encoding in each packet. In contrast, the average bit cost of DeltaINT increases from 1 bit to 1.01 bits and from 1 bit to 1.05 bits as the path length increases in Kentucky Datalink and Fat Tree topologies, respectively. The reason is that DeltaINT only embeds all device IDs of the traversed nodes into the first packet of each flow, while each of the subsequent packets only carries a one-bit bitmap.

(Exp#6) Convergence in path tracing. We compare the convergence of PINT and DeltaINT based on the average and 99th percentile (i.e., tail) numbers of the required packets sent by a flow to collect all static states.

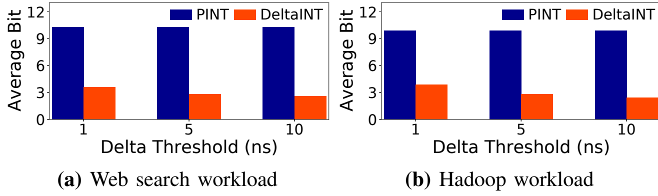
Figures 13a and 13b show that the average number of packets in PINT grows almost linearly with the path length to around



(a) Kentucky Datalink **(b)** Fat Tree

(c) Kentucky Datalink **(d)** Fat Tree

Figure 13: (Exp#6) Convergence in path tracing.



(a) Web search workload **(b)** Hadoop workload

Figure 14: (Exp#7) Bandwidth usage in latency measurement.

120 and 15 in Kentucky Datalink and Fat Tree, respectively, while Figures 13c and 13d show that the 99th percentile number of packets has a similar tendency that PINT requires around 350 and 45 packets in the two topologies, respectively. Since PINT employs sampling and distributed encoding, it needs to collect sufficient packets for path tracing. In contrast, both the average and 99th percentile numbers of packets in DeltaINT are always one, as DeltaINT must embed the device IDs of all traversed nodes into the first packet of each flow. Thus, DeltaINT can trace the path for a flow of any size with fast convergence (with only one packet).

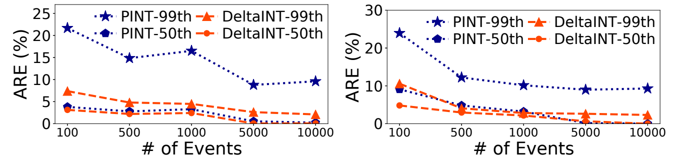
D. Latency Measurement

We evaluate DeltaINT in latency measurement. We again compare DeltaINT with PINT [8], using the same workloads as in §V-B. In the data plane, both systems use the same value approximation to obtain an 8-bit latency as in §V-B. In the control plane, both systems use the same quantile sketch called KLL [20], configured as in the PINT paper [8]. For DeltaINT, we set the delta threshold of the latency as 1 ns.

(Exp#7) Bandwidth usage in latency measurement. We compare the bandwidth usage of PINT and DeltaINT. We tune the delta threshold from 1 to 10 ns.

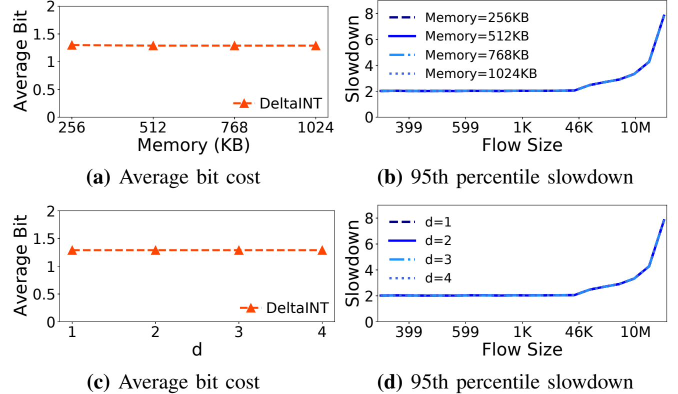
Figure 14 shows that the average bit cost of PINT is around 10.3 bits and 9.9 bits under web search and Hadoop workloads, respectively. In contrast, as the delta threshold rises, the average bit cost of DeltaINT decreases to around 2.6 bits and 2.4 bits in the two workloads, respectively. PINT always embeds the latency state into packets, while DeltaINT only embeds the latency state when the delta exceeds the threshold.

(Exp#8) Accuracy in latency measurement. We compare the measurement accuracy of PINT and DeltaINT. We consider the average relative error (ARE) of the 50th and 99th percentile latencies.



(a) Web search workload **(b)** Hadoop workload

Figure 15: (Exp#8) Accuracy in latency measurement.



(a) Average bit cost **(b)** 95th percentile slowdown

(c) Average bit cost **(d)** 95th percentile slowdown

Figure 16: (Exp#9) Sensitivity on sketching parameters.

Figure 15 shows that PINT always has a larger ARE than DeltaINT in both 50th and 99th percentile latencies. The reason is that PINT embeds latency by sampling, which could miss critical information. However, DeltaINT avoids sampling and embeds each critical latency state with non-negligible delta.

E. Sketching Parameter Sensitivity

(Exp#9) Sensitivity to sketching parameters. We validate the limited sensitivity of DeltaINT to sketching parameters. Since the results are similar for all four applications, we only present the results for congestion control in the interest of space. We keep the same configuration as §V-B and use the Hadoop workload with around 3.3 million flows.

Figures 16a and 16b show the results versus the memory usage of the sketch, where we fix the number of hash functions $d = 1$. As the memory usage increases, the average bit cost and the slowdown do not change. Figures 16c and 16d show the results versus d, where we fix the entire memory as 1 MB. The results are similar, since DeltaINT only focuses on the adjacent packets of each flow, while most conflicting flows do not have overlapping life cycles. Thus, the impact of hash collisions on DeltaINT is limited.

F. DeltaINT in Hardware

We implement DeltaINT in P4 [9] with less than 400 lines of code in each family of applications. We compile it in the Barefoot Tofino chipset [1] to demonstrate the feasible hardware deployment of DeltaINT. Specifically, we deploy DeltaINT in the egress pipeline. We realize each row of the sketch in the data plane as an array of registers. If the number of bits of the recorded data in each bucket exceeds the hardware limitation (e.g., at most 64 bits in Tofino), we split it into multiple register arrays with the same length. First, we calculate the hash results of the flowkey and store them in the Packet

	SRAM (KB)	No. stages	No. actions	No. ALUs	PHV size (bytes)
F1	224 KB (1.46%)	4 (33%)	9 (nil)	2 (4.17%)	115 (15%)
F2	224 KB (1.46%)	6 (50%)	10 (nil)	3 (6.25%)	111 (15%)
F3	304 KB (1.98%)	2 (16%)	4 (nil)	1 (2.08%)	104 (14%)
F4	224 KB (1.46%)	4 (33%)	9 (nil)	2 (4.17%)	115 (15%)
INT	176 KB (1.15%)	4 (33.%)	42 (nil)	0 (0%)	231 (30%)

TABLE I: (Exp#10) Hardware resource usage (percentages in brackets are fractions of total resource usage).

Header Vector (PHV) to avoid duplicate hash operations. Then for each array, we use the corresponding hash result to locate a register. All the three primitives about the register, including StateLoad, DeltaCalc, and StateUpdate, are performed in one operation of an ALU. Finally, according to the return value of DeltaCalc, we embed the states into the packet header via MetadataInsert. For each family, we use the same setting of the application as mentioned before.

(Exp#10) Hardware resource usage. Table I shows the resource usage in hardware. It includes computational overhead (i.e., SRAM consumption, the number of physical stages, actions, and stateful ALUs) and cross-stage message overhead (i.e., the PHV size). For DeltaINT, we present the results of all four families of applications, including per-packet-per-node monitoring (F1), per-packet aggregation (F2), static per-flow aggregation (F3), and dynamic per-flow aggregation (F4). DeltaINT needs at most 304 KB of SRAM, accounting for only 1.98% of the total SRAM. Although DeltaINT needs at most six physical stages (50% of total) to accommodate all tables, registers, and ALU operations in the data plane, it still leaves enough resources, including the unoccupied stages and the unused SRAM and ALUs of the occupied stages, for other network functions. Moreover, DeltaINT incurs at most 10 actions and consumes only at most three stateful ALUs (6.25% of total). Finally, DeltaINT uses at most 111 bytes of PHV (15% of total) to transmit messages between different stages. Note that nearly half of them are used for the basic network functions like packet forwarding.

We further use the baseline switch P4 program `switch.p4` provided by Barefoot to evaluate the hardware resource usage of the original INT. We disable all network functions except the original INT in `switch.p4`, and compile it in the same testbed as DeltaINT. Here, we focus on the performance of the original INT in F2, which incurs the most number of stages in DeltaINT. From Table I, DeltaINT incurs slightly more SRAM, stages, and stateful ALUs than the original INT due to the tracking of states. Nevertheless, the original INT incurs larger bandwidth overhead than DeltaINT, and hence uses more PHV size and actions to process and transmit INT information.

VI. Related Work

Bandwidth mitigation in the data plane. Some approaches embed INT information into only a subset of traversed packets. Selective-INT [22], Sel-INT [31], FS-INT-R [29], and PINT [8] build on sampling, and embed INT information into only sampled packets. The sampling rate can be adjusted according to state changes [22] or historical states [31]. However, sampling requires sufficient packets for event detection and has slow convergence (§I). BurstRadar [19] detects microbursts via INT by embedding INT information to the packets only when the egress queue length is larger than a predefined threshold. INT-path [26] sends probes based on source routing and generates a minimum number of non-overlapped paths, yet it still embeds the full states into packets (§V-A).

Some approaches reduce the INT information embedded in a packet by encoding or approximation. FS-INT-E [29] introduces a bitmap in a packet for per-node metadata to track the state changes of nodes, but the bitmap cannot be readily extended for packet-level or flow-level aggregation. PINT [8] exploits global hashing, distributed encoding, and approximation to combine INT metadata, yet it cannot readily support per-packet-per-node monitoring. Also, both the global hashing and distributed encoding of PINT are based on sampling, which leads to slow convergence. LightGuardian [35] piggybacks sketchlets (i.e., fragments of a sketch) into packets, such that each sketchlet encodes flow-level statistics in compact form, yet it only supports flow-level INT, but not packet-level INT.

Bandwidth mitigation in the control plane. Some approaches focus on reducing events in the control plane. Fast INTCollector [33] reports a small number of events to the controller only for significant state changes. Intel proposes a change detection algorithm [2] that monitors packets against specific conditions (e.g., performance thresholds) and reports events only when the conditions are met, so as to reduce the number of events. OmniMon [15] stores telemetry data in network entities and sends the telemetry data only periodically to the controller for analysis, but it only supports dynamic flow-level telemetry with specific aggregation functions (e.g., frequency and sum). DeltaINT also mitigates the bandwidth overhead in the control plane, since each sink extracts all INT information from the packet and reports it as an event to the control plane (§II-A).

VII. Conclusion

DeltaINT is a novel INT framework that achieves extremely low bandwidth overhead for INT and has the properties of generality, scalability, and compatibility. The key insight is to embed the states into each packet only if the state changes are non-negligible over the traversed packets. We theoretically derive the time/space complexities, the error probability, and the bounds of bandwidth mitigation. We show via software simulations that DeltaINT can mitigate INT bandwidth overhead by up to 93% with similar or even better application performance, and we further show via P4 hardware implementation that DeltaINT has limited usage of hardware resources in the Barefoot Tofino switch deployment.

Acknowledgment. We thank our shepherd, Gianni Antichi, and the anonymous reviewers for their comments. This work was supported by the National Key R&D Program of China (2019YFB1802600), National Natural Science Foundation of China (U20A20179), and National Natural Science Foundation of China (61802365). Qun Huang is the corresponding author.

REFERENCES

[1] Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino/.

[2] Change detection algorithm of Intel. https://builders.intel.com/docs/networkbuilders/in-band-network-telemetry-detects-network-performance-issues.pdf.

[3] Mininet. http://mininet.org/.

[4] Network Simulator3. https://www.nsnam.org/.

[5] P4 switch behavioral model. https://github.com/p4lang/behavioral-model.

[6] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proc. of ACM SIGCOMM*, 2014.

[7] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. of ACM SIGCOMM*, 2010.

[8] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *Proc. of ACM SIGCOMM*, 2020.

[9] 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. *ACM SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, 2014.

[10] B. Choi, S. B. Moon, R. L. Cruz, Z. Zhang, and C. Diot. Quantile sampling for practical delay monitoring in internet backbone networks. *Computer Networks*, 51(10):2701–2716, 2007.

[11] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends Databases*, 4(1–3):1–294, 2012.

[12] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. of USENIX NSDI*, 2014.

[13] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proc. of ACM HotOS*, 2017.

[14] Q. Huang, P. P. C. Lee, and Y. Bao. SketchLearn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proc. of ACM SIGCOMM*, 2018.

[15] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao. OmniMon: Re-architecting network telemetry with resource efficiency and full accuracy. In H. Schulzrinne and V. Misra, editors, *Proc. of ACM SIGCOMM*, pages 404–421. ACM, 2020.

[16] N. Ivkin, Z. Yu, V. Braverman, and X. Jin. QPipe: Quantiles sketch fully in the data plane. In *Proc. of ACM CoNEXT*, 2019.

[17] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *Proc. of ACM SIGCOMM*, 2014.

[18] C. Jia, T. Pan, Z. Bian, X. Lin, E. Song, C. Xu, T. Huang, and Y. Liu. Rapid detection and localization of gray failures in data centers via in-band network telemetry. In *Proc. of IEEE NOMS*, 2020.

[19] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo. BurstRadar: Practical real-time microburst monitoring for datacenter networks. In *Proc. of ACM APSys*, 2018.

[20] Z. S. Karnin, K. J. Lang, and E. Liberty. Optimal quantile approximation in streams. In *Proc. of IEEE FOCS*, 2016.

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

[22] Y. Kim, D. Suh, and S. Pack. Selective in-band network telemetry for overhead reduction. In *Proc. of IEEE CloudNet*, 2018.

[23] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better NetFlow for data centers. In *Proc. of USENIX NSDI*, 2016.

[24] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPCC: High precision congestion control. In *Proc. of ACM SIGCOMM*, 2019.

[25] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. of ACM SIGCOMM*, pages 15–28, 2017.

[26] T. Pan, E. Song, Z. Bian, X. Lin, X. Peng, J. Zhang, T. Huang, B. Liu, and Y. Liu. INT-path: Towards optimal path planning for in-band network-wide telemetry. In *Proc. of IEEE INFOCOM*, 2019.

[27] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proc. of ACM SIGCOMM*, 2015.

[28] S. Sheng, Q. Huang, and P. P. C. Lee. DeltaINT: Toward general in-band network telemetry with extremely low bandwidth overhead. Technical report, The Chinese University of Hong Kong, 2021. http://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_deltaint.pdf.

[29] D. Suh, S. Jang, S. Han, S. Pack, and X. Wang. Flexible sampling-based in-band network telemetry in programmable data plane. *ICT Express*, 6(1):62–65, 2020.

[30] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with PathDump. In *Proc. of USENIX OSDI*, 2016.

[31] S. Tang, D. Li, B. Niu, J. Peng, and Z. Zhu. Sel-INT: A runtime-programmable selective in-band network telemetry system. *IEEE Trans. on Network Service and Management*, 17(2):708–721, 2020.

[32] The P4.org Applications Working Group. In-band network telemetry (INT) dataplane specification version 2.1. https://github.com/p4lang/p4-applications/blob/master/docs/INT_latest.pdf, Nov 2020.

[33] J. Vestin, A. Kassler, D. Bhamare, K. Grinnemo, J. Andersson, and G. Pongrácz. Programmable event detection for in-band network telemetry. In *Proc. of IEEE CloudNet*, 2019.

[34] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *Proc. of ACM IMC*, pages 78–85, 2017.

[35] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, et al. LightGuardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *Proc. of USENIX NSDI*, 2021.

[36] Y. Zhu, N. Kang, J. Cao, A. G. Greenberg, G. Lu, R. Mahajan, D. A. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *Proc. of ACM SIGCOMM*, 2015.