Task Scheduling for Probabilistic In-band Network Telemetry

Wei Chen, Ye Tian, *Member, IEEE,* Zhongxiang Wei, Jiangyu Pan, Xinming Zhang, *Senior Member, IEEE*

***Abstract*—In-band Network Telemetry (INT) is a novel framework for monitoring network health in real-time, and its recent variant, Probabilistic INT (PINT), reduces its bandwidth consumption with a probabilistic approach. However, as we show in this paper, a PINT task can be successfully accomplished only when it is allocated a sufficient number of packets, and if there are many tasks executed in parallel, packets become a scarce resource. Meanwhile, today's production network generally executes multiple measurement tasks for tracing different network states simultaneously. Therefore, in such a context, scheduling parallel PINT tasks on one single INT flow that has a limited number of packets becomes a critical problem. In this paper, we address this problem for the first time. We propose an algorithm that efficiently schedules multiple parallel PINT tasks on a flow by allocating the flow's packets to the tasks and showing that the allocation is optimal. We realize the algorithm with a packet processing pipeline and implement it on software and hardware-programmable switches. Comprehensive evaluation on a FatTree testbed shows that at a low scheduling overhead, our algorithm can conduct parallel PINT tasks to detect various network faults in a timely and accurate manner. Additionally, the algorithm accomplishes more PINT tasks with higher quality than the alternative solutions.**

***Index Terms*—Network measurement, Probabilistic In-band Network Telemetry (PINT), task scheduling, resource allocation.**

I. Introduction

Faults and errors are inevitable in today's production networks. For troubleshooting network malfunctions, monitoring network health in real-time is essential. Sampling-based network monitoring solutions, such as NetFlow [1] and sFlow [2], have been successfully applied for decades; however, these methods miss many small flows, and thus, are inadequate today. Meanwhile, with the rapid development of software-defined networking (SDN) and data plane programmability, innovations in network telemetry have attracted increasing attention in recent years (e.g., [3] [4] [5] [6] [7] [8] [9] [10] [11]).

One critical problem in network telemetry is efficiently reporting measurement data to analyzers. There are two representative approaches: out-band network telemetry and in-band network telemetry (INT). The former employs dedicated channels to transfer measurement data to analyzing servers. However, such an approach may raise a scalability of concern, as a large volume of measurement data needs to be transmitted. Instead, in INT, telemetry data are carried within a packet header, and as the packet travels along a path, network state values on all the switches that are enroute can be collected by the sink. The benefit of INT is obvious, as no dedicated out-of-band channel is needed, while measurement data can be collected at the line rate.

However, by carrying network state values in the packet header, INT consumes nontrivial bandwidth, which considerably reduces a flow's goodput and increases its completion time. To overcome this problem, Probabilistic INT (PINT) was proposed in [12]. Rather than using one single packet to carry network state values from all the switches enroute, in PINT, a packet carries data from only one switch that is randomly selected along the path, and a path-wide measurement result is collectively gathered by several different packets. The chance that network state values on all the switches along a path can be successfully collected depends on the amount of packets that are used for collecting them. Our analysis in this paper shows that to ensure a high success probability, over $1,000$ packets are required for a path of only 5 hops. Obviously, with PINT, packets become a scarce resource.

Meanwhile, for troubleshooting various network faults, multiple network states are generally traced in parallel. For example, in data center networks, packet counters and switch IDs are traced for preventing forwarding loops and detecting silent blackholes [5] [13] [8] [7] [9] [10]. In addition, interarrival time, port utilization, and queue occupancy are traced for congestion control and preventing load imbalances [3] [14]. In industrial wireless networks, transmitting delay, signal strength, and interference are traced for QoS-aware routing and energy saving [15]. Note that different measurement tasks may have various requirements for collecting measurement data. Some tasks should be conducted more frequently and some network states should be collected within shorter intervals than others. For example, the static network state, such as switch ID, needs to be collected only once, as long as the path is unchanged. However, for some time-varying network states, such as packet counters and interarrival times, their values should be collected timely and at relatively high frequency to detect dynamic network events, such as microbursts and congestion.

Motivated by the above observation, in this paper, we consider the following problem: *When multiple parallel PINT*

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grants 61672486 and 62072425 and in part by the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant XDC02070300. *(Corresponding author: Ye Tian (yetian@ustc.edu.cn).)*

The authors are with the Anhui Key Lab on High Performance Computing, School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China, 230026. (e-mail: {szcw33, wz199758, pjiangyu}@mail.ustc.edu.cn; {yetian, xinming}@ustc.edu.cn)

tasks are imposed on a single INT flow[1] *that has a limited number of packets, how can the tasks be scheduled according to their diverse requirements for collecting measurement data?* To address this problem, we propose a scheduling algorithm that allocates the "right" numbers of packets to different tasks. We aim to meet three objectives with the algorithm. First, the algorithm should be *effective*, meaning that the algorithm should accomplish as many PINT tasks as possible. Moreover, a task should collect the network state at an appropriate frequency, and the collected sample data should meet the task's requirement regarding time proximity. Second, the algorithm should be *practical*. In particular, switches should not expect expensive interventions, such as rule insertions and updates, frequently from the network control plane. Third, the algorithm should be *realizable* with a Reconfigurable Match-Action Table (RMT) [16] [17] pipeline, which is essential for hardware implementation. Additionally, since P4 [18] has already become the de facto standard for data plane programmability, a P4 implementation is preferred. Driven by these objectives, we make two contributions in this paper.

- **Analysis and design**: We start by analyzing the relation between the number of packets of a flow that are allocated to a PINT task and the task's success probability. Based on the analysis, we formulate the PINT task scheduling problem and solve it with an algorithm that allocates optimal numbers of packets to different tasks. We also propose a batch allocation scheme to preserve the time proximity of the collected sample data and improve the task success probabilities.
- **Realization and evaluation**: We propose realizing the task scheduling algorithm with a five-stage RMT pipeline and implementing the pipeline with P4 on software and hardware-programmable switches [19]. [20] Comprehensive evaluations show that our proposed algorithm outperforms other solutions with respect to both quality and quantity when accomplishing PINT tasks. Experiments on a FatTree testbed confirm that our method can be practically applied to schedule parallel PINT tasks for detecting various real-world network faults, and the evaluation on the hardware switch suggests that the overhead caused by PINT task scheduling is insignificant.

Previous works on resource allocation among network measurement tasks are focused mainly on in-switch memories, such as TCAM [21] and SRAM [3] [22]. To the best of our knowledge, we are the first to consider a flow's packets as a scarce resource and allocate them among parallel measurement tasks. For the remainder of this paper, we give a brief introduction to INT-based network diagnosis and discuss the related works in Sec. II. Sec. III formulates the PINT task scheduling problem and presents our solution. A performance evaluation is given in Sec. IV, and we conclude this paper in Sec. V.

[1]As in PINT [12], in this work, we refer to an INT flow as all the packets traveling from the same origin to the same destination over a fixed single path. Note that an INT flow may contain packets from multiple application sessions. We use the terms "flow" and "INT flow" interchangeably in this work.

II. Background and Related Work

A. INT-based Network Diagnosis

Today's production network is composed of a wide range of network elements, including many types of switches and middleboxes, and faults arise from any single or combination of these elements. In the following, we describe a number of network faults that commonly occur and how INT-based methods can detect them.

- **Path deviation**: Switches or middleboxes mistakenly forward packets, causing the actual forwarding paths to deviate from the planned paths. Such a fault can be detected by checking the switch IDs of all the switches that are enroute between the origin and the destination [4] [5] [8].
- **Forwarding loop**: A faulty switch or middlebox may throw packets back toward their origin, resulting in forwarding loops. Such a fault can be detected by comparing the counters of forwarded packets on all the switches along the path that contains loop [4] [5] [8] [10].
- **Silent blackhole**: A switch with buggy software or faulty hardware randomly drops packets without reporting. Such a fault can be detected by comparing the counters of received packets on all the switches that are enroute. A hop whose downstream switches received substantially fewer packets is considered a blackhole switch [6] [12].
- **Congestion**: Bursting traffic and load imbalances can cause congestion, which leads to longer RTTs, higher packet loss rates, and lower throughputs. Congestion can be detected by comparing packet interarrival time distributions on switches or by comparing queue occupancies of switch ports [14] [12].
- **DDoS attack**: A DDoS attack occurs when numerous attacking end hosts across the network collectively send a large volume of traffic to the victim. The attackers' positions can be detected by determining the traffic contribution from all ingress switches that reach a DDoS victim switch [13].

B. Related Work

Existing studies focus on two aspects for improving data plane visibility in production networks. Efforts are made to enable switches to trace network states more efficiently. There is a rich literature on sketch-based measurement frameworks that implement a probabilistic data structure (i.e., sketch) for tracing traffic characteristics [23]. In particular, Yu et al. [3] presented a software-defined traffic measurement architecture named OpenSketch and designed a three-stage pipeline that contained various sketches to support different measurement tasks. Yang et al. [24] introduced a generic sketch named Elastic Sketch that was fast and accurate in network measurements, and was adaptive to traffic characteristics. Zhou et al. [25] proposed a set of common frameworks, each for a family of traffic measurement solutions that shared the same implementation structure. Zhang et al. [26] designed a structure named CocoSketch that was capable of supporting partial key queries.

The other aspect describes how to report traced measurement data to analyzers. The approaches can be classified into two categories: out-band and in-band. For out-band network telemetry, Handigol et al. [5] used a separate VLAN to collect packet histories from OpenFlow switches for network diagnosis. Zhu et al. [6] exploited the "match and mirror" functionality of commodity switches to mirror packets to analyze servers through dedicated links. However, such an approach may raise a scalability concern, as it could introduce a large volume of measurement traffic. To reduce measurement overhead, some systems proposed retrieving the entire or part of a sketch via the southbound interface of the controller (e.g., [3] [7]) and carefully partition the measurement data stored at end-hosts and switches (e.g., [27]).

Furthermore, in-band network telemetry (INT), which keeps measurement data in packets, has attracted increasing attention in the last few years. Tammana et al. [8] proposed embedding packet trajectory information into packet headers at each switch that was enroute and analyzing it at the end-hosts. Moreover, switch memory is used to store pointers to end-hosts where relevant telemetry data are stored [9]. Jeyakumar et al. [4] proposed allowing packets to access and carry switch state information and designing a concise set of instructions for switches to collect the states at the line rate. Sonchack et al. [28] proposed carefully partitioning processing between ASIC hardware and application software, and developed a switch accelerated telemetry system that could monitor network traffic at the line rate. Niu et al. [29] proposed an INT system to visualize an IP-over-optical network in real-time. Basat et al. [12] presented the Probabilistic In-band Network Telemetry (PINT) framework, which employed a number of probabilistic techniques to reduce the bandwidth consumption in carrying INT data. Zhao et al. [11] combined INT and device-local sketches by splitting sketches on switches into sketchlets embedded in packet headers and aggregated the sketchlets to restore the sketches at the end-hosts for analyzing flow statistics. Sheng et al. [30] presented DeltaINT, which reduced INT bandwidth consumption by selectively carrying network states only when their values changed substantially. Yang et al. [31] proposed constructing a novel sketch named TowerSketch at the end-host with an INT approach. A comprehensive survey on INT was presented in [32], and INT was standardized by state-of-the-art programmable data planes, such as P4 [33].

With regard to resource allocation in network measurements, Yu et al. [3] allocated SRAM resources on switches used in multiple measurement tasks. Moshref et al. [21] [22] proposed an adaptive measurement framework that dynamically adjusted a switch's TCAM and SRAM devoted to each measurement task while ensuring a user-specified level of accuracy. However, most previous works focused on allocating in-switch memory resources, and as far as we know, we are the first to consider an INT flow's packets as a scarce resource and allocate them among different measurement tasks.

III. PINT Task Scheduling

In this section, we first analyze the relation between the number of packets allocated to a PINT task and the task's success probability. Then, based on the analysis, we formulate the task scheduling problem and present our solution.

TABLE I

Notation	Meaning
$(p_1, \cdots, p_n)$	An INT flow of n packets, p_i is the i^{th} packet.
$(s_1, \cdots, s_m)$	An m-hop network path, s_j is the j^{th} switch.
$v_k(p_i, s_j)$	Value of k^{th} state on switch s_j observed by packet p_i.
$S_m(x)$	Lower bound of success probability of accomplishing a task on a path of length m by allocating x packets.
K	Total num. of PINT tasks.
$\{u_k\}$	Utilities of PINT tasks.
$\{q_k\}$	Switch parameters for allocating packets to tasks.
$\mathbf{h}(p_i, j)$	Hashing function for selecting a switch in a PINT task.
$\mathbf{g}(p_i)$	Hashing function for allocating p_i to a PINT task.

A. PINT Task Success Probability

We consider a per-flow measurement task as in [12], where an INT flow containing n packets travels a network path composed of m switches. As listed in Table I, the flow is denoted as $(p_1, \cdots, p_n)$, with p_i representing the i^{th} packet in the flow, and the path is denoted as $(s_1, \cdots, s_m)$, with s_j representing the j^{th} switch that is enroute along the path. Each switch has a number of states, such as switch ID, packet and byte counters, packet interarrival time, port utilization, queue occupancy, etc. In particular, many network states, such as counter, time interval, utilization and occupancy ratios, are time-varying and require being repeatedly collected. In INT [4] [33] [34], when a switch receives a packet, it can record the current values of its states in the INT field of the packet's header, and the packet carries the data to the destination. We use $v_k(p_i, s_j)$ to denote the value of the k^{th} state on a switch s_j that can be recorded in packet p_i's header.

An INT task is accomplished by using one packet to carry the state values from all the switches that are enroute, i.e., $v_k(p_i, s_1), \cdots, v_k(p_i, s_m)$. However, INT consumes nontrivial bandwidth by carrying up to m state values in each packet's header, which considerably reduces the flow's goodput and increases its completion time. In PINT [12], a packet carries the state value from only one switch to reduce the bandwidth consumption, and since there are m switches along the path, which switch's state value will be carried is determined in a probabilistic way.

More specifically, in PINT, when a switch s_j receives a packet p_i, it computes a hash value $\mathbf{h}(p_i, j)$ in $[0, 1)$ based on the packet data and the switch's position index. If $\mathbf{h}(p_i, j) \leq \frac{1}{j}$, s_j overwrites the INT field in p_i's header with its state value $v_k(p_i, s_j)$, regardless of whether the field has been written by an upstream switch. Simple analysis shows that when a packet reaches its destination, it has an equal chance to carry each switch's state value, i.e., the probability of carrying $v_k(p_i, s_j)$ in p_i's header is $\frac{1}{m}$, for $\forall s_j \in \{s_1, \cdots, s_m\}$. Clearly, after receiving a sufficient number of packets, all the state values of the switches that are enroute can be collectively gathered.

The procedure of accomplishing a PINT task (i.e., collecting a state's values from all the switches enroute) is a *coupon collecting game* [35]. Suppose n' is the number of packets

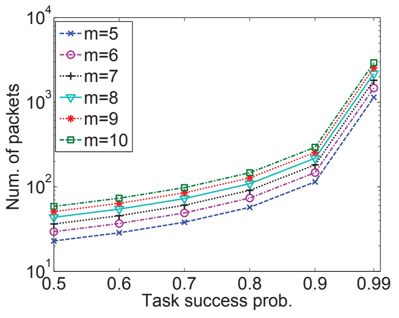


Fig. 1. Number of packets required for accomplishing a PINT task on a path of different lengths (m) above various success probabilities.

required for accomplishing a PINT task on an m-hop path; then, its expectation can be computed as follows:

$$h_m = E[n'] = m \ln m + \gamma m + 1/2 + O(1/m) \tag{1}$$

where $\gamma \approx 0.5772$. Note that h_m depends only on the path length m.

According to Markov's inequality [36], for any $x \geq h_m$, we have $\Pr[n' \geq x] \leq \frac{h_m}{x}$. In other words, if x packets are allocated to a PINT task, then the probability that the task can be successfully accomplished is lower bounded as follows:

$$S_m(x) = 1 - \frac{h_m}{x} \tag{2}$$

Following (2), we present the numbers of packets required for accomplishing a PINT task on network paths of different lengths with various success probabilities in Fig. 1. The figure shows that to achieve a certain success probability, accomplishing a task on a longer path requires more packets, as state values on more switches need to be collected. Moreover, if we want to ensure a higher success probability, more packets should be allocated. For example, for a PINT task on a 5-hop path to be accomplished with a chance above 80%, we need to allocate at least 58 packets; if we want to ensure a 99% success probability, as many as $1,144$ packets are needed. Obviously, for PINT, making good use of the limited packets in an INT flow becomes a critical issue.

B. Multiple Tasks and Problem Formulation

Our previous analysis considered only one single PINT task. However, as we discuss in Sec. I, a production network generally traces multiple network states to troubleshoot various faults. In this section, we consider that a switch is capable of tracing a total number of K different states, and collecting each state corresponds to a different PINT task. We recognize that different measurement tasks have various requirements in collecting measurement data [37] [38]. For example, a static network state, such as a switch ID, needs to be collected only once, while for some time-varying states, such as packet counters and interarrival times, the value should be collected more frequently.

With the above observation, in task scheduling, we associate each PINT task, such as $task_k$, with a nonnegative utility u_k, which reflects its requirement in tracing the corresponding network state. Generally, a task with a higher utility means that the measurement data should be collected more frequently.

To enable a switch to conduct multiple tasks in parallel, we introduce in every switch a uniform hash function $\mathbf{g}(.)$ in $[0, 1)$ and a set of parameters $\{q_k\}_1^K$ with $\sum_k q_k = 1$. Upon receiving a packet p_i, the switch computes a hash value $\mathbf{g}(p_i)$, and if $\mathbf{g}(p_i)$ falls in the interval of $[q_1+\cdots+q_{k-1}, q_1+\cdots+q_k)$, p_i is allocated to $task_k$. In other words, packets are allocated to $task_k$ with a probability of q_k, and for a flow of n packets, statistically $n_k = n \times q_k$ packets are allocated to $task_k$.

For scheduling multiple PINT tasks on a network flow, we seek to allocate an appropriate number of packets to each task to ensure that the tasks can be accomplished according to their utilities with high success probabilities. Formally, we present the objective of maximizing the weighted overall utility of the PINT tasks, which is defined as follows:

$$U_m = \sum_{k=1}^{K} S_m(n_k) \times u_k = \sum_{k=1}^{K} \left(1 - \frac{h_m}{n_k}\right) \times u_k \tag{3}$$

where $S_m(n_k)$ is the success probability of $task_k$ with n_k packets allocated, as formulated in (1), and u_k is $task_k$'s utility given by the network administrator.

For a network flow of n packets traversing an m-hop path, we formulate the PINT task scheduling problem as follows:

$$\begin{array}{ll} \max & U_m \\ s.t. & \sum_{k=1}^{K} n_k = n \\ & n_k \in \mathbb{N}^0 \end{array} \tag{4}$$

where the nonnegative integer n_k is the number of packets allocated to $task_k$.

C. Task Scheduling Algorithm

To solve the integer optimization problem in (4), we first relax the conditions by allowing the variables $\{n_k\}$ to be real numbers. Moreover, we use $\{q_k\}$ to replace $\{n_k\}$ in the problem, which is now formulated as follows:

$$\begin{array}{ll} \max & U_m = \sum_{k=1}^{K} \left(1 - \frac{h_m}{n \times q_k}\right) \times u_k \\ s.t. & \sum_{k=1}^{K} q_k = 1 \\ & q_k \geq 0 \end{array} \tag{5}$$

The Lagrangian function for the above problem is as follows:

$$\Lambda = \sum_{k=1}^{K} \left(1 - \frac{h_m}{n \times q_k}\right) \times u_k + \lambda \left(1 - \sum_{k=1}^{K} q_k\right) \tag{6}$$

Since the problem is convex [39], by applying the Karush–Kuhn–Tucker (KKT) conditions, we obtain the following:

$$\frac{\partial \Lambda}{\partial q_k} = 0, \frac{\partial \Lambda}{\partial \lambda} = 0 \tag{7}$$

By solving (7), we can obtain the global optimal solution as follows:

$$q_k = \frac{\sqrt{u_k}}{\sum_{i=1}^{K} \sqrt{u_i}} \tag{8}$$

for $\forall k \in \{1, \cdots, K\}$.

The solution in (8) does not require the knowledge of path length m and flow size n; it depends only on the tasks' utilities

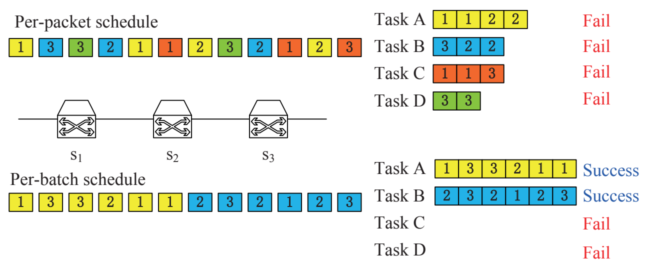


Fig. 2. A demonstrating example of task scheduling. Each packet is represented as a small rectangle, and its color represents which task the packet is allocated to and the number in the rectangle indicates from which switch the network state value is collected from.

$\{u_k\}_1^K$. Note that this is a desired property, as switches can allocate packets to PINT tasks using a fixed set of parameters, $\{q_k\}_1^K$, as long as the measurement tasks' utilities are the same on each switch.

In our previous discussion, we assume that each packet is allocated to a task individually; however, in practice, such per-packet scheduling has two drawbacks. First, per-packet scheduling cannot preserve the time proximity of the collected sample data, which is required when collecting some time-varying network states (e.g., interarrival interval, port utilization). For example, suppose that there are 10 tasks of the same utility. On average, a switch's network state can be collected only once in every 10 packets, and it takes as long as $10 \times h_m$ packet time to sample from all the switches that are enroute. The second drawback is that per-packet scheduling cannot ensure successful accomplishments of the PINT tasks when an INT flow has only a limited number of packets. Additionally, in the above example, if the flow has only 50 packets, equally allocating 5 packets to each task is meaningless, as all the 10 tasks will fail with so few packets allocated.

Motivated by the above observation, we propose grouping packets in *batches* and allocating an entire batch of packets to one single PINT task each time. Fig. 2 presents an example, where an INT flow containing 12 packets travels along a path composed of 3 switches. Suppose that there are 4 PINT tasks with the same utility, and under the per-packet scheduling, each packet randomly collects a network state from a random switch with an equal chance. As a result, each task has an average of 3 packets allocated, and each switch is selected by 4 packets on average, but overall, none of the tasks can collect state values from all the switches that are enroute, and thus, fail. However, if we group 6 consecutive packets into a batch, and allocate a batch of packets to one single task at a time, as shown in the figure, 2 of the 4 tasks can be successfully accomplished. In this example, we can see that with batch allocating, more packets can be concentrated on one single task to ensure its success and preserve the time proximity of the collected sample data.

To enable the per-batch task scheduling, when the first switch of a path receives a packet p_i that is not allocated to any task, it allocates this packet, as well as the $B-1$ subsequent packets, to a task according to $\mathbf{g}(p_i)$ and $\{q_k\}_1^K$. More specifically, the switch maintains a counter initialized as B and records the ID of the assigned task in the packet header. All the downstream switches along the path read the task ID from the packet and directly allocate the packet to the corresponding task. The first switch decrements the batch counter on receiving each packet, and when the counter becomes zero, it assigns a new task to the next batch of B packets, and resets the counter. Algorithm 1 presents the complete algorithm.

Algorithm 1: Task scheduling algorithm

```
Algorithm Algorithm runs on switch $s_j$
    Input : packet $p_i$
    Find $flow$ that $p_i$ belongs to;
    if $s_j$ is the flow's first switch then
        if $flow.batch\_counter == 0$ then
            Compute $\mathbf{g}(p_i)$;
            Find $task_k$ that $p_i$ is allocated to;
            Set $flow.batch\_counter = B$;
        Decrement $flow.batch\_counter$ by 1;
        Write task ID $k$ to $p_i$'s header;
    else
        Read task ID $k$ from $p_i$'s header;
    Compute $\mathbf{h}(p_i, j)$;
    if $\boldsymbol{h}(p_i, j) \leq \frac{1}{j}$ then
        Write $v_k(p_i, s_j)$ to $p_i$'s header;
```

We set the batch size B as follows:

$$B = \lceil c \times h_m \rceil \tag{9}$$

where h_m is the expected number of packets for accomplishing a task, as in (1), and we refer to c ($c > 0$) as the *batch size scaling factor* for controlling the batch size. For example, in Fig. 2, if $c = 1$, the batch size B should be $\lceil c \times h_3 \rceil = 6$ packets.

D. Pipeline Design and Realization

We design a packet processing pipeline to realize Algorithm 1 and implement the pipeline with the P4 language [18], which is currently the de facto standard for data plane programmability. Note that all the resources and operations required by Algorithm 1, such as the self-defined INT field, hash functions, and counters, are natively supported by P4. Fig. 3 illustrates the ingress pipeline of a flow's first switch, which is composed of five stages: Stage 1 checks the source and destination addresses of a received packet p_i and decides which INT flow the packet belongs to, as well as the position index of the switch on the flow's forwarding path. If the switch is the first hop, Stage 2 maintains a counter for the flow and decides whether the incoming packet belongs to the current batch or to a new batch. If the packet is the first packet of a new batch, Stage 3 computes $\mathbf{g}(p_i)$, decides which task the new batch of packets should be allocated to, and writes the task ID into the packet header. In Stage 4, the hash value $\mathbf{h}(p_i, j)$ is computed, and in Stage 5, $\mathbf{h}(p_i, j)$ is compared against $\frac{1}{j}$ to decide whether the switch's state value $v_k(p_i, s_j)$ should be written to p_i's INT field. Other switches on the path do not

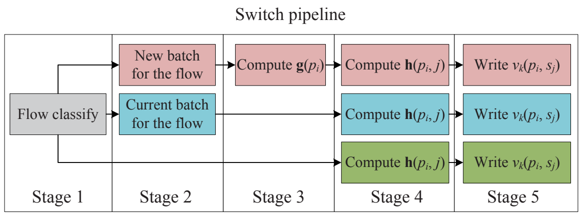


Fig. 3. Pipeline layout illustration: For the first-hop switch, the first packet of a new batch is processed by Stages 1-5, while the subsequent packets are processed by Stages 1-2 and 4-5. The non-first hop switches process packets with Stages 1 and 4-5.

maintain the batch counter, and their pipelines only contain Stage 1, Stage 4, and Stage 5, where Stage 1 confirms that the switch is not the first hop, and Stage 4 and Stage 5 execute the corresponding PINT task according to the task ID in the packet header embedded by the first-hop switch.

The pipeline is populated with rule entries from the network controller. Specifically, since the controller oversees the global topology, when planning a path from an origin to a destination, it knows each switch's position on the path. The controller uses this knowledge to compute all the parameters related to PINT tasks and task scheduling, constructs rule entries for differently positioned switches, and installs them in the switches' pipelines.

In Sec. III-A and III-B, we define the two hash functions of $\mathbf{h}(.)$ and $\mathbf{g}(.)$ in $[0, 1)$ and compare the hash values $\mathbf{h}(p_i, j)$ and $\mathbf{g}(p_i)$ against real number thresholds and ranges. However, in practice, P4-provided functions only return integers. To bridge this gap, in our implementation, we map real numbers in $[0, 1)$ to integers. Specifically, we use the P4-provided `crc32` hash function in $\{0 \cdots 65,535\}$ as $\mathbf{h}(.)$, and use the integer $round(65,535 \times \frac{1}{j})$ as the threshold. Similarly, we use the P4-provided `random` function in $\{0 \cdots 999\}$ as $\mathbf{g}(.)$, and compare its returned value against the integer range of $[round(999 \times \sum_{i=1}^{k-1} q_i), round(999 \times \sum_{i=1}^{k} q_i)]$ to decide whether the current batch executes $task_k$.

We implement Algorithm 1 on the `bmv2` software switch [19] and the Barefoot Tofino hardware switch [20], and share the P4 code with the community[2].

E. Discussion

The proposed PINT task scheduling algorithm achieves the objectives discussed in Sec. I: First, the algorithm is effective, as it basically follows the optimal solution in (8) to allocate packets to different PINT tasks. In Sec. IV, we further illustrate its effectiveness with a comprehensive evaluation.

Second, the algorithm is practical, as it does not require frequent rule insertions and updates on the data plane from the control plane. From Algorithms 1 and (8), we can see that the rules for task scheduling on a switch are updated only when the tasks' utilities, i.e., $\{u_k\}_1^K$, are changed. Since tasks generally do not change their requirements in collecting measurement data, the task scheduling rules are updated infrequently.

[2]https://github.com/DpintHPCC416/Batch-Pint

Third, as discussed in the above section, the algorithm can be realized with a packet processing pipeline, which is implemented on programmable switches with the P4 language.

Thus far, we consider the case in which a packet can collect and carry only one state value from a switch that is enroute. If each packet is capable of carrying multiple state values in its header, we can view each packet as multiple "virtual" packets and view an INT flow as multiple "virtual" INT flows on the same path. By applying Algorithm 1 to allocate packets on each "virtual" INT flow independently, the problem can be solved.

For the issue of assigning utilities to various tasks, as we have discussed in Sec. I, a task's utility is closely related to the nature of the network state that it aims to collect, and it also depends on how critical the corresponding malfunction is to the network health. For example, a network administrator may assign a task that collects packet counters with a high utility if the network has an extremely low tolerance to packet losses, or they may assign a task that monitors queue occupancy with a high utility if latency is one of the most important QoS metrics. Clearly, assigning utilities to parallel PINT tasks is empirical and case-by-case, and this topic is beyond the scope of this work.

IV. Evaluation

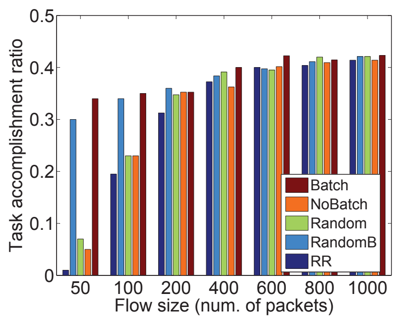
In this section, we evaluate our proposed PINT task scheduling algorithm with experiments. We first compare the algorithm with a number of alternative solutions in various conditions (Sec. IV-A). Then, we construct a data center network testbed with the FatTree topology [40] and apply our algorithm to run four real-world measurement tasks for troubleshooting four different network faults in parallel (Sec. IV-B). Finally, we implement the algorithm on a hardware switch and evaluate its impact on the switch's forwarding performance (Sec. IV-C). The evaluations show that our algorithm outperforms alternative solutions and can be practically applied in the real world.

A. Algorithm Evaluation

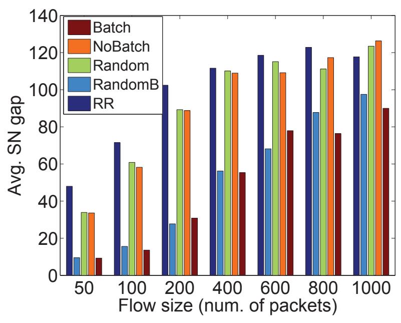
1) Experiment setup: We consider a simple scenario where an INT flow of n packets traverses an m-hop path and compare the following methods to schedule K parallel PINT tasks on the flow.

- **Batch**: This is exactly our proposed algorithm, as elaborated on in Sec. III and Algorithm 1.
- **NoBatch**: This method applies a per-packet allocation according to the parameters in (8).
- **Random**: In this method, packets are allocated to tasks randomly with a probability of $\frac{1}{K}$ for each task.
- **Random batch (RandomB)**: In this method, a batch of packets are randomly allocated to a task at a probability of $\frac{1}{K}$, and the batch size is set the same as in **Batch**.
- **Round robin (RR)**: In this method, packets are allocated to tasks in a round-robin way. Formally, the i^{th} packet of a flow is allocated to the k^{th} task with $k = i \mod K$.

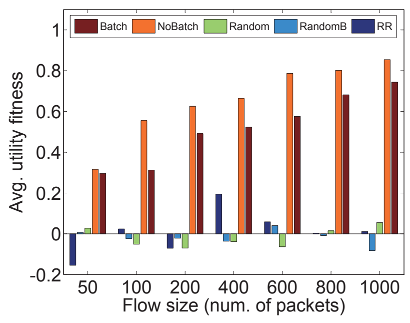
Similar to the **Batch** algorithm, we implement **NoBatch**, **Random**, **RandomB**, and **RR** with P4. Since we are the first



(a) Task accomplishment ratio



(b) SN gap



(c) Utility fitness

Fig. 4. Comparison of five different PINT task scheduling methods under various flow sizes (number of packets) regarding (a) task accomplishment ratio, (b) averaged SN gap, and (c) averaged utility fitness.

to address the problem of scheduling multiple PINT tasks in parallel, we develop the following metrics to assess the scheduling result.

- *Task accomplishment ratio*: This metric is defined as the ratio between the number of actually accomplished PINT tasks and the maximum number of tasks that can be accomplished in theory, which is $\frac{n}{m}$. For instance, considering a flow of 20 packets on a 5-hop path, the maximum number of tasks that can be accomplished in theory is $\frac{20}{5} = 4$, as each task requires at least 5 packets. If the flow actually accomplishes one task, then the task accomplishment ratio is 0.25. In other words, the task accomplishment ratio reflects how well a scheduling algorithm can make use of an INT flow's packets without wasting them, and we use it as the primary metric for evaluating a task scheduling scheme.
- *Sequence number gap (SN gap)*: Each time a task is accomplished, the SN gap is defined as the difference between the sequence numbers of the last and first packets that are used to accomplish this task. Clearly, the smaller the SN gap is, within a shorter interval the corresponding task is accomplished. In other words, this metric measures the time proximity of the collected sample data under a task scheduling scheme.
- *Utility fitness*: For a flow with K tasks scheduled, a task may be accomplished multiple times. We use a_k to denote the frequency of accomplishing $task_k$, and the utility fitness is defined as the Pearson's correlation between two normalized vectors ,$< a_k >$ and $< q_k >$, where q_k is the parameter for allocating packets to $task_k$ as in (8). This metric reflects how well tasks are accomplished according to their diverse requirements in collecting measurement data.

In the following, we use the three metrics to evaluate and compare the different PINT task scheduling methods and discuss the impact of various factors on their performance. Unless otherwise specified, we fix the flow size $n = 200$ packets, path length $m = 5$ hops, and tasks are assigned with linear utilities as $u_k = k$ $(k = 1, \cdots, K)$ for $K = 12$. We set the batch size scaling factor as $c = 1$, and according to (9), a batch contains $B = \lceil c \times h_5 \rceil = 12$ packets.

2) Impact of the flow size (number of packets): We first consider the impact of the flow size by varying n from 50 to $1,000$ packets. Fig. 4 presents the five PINT task scheduling methods' performances under various flow sizes. Each experiment was repeated 20 times. From Fig. 4(a) and (b), we can see that when the flow size is small, **RandomB** and **Batch**, which allocate packets in batches, have higher task accomplishment ratios and smaller SN gaps than the per-packet allocation schemes of **NoBatch**, **Random**, and **RR**. This observation conforms to our analysis that batch allocation can ensure task success and preserve the time proximity of the collected sample data.

As the flow size increases, all the solutions achieve high task accomplishment ratios, but the accomplished tasks have larger SN gaps. This is because when the flow has more packets, a task, which used to be unaccomplished with a smaller flow, can be accomplished now by waiting for more packets to be allocated to it.

Fig. 4(c) shows that **Batch** and **NoBatch** outperform **Random**, **RandomB**, and **RR** regarding utility fitness. The reason for this is that the methods of **Random**, **RandomB**, and **RR** are agnostic to tasks' utilities. Moreover, for **Batch** and **NoBatch**, their utility fitness increases with the flow size. This is because **Batch** and **NoBatch** allocate packets according to (8); thus, their scheduling results approach the optimal ones when there are more packets for allocation. Finally, **NoBatch** outperforms **Batch** as it allocates packets in a finer granularity.

3) Impact of the path length: To assess the impact of the path length, we vary m from 3 to 15 hops. It was found that when the path becomes longer, we have lower task accomplishment ratios and larger SN gaps for all five methods. This is easy to understand, as a longer path means that more packets are required for accomplishing a task. Given the fixed flow size, fewer tasks can be accomplished on longer paths.

We also find that the **Batch** and **RandomB** methods, which can concentrate a batch of packets on one single task, achieve higher task accomplishment ratios and smaller SN gaps than the per-packet allocating methods of **NoBatch**, **Random**, and **RR**. In additional, **Batch** and **NoBatch** outperform **Random**, **RandomB**, and **RR** regarding utility fitness. We omit the detailed experimental results because they are intuitive and easy to understand.

4) Impact of the number of tasks: In this section, we examine the impact of the task number K by assigning 8 to 20 parallel PINT tasks on an INT flow. Fig. 5 presents

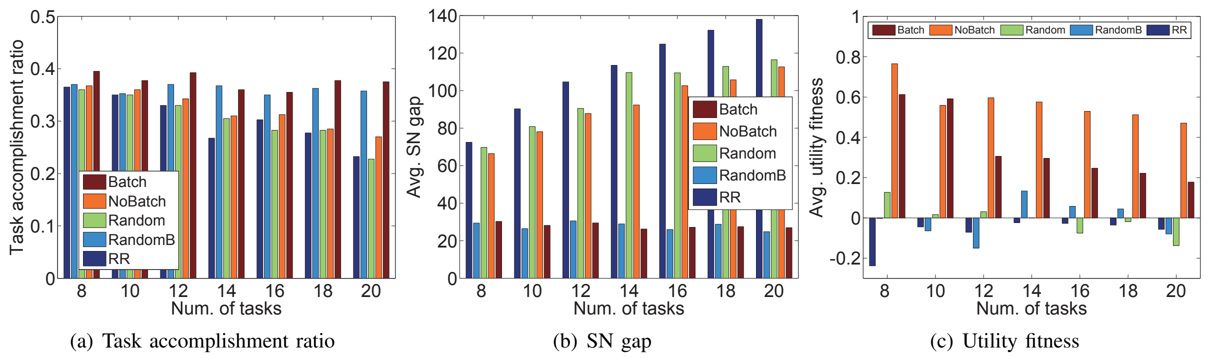


Fig. 5. Comparison of five different PINT task scheduling methods under various numbers of parallel tasks regarding the (a) task accomplishment ratio, (b) averaged SN gap, and (c) averaged utility fitness.

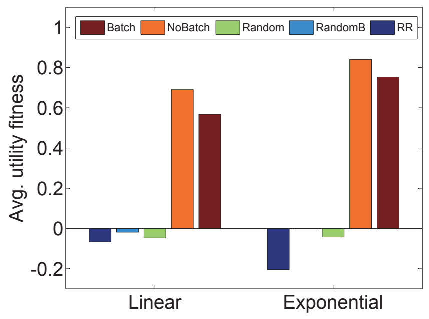


Fig. 6. Comparison of the utility fitness of five different PINT task scheduling methods under linear and exponential utility patterns.

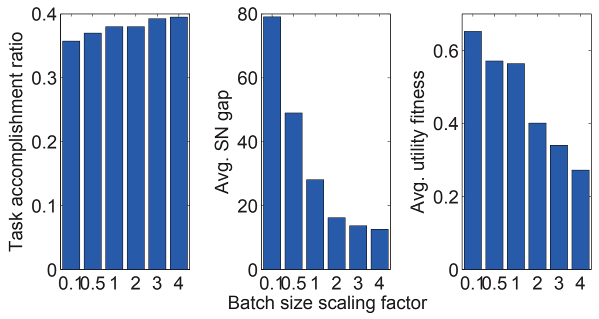


Fig. 7. Impact of the batch size on the performance of the **Batch** method

the experimental results. From Fig. 5(a) and (b), it can be seen that for the **NoBatch**, **Random**, and **RR** methods, their performances become worse when more tasks are scheduled. This is because the limited packets have to be distributed among more tasks, with each task less likely to be successfully accomplished.

However, the methods of **Batch** and **RandomB** are not impacted by the task number regarding the task accomplishment ratio and the SN gap. This is because the two methods allocate a batch of packets to one single task each time, and thus, can successfully accomplish a task with a high probability.

Fig. 5(c) shows that compared with **NoBatch**, the **Batch** method is more impacted by the task number regarding utility fitness because when there are more tasks and packets are allocated in batches, tasks with low utilities may have no packets allocated at all, resulting in poor utility fitness.

5) Impact of the utility pattern: Previous experiments assume that PINT tasks are assigned linear utilities, i.e., $u_k = k$; however, in practice, some PINT tasks require measurement data to be collected much more frequently than other tasks, and these tasks should have much higher utilities. In this section, we consider an exponential utility pattern, where $task_k$'s utility $u_k = \alpha^k$ with $\alpha = 1.5$. Note that when there are $K = 12$ tasks, the highest utility u_{12} is over 86 times larger than the lowest utility u_1.

Since the utility pattern does not impact the task accomplishment ratio and the SN gap, we only present the utility fitness of different task scheduling methods in Fig. 6. We can see that when tasks' utilities become more imbalanced under the exponential pattern, the methods of **Batch** and **NoBatch** preserve their good performances, as they apply the optimal solution in (8) to allocate packets to tasks, and thus, are adaptive to the highly imbalanced task utilities.

6) Impact of batch size: As in (9), our proposed PINT task scheduling method (i.e., **Batch**) has a scaling factor c for controlling the batch size. In this section, we examine how the batch size impacts the performance by varying c from 0.1 to 4, which indicates that the number of packets in a batch varies from 2 to 46.

In Fig. 7, we present the three metrics achieved by the **Batch** method under various batch size scaling factors. We can see that a tradeoff exists; when the batch size increases, a higher task accomplishment ratio and a smaller SN gap can be achieved. This is easy to understand, as a larger batch size means that more packets are allocated to the current task without being wasted on other tasks. Note that when the batch size is large enough, a task may be accomplished multiple times within a batch, which explains the small SN gap in Fig. 7.

However, the metric of the utility fitness decreases with the batch size scaling factor c. This is because as packets are allocated in a coarser granularity with a larger batch size, the actual packet allocation result deviates more from the optimal solution in (8), leading to a lower utility fitness.

Summary: From the experiments, we conclude that compared with the alternative approaches, our proposed PINT task scheduling algorithm, i.e., **Batch**, accomplishes more PINT tasks, better preserves the time proximity of the collected sample data, and better fits the tasks' diverse requirements in collecting the measurement data. Moreover, these merits persist under varied combinations of flow size, path length, utility pattern, and task number. The algorithm is also flexible by allowing a tradeoff between the quality and the quantity of the tasks being accomplished.

B. Network Evaluation

1) Experimental setup: In this section, we emulate a data center network with a FatTree topology [40] using Mininet [41], and schedule four parallel PINT tasks to detect four real world faults in the network. As shown in Fig. 8(a), the testbed is composed of 16 hosts and 20 P4 software switches. In the experiment, each host sends a flow at a constant rate of $1,000$ pps to another host in the network, and we program the switches with rules to forward packets of the flows according to the routing policy in [40]. More specifically, for a pair of hosts, the FatTree topology contains multiple equal-cost paths, and the algorithm in [40] selects a path based on the parity of the source address. As a result, 192 paths containing 5 hops, 32 paths containing 3 hops, and 16 paths containing 1 hop that interconnect all the hosts on the FatTree network are planned, and the paths are evenly distributed among the switches.

We introduce four different faults in the FatTree network.

- **Inflated path**: We introduce an inflated path, which is a case of path deviation fault as described in Sec. II-A, by modifying the forwarding rules at s_4 and s_7. More specifically, we instruct s_4 to forward packets destined for odd-indexed hosts to s_7, which forwards these packets to s_5, and s_5 handles them normally as in [40]. As a result, the impacted flows have paths longer than 5.
- **Forwarding loop**: We introduce a loop by modifying the forwarding rules at s_{13} and s_{15}. When receiving a packet from s_{12} that is destined for h_1 or h_7, s_{15} forwards the packets to s_{13}, which forwards them back to s_{15}, and s_{15} handles them normally as in [40].
- **Blackhole**: We introduce two blackholes at s_{11} and s_{14}. s_{11} is configured to randomly drop packets on its port connecting to s_{18}, and s_{14} randomly drops packets on its port connecting to s_{16}. Both switches drop packets at a probability of 8%. Note that although the blackhole definition applies to the switch rather than to the switch port, here we configure the switch ports to randomly drop packets for partially imitating a faulty switch's behaviors in the experiment.
- **Congestion**: We introduce three congested links $s_1 - s_3$, $s_7 - s_{19}$, and $s_{10} - s_9$ by imposing a random latency before sending packets on s_1, s_7, and s_{10}. The latency is chosen randomly between $20\,\text{ms}$ and $40\,\text{ms}$.

Each network fault can be detected with a different PINT task, more specifically.

- To detect inflated paths, a PINT task is scheduled to collect switch IDs of all the switches traversed by a flow. If the number of unique switches on a path exceeds 5, an inflated path is detected.
- To detect forwarding loops, a switch maintains a per-flow packet counter at each of its egress ports, and a PINT task is scheduled to collect the counters from all the switches traversed by a flow. For a pair of two consecutive switches (s_i, s_{i+1}), if $\frac{counter_{i+1}}{counter_i}$ exceeds a threshold of 1.8, we declare a forwarding loop starting at s_{i+1}.
- To detect blackhole switches, a switch maintains a per-flow packet counter at each of its ingress ports, and a PINT task is scheduled to collect the counters from all the switches traversed by a flow. For a switch s_i, if all its downstream switches have counters no more than $0.93 \times counter_i$, we declare that s_i is a blackhole.
- To detect congestion, a switch records the interarrival time between the two consecutive packets it recently received, and a PINT task is scheduled to collect the interarrival times on all the switches traversed by a flow. With the collected samples, the analyzing server derives an interarrival time distribution on each hop and computes a Chi-square distance between the distributions of any two consecutive switches. The Chi-square distance is defined as follows:
$$\sum_{i=1}^{n} \frac{(x_i - y_i)^2}{(x_i + y_i)} \tag{10}$$
where n is the number of bins, and x_i and y_i are the number of samples in the i^{th} bin for the two distributions respectively [42]. In our experiment, we put the collected samples into $n = 4$ bins with each bin of $15\,\text{ms}$, and declare that a link (s_i, s_{i+1}) is congested when the distance between the distributions of the two switches s_i and s_{i+1} exceeds a threshold of 0.3.

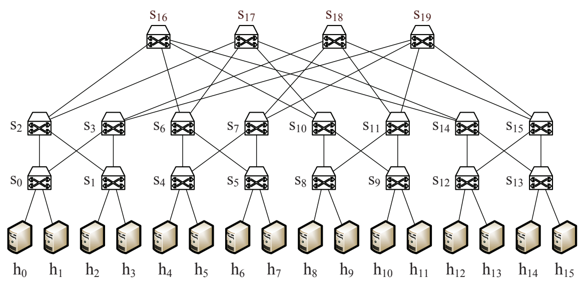
Before presenting the experimental results, we first discuss how the utilities of the four parallel PINT tasks should be assigned in principle. First, for detecting inflated paths, since a switch ID needs to be collected only once, the corresponding task should have a low utility.

The utility of the task for detecting forwarding loops should also be low, as a switch in a loop has a counter that is twice as large as the one on an off-loop switch, and such a difference is easy to detect.

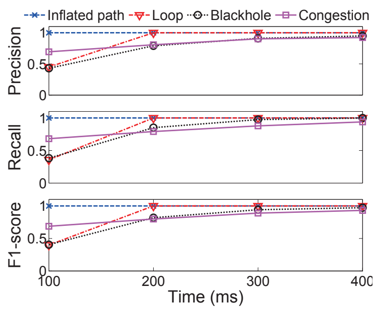
However, the utility of the task for detecting blackholes should be high; as a blackhole switch drops packets at a low probability, its counter value is only slightly lower than that of a normal switch. Since a packet counter increases over time and counters on different switches are collected asynchronously, the PINT task should collect a sufficient number of samples on each hop to offset the impact that counters on different switches are collected at different moments.

Finally, for detecting congestion, the utility of the task should be high, as we need to accumulate a sufficient number of samples to derive an interarrival time distribution on each hop and compute the chi-square distance with the distributions.

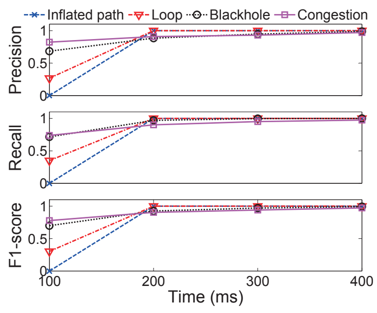
2) Result: We evaluate two utility settings in our experiment. We first ignore the differences in the PINT tasks by assigning each task an equal utility of 0.25. After every $100\,\text{ms}$, we detect the network faults with the measurement data collected thus far. For each fault, we present the detection results in terms of the precision, recall, and F1-score in Fig. 8(b). In the figure, we can see that the inflated path is accurately detected at $100\,\text{ms}$, and the forwarding loop is detected at $200\,\text{ms}$. However, for the two blackhole switches and the



(a)



(b)



(c)

Fig. 8. (a) FatTree topology; (b) precision, recall, and F1-score for detecting four network faults under task utilities of $(0.25, 0.25, 0.25, 0.25)$; (c) precision, recall, and F1-score for detecting four network faults under task utilities of $(0.1, 0.1, 0.4, 0.4)$.

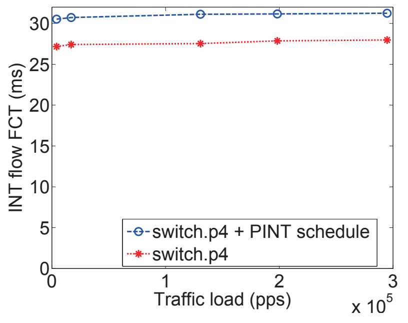


Fig. 9. Comparison of forwarding performances with and without the PINT task scheduling and execution pipeline on Barefoot Tofino switch.

three congested links, we do not have accurate detection results with F1-scores above 0.9 until $400\,\mathrm{ms}$.

We then take the different requirements of the PINT tasks in collecting measurement data into consideration, and assign the utilities of the tasks for detecting inflated paths, forwarding loops, blackholes, and congestion as 0.1, 0.1, 0.4, and 0.4 respectively. We believe that this setting is more reasonable by assigning higher utility values to the tasks for detecting blackholes and congestion. Fig. 8(c) presents the results, from where it can be seen that we have successfully detected all the network faults with F1-scores above 0.9 at the $200\,\mathrm{ms}$, which is much earlier than under the equal utility setting.

The experimental results in this section suggest that first, our proposed algorithm can be practically applied to schedule parallel PINT tasks for detecting real world network faults; second, when multiple PINT tasks are scheduled in parallel, their utility values should be carefully selected to improve the overall network diagnosis efficiency.

C. Hardware Evaluation

Our proposed pipeline demonstrated in Fig. 3 computes two hash values and accesses memory at most once for each INT flow packet. In this section, we realize the pipeline on the commodity Edgecore Wedge 100BF Tofino-based programmable switch and evaluate the impact of the overhead on the switch's forwarding performance.

In particular, we append our five-stage pipeline to the ingress pipeline of `switch.p4`, which is a baseline P4 implementation for the L2/L3 switch [43]. We run the program on the Tofino hardware switch and compare the switch's forwarding performance with the case when the switch runs the baseline `switch.p4`. To assess the forwarding performance, we send a traffic load varying from $4,000$ pps to $30,000$ pps and send an INT flow containing $127,834$ packets back-to-back to the switch. In our five-stage pipeline, INT flow packets are allocated to $K = 8$ PINT tasks for carrying different state values, such as the switch ID, port number, packet counter, byte counter, timestamp, etc., while with the original `switch.p4`, the INT flow is treated as an ordinary flow.

We record the INT flow's flow completion times (FCTs) under various background traffic loads, with and without the PINT task scheduling pipeline, and compare the results in Fig. 9. In the figure, we can see that after appending our proposed pipeline, the FCTs increase no more than 10%. The result suggests that the overhead introduced by our proposed pipeline for PINT task scheduling and execution is insignificant, and our proposed algorithm can be practically applied in production networks.

V. Conclusion

In this paper, we assert that for the first time, with Probabilistic In-Network Telemetry (PINT), packets in a flow will be a scarce resource and require smart allocation among parallel PINT tasks. Based on the analysis of the relation between the number of packets allocated to a PINT task and the task's success probability, we formulate the PINT task scheduling problem and solve it by allocating the optimal numbers of packets to different measurement tasks. We propose a PINT task scheduling algorithm that requires few interventions from the control plane, realize it with a five-stage pipeline, and implement it with P4. Comprehensive performance evaluation shows that at a low scheduling overhead, our proposed algorithm can accomplish more PINT tasks with higher quality than other alternative solutions, and it can be applied to schedule parallel PINT tasks for efficiently detecting real world network faults.

References

[1] B. Claise, "Cisco systems netflow services export version 9," RFC 3954, Oct. 2004. [Online]. Available: https://www.ietf.org/rfc/rfc3954.txt

[2] P. Phaal and M. Lavine, “sflow version 5,” Jul. 2004. [Online]. Available: https://sflow.org/sflow_version_5.txt
[3] M. Yu, L. Jose, and R. Miao, “Software defined traffic measurement with OpenSketch,” in *Proc. NSDI’13*, Lombard, IL, USA, Apr. 2013.
[4] 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. SIGCOMM’14*, Chicago, IL, USA, Aug. 2014.
[5] 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. NSDI’14*, Seattle, WA, USA, Apr. 2014.
[6] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, “Packet-level telemetry in large datacenter networks,” in *Proc. SIGCOMM’15*, London, UK, Aug. 2015.
[7] Y. Li, R. Miao, C. Kim, and M. Yu, “FlowRadar: A better NetFlow for data centers,” in *Proc. NSDI’16*, Santa Clara, CA, USA, Mar. 2016.
[8] P. Tammana, R. Agarwal, and M. Lee, “Simplifying datacenter network debugging with pathdump,” in *Proc. OSDI’16*, Savannah, GA, USA, Nov. 2016.
[9] ——, “Distributed network monitoring and debugging with SwitchPointer,” in *Proc. NSDI’18*, Reton, WA, USA, Apr. 2018.
[10] J. Kučera, R. B. Basat, M. Kuka, G. Antichi, M. Yu, and M. Mitzenmacher, “Detecting routing loops in the data plane,” in *Proc. CoNEXT’20*, Barcelona, Spain, Dec. 2020.
[11] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, and N. Zhang, “LightGuardian: A full-visibility, lightweight, in-band telemetry system using sketchlets,” in *Proc. NSDI’21*, Apr. 2021.
[12] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, “PINT: Probabilistic in-band network telemetry,” in *Proc. SIGCOMM’20*, Virtual Event, NY, USA, Aug. 2020.
[13] S. Narayana, M. T. Arashloo, J. Rexford, and D. Walker, “Compiling path queries,” in *Proc. NSDI’16*, Santa Clara, CA, USA, Mar. 2016.
[14] 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. SIGCOMM’19*, Beijing, China, Aug. 2019.
[15] A. Karaagac, E. D. Poorter, and J. Hoebeke, “In-band network telemetry in industrial wireless sensor networks,” *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 1, 2019.
[16] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, “Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN,” in *Proc. SIGCOMM’13*, Hong Kong, China, Aug. 2013.
[17] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall1, “dRMT: Disaggregated programmable switching,” in *Proc. SIGCOMM’17*, Los Angeles, CA, USA, Aug. 2017.
[18] “P4 open source programming language,” accessed on May 20, 2021. [Online]. Available: https://p4.org/
[19] “bmv2, the behavioral model for P4,” accessed on May 20, 2021. [Online]. Available: https://github.com/p4lang/behavioral-model
[20] “Intel Tofino Series,” accessed on Apr. 20, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html
[21] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, “DREAM: Dynamic resource allocation for software-defined measurement,” in *Proc. SIGCOMM’14*, Chicago, Illinois, USA, Aug. 2014.
[22] ——, “SCREAM: Sketch resource allocation for software-defined measurement,” in *Proc. CoNEXT’15*, Heidelberg, Germany, Dec. 2015.
[23] S. Li, L. Luo, D. Guo, Q. Zhang, and P. Fu, “Sketch for traffic measurement: design, optimization, application and implementation,” *CoRR*, vol. abs/2012.07214, 2020. [Online]. Available: https://arxiv.org/abs/2012.07214
[24] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, “Elastic Sketch: Adaptive and fast network-wide measurements,” in *Proc. SIGCOMM’18*, Budapest, Hungary, Aug. 2018.
[25] Y. Zhou, Y. Zhang, C. Ma, S. Chen, and O. O. Odegbile, “Generalized sketch families for network traffic measurement,” *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 3, 2019.
[26] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, “CocoSketch: High-performance sketch-based measurement over arbitrary partial key query,” in *Proc. SIGCOMM’21*, Virtual Event, USA, Aug. 2021.
[27] 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 *Proc. SIGCOMM’20*, Virtual Event, NY, USA, Aug. 2020.
[28] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, “Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow,” in *Proc. USENIX ATC’18*, Boston, MA, USA, Jul. 2018.
[29] B. Niu, J. Kong, S. Tang, Y. Li, and Z. Zhu, “Visualize your IP-over-optical network in realtime: A p4-based flexible multilayer in-band network telemetry (ML-INT) system,” *IEEE Access*, vol. 7, 2019.
[30] S. Sheng, Q. Huang, and P. P. C. Lee, “DeltaINT: Toward general in-band network telemetry with extremely low bandwidth overhead,” in *Proc. ICNP’21*, Virtual Event, Nov. 2021.
[31] K. Yang, Y. Li, Z. Liu, T. Yang, Y. Zhou, J. He, J. Xue, T. Zhao, Z. Jia, and Y. Yang, “SketchINT: Empowering INT with TowerSketch for per-flow per-switch measurement,” in *Proc. ICNP’21*, Virtual Event, Nov. 2021.
[32] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, “In-band network telemetry: A survey,” *Comput. Netw.*, vol. 186, no. 107763, 2021.
[33] “In-band network telemetry (int) dataplane specification,” The P4.org Applications Working Group, Tech. Rep., Nov. 2020. [Online]. Available: https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf
[34] “In-band network telemetry detects network performance issues,” Intel, White Paper, Dec. 2020. [Online]. Available: https://networkbuilders.intel.com/solutionslibrary/in-band-network-telemetry-detects-network-performance-issues
[35] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge, UK: Cambridge University Press, 1995.
[36] E. M. Stein and R. Shakarchi, *Real Analysis: Measure Theory, Integration, and Hilbert Spaces*. Princeton, NJ, USA: Princeton University Press, 2005.
[37] Y. Kim, D. Suh, and S. Pack, “Selective in-band network telemetry for overhead reduction,” in *Proc. IEEE International Conference on Cloud Networking (CloudNet)*, Tokyo, Japan, Oct. 2018.
[38] E. Song, T. Pan, C. Jia, W. Cao, J. Zhang, T. Huang, and Y. Liu, “INT-label: Lightweight in-band network-wide telemetry via interval-based distributed labelling,” in *Proc. INFOCOM’21*, Vancouver, BC, Canada, May 2021.
[39] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, UK: Cambridge University Press, 2004.
[40] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *Proc. SIGCOMM’08*, Seattle, WA, USA, Aug. 2008.
[41] “Mininet,” accessed on May 20, 2021. [Online]. Available: http://mininet.org/
[42] Y. Dodge, *The Concise Encyclopedia of Statistics*. New York, NY, USA: Springer, 2008.
[43] “Consolidated switch repo,” accessed on May 1, 2022. [Online]. Available: https://github.com/p4lang/switch

Wei Chen received the bachelor’s degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2020. He is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, USTC. His research interest is focused on programmable networks.

Ye Tian received the bachelor's degree in electronic engineering and the master's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2001 and 2004, respectively, and the Ph.D. degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, in 2007. He joined USTC in 2008 and is currently an Associate Professor with the School of Computer Science and Technology, USTC. His research interests include programmable networks, network and Internet measurements, and network softwarization. He has published over 70 papers and co-authored a research monograph published by Springer. He is the winner of the Wilkes Best Paper Award of Oxford The Computer Journal in 2016. He is currently serving as a Young Associate Editor of Springer Frontiers of Computer Science Journal. He is a member of the IEEE.

Zhongxiang Wei received the bachelor's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2020. He is currently working toward the master's degree with the School of Computer Science and Technology, USTC. His research interests include programmable networks and network measurement.

Jiangyu Pan received the bachelor's and master's degrees in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2019 and 2022 respectively. His research interests include wireless networks and programmable networks. He will join TP-Link Technologies in July 2022.

Xinming Zhang received the BE and ME degrees in electrical engineering from China University of Mining and Technology, Xuzhou, China, in 1985 and 1988, respectively, and the PhD degree in computer science and technology from the University of Science and Technology of China (USTC), Hefei, China, in 2001. Since 2002, he has been with the faculty of USTC, where he is currently a Professor with the School of Computer Science and Technology. From September 2005 to August 2006, he was a visiting Professor with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea. His research interest includes wireless networks, deep learning, and intelligent transportation. He has published more than 100 papers. He won the second prize of Science and Technology Award of Anhui Province of China in Natural Sciences in 2017. He won the awards of Top reviewers (1%) in Computer Science & Cross Field by Publons in 2019. He is a senior member of the IEEE.