A General Delta-based In-band Network Telemetry Framework with Extremely Low Bandwidth Overhead

Siyuan Sheng[a], Qun Huang[b,*], Patrick P. C. Lee[a]

[a]*Department of Computer Science and Engineering, The Chinese University of Hong Kong, China*
[b]*Department of Computer Science and Technology, Peking University, China*

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 the complete state information into a packet only when the state change is deemed significant. We propose two variants for DeltaINT that trade between bandwidth usage and measurement accuracy, while both variants achieve significantly lower bandwidth overhead than the original INT framework. We theoretically derive the time/space complexities and the guarantees of bandwidth mitigation for DeltaINT. We implement DeltaINT in both software and P4. Our evaluation shows that DeltaINT significantly mitigates the bandwidth overhead, and the deployment in a Tofino switch incurs limited hardware resource usage.

Keywords: In-band network telemetry; network measurement

Note: An earlier version of this paper appeared at the 29th Annual IEEE International Conference on Network Protocols (ICNP'21) [34]. In this extended version, we extend DeltaINT to support monitoring with full accuracy, while still significantly reducing the bandwidth overhead of the original INT framework. We now propose two variants of DeltaINT, namely DeltaINT-O (proposed in our conference version) and DeltaINT-E (proposed in this extended version), that trade between bandwidth usage and measurement accuracy. We also add new evaluation results for both software and hardware implementations.

1. Introduction

In-band Network Telemetry (INT) [39] 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

*Corresponding author
Email addresses: sysheng21@cse.cuhk.edu.hk (Siyuan Sheng), huangqun@pku.edu.cn (Qun Huang), pclee@cse.cuhk.edu.hk (Patrick P. C. Lee)

(called *nodes*) to collect various device-internal states (e.g., device ID, link utilization, and 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 [39]. Such INT information can be collected by the destinations of the forwarding path for aggregate analysis, at packet-level [14, 32, 44] or flow-level [9, 17, 29] granularities. In particular, INT aims for *generality* and supports various network management applications, such as load balancing [7], latency profiling [27], microburst detection [25], congestion control [9, 30], and network troubleshooting [14, 23]; it is also deployed in production data centers [30].

A drawback of the original INT framework [39] 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., [9, 18, 25, 28, 32, 36, 38, 40, 43]) have proposed different INT bandwidth mitigation approaches. A straightforward approach is *sampling* [9, 28, 36, 38], 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 [9] 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 Section 2.3, existing studies often sacrifice generality for INT bandwidth mitigation.

We design DeltaINT, a novel INT framework that aims to support general network management 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 the complete state information into packets only when there exist significant state changes that exceed some pre-defined thresholds. This allows DeltaINT to save significant INT bandwidth, without compromising the functionalities of network management applications. DeltaINT comprises two variants that trade between bandwidth usage and measurement accuracy: (i) DeltaINT-O, which omits the state from the packets if the state change is within some pre-specified threshold, and (ii) DeltaINT-E, which encodes the state change with a limited number of bits into the packets if the state change is within some pre-specified threshold. DeltaINT-O achieves the minimum possible bandwidth usage as it excludes any state changes that are considered negligible, yet its measurement accuracy may degrade for a larger threshold (note that the accuracy is maintained in the common case since a reasonably small threshold can work well for bandwidth mitigation; see Section 5). In contrast, DeltaINT-E maintains the full measurement accuracy independent of the threshold, although it incurs slightly larger bandwidth usage than DeltaINT-O. Nevertheless, both DeltaINT-O and DeltaINT-E achieve significantly lower bandwidth overhead

than the original INT framework. To this end, 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 [32] and value approximation [9], for further bandwidth savings. We summarize our contributions as follows.

- We design DeltaINT, a general delta-based 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 using bmv2 [6] and NS3 [5] on DeltaINT for various applications, and compare DeltaINT with INT-Path [32] and PINT [9]. DeltaINT significantly reduces the average per-packet bit overhead (e.g., by up to 93% and 85% in gray failure detection for DeltaINT-O and DeltaINT-E compared with INT-Path [32], respectively), while showing fast convergence and being compatible with existing INT techniques.
- We provide the P4 hardware implementation (compiled by a Tofino switch [2]) for DeltaINT to show that it has limited hardware resource usage.

We open-source our DeltaINT prototype, including the code for both software simulations and Tofino-based hardware implementation, at **http://adslab.cse.cuhk.edu.hk/software/deltaint**.

The rest of the paper proceeds as follows. Section 2 presents the background and motivation for the bandwidth mitigation problem in INT. Section 3 presents the design of DeltaINT, including the two variants DeltaINT-O and DeltaINT-E. Section 4 presents the theoretical analysis for both DeltaINT-O and DeltaINT-E. Section 5 presents the evaluation results for DeltaINT using both software simulation and hardware implementation. Finally, Section 6 summarizes the evaluation results and concludes the paper.

2. Background and Motivation

We introduce the basics of the original INT framework (Section 2.1) and different families of INT applications (Section 2.2). We review existing solutions on mitigating the INT bandwidth and state their limitations (Section 2.3). Finally, we provide an overview for our solution (Section 2.4).

2.1. Basics of INT

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

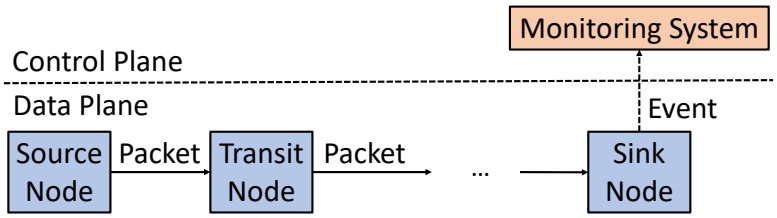


Figure 1: INT framework.

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. Note that if any forwarding path of a packet changes, the controller needs to reconfigure the roles of the affected nodes (along both the old and new paths). How to detect the path changes or reconfigure the affected nodes by the controller is beyond the scope of the paper.

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.

2.2. 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 [9]. In the following, we

provide the definitions, as well as the representative applications that are used by existing INT solutions, for different families 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 major application is *fine-grained monitoring* [20, 27], which monitors the data traffic by tracking the accurate packet-level INT states on a per-node manner for network management. For example, we can leverage the original INT framework [27] to collect per-packet-per-node latency states for debugging network incidents. Another application is the *detection of gray failures* [16], which refer to the components that remain functional (i.e., bypassing failure detection) but suffer from performance anomalies. To detect gray failures, INT-Detect [24] 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 [30]. To support fine-grained congestion control, HPCC [30] 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 in 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* [23], which discovers the forwarding path traversed by a flow for path-level performance control (e.g., load balancing [29] and network debugging [37]). To trace the path of a flow, PathDump [37] 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 $p_1, p_2, \cdots$ in 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 [22] and delay monitoring [11]. For efficient statistics collection, PINT [9] collects per-node latency statistics carried by a packet into a compact data structure based on the quantile sketch [26].

2.3. Related Work

Many existing studies have been proposed in the literature to mitigate the bandwidth cost of the original INT framework. However, we argue that none of them can generally address all the aforementioned families of applications.

Bandwidth mitigation in the data plane. Some approaches embed INT information into only a subset of traversed packets. Selective-INT [28], Sel-INT [38], FS-INT-R [36], and PINT [9] build on sampling, and embed INT information into only sampled packets. The sampling rate can be adjusted according to state changes [28] or historical states [38]. However, sampling requires sufficient packets for event detection and has slow convergence (Section 1). BurstRadar [25] 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 [32] sends probes based on source routing and generates a minimum number of non-overlapped paths, yet it still embeds the full states into packets (Section 5.1).

Some approaches reduce the INT information embedded in a packet by encoding or approximation. FS-INT-E [36] 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 [9] 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 [43] 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 [40] reports a small number of events to the controller only for significant state changes. Intel proposes a change detection algorithm [1] 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 [18] 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). SketchINT [41] collects packet-level states by the original INT framework, aggregates them at the flow level into a sketch deployed in each sink, and periodically reports the sketch to the monitoring system. 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 (Section 2.1).

2.4. Our Solution

DeltaINT is an INT framework that achieves extremely low bandwidth overhead and aims for generality, convergence, and compatibility (Section 1). Recall that a node collects multiple device-internal states (Section 2.1). 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

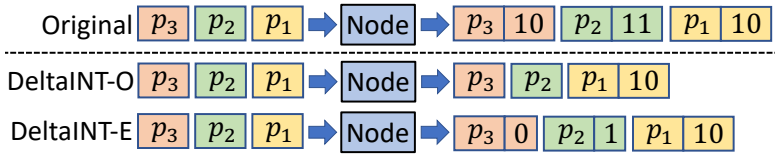
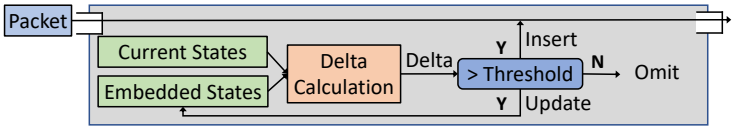


Figure 2: Motivating example.

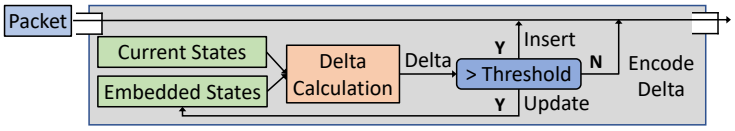
some pre-specified threshold) at most time in typical applications. For example, the queue occupancy and access latency remain relatively stable unless a microburst occurs [25, 42]; the link utilization remains stable unless under network congestion [30]; some static states, such as device ID and ingress/egress ports, are well-defined and remain unchanged.

Under DeltaINT, each node embeds the complete state information to only a subset of traversed packets, thereby reducing the INT bandwidth usage. DeltaINT comprises two variants, DeltaINT-O and DeltaINT-E, to trade between bandwidth usage and measurement accuracy, while both variants still significantly reduce the bandwidth overhead of the original INT framework. In actual deployment, DeltaINT selects one of the variants to use. Specifically, DeltaINT-O omits the state from the packets if the delta of the state is negligible, so as to reduce the bandwidth overhead as much as possible. It may incur slight degradations in measurement accuracy due to the omission of the state, yet it preserves the accuracy of common applications that are only interested in the significant delta of a state. For example, for fast gray failure detection, it is possible to identify only the network paths with significant timeouts [24]. On the other hand, DeltaINT-E encodes the delta of a state into the packets with much fewer bits than the complete state information. This maintains the full accuracy, as the current state can be reconstructed from the latest embedded state and its delta. While DeltaINT-E incurs slightly more bandwidth usage than DeltaINT-O, it is useful for the applications that require fine-grained network monitoring as targeted by the original INT framework, such as debugging network incidents [27].

Figure 2 shows an example of DeltaINT and how its variants DeltaINT-O and DeltaINT-E reduce the INT bandwidth overhead compared with the original INT framework. 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 complete current 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 complete current state. Specifically, DeltaINT-O directly omits the state from p_2 and p_3, so it saves roughly two-thirds of bandwidth usage compared with the original INT framework (note that DeltaINT additionally adds a bitmap into each traversed packet to track which states are carried; see Section 3.2 for details). On the other hand, DeltaINT-E encodes the deltas one and zero into p_2 and p_3, respectively. In this example, each encoded delta can represent one in two bits and zero in one bit (Section 3.2), so DeltaINT-E only incurs slightly more bandwidth usage than DeltaINT-O (i.e., three more bits



(a) DeltaINT-O



(b) DeltaINT-E

Figure 3: DeltaINT per-node architecture.

in total), while it still saves almost two-thirds of bandwidth usage compared with the original INT framework. Note that DeltaINT-E can retrieve the current states of p_2 and p_3 by adding the encoded deltas to the embedded state (i.e., 10) of p_1 for full accuracy.

3. DeltaINT Design

We present the design details of DeltaINT and its two variants DeltaINT-O and DeltaINT-E. Note that the two variants DeltaINT-O and DeltaINT-E mainly differ in the treatment of the negligible delta of a state, while their operations in other aspects are identical. Thus, we use DeltaINT to collectively refer to both variants when we discuss their common operations, while referring to each variant separately when there are differences in the design details.

In the following, we first introduce the architecture of DeltaINT (Section 3.1). We elaborate the design details of the DeltaINT framework (Section 3.2), including the data structure, update algorithm, and state reconstruction. We discuss the design issues in Section 3.3, and finally explain how DeltaINT can fit into different network management applications (Section 3.4).

3.1. Architecture

Figure 3 shows the per-node architectures in the data plane of both variants of DeltaINT (i.e., DeltaINT-O and DeltaINT-E). For each traversed packet, DeltaINT (including both of its variants) 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 pre-specified threshold, it 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 complete current state. If the delta is within the threshold (i.e., the delta is negligible), the two variants DeltaINT-O and DeltaINT-E behave differently: DeltaINT-O simply omits the state from the packet, while DeltaINT-E encodes the delta into the packet. Finally, if the sink receives a packet, DeltaINT extracts the INT information and sends an event to the monitoring system in the control plane, as in the original INT framework.

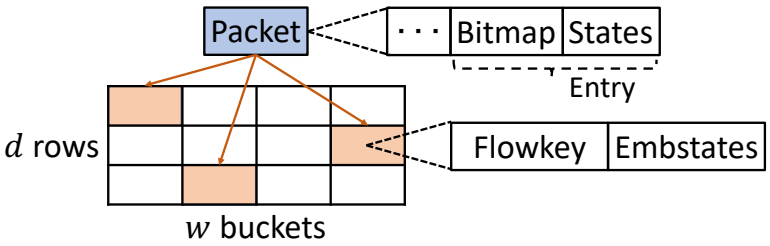


Figure 4: Sketch data structure for tracking embedded states.

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 [31] 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 Section 3.2.

DeltaINT deploys a monitoring system in the control plane. It in essence has the same design as the original INT framework, except for the state reconstruction (Section 3.2) and dynamic flow-level aggregation (Section 3.4).

3.2. 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 (Section 3.1). 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 for each node along the forwarding path of the packet. The bitmap tracks which state is being carried for the corresponding node. 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.

Algorithm 1 Update Algorithm of DeltaINT Framework

```
Require: Packet p; Threshold 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, φ)
 9:     delta ← curstate − embstate
10:     if |delta| ≤ φ 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] ← state
17:     if A[row][h_i(x)].flowkey ≠ x then
18:         A[row][h_i(x)].flowkey ← x
19:     end if
20: end function
21: function MetadataInsert(bitmap, curstates, embstates, p)
22:     Insert bitmap into p
23:     for all 1 ≤ idx ≤ #curstates do
24:         if bitmap[idx] = 1 then
25:             Insert curstates[idx] into p
26:         else if DeltaINT-E is used then
27:             delta ← curstates[idx] − embstates[idx]
28:             Encode delta into p by Huffman coding
29:         end if
30:     end for
31: end function
32: procedure Update(p)
33:     Extract a flowkey x from p
34:     Load curstates from the deployed node
35:     embstates ← StateLoad(x)
36:     Initialize bitmap with one
37:     if embstates exists then
38:         for all 1 ≤ i ≤ #curstates do
39:             if DeltaCalc(curstates[i], embstates[i], Φ[i])
    then
40:                 bitmap[i] ← 0
41:             end if
42:         end for
43:     end if
44:     for i = 1, 2, ..., d do
45:         notrecorded ← A[i][h_i(x)].flowkey ≠ x
46:         for all 1 ≤ idx ≤ #curstates do
47:             if bitmap[idx] = 1 then
48:                 StateUpdate(i, x, idx, curstates[idx])
49:             else if notrecorded then
50:                 StateUpdate(i, x, idx, embstates[idx])
51:             end if
52:         end for
53:     end for
54:     MetadataInsert(bitmap, curstates, embstates, p)
55: end procedure
```

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).

- DeltaCalc (Lines 8-14): It performs delta calculation for each state based on the difference between the current state *curstate* and the embedded state *embstate* (Line 9). It returns true if the absolute value of 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 21-31): It inserts INT information, including the bitmap and the states with non-negligible deltas, into packet p. If the delta of a state is negligible, the function behaves differently for DeltaINT-O and

DeltaINT-E: if DeltaINT-O is used, the state is not inserted; if DeltaINT-E is used, the function encodes the delta into packet p (see the delta encoding details below).

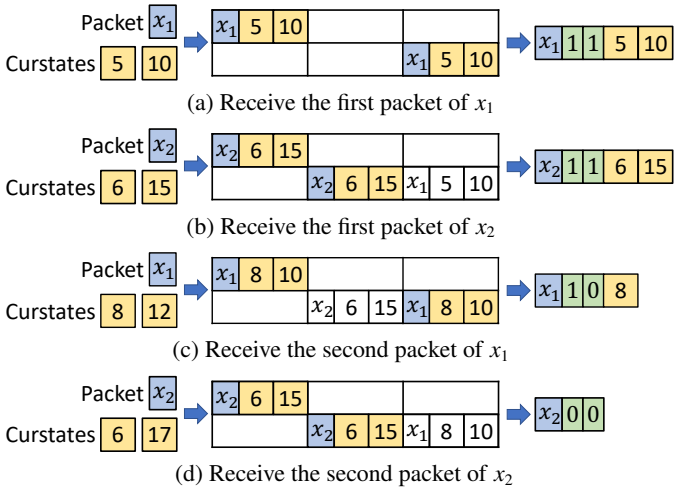
Update algorithm. Algorithm 1 shows how DeltaINT works in a node based on the four primitives mentioned above. DeltaINT calls the UPDATE procedure for each received packet p (Lines 32-55). Initially, the UPDATE procedure first extracts a flowkey x from p (Line 33) and loads the current states *curstates* from the node (Line 34). It also calls STATELOAD to load the embedded states into *embstates* from the stateful memory (Line 35). It further initializes all bits in *bitmap* as ones, meaning that each state is to be embedded by default (Line 36).

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 39-41). 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 44-53) 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 54).

Delta encoding in DeltaINT-E. Recall that if DeltaINT-E is used, the negligible delta of a state is encoded into a packet (Line 28 in Algorithm 1). We implement the encoding method based on Huffman coding [19], which realizes optimal prefix encoding to represent information with the fewest possible bits for a given probability distribution. Here, we assume that the delta value is zero with the largest probability, while each non-zero delta value within a pre-specified threshold $\phi > 0$ (i.e., any value in $[-\phi, \phi]\backslash\{0\}$) has an equal remaining probability. Based on Huffman coding, DeltaINT-E uses a single '0' bit to represent the zero delta, and uses the '1' bit followed by $\lceil \log_2(2\phi) \rceil$ bits to encode a non-zero delta. As a small ϕ works for a negligible delta in practice (Section 5), the bit cost of delta encoding is generally much smaller than that of the original INT framework. For example, if we set the threshold $\phi = 1$, DeltaINT-E uses bit "0", bits "10", and bits "11" to encode the delta of 0, -1, and 1, respectively. In contrast, the original INT framework always uses 4 bytes to embed a complete state. Thus, DeltaINT-E uses much fewer bits in delta encoding. Note that if the threshold $\phi = 0$, DeltaINT-E does not need to explicitly encode any a zero delta, as it can distinguish between the zero delta or the complete state based on the corresponding bit in the bitmap; in other words, DeltaINT-E omits the zero delta as in DeltaINT-O.

Update examples. We depict via examples how the UPDATE procedure works in DeltaINT-O and DeltaINT-E. In the examples, we consider a sketch A that has $d = 2$ rows with $w = 3$ buckets each. The sketch tracks two states, with thresholds 0 and 2, respectively. We consider two flows x_1 and x_2 with two packets each. Figure 5 shows the steps of the UPDATE procedure in DeltaINT-O:

- First, DeltaINT-O receives the first packet of x_1 with the current states of 5 and 10 (Figure 5(a)). It locates two buckets, say $A[1][1]$ and $A[2][3]$. Since no bucket matches x_1, DeltaINT-O 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.



(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

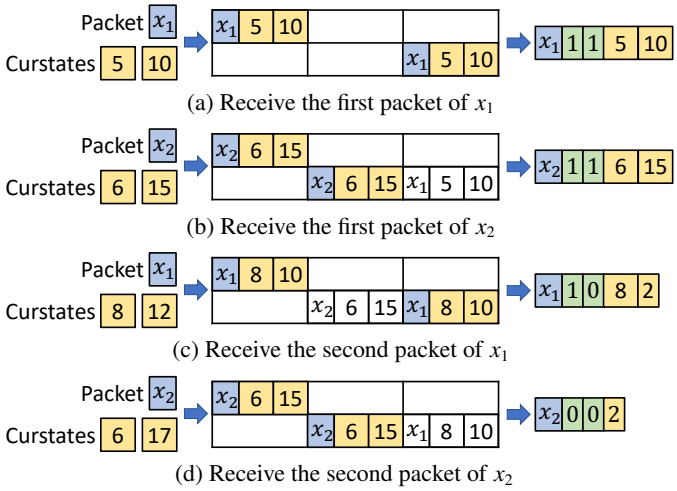
Figure 5: Update example of DeltaINT-O.

- Second, DeltaINT-O receives the first packet of x_2 with the current states of 6 and 15 (Figure 5(b)). 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.

- Third, DeltaINT-O receives the second packet of x_1 with the current states of 8 and 12 (Figure 5(c)). 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 delta is within the threshold. Thus, DeltaINT-O 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-O receives the second packet of x_2 with the current states of 6 and 17 (Figure 5(d)). 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 deltas are within the thresholds, DeltaINT-O updates the hashed buckets by $\langle x_2, 6, 15\rangle$ and only inserts a bitmap $\langle 0, 0\rangle$ into the packet.

Figure 6 shows the steps of the UPDATE procedure in DeltaINT-E. The first two steps in DeltaINT-E (i.e., Figures 6(a) and 6(b)) are the same as those in DeltaINT-O, as DeltaINT-E also embeds the current states into p_1 and p_2 and updates the embedded states in the sketch. The last two steps in DeltaINT-E are different from those in DeltaINT-O in the processing of deltas. Specifically, for p_3, after calculating the deltas (i.e., 3 and 2), DeltaINT-E finds that the delta of the second state is within the threshold (note that the thresholds of the two states are 0 and 2, respectively). It thus encodes the delta 2 into p_3 with the first current state 8 and a bitmap with bits $\langle 1, 0\rangle$. In particular, it represents the delta 2 in three bits "1xx", where "xx" are the two bits to encode a non-zero delta value for a threshold 2. For p_4, both deltas (i.e., 0 and 2) of the two states are within the thresholds. DeltaINT-E omits the zero delta for the first state (with a threshold zero) and encodes the delta 2 (in bits "1xx") for the second state into p_4, together with a bitmap of bits $\langle 0, 0\rangle$.



(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 6: Update example of DeltaINT-E.

State reconstruction. The sink extracts the INT information for each state (i.e., the complete state if the delta is non-negligible, or an empty state for DeltaINT-O or an encoded delta for DeltaINT-E if the delta is negligible) from each received packet. It then sends an event that includes all state information to the monitoring system in the control plane. In the monitoring system, DeltaINT maintains the latest embedded state for each flow. If the received event contains the complete state, DeltaINT updates the latest embedded state with the complete state; otherwise, DeltaINT-O keeps the latest embedded state to approximate the complete state, while DeltaINT-E adds the encoded delta to the latest embedded state to reconstruct the complete state.

3.3. 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 (Section 4). In fact, DeltaINT works well even with $d = 1$ in our evaluation (Section 5), as the number of conflicting flows with overlapping life cycles is small in practice, while using more rows in the sketch (i.e., $d > 1$) can provide more robust monitoring for general scenarios.

Sensitivity to the delta threshold. We argue that DeltaINT is insensitive to the choice of the delta threshold for practical network management applications. 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 complete states into packets.

Note that for fine-grained monitoring where accurate packet-level INT states need to be tracked on a per-node manner (Section 2.2), DeltaINT-O may incur large measurement errors under large delta thresholds since it omits the embedding of complete state information. In this case, DeltaINT-E should be adopted to maintain full measurement accuracy at the expense of incurring more bandwidth than DeltaINT-O (while still incurring less bandwidth than the original INT framework).

Setting the most appropriate delta threshold in practice relies on the domain knowledge for various applications. Our evaluation (Section 5) shows that DeltaINT works well even with a threshold of one for the integer states (32 bits). How to automatically set the delta threshold for general scenarios is posed as our future work.

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 states [39]. Also, practical applications only have at most three states [9]. Furthermore, as we show in Section 3.4, 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 functionalities of network management applications.

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 framework [39], but we do not make this assumption in our design.

Sketching overhead. Although DeltaINT introduces sketch-based techniques to track the embedded states for delta calculation, we emphasize that the extra overhead is limited. Note that the sketch itself is a lightweight data structure with resource efficiency guarantees (Section 3.1). Due to the limited number of hash collisions (see above), the sketch in DeltaINT requires limited amount of memory (e.g., 1 MB in evaluation (Section 5)) and limited computational overhead (e.g., only $\approx$0.01 μs on average to hash a flowkey by MurmurHash [4] used in software simulation, which is consistent with the prior findings [21]).

Extensions to other domains. DeltaINT can be extended to other domains, including wireless networks and multi-application scenarios. In wireless networks, mitigating the INT bandwidth cost can save the energy consumption of sending radio-frequency waves. Note that the monitoring system of DeltaINT may not have the latest embedded state for state reconstruction (Section 3.2) due to packet losses under unreliable wireless channels. To resolve the issue, one solution is to maintain a timeout-and-retry mechanism in each node, such that the latest embedded state can be reliably sent to the monitoring system. How to address the trade-off between the INT bandwidth cost and the reliability is our future work. In multi-application scenarios, the applications can pose different requirements on bandwidth mitigation.

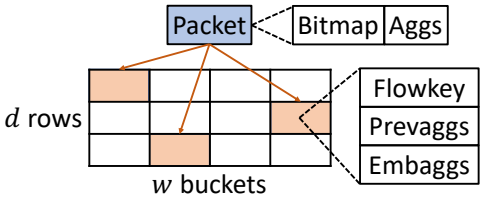


Figure 7: Sketch in per-packet aggregation.

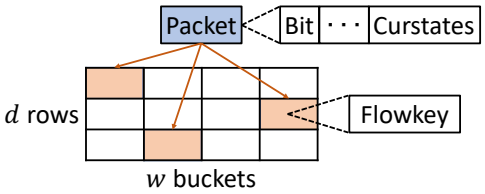


Figure 8: Sketch in static per-flow aggregation.

DeltaINT can preserve a bit in each packet, such that each application can specify which variant (i.e., DeltaINT-O or DeltaINT-E) should be used.

3.4. Fitting DeltaINT into Applications

We explain how we apply the DeltaINT framework into each family of applications described in Section 2.2.

Per-packet-per-node monitoring. The DeltaINT framework can be directly applied, such that each packet embeds the complete states with non-negligible deltas (or encodes the negligible deltas for DeltaINT-E) in 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 (Section 2.2). 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 7). 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 the complete aggregate state is not carried by the packet, DeltaINT uses the previous aggregate state recorded in the sketch (and adds the encoded delta carried in the packet to the previous aggregate state if DeltaINT-E is used). DeltaINT 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 8 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) [9]. 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 (Section 2.2). 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. For each flow in a node, to avoid enumerating all collected states, DeltaINT maintains the critical states in a *quantile sketch* [26] in the control plane as in PINT [9]. The control plane extracts all states of a flow in different nodes from each event and updates them into each corresponding 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 corresponding quantile sketch for each flow in a node, such that the quantile sketch provides approximate quantile information by enumerating a limited number of the stored states.

4. Theoretical Analysis

In this section, we present theoretical analysis of the 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 flowkey space size.*

Proof:. Each packet triggers the calculation of d hash functions and the updates for d buckets. Note that the complexity of updating one bucket is O(1). Thus, 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 since the hash collisions are the only cause of errors in DeltaINT. We characterize the errors with two user-specified parameters: (i) the maximum accepted error probability σ and (ii) the approximation coefficient ϵ that specifies 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 $d = \log \frac{1}{\epsilon\sigma}$ 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 Sections 5.2.5 and 5.3.1 in [12] and we omit the details here. □

Theorems 3 and 4 give the theoretical guarantees of the bit overhead (i.e., the number of bits for INT information) of DeltaINT-O and DeltaINT-E based on Lemma 1, respectively. 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 complete current state is $P_s = (1 - \epsilon\sigma)(1 - \alpha)$, where α is the probability that the delta exceeds the pre-specified threshold of delta calculation.*

Proof:. For each packet of a flow, DeltaINT does not embed the complete current state if and only if: (i) we can load the embedded state of the flow from the sketch A, and (ii) the delta does not exceed the threshold. The first condition means that at least one of all the d mapped buckets records the embedded state of the flow. Thus, the probability that the first condition holds is equal to $1 - (1 - e^{-\frac{n}{w}})^d = 1 - \epsilon\sigma$. For the second condition, its probability is $1 - \alpha$ based on the definition of α. Since the two conditions are independent, we have $P_s = (1 - \epsilon\sigma)(1 - \alpha)$. □

Theorem 3. *We define the* bandwidth ratio *as the ratio between the bit overhead of DeltaINT and that of the original INT framework. For DeltaINT-O, the bandwidth ratio approximates $\frac{1}{b} + 1 - P_s$, where b is the number of bits of the state.*

Proof:. Given a flow x with $z(x)$ packets, the bit overhead of flow x consists of two parts: (i) the first packet and (ii) the subsequent packets. For the first packet, DeltaINT-O must embed the state, as no embedded state is available in the sketch. Note that the bitmap requires one bit for each state, so DeltaINT-O costs $b + 1$ bits in the first part. For each of the $z(x) - 1$ subsequent packets, if DeltaINT-O can load the embedded state from the sketch (i.e., no hash

collision) and the delta is within the threshold, it embeds only one bit for the bitmap; otherwise, it embeds one bit for the bitmap and b bits for the complete state. Based on Lemma 1, the bit overhead of flow x in DeltaINT-O (denoted by $B_O(x)$) is given by $B_O(x) = (b+1) + (z(x)-1)P_s + (z(x)-1)(b+1)(1-P_s) = bP_s + z(x)(b+1-bP_s)$. Let n be the total number of flows and S be the total number of packets. By summing over all flows, the bandwidth ratio of DeltaINT-O is equal to $\frac{\sum_x B_O(x)}{bS} = \frac{bP_s n + S(b+1-bP_s)}{bS}$. Assuming that n is often much smaller than S, the bandwidth ratio approximates $\frac{1}{b} + 1 - P_s$. □

Theorem 4. *Given a delta threshold ϕ, if $\phi = 0$, the bandwidth ratio of DeltaINT-E approximates $\frac{1}{b}+1-P_s$; otherwise, it approximates $\frac{1}{b} + 1 - P_s + \frac{1+\lceil\log(2\phi)\rceil(1-\beta)}{b}P_s$, where β is the probability that a delta is zero.*

Proof:. Recall that if the delta threshold is zero, DeltaINT-E omits the zero delta from a packet as in DeltaINT-O (Section 3.2). Thus, the bandwidth ratio of DeltaINT-E is $O(\frac{1}{b} + 1 - P_s)$ as in Theorem 3. For a non-zero delta threshold, the proof is similar as Theorem 3 except with one change. To encode the delta of a state, in addition to keeping one bit in the bitmap, DeltaINT-E needs one extra bit for a zero delta, or $\lceil\log(2\phi)\rceil + 1$ extra bits for a non-zero delta. Based on Lemma 1, the bit overhead of flow x in DeltaINT-E (denoted by $B_E(x)$) is given by $B_E(x) = (b+1) + (z(x)-1)(1+1)\beta P_s + (z(x)-1)(\lceil\log(2\phi)\rceil + 1 + 1)(1-\beta)P_s + (z(x)-1)(b+1)(1-P_s)$. Following the proof of Theorem 3, by summing over all flows and assuming that the number of flows n is much smaller than the number of packets S, the bandwidth ratio of DeltaINT-E is $\frac{\sum_x B_E(x)}{bS} \approx \frac{1}{b} + 1 - P_s + \frac{1+\lceil\log(2\phi)\rceil(1-\beta)}{b}P_s$. □

Remarks. We use an example to understand the implications of the bandwidth ratios of DeltaINT-O and DeltaINT-E. We consider the measurement of a 32-bit state with a delta threshold of one, and assume that the state stays unchanged in 90% of the time. It means that $b = 32$, $\phi = 1$, $\alpha = 0.1$, and $\beta = 0.9$. Given $\epsilon = 0.1$ and $\sigma = 0.1$, we have $P_s = (1-\epsilon\sigma)(1-\alpha) = 0.99 * (1-0.1) = 0.891$. Then the bandwidth ratio of DeltaINT-O is $\frac{1}{b} + 1 - P_s = 14\%$, while the bandwidth ratio of DeltaINT-E is $\frac{1}{b} + 1 - P_s + \frac{1+\lceil\log(2\phi)\rceil(1-\beta)}{b}P_s = 17\%$. In practice, α can be smaller (i.e., larger P_s) and β can be larger, so the bandwidth mitigation of both DeltaINT-O and DeltaINT-E can be higher.

5. Evaluation

We evaluate the DeltaINT framework, including its two variants DeltaINT-O and DeltaINT-E, on several representative applications described in Section 2.2.

Methodology. We evaluate the DeltaINT framework in both software and P4 hardware [10]. For the former, we conduct simulations using both bmv2 [6] and NS3 [5]; for the latter, we compile the framework in a Tofino switch [2] connected with 40-Gbps NICs of 16 servers. We choose P4 for our hardware implementation due to its widespread usage, while DeltaINT can also be implemented by other protocol-independent languages [13, 15, 35]. We configure the framework with 1 MB memory and one hash function in each node. We also evaluate different choices of the two parameters in Section 5.6. We focus on 5-tuple flowkeys.

5.1. Gray Failure Detection

We first evaluate DeltaINT-O and DeltaINT-E in gray failure detection. We compare them with the state-of-the-art probing-based INT framework INT-path [32]. 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 [24] to implement the data planes of DeltaINT-O, DeltaINT-E, and INT-path. For fair comparisons, we configure the data plane of each scheme to 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 each data plane. 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 period of time (called the *aging time*), the control plane reports a potential gray failure.

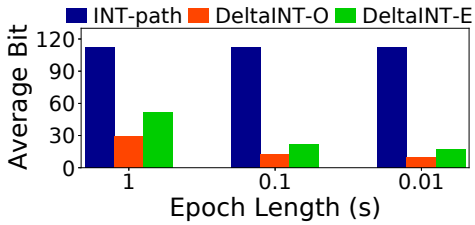
We have the following configuration for each scheme based on the original INT-path paper [32]. We set the epoch length as 0.01 s to keep the probe sending rate at 100 packets per second. We set the heavy threshold as 1 μs for heavy latency detection. To mimic gray failures, we inject link down and heavy latency events. Each scheme collects the 8-bit device ID, 8-bit ingress port, 8-bit egress port, and 32-bit latency as the states. By default, for DeltaINT-O and DeltaINT-E, we fix the delta thresholds of the device ID, ingress port, and egress port as zero, and that of the latency as 1 ns. We use the spine-leaf topology and run the experiment for 10 s.

(Exp#1) Bandwidth usage in gray failure detection. We compare the bandwidth usage of DeltaINT-O, DeltaINT-E, and INT-path. 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-O and DeltaINT-E, we also count the number of bits in the bitmap of each packet. Note that the average bit cost excludes the metadata header (Section 2.1) in the calculation, as the control plane can directly configure which states are tracked [9].

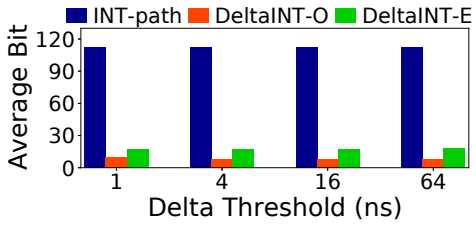
Figure 9(a) shows the average bit cost versus the epoch length, with a fixed delta threshold of 1 ns. The average bit costs of DeltaINT-O and DeltaINT-E decrease as the epoch length decreases, mainly because they only need to embed all complete states for the first probe packet of each path (which has no embedded states at the beginning), and no longer needs to embed the complete 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 9(b) shows the average bit cost versus the delta threshold, with a fixed epoch length of 0.01 s. The average bit cost of DeltaINT-O decreases as the delta threshold increases, as a larger delta threshold implies fewer latency states being embedded after delta calculation. On the other hand, the average bit cost of DeltaINT-E increases as the delta threshold increases, since more bits are needed to encode a non-zero delta under a larger delta threshold.

Overall, the average bit cost of INT-path is always 112 bits. The reason is that each path in the topology has three nodes, and each node in INT-path 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

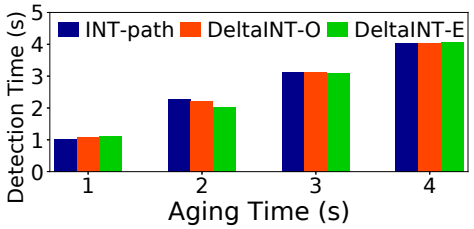


(a) Different epoch lengths

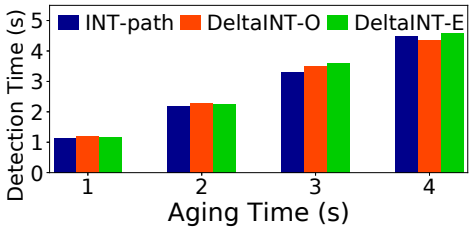


(b) Different thresholds

Figure 9: (Exp#1) Bandwidth usage in gray failure detection.



(a) Link down



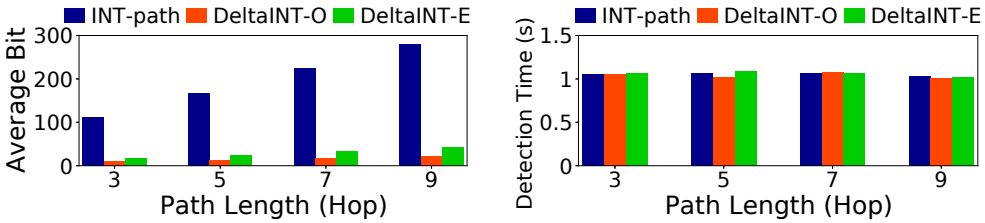
(b) Heavy latency

Figure 10: (Exp#2) Detection time in gray failure detection.

information of size 56 bits, 112 bits and 168 bits, so the average bit cost in INT-path is 112 bits. The average bit cost of INT-path is significantly larger than those of DeltaINT-O and DeltaINT-E. Specifically, the average bit cost of DeltaINT-O is as low as 8.1 bits when the delta threshold is 16 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 in each node, while DeltaINT-O only embeds those with non-negligible deltas. On the other hand, the average bit cost of DeltaINT-E is as low as 16.8 bits when the delta threshold is 1 ns and the epoch length is 0.01 s (i.e., at most 85% bandwidth mitigation). While DeltaINT-E incurs higher bandwidth overhead than DeltaINT-O due to delta encoding, it still incurs significantly less bandwidth than INT-path by avoiding the embedding of complete states.

(Exp#2) Detection time in gray failure detection. We follow INT-Detect [24] to evaluate the detection time of gray failures (including link down and heavy latency) versus the aging time. Figures 10(a) and 10(b) show that DeltaINT-O and DeltaINT-E have almost the same detection time as INT-path, as they still collect 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.

(Exp#3) Impact of path length on gray failure detection. We evaluate the impact of path length of DeltaINT-O, DeltaINT-E, and INT-path by increasing the lengths of non-overlapping paths through more leaf layers. Figure 11(a) shows that as the path length increases from 3 to 9 hops, the average bit of INT-path increases from 112 to 280 bits, while those of DeltaINT-O and DeltaINT-E increases from 9.5 to 21.2 bits and from 16.8 to 41.9 bits, respectively. The reason is that INT-path always embeds the complete states from each traversed node into each packet, and the overhead increases as the path length increases. On the other hand, as the path length increases, DeltaINT-O only embeds more bitmaps, while DeltaINT-E only embeds more bitmaps and negligible deltas. Thus, the increases in



(a) Bandwidth usage (b) Detection time

Figure 11: (Exp#3) Impact of path length on gray failure detection.

bandwidth overhead in both DeltaINT-O and DeltaINT-E are significantly less than that in INT-path.

We also consider the detection time, where we focus on the detection of link down events and fix the aging time as 1 s (similar results are observed for the detection of heavy latency events). Figure 11(b) shows that as the path length increases, DeltaINT-O and DeltaINT-E can still achieve almost the same detection time (i.e., $\approx$1 s) as INT-path. The reason is that all the three methods periodically send probe packets to monitor the network state regardless of the path length.

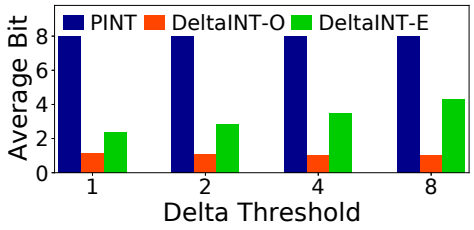
5.2. Congestion Control

We evaluate DeltaINT-O and DeltaINT-E in congestion control. We compare them with the state-of-the-art sampling-based INT framework PINT [9]. We follow PINT to implement DeltaINT-O and DeltaINT-E in NS3 [5] with the same Fat Tree topology. For fair comparisons, all schemes use the same approach of value approximation to obtain link utilization. We fix each scheme 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, each scheme uses HPCC [30] for congestion control.

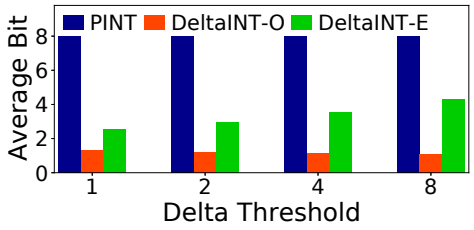
We use the following configuration. We follow the same flow size (i.e., number of bytes in a flow) distribution to generate traffic, including web search workload [8] and Hadoop workload [33], as in the PINT paper [9]. For DeltaINT-O and DeltaINT-E, we set the default delta threshold of the 8-bit link utilization as one.

(Exp#4) Bandwidth usage in congestion control. Figure 12 compares DeltaINT-O, DeltaINT-E, and PINT on INT bandwidth usage. It shows that the average bit cost of DeltaINT-O decreases to nearly one bit in both web search and Hadoop workloads, as the link utilization remains stable at most time and DeltaINT-O only needs to embed a limited number of states. On the other hand, the average bit cost of DeltaINT-E increases from 2.4 bits to 4.3 bits and from 2.6 bits to 4.3 bits as the delta threshold increases in the web search and Hadoop workloads, respectively, since it includes more bits for delta encoding under a larger delta threshold. Note that both DeltaINT-O and DeltaINT-E incur smaller bandwidth overhead than PINT, whose average bit cost is always 8 bits due to the embedding of maximum link utilization into each packet.

(Exp#5) Flow completion time in congestion control. We evaluate the flow completion times of DeltaINT-O, DeltaINT-E, and PINT based on the *slowdown* of a flow [9], 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

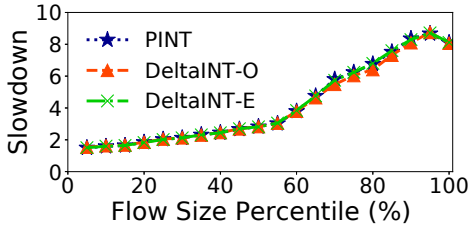


(a) Web search workload

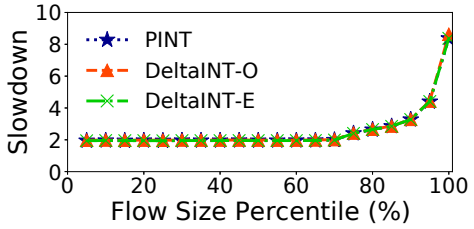


(b) Hadoop workload

Figure 12: (Exp#4) Bandwidth usage in congestion control.



(a) Web search workload



(b) Hadoop workload

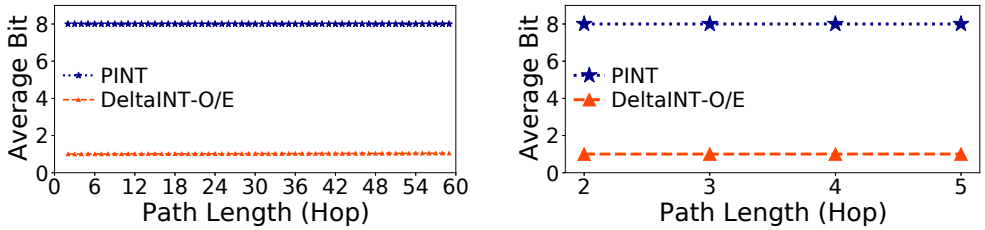
Figure 13: (Exp#5) Flow completion time in congestion control, measured by the 95th percentile slowdown (note that the flow sizes of the web search and Hadoop workloads range from 1 byte to 30 MB and from 1 byte to 10 MB, respectively).

control. We sort the flow sizes (excluding the packet headers) in ascending order, and consider the 95th percentile *slowdown* for every 5% flow size percentile. Figures 13(a) and 13(b) show that both DeltaINT-O and DeltaINT-E have similar slowdowns to PINT. The reason is that they embed all critical link utilization states with non-negligible deltas to ensure that the flow sending rate is properly adjusted for congestion control.

5.3. Path Tracing

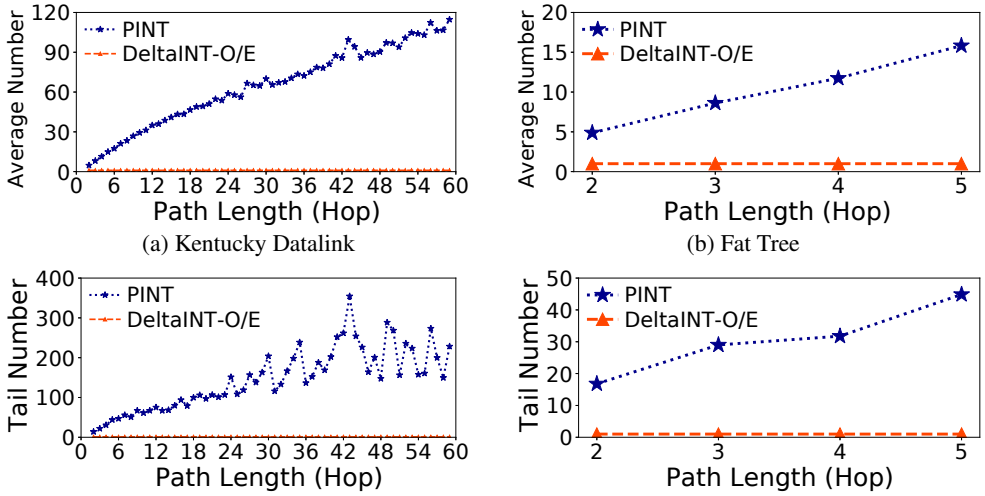
We evaluate DeltaINT-O and DeltaINT-E in path tracing. We compare them with PINT [9]. For each flow, PINT amortizes all device IDs of the path into sampled packets by distributed encoding. We follow PINT [9] to implement DeltaINT-O and DeltaINT-E in bmv2 [6] and build all schemes in Mininet [3]. We consider 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 [9]. For DeltaINT-O and DeltaINT-E, we fix the delta threshold of device ID as zero. Recall that DeltaINT-E does not encode any zero delta in a packet if the delta threshold is zero (Section 3.2), so it has the same bandwidth overhead as DeltaINT-O. In Exp#6 and Exp#7, we use DeltaINT-O/E to collectively refer to both DeltaINT-O and DeltaINT-E, and compare the results with PINT.

(Exp#6) Bandwidth usage in path tracing. We compare the INT bandwidth usage of DeltaINT-O/E and PINT. 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 14(a) and 14(b) 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-O/E increases



(a) Kentucky Datalink (b) Fat Tree

Figure 14: (Exp#6) Bandwidth usage in path tracing. Note that the results of DeltaINT-O and DeltaINT-E are identical.



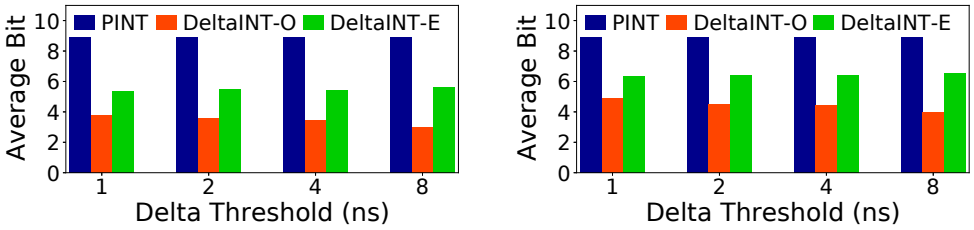
(a) Kentucky Datalink (b) Fat Tree

(c) Kentucky Datalink (d) Fat Tree

Figure 15: (Exp#7) Convergence in path tracing. Note that the results of both DeltaINT-O and DeltaINT-E are identical.

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 in DeltaINT-O/E, only the first packet of each flow carries all device IDs of the traversed nodes, while each of the subsequent packets only carries a one-bit bitmap.

(Exp#7) Convergence in path tracing. We compare the convergence of DeltaINT-O/E and PINT 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 15(a) and 15(b) show that the average number of packets in PINT grows almost linearly with the path length to around 120 and 15 in Kentucky Datalink and Fat Tree, respectively, while Figures 15(c) and 15(d) 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-O/E are always one, as DeltaINT-O/E must embed the device IDs of all traversed nodes into the first packet of each flow. Thus, DeltaINT-O/E can trace the path for a flow of any size with fast convergence (with only one packet).



(a) Web search workload (b) Hadoop workload

Figure 16: (Exp#8) Bandwidth usage in latency measurement.

5.4. Latency Measurement

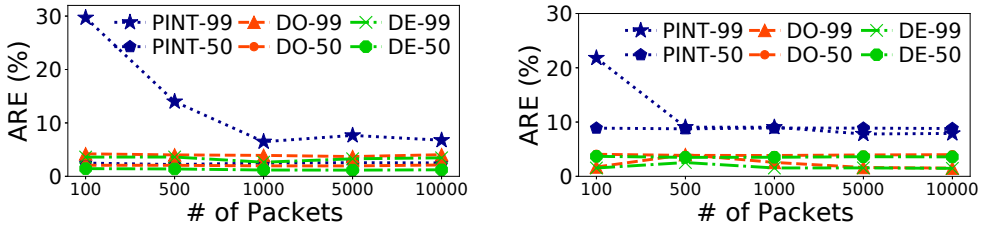
We evaluate DeltaINT-O and DeltaINT-E in latency measurement. We again compare them with PINT [9], using the same workloads as in Section 5.2. In the data plane, all schemes use the same value approximation to obtain an 8-bit latency as in Section 5.2. In the control plane, for each flow in a node, all schemes use the same quantile sketch called KLL [26], configured as in the PINT paper [9]. For DeltaINT-O and DeltaINT-E, we set the default delta threshold of the latency as 1 ns.

(Exp#8) Bandwidth usage in latency measurement. We compare the bandwidth usage of DeltaINT-O, DeltaINT-E, and PINT. We vary the delta threshold from 1 ns to 8 ns. Figure 16 shows the average bit costs of DeltaINT-O, DeltaINT-E, and PINT. Specifically, the average bit cost of PINT is around 8.9 bits under both web search and Hadoop workloads. In contrast, as the delta threshold increases, the average bit cost of DeltaINT-O decreases to around 3 bits and 4 bits in the two workloads, respectively, due to the omission of more states with negligible deltas. On the other hand, as the delta threshold increases, the average bit cost of DeltaINT-E increases from 5.4 bits to 5.6 bits and from 6.3 bits to 6.5 bits in the web search and Hadoop workloads, respectively, due to the use of more bits for delta encoding under a larger threshold. Although PINT reduces the bandwidth overhead by sampling, both DeltaINT-O and DeltaINT-E still have smaller average bit costs than PINT as they only embed the complete latency state into a packet when the delta exceeds the threshold.

(Exp#9) Accuracy in latency measurement. We compare the accuracy of latencies estimated by the quantile sketches deployed in the control plane for DeltaINT-O, DeltaINT-E, and PINT. We consider the average relative error (ARE) of the 50th and 99th percentile latencies. Figure 17 shows that PINT always has a larger ARE than DeltaINT-O and DeltaINT-E, especially in the 99th percentile latency. The reason is that PINT embeds latency by sampling, which could miss critical information. However, DeltaINT-O and DeltaINT-E avoid sampling and embed any complete critical latency state with non-negligible delta.

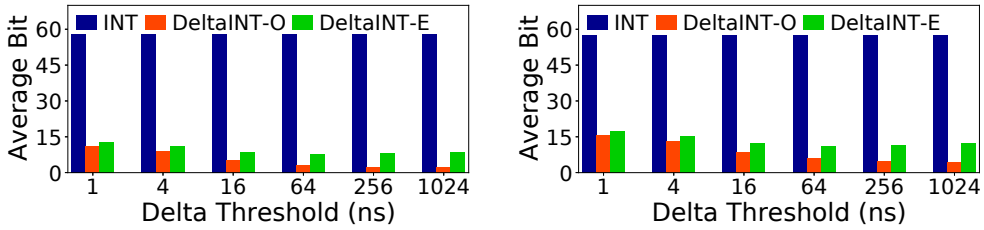
5.5. Fine-grained Monitoring

We evaluate DeltaINT-O and DeltaINT-E in fine-grained monitoring. We compare them with the original INT framework [27], using the same workloads as in Section 5.2. We consider a network management application that collects the latency state from each node and reports the latency information to the monitoring system for network



(a) Web search workload (b) Hadoop workload

Figure 17: (Exp#9) Accuracy in latency measurement (DO stands for DeltaINT-O and DE stands for DeltaINT-E).



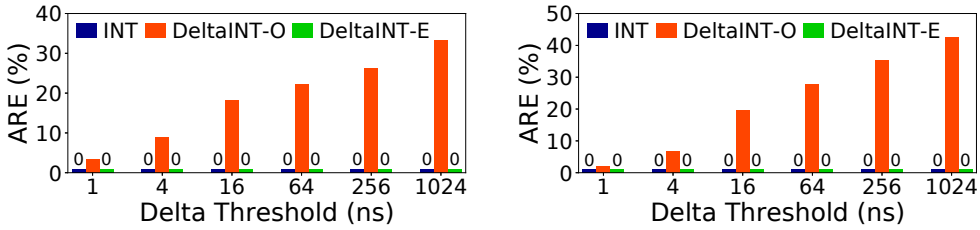
(a) Web search workload (b) Hadoop workload

Figure 18: (Exp#10) Bandwidth usage in fine-grained monitoring.

debugging. Specifically, we configure DeltaINT-O, DeltaINT-E, and the original INT framework to track the 32-bit latency directly provided by each node for each traversed packet. For DeltaINT-O and DeltaINT-E, we vary the delta threshold of the latency state from 1 ns to 1024 ns.

(Exp#10) Bandwidth usage in fine-grained monitoring. Figure 18 shows the average bit costs of DeltaINT-O, DeltaINT-E, and the original INT framework for different delta thresholds. Specifically, as the delta threshold increases, the average bit cost of DeltaINT-O decreases from 11.2 bits to 2.1 bits and from 15.8 bits to 4.3 bits in the web search and Hadoop workloads, respectively. On the other hand, the average bit cost of DeltaINT-E in the web search workload first decreases from 12.8 bits to 7.7 bits as the delta threshold increases from 1 ns to 64 ns, and then increases to 8.7 bits as the delta threshold increases to 1024 ns. The trend is similar to that in the Hadoop workload, where the average bit cost of DeltaINT-E first decreases from 17.2 bits to 11.2 bits as the delta threshold increases from 1 ns to 64 ns, and then increases to 12.4 bits as the delta threshold increases to 1024 ns. The reason is that as the delta threshold first increases, DeltaINT-E avoids embedding more complete latency states and the extra bit cost for delta encoding remains limited. However, as the delta threshold continues to increase, while DeltaINT-E already avoids embedding the complete latency states, while the extra bit cost for encoding the non-zero deltas becomes more significant. Nevertheless, both DeltaINT-O and DeltaINT-E significantly reduce the bandwidth overhead of the original INT framework, whose average bit cost is 57.8 bits and 57.7 bits in the web search and Hadoop workloads, respectively.

(Exp#11) Accuracy in fine-grained monitoring. We compare the accuracy of the latency states collected in the data plane for DeltaINT-O, DeltaINT-E, and the original INT framework. We consider the average relative error (ARE) of the latency state of a node reported by each scheme with respect to the actual latency state of a node. Recall that both



(a) Web search workload (b) Hadoop workload

Figure 19: (Exp#11) Accuracy in fine-grained monitoring.

DeltaINT-E and the original INT framework have zero AREs. In contrast, DeltaINT-O does not report the latency state if the delta of a state is within the pre-specified threshold, but directly uses the latest embedded state as the current state. Thus, it incurs a non-zero ARE. In this experiment, we study the severity of the ARE in DeltaINT-O in fine-grained monitoring.

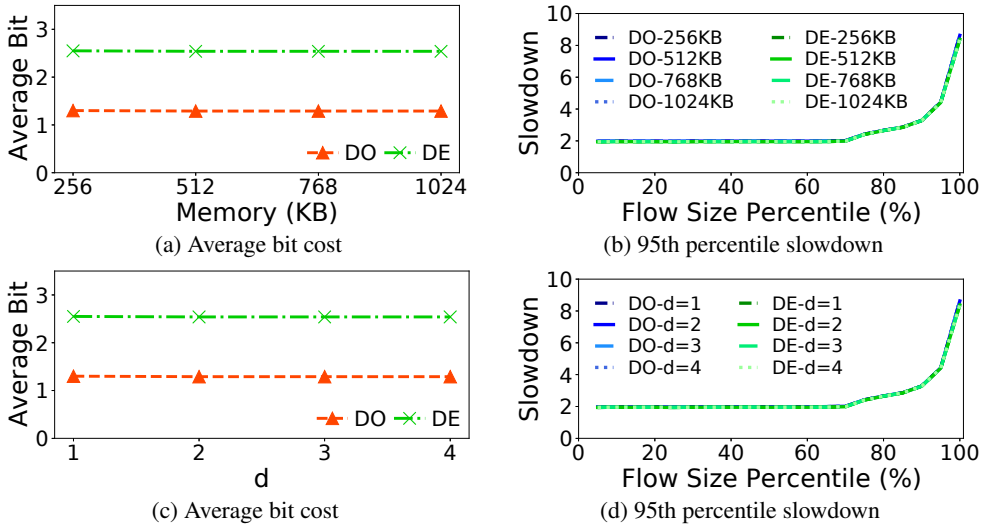
Figure 19 shows that as the delta threshold increases, the ARE of DeltaINT-O increases from 3.5% to 33.2% and from 2.1% to 42.7% in the web search and Hadoop workloads, respectively. We emphasize that while DeltaINT-O has a high ARE especially for a large delta threshold, its limitation only appears in the applications that require fine-grained measurement. For common applications (e.g., the applications in Section 5.1 to Section 5.4), DeltaINT-O does not compromise the functionality of network management for a reasonably small delta threshold, while achieving significant bandwidth mitigation.

5.6. Sketching Parameter Sensitivity

(Exp#12) Sensitivity to sketching parameters. We validate the limited sensitivity of DeltaINT-O and DeltaINT-E 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 in Section 5.2 and use the Hadoop workload with around 3.3 million flows. Figures 20(a) and 20(b) 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 of each scheme do not change. Figures 20(c) and 20(d) show the results versus d, where we fix the entire memory as 1 MB. The results are similar, since DeltaINT-O and DeltaINT-E only focus on the adjacent packets of each flow, while most hash-colliding flows do not have overlapping life cycles. Thus, the impact of hash collisions on DeltaINT-O and DeltaINT-E is limited.

5.7. Evaluation in Hardware

We implement each of DeltaINT-O and DeltaINT-E in P4 [10] with less than 400 lines of code in each family of applications. We compile them in the Tofino chipset [2] to demonstrate the feasible hardware deployment. Specifically, we deploy each of DeltaINT-O and DeltaINT-E in the egress pipeline. We realize each row of the sketch in the data plane as an array of registers. As the number of buckets must be a power of two for each register array due to the constraints of the Tofino switch, we assign the maximum possible number of buckets within the 1-MB on-chip



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

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

Figure 20: (Exp#12) Sensitivity to sketching parameters (DO stands for DeltaINT-O and DE stands for DeltaINT-E).

memory. If the number of bits of the recorded data in each bucket exceeds the hardware limitation (e.g., at most 64 bits in the Tofino switch), we split it into multiple register arrays with the same length. For each register array, we hash the flowkey to locate the corresponding register. All the three primitives about the register, including StateLoad, DeltaCalc, and StateUpdate, are performed in one ALU. For DeltaCalc, in order to compare the absolute value of delta with the threshold (i.e., $|delta| \leq \phi$) in a single ALU, we compute "$delta \geq -\phi$ and $delta \leq \phi$", which involves two relational operations and one logical operation. Finally, according to the return value of DeltaCalc, we embed the complete states (or encode the deltas for DeltaINT-E) into the packet header via MetadataInsert. For each family, we use the same setting of the application as mentioned in Section 5.

(Exp#13) Hardware resource usage. Table 1 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 both DeltaINT-O and DeltaINT-E, we present the results of all four families of applications, including per-packet-per-node monitoring (F1) (based on gray failure detection), per-packet aggregation (F2) (based on congestion control), static per-flow aggregation (F3) (based on path tracing), and dynamic per-flow aggregation (F4) (based on latency measurement). Specifically, DeltaINT-O needs at most 1,264 KB of SRAM for both sketching and match-action tables, accounting for only 8.23% of the total SRAM. Although DeltaINT-O 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-O incurs at most 23 actions and consumes only at most six stateful ALUs (12.5% of total). Finally, DeltaINT-O uses at most 126 bytes of PHV (16% of total) to transmit messages between different stages. Note that nearly half of them are used for the basic network functions like packet forwarding. For DeltaINT-E, its PHV usage is slightly larger than that of DeltaINT-O due to delta encoding.

Table 1: (Exp#13) Hardware resource usage (percentages in brackets are fractions of total resource usage; DO stands for DeltaINT-O and DE stands for DeltaINT-E).

	SRAM (KB)	No. stages	No. actions	No. ALUs	PHV size (bytes)
DO-F1	1152 KB (7.50%)	4 (33%)	23 (nil)	6 (12.5%)	126 (16%)
DO-F2	1264 KB (8.23%)	6 (50%)	13 (nil)	5 (10.42%)	112 (15%)
DO-F3	975 KB (6.35%)	3 (25%)	10 (nil)	3 (6.25%)	105 (14%)
DO-F4	848 KB (5.52%)	3 (25%)	10 (nil)	4 (8.33%)	119 (15%)
DE-F1	1152 KB (7.50%)	4 (33%)	23 (nil)	6 (12.5%)	129 (17%)
DE-F2	1264 KB (8.23%)	6 (50%)	13 (nil)	5 (10.42%)	112 (15%)
DE-F3	975 KB (6.35%)	3 (25%)	10 (nil)	3 (6.25%)	105 (14%)
DE-F4	848 KB (5.52%)	3 (25%)	10 (nil)	4 (8.33%)	120 (16%)
INT	16 KB (0.10%)	2 (16%)	4 (nil)	0 (0%)	112 (15%)

We further use the baseline switch P4 program `switch.p4` provided by the Tofino switch to evaluate the hardware resource usage of the original INT framework. We disable all network functions except the original INT framework in `switch.p4`, and compile it in the same testbed as DeltaINT-O and DeltaINT-E. Here, we focus on the performance of the original INT framework in F2, which incurs the most number of stages in DeltaINT-O and DeltaINT-E. From Table 1, DeltaINT-O and DeltaINT-E incur more SRAM, stages, stateful ALUs, and PHV size than the original INT framework due to the tracking of states and delta calculation for bandwidth mitigation, while the hardware resource usage is still limited compared with the total available resources in the Tofino switch.

6. Summary of Results and Conclusions

We summarize the evaluation results for our DeltaINT framework (Section 5) as follows:

- In gray failure detection, DeltaINT-O and DeltaINT-E incur 93% and 85% less INT bandwidth than the probing-based INT framework INT-path [32], respectively (Exp#1), with similar detection time (Exp#2). Also, as the path length increases, the increases in the INT bandwidth in both DeltaINT-O and DeltaINT-E are significantly less than that in INT-path, while maintaining similar detection time (Exp#3).
- In congestion control, DeltaINT-O consumes only one bit per packet on average and DeltaINT-E consumes as low as 2.4 bits (and up to 4.3 bits) compared with the sampling-based INT framework PINT [9], which requires 8 bits (Exp#4), with similar flow completion time (Exp#5).
- In path tracing, both DeltaINT-O and DeltaINT-E consume nearly one bit per packet on average, while PINT requires 8 bits (Exp#6). They also have better convergence than PINT (Exp#7).
- In latency measurement, DeltaINT-O and DeltaINT-E consume 3 bits and 5.4 bits per packet on average, respectively, while PINT requires at least 8.9 bits (Exp#8). They also incur smaller errors in latency measurement (estimated by the quantile sketches in the control plane) than PINT (Exp#9).

- In fine-grained monitoring, DeltaINT-O and DeltaINT-E consume as low as 2.1 bits and 7.7 bits per packet on average, respectively, while the original INT framework requires at least 57.7 bits (Exp#10). DeltaINT-E also maintains full accuracy in the latency states collected from the data plane as in the original INT framework (Exp#11).
- For INT bandwidth usage and the performance of each representative application (Section 2.2), both DeltaINT-O and DeltaINT-E are insensitive to sketching parameters (Exp#12).
- Both DeltaINT-O and DeltaINT-E consume limited resources when being deployed in a Tofino switch (Exp#13).

DeltaINT is a novel INT framework that achieves extremely low bandwidth overhead for INT and has the properties of generality, convergence, and compatibility. The key insight of DeltaINT is to embed the complete states into each packet only if the state changes are non-negligible over the traversed packets. DeltaINT comprises two variants to make a trade-off between bandwidth usage and measurement accuracy: (i) DeltaINT-O, which omits the state from the packets if the state changes are within pre-specified thresholds and hence achieves the lowest possible bandwidth usage, and (ii) DeltaINT-E, which encodes the state changes that are within pre-specified thresholds and hence maintains the full measurement accuracy. We theoretically derive the time/space complexities, the error probability, and the guarantees of bandwidth mitigation for both DeltaINT-O and DeltaINT-E. We show via software simulations that DeltaINT-O and DeltaINT-E can significantly mitigate the bandwidth overhead compared with state-of-the-art INT schemes. We further show via P4 hardware implementation that DeltaINT-O and DeltaINT-E have limited usage of hardware resources in the Tofino switch deployment.

Acknowledgment

This work was supported by the National Key R&D Program of China (2019YFB1802600), Joint Funds of the National Natural Science Foundation of China (U20A20179), and National Natural Science Foundation of China (62172007).

References

[1] Change detection algorithm of Intel. `https://builders.intel.com/docs/networkbuilders/in-band-network-telemetry-detects-network-performance-issues.pdf`.
[2] Intel Tofino. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html`.
[3] Mininet. `http://mininet.org/`.
[4] Murmurhash. `https://github.com/aappleby/smhasher/`.
[5] Network Simulator3. `https://www.nsnam.org/`.
[6] P4 switch behavioral model. `https://github.com/p4lang/behavioral-model`.
[7] 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.
[8] 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.
[9] 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.
[10] 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.
[11] 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.

[12] 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.

[13] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta. Switch code generation using program synthesis. In *Proc. of ACM SIGCOMM*, 2020.

[14] 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.

[15] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker. Modular switch programming under resource constraints. In *Proc. of USENIX NSDI*, 2022.

[16] 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.

[17] 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.

[18] 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.

[19] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[20] J. Hyun, N. Van Tu, J.-H. Yoo, and J. W.-K. Hong. Real-time and fine-grained network monitoring using in-band network telemetry. *International Journal of Network Management*, 29(6):e2080, 2019.

[21] D. Ivanchykhin, S. Ignatchenko, and D. Lemire. Regular and almost universal hashing: an efficient implementation. *Software: Practice and Experience*, 47(10):1299–1323, 2017.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] S. Sheng, Q. Huang, and P. P. Lee. DeltaINT: Toward general in-band network telemetry with extremely low bandwidth overhead. In *Proc. of IEEE ICNP*, 2021.

[35] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *Proc. of ACM SIGCOMM*, 2016.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] K. Yang, Y. Li, Z. Liu, T. Yang, Y. Zhou, J. He, T. Zhao, Z. Jia, Y. Yang, et al. SketchINT: Empowering INT with TowerSketch for per-flow per-switch measurement. In *Proc. of IEEE ICNP*, 2021.

[42] 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.

[43] 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.

[44] 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.