OmniPath Ping: Active Network Measurement in the Era of Packet Spraying

Kaicheng Yang[†§], Zongwei Lv[†], Peijun Huang[†], Kaitai Zhang[†], Qiuheng Yin[†], Yaoming Li[†]
Feiyu Wang[†], Zhuochen Fan[†], Yikai Zhao[§], Chen Sun[§], Xia Zhu[§], Tong Yang[†]
[†]State Key Laboratory of Multimedia Information Processing, School of Computer Science, Peking University
[§]Huawei Technologies Co., Ltd.

ABSTRACT

Ping and Traceroute constitute the fundamental toolkit for network monitoring. Built upon these primitives, initial active probing systems focus on achieving sufficient coverage of switches and links. To further diagnose subtle failures (*e.g.*, misconfigurations) affecting specific service flows, recent systems have proposed service tracing to measure their connectivity and performance. These systems operate effectively under the widely adopted flow-level LB scheme (*i.e.*, ECMP), which binds each flow to a deterministic path. However, the paradigm shift towards packet-level LB (*e.g.*, packet spraying) in next-generation networks invalidates this foundation. By dispersing packets of a single flow across multiple paths, packet spraying causes traditional primitives to suffer from path ambiguity in failure localization and incurs prohibitive overhead for full path coverage.

In this paper, we propose OmniPath Ping (OPP), the first diagnostic primitive designed for service tracing under packet spraying. OPP overcomes these challenges through a switch-host co-design featuring two key techniques: (1) an in-network OmniPath cache that captures per-hop telemetry to resolve path ambiguity; and (2) in-network duplication and deduplication mechanisms that enable full path coverage with zero redundancy. To further ensure scalability for network-wide service tracing, OPP incorporates (3) a topology-aware concurrency control mechanism to balance OmniPath cache SRAM usage and probe timeliness. Extensive results demonstrate that OPP achieves 10 ms-level service tracing with full path coverage at minimal host and network overhead, requiring only 181 KB of switch SRAM.

CCS CONCEPTS

• **Networks** → **Network measurement**; **Network monitoring**; **Network reliability**.

KEYWORDS

Active Probing; Service Tracing; Fault Localization; Packet Spraying

*This work was done while Kaicheng Yang was an intern at Huawei. Chen Sun (sunchen48@huawei.com) and Tong Yang (yangtong@pku.edu.cn) are the corresponding authors.

This work is licensed under a Creative Commons Attribution 4.0 International License.
ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA, 2026.
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829105

ACM Reference Format:
Kaicheng Yang, Zongwei Lv, Peijun Huang, Kaitai Zhang, Qiuheng Yin, Yaoming Li, Feiyu Wang, Zhuochen Fan, Yikai Zhao, Chen Sun, Xia Zhu, and Tong Yang. 2026. **OmniPath Ping: Active Network Measurement in the Era of Packet Spraying** . In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 26 pages. https://doi.org/10.1145/3789240.3829105

1 INTRODUCTION

Ping [1] and its companion Traceroute constitute the fundamental toolkit for network monitoring. While Ping is used to monitor end-to-end performance, Traceroute complements it by providing hop-by-hop path visibility for fault localization. Built upon these primitives, active probing systems like Pingmesh [2] have been widely deployed in production data center networks (DCNs), establishing themselves as the de facto standard for diagnosing network failures.

Initial probing systems are designed primarily for link and switch coverage [3–5] to diagnose physical component failures. To further diagnose subtle failures caused by misconfigured or erroneous table entries (§ 2.1), which typically impact only a small subset of flows, recent systems [6, 7] propose and adopt *service tracing* to measure connectivity and performance of service flows.

The effectiveness of Ping and Traceroute for service tracing depends critically on the flow-level load balancing (LB) feature of contemporary DCNs. The most widely adopted LB scheme, *i.e.*, ECMP [8], routes packets based on flow IDs (*e.g.*, 5-tuples), ensuring that all packets of the same flow follow a single forwarding path. This flow-to-path binding establishes a deterministic mapping where each Time-To-Live (TTL) value corresponds to a unique switch. Consequently, once Ping detects packet loss, Traceroute leverages this determinism for precise failure localization: by iteratively varying the TTL, a probe failure at a specific TTL definitively pinpoints the corresponding faulty switch.

However, the LB scheme in DCNs is fundamentally shifting from flow-level towards packet-level (§ 2.2). As a typical representative, packet spraying [9] leverages switches to randomly disperse packets of a flow across all its equal-cost paths. The primary driver for this shift is the potential to significantly enhance network performance, evidenced by a 2.6× improvement under All-to-All workloads [10]. With major players like NVIDIA, Google, and UEC actively deploying corresponding solutions [11–13], packet-level LB has become the norm for next-generation DCNs.

Unfortunately, Ping and Traceroute fall short under packet spraying, encountering two key challenges:

Challenge 1: path ambiguity. Packet spraying breaks the 1-to-1 mapping between TTL value and switch. Instead, a single TTL value maps to a set of candidate switches, and a Traceroute probe

with a fixed TTL may randomly traverse any one of these candidates. Therefore, when packet loss occurs, Traceroute can no longer identify which specific switch among the candidates causes the failure.

Challenge 2: costly path coverage. To monitor network failures and service flow performance under packet spraying, Pingmesh-like systems must achieve full path coverage by traversing every equal-cost path with Ping probes. This corresponds to a classic coupon collector's problem [14], where the required number of probes scales super-linearly ($O(n \ln n)$) with the number of equal-cost paths n. Given that n can readily exceed 1000 in modern DCNs (*e.g.*, 1024 paths in 64-ary Fat-tree [15]), building service tracing systems upon Ping incurs prohibitive probe overhead, since probe agents can be overloaded with just 4 tasks per server pair [16].

We observe that the root cause of these problems is the *host's inherent lack of visibility into and control over in-network behavior under packet spraying*, whereas switches possess the ground truth of forwarding decisions. To tackle this problem, our key insight is that instead of relying solely on host-side mechanisms as in traditional Ping, switches must participate by providing path *visibility* of Ping probes for explicit failure localization, and *controlling* the paths of Ping probes for low-cost path coverage.

Driven by this insight, we propose *OmniPath Ping* (OPP), the first diagnostic primitive designed for service tracing under packet spraying. OPP overcomes two key challenges via a switch-host co-design, featuring two key techniques:

Technique 1: in-network OmniPath cache (OP cache). To resolve path ambiguity, we deploy OP cache on the switch data plane to capture per-hop telemetry in situ. As a probe traverses the network, it visits the OP cache on each switch to record per-hop measurement results. These results are then aggregated and carried back by the corresponding ACK. In this way, OPP provides enriched measurement results with precise path information for fault localization.

Technique 2: in-network duplication and deduplication (Figure 1). To reduce path coverage cost, OPP shifts the responsibility of probe generation and path exploration from the host to the switch by explicitly regulating forwarding behaviors of Ping probes. Specifically, OPP performs in-network duplication and deduplication: a single probe is duplicated to cover all equal-cost paths at upstream switches, while redundant instances of the same probe are deduplicated at downstream switches. In this way, OPP ensures that one distinct probe traverses every link along the equal-cost paths exactly once, thereby achieving full path coverage with zero redundancy.

While the above design enables efficient individual probes, scaling OPP to a network-wide service tracing system (akin to Pingmesh) reveals a new challenge:

Challenge 3: OP cache explosion. Given the severe scarcity of switch SRAM, the capacity of OP cache must be strictly constrained. However, since every in-flight probe consumes dedicated OP cache resources along its paths, this creates a fundamental trade-off between maximizing probe timeliness to capture transient failures and limiting OP cache resource overhead.

In response, we propose the following key technique.

Technique 3: topology-aware concurrency control. We propose a topology-aware concurrency control mechanism to regulate the number of concurrent in-flight probes. We customize this mechanism for Clos and switchless (*e.g.*, torus [17]) topologies, leveraging their unique structural characteristics to improve concurrency under limited SRAM resources, and further generalize the mechanism to arbitrary topologies. In this way, OPP balances resource overhead and probe timeliness, ensuring scalability in large-scale networks.

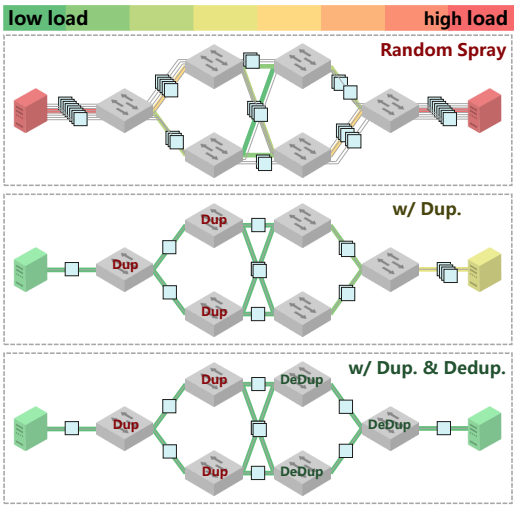


Figure 1: Under packet spraying, covering all paths with host-generated probes significantly burdens links and hosts. OPP minimizes probe overhead with in-network duplication and deduplication. Link and host colors reflect load levels.

We evaluate OPP through comprehensive experiments, comprising large-scale simulations in NS-3 [18] and a testbed prototype. Results show that: 1) OPP achieves full coverage with minimal host and network overhead, outperforming the baseline by reducing host overhead by 99.90% and network overhead by 93.33%. 2) OPP attains almost full accuracy in failure detection and localization, outperforming the baseline by reducing host overhead by 99.99%. 3) OPP achieves real-time service tracing (every 10 ms) using negligible switch SRAM (181 KB) for a 64-ary Fat-tree simulating a distributed training cluster with 65536 GPUs, proving its scalability. We have released all related source code on GitHub [19].

OPP is the first step towards conquering the measurement challenges brought by packet spraying from the perspective of active probing. As packet spraying disperses each flow to thousands of paths, each switch now carries three orders of magnitude more flows. As a result, Flow Table [20] used to track flows will be easily overloaded. Moreover, packet spraying results in non-trivial out-of-order issues [21] and creates new measurement requirements. We hope OPP can attract more research efforts to jointly solve these challenges.

Ethics: This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

2.1 Network Failures and Service Tracing

Network failures in DCNs are ubiquitous and diverse. We classify network failures into two categories according to blast radius: *device-level failures* and *flow-level failures*.

Device-level failures arise from the malfunctions (*e.g.*, link flapping) or capacity constraints (*e.g.*, congestion) of physical components. A salient feature of these failures is that they tend to impact

No.	Category	Root cause	Consequence
1	Device-level	Link flapping	Packet loss
2		Port down	Packet loss
3		Transceiver malfunctioning [22]	Packet loss
4		Congestion	Abnormal latency / Packet loss
5		Switch malfunctioning	Packet loss
6	Flow-level	Policer misconfiguration [23, 24]	Packet loss
7		ACL DSCP misconfiguration	Abnormal latency
8		ACL drop rule misconfiguration [25]	Packet loss
9		Routing misconfiguration	Packet loss (forwarding loop)
10		PFC unconfigured/misconfigured [6]	Packet loss
11		Table lookup misses caused by bit flip [26]	Packet loss

Table 1: Representative network failures.

any traffic that traverses the corresponding component. Leveraging this feature, most active probing systems [3, 4, 16] prioritize achieving sufficient coverage of all physical components (*i.e.*, links and switches), thereby ensuring failure visibility while minimizing probe overhead.

In contrast, flow-level failures arise from erroneous switch table entries [27, 28], including:

ACL misconfiguration [25]. A network operator wrongly configures an ACL entry. If the action associated with the incorrect entry is to drop, flows associated with this ACL entry experience packet loss. Alternatively, if the entry incorrectly downgrades the DSCP of a VIP customer's service from a high-priority class to a low-priority one, flows associated with this customer (matched by IP) and service (matched by port) experience abnormal latency.

Policer misconfiguration [23, 24]. A network operator misconfigures a dedicated policer entry for a specific service. For instance, if a rate limit of 10 Gbps is mistakenly set to 1 Gbps, flows associated with this customer (matched by IP) and service (matched by port) encounter severe packet loss.

Bit flip [26]. A table entry is corrupted due to switch memory bit flip. The entry happens to reside outside the switch detection zone and therefore is not reported to the switch control plane. As a result, specific flows with IDs matching the corrupted entry suffer from silent packet loss, driven by table lookup misses caused by parity errors.

Unlike device-level failures, flow-level failures affect only specific subsets of traffic. Consequently, Pingmesh-like systems designed solely for link/switch coverage are inherently inadequate for detecting such failures, as their probes are unlikely to match the specific erroneous entries.

To address this problem, R-Pingmesh [6] proposes *service tracing*, which traces the performance of each service flow by injecting tailored probes that mimic the target flow's ID. Probe results can be further leveraged to (1) monitor service SLA; (2) assess the impact of failures on services and determine whether immediate intervention is required.

2.2 Evolution towards Packet-Level LB

Traditional flow-level LB (*i.e.*, ECMP) maps flows to static forwarding paths, thus suffering from hash collisions that cause congestion [29, 30]. In modern AI clusters, such network-induced stalls are economically prohibitive given the high cost of accelerators. Consequently, both industry and academia have pivoted to packet-level LB [10, 31], which mitigates flow hash collisions by distributing traffic at the packet granularity.

At a high level, packet-level LB falls into two categories based on where the packet forwarding path is determined.

Host-based LB. Typical solutions, such as MP-RDMA [32], REPS [12], and Falcon [13], determine forwarding paths at end-hosts. They inject varying entropy values into packet headers (*e.g.*, the IPv6 flow label in Falcon), which interact with standard ECMP hashing on switches to disperse packets across equal-cost paths. In host-based LB networks, hosts have control over the paths of packets, which to some extent resembles flow-level LB networks.

Switch-based LB. These techniques (*e.g.* packet spraying [9] and Adaptive Routing (AR) [11]) determine forwarding paths of each packet directly inside the switch by round-robin scheduling across equal-cost ports, random selection, or congestion-aware port selection. Switch-based LB introduces unique difficulty for service tracing compared to host-based LB, as it precludes hosts from packet path selection (§ 1).

Therefore, this paper focuses on enhancing Ping and service tracing capabilities in switch-based LB networks. Note that OPP remains effective under host-based LB networks. We defer this discussion to Appendix G.

2.3 Service Tracing Under Packet Spraying

Packet spraying undermines the effectiveness of Ping and Traceroute for service tracing in terms of path visibility and control (§ 1). Next, we discuss two possible enhancements to Ping and Traceroute using commodity measurement techniques and analyze their effectiveness under packet spraying.

Ping + Mirror/INT for enhanced path visibility. We observe that two telemetry features, which are already supported by commodity switches, can potentially enhance service tracing: packet mirroring (*e.g.*, ERSPAN [33]) and In-band Network Telemetry (INT) [34]. Packet mirroring duplicates the probe at each traversed switch, encapsulates it with the switch ID and timestamp, and forwards the copy to an analyzer. This mechanism reveals the forwarding path and offers insights into link congestion and packet loss. INT in postcard mode [35] provides similar visibility by configuring switches to generate lightweight telemetry reports (*i.e.*, postcards) for each probe.

However, even with the aid of Mirror/INT, achieving sufficient path coverage still incurs high probe overhead (Challenge 2): the number of probes required remains $O(n \ln n)$, where n denotes the number of equal-cost paths. Experimental results (Figure 6) show that this combination incurs 1000× higher overhead compared to OPP.

Ping + IP-in-IP for path control. Some probing systems [3, 4, 36] use source routing protocols, such as IP-in-IP [37, 38], to achieve switch and link coverage by assigning the path that probes should follow. However, IP-in-IP-based solutions often use the target switch's IP address as the destination, which is different from that of service flows. Therefore, source routing is not suitable for service tracing. Similarly, Bidirectional Forwarding Detection (BFD) [39] uses the target switch's IP address as the destination, making it unsuitable for service tracing.

In conclusion, even with the state-of-the-art commodity measurement techniques, it is still challenging to achieve efficient service tracing under packet spraying.

3 OMNIPATH PING OVERVIEW

Our key insight is that the limitations of traditional probing primitives under packet spraying stem from the host's inherent lack of visibility into and control over in-network forwarding behaviors. Armed with this insight, we propose OPP, the first diagnostic primitive designed for service tracing under packet spraying. Through switch-host co-design, OPP enables real-time service tracing with full path coverage at minimal overhead.

In this section, we first delve into the challenges posed by packet spraying and the corresponding techniques to address them. Then, we outline OPP's workflow.

3.1 Challenges and Techniques

Challenge 1: path ambiguity. Packet spraying renders Traceroute incapable of directly pinpointing the exact failure location based solely on lost probes.

Technique 1: in-network OP cache. We deploy OP cache on the switch data plane to capture per-hop telemetry in situ. As a probe traverses the network, it initializes an OP cache entry on each traversed switch to capture per-hop measurement results (*e.g.*, queueing delay). Subsequently, as the corresponding ACK returns, it aggregates and reports the results stored in OP cache entries across the traversed switches. To address failure scenarios involving lost probes or ACKs, we introduce an entry expiration mechanism. Entries that remain uncollected after a predefined threshold proactively trigger an ACK to the probe sender. By strategically configuring the expiration timers, we ensure that the entry on the switch closest to the failure point expires earliest, thus accurately pinpointing the failure location. (§4)

Challenge 2: costly path coverage. Achieving full coverage of n paths requires $O(n \ln n)$ probes, imposing prohibitive probe overhead in large-scale topologies.

Technique 2: in-network duplication & deduplication. OPP shifts probe generation and path exploration from host to switch by explicitly regulating probes' in-network behaviors. To achieve full coverage, OPP performs in-network duplication, leveraging the switch to multicast a probe to all equal-cost ports. To faithfully mimic unicast forwarding behaviors for service tracing, we design a unicast-based simulated multicast operation on programmable data plane instead of using native hardware multicast. To prevent duplicate probes from converging on downstream links and causing receiver-side incast, OPP further performs in-network deduplication: only the first arriving instance of a distinct probe is forwarded downstream, while the remaining copies are dropped. In the reverse direction, ACKs undergo symmetric deduplication and duplication, while aggregating and propagating measurement results stored in OP cache along the reverse equal-cost paths. Ultimately, OPP ensures that a distinct probe/ACK traverses every equal-cost link exactly once, minimizing host and network overhead. (§4)

Challenge 3: OP cache explosion. Since every in-flight probe consumes dedicated OP cache resources, a trade-off arises between maximizing probe timeliness and minimizing resource overhead.

Technique 3: topology-aware concurrency control. Observing that each in-flight probe consumes OP cache resources exclusively along its equal-cost paths, we propose a topology-aware concurrency control mechanism to regulate the number of concurrent in-flight probes. We customize this mechanism for Clos and switchless (*e.g.*, torus [17]) topologies, leveraging their unique structural characteristics to improve concurrency under limited SRAM resources, and further generalize the mechanism to arbitrary topologies. In this way, OPP ensures real-time service tracing and scalability in large-scale networks. (§5)

3.2 OmniPath Ping Workflow

OPP performs network-wide service tracing through 4 steps.

Step 1: probe generation (§ 5.1). A probe agent residing on a host maintains a list of locally active service flows. For each service flow, the agent initiates a periodic probe task (*e.g.*, probe every 10 ms), with probe injection timing regulated by the concurrency control mechanism (§ 5.2).

Step 2: in-network probe processing (§ 4.3). At each traversed switch, the probe is conditionally duplicated or deduplicated. Meanwhile, the probe initializes an OP cache entry to record local measurement results.

Step 3: in-network ACK processing (§ 4.4). Upon receiving the probe, the probe agent at the destination host replies with a corresponding ACK. Similarly, the ACK undergoes conditional duplication or deduplication at each switch. Meanwhile, it accesses the corresponding OP cache entry to collect and aggregate the recorded measurement results.

Step 4: failure localization (§ 4.6). For each ACK, the probe agent analyzes the carried measurement results to determine whether a failure exists and pinpoint its location.

4 OMNIPATH PING DESIGN

In this section, we present the design of OPP. We begin by detailing the core data structures, including the OP cache and the probe/ACK formats. We then elaborate on the switch processing logic, describing the operations on OP cache and the triggering conditions for duplication and deduplication. Additionally, we discuss the failure localization mechanism based on the collected measurement results. Finally, we discuss the deployment of OPP on commodity switch-silicon families.

4.1 OmniPath Cache

OP cache is a stateful table deployed at the ingress pipeline of the programmable data plane, designed to provide measurement results enriched with precise location information for failure localization. To ensure that each in-flight probe can reserve its OP cache entry on every switch along its forwarding paths, we implement OP cache as a direct-access table, with entry fields listed in Table 2.

OPP uses `Token_ID` (§ 5.2), carried in both probes and their corresponding ACKs, to directly index the entry, thereby avoiding hash

Field	Description
Token_ID_C	Records the dynamically assigned Token_ID.
Token_Seq_No_C	Records Token_Seq_No of the probe to distinguish active probes from outdated ones.
Flow_ID_C	Records the flow ID, a 5-tuple (source IP, port, and Queue Pair Number (QPN), destination IP and QPN) identifying a flow in RoCE RC mode, with destination port fixed at 4791.
Input_Ports_C	Records ingress ports of probes received by the switch as a bitmap.
Output_Ports_C	Records egress ports of probes forwarded via multicast as a bitmap.
Equal_Cost_Path_Count_C	Records the number of equal-cost paths from the switch to the destination host.
Maximum_QDelay_C	Records the maximum single-hop queueing delay across equal-cost paths.
Congested_Link_C	Records the congested link (switch ID and egress port) with the maximum queueing delay.
Entry_Expiration_Time_C	Records the timestamp when the entry expires, used to detect failures like packet loss.
Is_Expired_C	Indicates if the entry has expired.
Faulty_Link_C	Records the faulty link (switch ID and egress port) where packet loss occurred.
Remaining_Hop_Count_C	Indicates the number of switches remaining to the destination host, used for loop detection.
Is_Loop_C	Indicates if there is a loop.

Table 2: Fields in an OP cache entry.

Field	Description
Token_ID_P	Records the dynamically assigned Token_ID, used to index the corresponding OP cache entry.
Token_Seq_No_P	Records the Token_Seq_No to distinguish active probes from outdated ones.
Source_QPN_P	Records the source QPN of the associated service flow, essential for ACK routing as RC packets specify only the destination QPN.
Remaining_Hop_Count_P	Indicates the number of switches remaining to the destination host. Initialized to the number of switches along the source-to-destination path.
Previous_Hop_QDelay_P	Records the queueing delay at the previous switch.
Previous_Hop_Link_P	Records the link traversed at the previous switch (switch ID and egress port).
Is_USM_P	Records whether the probe performs unicast-based simulated multicast or native multicast.

Table 3: Supplementary fields in probe packet header.

Field	Description
Token_ID_A	Records the Token_ID of the corresponding probe. Initialized to Token_ID_P.
Token_Seq_No_A	Records the Token_Seq_No of the corresponding probe. Initialized to Token_Seq_No_P.
Equal_Cost_Path_Count_A	Records the number of equal-cost paths discovered. Initialized to 1, representing the path from the destination ToR switch to the host.
Maximum_QDelay_A	Records the maximum queueing delay experienced by the probe at any switch. Initialized to Previous_Hop_QDelay_P, reflecting the delay at the destination ToR switch.
Congested_Link_A	Records the congested link (switch ID & egress port) where the maximum queueing delay occurred. Initialized to Previous_Hop_Link_P.
Faulty_Link_A	Records the faulty link where packet loss occurred. Initialized to 0 (null).
Is_Loop_A	Indicates if there is a loop.

Table 4: Supplementary fields in ACK packet header.

collisions. To further prevent collisions due to multiple concurrent in-flight probes using the same Token_ID, OPP incorporates a concurrency control mechanism (§ 5.2) that enforces a one-to-one mapping between a Token_ID and an in-flight probe. Finally, OPP uses Token_Seq_No to distinguish generations of probes. A higher Token_Seq_No signifies a newer probe, indicating the completion of the previous one. Therefore, OPP allows the newer probe to overwrite the entry.

4.2 Probe & ACK Format

Probe format. The probe format design adheres to two key principles. First, probes must mirror the essential header fields of the target service flow to accurately trace its forwarding behavior. Therefore, probes follow the packet format of service flows, including but not limited to TCP, UDP, RoCE, *etc.* We use RoCE Reliable Connection (RC) as an example in the rest of this paper. Second, probes must include a unique identifier to distinguish them from service traffic to prevent ambiguity and cross-interference. Refer to Appendix A for the detailed probe format.

Moreover, the probe header includes several supplementary fields (Table 3), initialized and placed at the beginning of payload, to ensure proper functionality. Specifically, the probe carries Token_ID_P and Token_Seq_No_P to locate the corresponding OP cache entry and identify the probe generation. The allocation of these values to each probe is performed by the concurrency control (§ 5.2), in order to limit the OP cache entry requirement.

ACK format. Upon receiving a probe, the probe agent uses the information carried by the probe to construct and transmit an ACK formatted as an RC packet. The ACK header also includes several supplementary fields (Table 4), initialized and placed at the beginning of the payload.

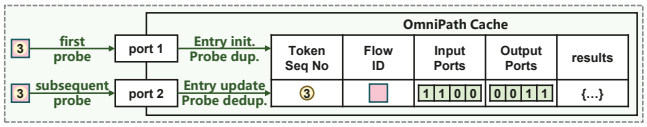


Figure 2: Probe processing logic.

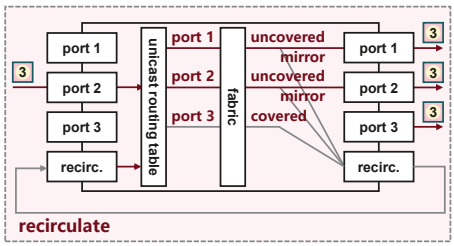


Figure 3: Unicast-based simulated multicast.

4.3 Switch-side Probe Processing

Upon receiving a probe, the switch queries and updates the OP cache entry indexed by the `Token_ID_P` field. Based on the cache state, the switch then determines whether to perform probe duplication or deduplication. The core processing workflow is outlined below (Figure 2).

Entry initialization & Probe duplication: Upon arrival of a new distinct probe (`Token_Seq_No_P` > `Token_Seq_No_C`), the switch initializes the cache entry to record probe-specific metadata, including the token sequence number (`Token_Seq_No_C`), the flow ID (`Flow_ID_C`), ingress/egress port bitmaps (`Input/Output_Ports_C`), remaining hop count (`Remaining_Hop_Count_C`), *etc.* An expiration timestamp (`Entry_Expiration_Time_C`) and two failure indicators (`Is_Expired_C` and `Is_Loop_C`) are simultaneously initialized to facilitate failure localization (§ 4.5 and § 4.6).

Unicast-based simulated multicast (Figure 3): After initialization, the switch triggers multicast to duplicate the probe to all equal-cost ports for full coverage. A straightforward approach is to use native multicast. However, native multicast is unsuitable for service tracing, because its forwarding logic diverges from that of unicast and therefore cannot faithfully emulate unicast service traffic. Specifically, when handling native multicast, a switch looks up its multicast routing table to obtain a group ID, and a replication engine in the switch's MMU then duplicates the packet to the set of egress ports associated with that group [40, 41]. This deviation from unicast forwarding can mask unicast-specific failures. For instance, a misconfigured unicast routing table that introduces a forwarding loop may go entirely undetected, as native multicast continues to follow the correct multicast-group port set and thus never exposes the fault.

To address this, OPP synthesizes multicast behavior using unicast primitives. Specifically, for a new distinct probe, the switch extracts a candidate bitmap of all equal-cost ports from the routing table, and initializes the egress port bitmap (`Output_Ports_C`) to the candidate bitmap. The switch then enters an iterative processing loop: it performs a standard unicast lookup to select an egress port; if this port exists in the candidate bitmap, it is cleared from the bitmap and the probe is forwarded. Simultaneously, a copy of the probe is mirrored back to the ingress pipeline via a recirculation port to explore remaining paths. To safeguard against infinite loops caused by configuration errors in the candidate bitmap, the switch terminates the process if the iteration number exceeds a recirculation threshold T_{recirc} (*e.g.*, 1000).

There is an inherent trade-off between unicast-based simulated multicast (USM) and native multicast. USM consumes recirculation bandwidth and increases probe delay due to repeated traversals of the switch pipeline, whereas native multicast may fail to expose unicast-specific failures. To enable per-probe selection between them, OPP reserves the `Is_USM_P` field in the probe header (Table 3).

Entry update & Probe deduplication: For subsequent probes associated with the current one (`Token_Seq_No_P` = `Token_Seq_No_C`), the switch updates the existing entry to record the maximum queuing delay and the corresponding link (`Maximum_QDelay_C` and `Congested_Link_C`), while also updating the ingress port bitmap (`Input_Ports_C`) to trace path connectivity. Next, these subsequent probes are dropped (deduplication), thereby eliminating redundancy and minimizing probe overhead.

During deduplication, if the switch identifies a discrepancy in the remaining hop count between probes (`Remaining_Hop_Count_P` ≠ `Remaining_Hop_Count_C`), it interprets this as a routing loop failure, and explicitly sets the loop indicator (`Is_Loop_C`) to 1.

Further details are elaborated in Appendix B.1.

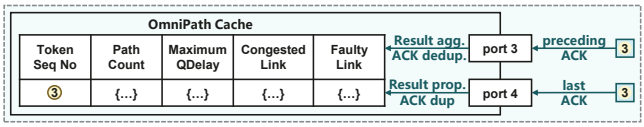


Figure 4: ACK processing logic.

4.4 Switch-side ACK Processing

Upon receiving an ACK, the switch queries and updates the corresponding OP cache entry indexed by the `Token_ID_A` field. Based on the cache state, the switch then determines whether to perform ACK duplication or deduplication. The core processing workflow is outlined below (Figure 4).

Result aggregation & ACK deduplication: For ACKs matching the current probe (`Token_Seq_No_A` = `Token_Seq_No_C`), the switch updates the entry to aggregate the measurement results, which include recording the maximum queuing delay and its corresponding link, recording the faulty link (`Faulty_Link_C`) where packet loss occurred, and accumulating the equal-cost path count (`Equal_Cost_Path_Count_C`). Simultaneously, the switch tracks the aggregation progress by clearing the bit corresponding to the ACK's ingress port in the egress port bitmap (`Output_Ports_C`), thereby confirming the receipt of the ACK from that port. Crucially, if the egress port bitmap remains non-zero, it indicates that some ACKs for the probe have not arrived yet. The switch then drops the received ACK (deduplication) until the last ACK arrives and the egress port bitmap is cleared.

Result propagation & ACK duplication: Once the egress port bitmap is cleared to zero, the switch confirms that the measurement results for the downstream paths have been fully aggregated. Consequently, the switch populates the ACK's supplementary fields with the fully aggregated results retrieved from the OP cache. Finally, it triggers multicast to duplicate the ACK to all ports encoded in the ingress port bitmap (`Input_Ports_C`) to propagate the results to the upstream switches for subsequent aggregation.

Further details are elaborated in Appendix B.2.

Table 5: Mapping switch-programming capabilities required by OPP to commodity switch-silicon families.

Required capability	Intel Tofino	Huawei NP	Juniper Trio	Broadcom Trident
Stateful memory RMW (P1)	Stateful register arrays + stateful ALUs	Stateful register arrays + memory-access engines	Shared Memory System + RMW engines	BroadScan's flow table + ALU operators
Repeated table lookups (P2)	Recirculation	Repeated memory access	Repeated memory access	Recirculation
Packet replication (P2)	Mirroring	Packet duplication via new RTC thread	PPE packet creation	Mirroring
Entry scanning (P3)	Packet generator + recirculation	Background threads	Background threads	BroadScan's hardware processes
Timestamping (P3)	Intrinsic metadata	Intrinsic metadata	Intrinsic metadata	Intrinsic metadata

4.5 OP Cache Entry Expiration

An ACK is multicast/duplicated for result aggregation only when the egress port bitmap is fully cleared to zero. However, if a probe is lost on a faulty link, the switch will fail to receive the corresponding ACK, and thus the egress port bitmap will never reach 0. To address this, OPP implements an expiration mechanism for OP cache entries, involving: (1) setting an expiration timestamp for each entry; (2) scanning entries to detect expiration; (3) generating ACKs for expired entries to trigger result aggregation.

Entry expiration timestamp: Upon initializing an OP cache entry, the switch sets the expiration timestamp (`Entry_Expiration_Time_C`) to the current time plus the product of the remaining hop count and a per-hop timeout threshold T_{expire}. This ensures that entries closer to the destination host expire sooner, as they are expected to receive ACKs earlier. T_{expire} is set to accommodate typical maximum per-hop queueing delay (*e.g.*, 10 ms), thereby preventing the misinterpretation of network congestion as loss.

Timely entry expiration: To efficiently identify expired entries, OPP iteratively scans the OP cache using recirculated scanner packets. Specifically, the switch maintains a stateful pointer that records the index of the last scanned entry, and advances the pointer by one on each scanner-packet pass to check the next entry. If the switch's current timestamp exceeds the expiration timestamp, the entry is flagged as expired by setting the expiration indicator (`Is_Expired_C`) to 1. At this point, any bit remaining set to 1 in the egress port bitmap represents a faulty link, and the switch randomly selects one of them to populate `Faulty_Link_C`.

Consider a $k = 64$ Fat-tree topology requiring 4426 entries (Figure 11) with T_{expire} set to 10 ms. Timely scanning needs a packet rate of 442 Kpps. The switch uses its hardware packet generator to inject one 64-B scanner packet every 1 ms. The packet is recirculated 442 times to advance the pointer, incurring negligible recirculation bandwidth (≈0.2 Gbps).

ACK generation for expired entries: For an expired entry, the switch creates an ACK packet, configuring its 5-tuple using `Flow_ID_C` and MAC addresses via ARP table for accurate routing. Then, the switch populates its supplementary fields with the aggregated values retrieved from the entry and multicasts it to all ports in the ingress port bitmap.

4.6 Failure Localization

For each injected probe, the probe agent in the host receives one corresponding ACK that aggregates measurement results. The agent analyzes the ACK and periodically reports failures under the following conditions:

Congestion: If `Maximum_QDelay_A` exceeds a predefined threshold T_{qdelay}, the agent reports congestion on `Congested_Link_A` for the associated service flow.

Packet loss: If `Faulty_Link_A` is non-zero, the agent reports packet loss on that link for the associated service flow. Note that to detect potential packet loss between the source host and source ToR switch, which would prevent ACK receipt, the probe agent maintains a local expiration timer for each probe, set to expire after `Remaining_Hop_Count_P` $\cdot\ T_{expire}$. If no ACK is received within this duration, the agent reports packet loss on the link between the source host and the source ToR switch for the associated service flow.

In this way, OPP effectively localizes both device-level and flow-level failures listed in Table 1. For forwarding loops, the reported faulty link may not identify the misconfigured switch, and we detail the localization of loops in Appendix C.

4.7 Deployability on Commodity Silicon

The design in § 4 targets Intel Tofino [41], a widely used programmable switch-silicon family. Appendix H provides the complete analysis of OPP's deployability on commodity switch-silicon families and summarizes the required switch-programming capabilities. In brief, to assess the deployability of OPP beyond Tofino, we decompose OPP into three core data-plane primitives: (P1) OP cache access and update, (P2) USM, and (P3) OP cache entry expiration. These primitives require five switch-programming capabilities: (1) stateful memory read-modify-write (RMW) for P1; (2) repeated lookups of the unicast routing table and (3) packet replication for P2; and (4) entry scanning and (5) timestamping for P3.

Table 5 maps these capabilities to representative commodity switch-silicon families, including Intel Tofino [41], Huawei NP [42], Juniper Trio [43], and Broadcom Trident [44]. For each capability, the table reports the mechanism used by each switch-silicon family to support it. Refer to Appendix H for a detailed explanation of the table entries.

5 NETWORK-WIDE SERVICE TRACING

In this section, we leverage OPP to conduct network-wide service tracing. We first specify the design goal and workload of network-wide service tracing, and then propose a topology-aware concurrency control mechanism to reconcile the fundamental tension between the strict SRAM constraints of OP cache and the need for high probe timeliness. Finally, we discuss the generalization of the concurrency control mechanism to arbitrary topologies.

5.1 Design Goal and Workload

Goal: The design goal of network-wide service tracing is defined as monitoring every active service flow within the network. Thus, every host-resident probe agent locally maintains a list of active service flows. Specifically, OPP leverages the lightweight eBPF-based mechanism from R-Pingmesh [6] to detect when service flows are established and closed in real time, and to obtain their 5-tuples with low overhead.

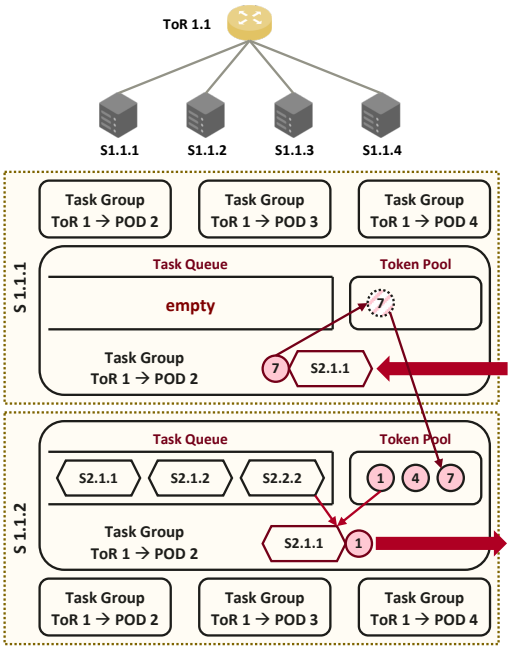


Figure 5: Real-time task scheduling.

Workload: The workload is directly determined by the design goal. For each active service flow, the agent generates a regular probe task to inject a probe every probe period (*e.g.*, every 10 ms). To manage this workload, we next introduce a topology-aware concurrency control mechanism that strictly regulates the injection timing of every probe.

5.2 Topology-aware Concurrency Control

Since OP cache operates on the switch data plane with limited SRAM resources, unthrottled probe injection causes severe entry contention, leading to measurement failures. To address this, we propose a concurrency control mechanism that regulates the number of concurrent in-flight probes, thereby ensuring that every in-flight probe deterministically acquires an entry. This mechanism consists of two phases.

Phase 1: topology-aware task grouping and entry allocation. Driven by the observation that each probe consumes OP cache entries exclusively along its equal-cost paths, we partition network-wide probe tasks into distinct task groups based on network topology and path features. To optimize resource utilization, switches allocate OP cache entries exclusively to the task groups likely to traverse them. Since the concurrency of each task group is directly constrained by its allocated entry count, the goal of task group configuration is to maximize overall concurrency under limited SRAM resources. The configuration is inherently topology-dependent. Specifically, in a Fat-tree topology, a task group is configured to aggregate all probes originating from a specific source ToR and destined for a specific destination PoD (§ D.1). Fundamentally, we require every task group to aggregate probes originating from a common source ToR to maximize the benefit of shared path features.

Phase 2: real-time task scheduling (Figure 5).

- *Setup.* For each task group, we generate a set of tokens matching the allocated entry count. Each token, identified by a `Token_ID`, serves as a unique handle for the corresponding OP cache entry on every switch along the equal-cost paths. It also carries a sequence number (`Token_Seq_No`, initialized to 0) to identify probe generations. For each task group, every probe agent under its source ToR maintains a dedicated task queue to buffer its probes awaiting injection, alongside a corresponding token pool to manage its available tokens. Upon initialization, the group's tokens are distributed evenly across these token pools.
- *Probe injection.* At the beginning of each probe period, the agent generates new probes and enqueues them into the appropriate task queues based on their task group affiliation. To balance recirculation overhead and the detection latency of unicast-specific failures, OPP uses *USM sampling*. Specifically, the agent marks a configurable fraction p of probes (*e.g.*, $p = 10\%$) as USM probes by setting the `Is_USM_P` field to 1, while the remaining probes use native multicast. Using round-robin scheduling, each probe task is guaranteed at least one USM probe every $\lceil 1/p \rceil$ probes, thereby ensuring that any unicast-specific failure is detected within $\lceil 1/p \rceil$ periods. For each queue, the agent continuously attempts to assign an available token (`Token_ID`) from the corresponding token pool to the probe at the head of the queue. Upon assignment, the agent increments the associated `Token_Seq_No` by 1 and embeds both the `Token_ID` and `Token_Seq_No` into the probe header. The probe is then dequeued and injected, and the token becomes unavailable for reuse until the corresponding ACK is received.
- *Token transfer.* If a queue runs empty, the agent transfers the available tokens in the corresponding token pool to a neighboring agent under the same ToR to avoid wasting probing opportunities.
- *Periodic reset.* At the end of each probe period, a per-ToR barrier is enforced to synchronize all agents within the rack and restore the initial token distribution.

This design yields three key properties: (1) The memory usage of OP cache is strictly bounded; (2) Each in-flight probe possesses a unique token, which prevents collisions in OP cache; (3) The management of tokens is mostly decentralized, involving only infrequent intra-rack communication comprising token transfer (triggered only upon queue exhaustion) and synchronization (triggered once per period), thus minimizing control-plane overhead.

Given this design, the configuration of task groups and the mapping from `Token_ID` to entry index are inherently topology-dependent. We focus on the configuration for two typical topologies: (1) Clos topologies and (2) switchless topologies. Configuration details are provided in Appendix D, and we summarize the core properties below.

Clos: For Clos topologies, which are the most prevalent architecture in modern DCNs, we design memory-efficient configurations for the widely-used Fat-tree and Leaf-spine. By using path features (*e.g.*, source ToR, destination ToR, and destination PoD) to configure task groups and `Token_ID`, OPP strikes a good balance between probe concurrency and memory usage. For a k-ary Fat-tree (§ D.1), OPP requires $(m_1 + m_2 \cdot \frac{k}{2} + m_3 \cdot k^2)$ entries per switch to support $(m_1 + m_2 + m_3 \cdot (k-1))$ concurrent in-flight probes per source ToR: m_1 intra-rack, m_2 intra-PoD, and m_3 inter-PoD (per destination PoD). Similarly, for a Leaf-spine with k ToR (§ D.2), OPP requires

$(m_1 + 2 \cdot m_2 \cdot k)$ entries per switch to support $(m_1 + m_2 \cdot (k-1))$ concurrent in-flight probes per source ToR: m_1 intra-rack and m_2 inter-rack (per destination rack).

Switchless: Switchless topologies, such as 2D/3D torus used in Google's TPU superPoDs [17, 45], are increasingly adopted for AI accelerator interconnects due to their high bisection bandwidth. As each node in a switchless topology is equipped with forwarding capability, we model each node as a combination of one host and one ToR. Given the absence of prominent path features, for these topologies, only the source ToR is used to configure task groups. OPP requires $m \cdot d$ entries per switch to support m concurrent in-flight probes per source ToR. Given that d is typically limited for ultra-low latency (*e.g.*, $d < 100$ [46]), such configuration does not pose scalability issues. Refer to Appendix D.3 for more details.

5.3 Generalization to Arbitrary Topologies

Beyond the topology-specialized configurations above, we generalize the task-group configuration problem to arbitrary topologies. Appendices D.4–D.6 provide the complete formulation, heuristic design, evaluation results, and practical considerations; we summarize their key points below.

In Appendix D.4, we formulate this problem as a general task-group partitioning problem. Let $\mathcal{T}$ denote the set of probe tasks and $\mathcal{V}$ denote the set of switches. Each probe task $\tau \in \mathcal{T}$ has a task footprint $D_\tau \subseteq \mathcal{V}$, *i.e.*, the set of switches that appear on at least one of its forwarding paths. OPP partitions $\mathcal{T}$ into task groups $\mathcal{G} = \{G_1, \ldots, G_r\}$, and each group allows at most one in-flight probe at a time. For a group G_i, its footprint is $D(G_i) = \bigcup_{\tau \in G_i} D_\tau$. Since only one task in G_i can be active at any moment, G_i consumes one OP cache entry on every switch in $D(G_i)$. Thus, any valid configuration must satisfy the per-switch capacity constraint $\sum_{i=1}^{r} \mathbf{1}[v \in D(G_i)] \le C_v$ for every $v \in \mathcal{V}$, where C_v is the OP cache capacity of switch v. Because tasks within the same group are executed sequentially, the worst-case completion time under unit probe delay is determined by the largest group size. The objective is therefore to find a valid task-group configuration that minimizes $\max_{G_i \in \mathcal{G}} |G_i|$.

In Appendix D.5, we cast this problem as a constrained hypergraph partitioning problem and develop a merge-based heuristic that reduces switch overload by merging task groups subject to a target group-size bound. The results in Table 6 show that, across six representative topology classes, the heuristic completes within one minute in all evaluated cases, including Dragonfly+ with over 2,000 switches and more than one million tasks. Compared with a topology-oblivious baseline, it reduces per-switch OP cache capacity demand by 3.88×–255.5×. It also achieves capacity demand close to that of the topology-specialized configurations above.

In Appendix D.6, we further discuss how to realize the heuristic-generated task groups in OPP by adding an exact-match table to the switches, which incurs less than 40 bits of additional SRAM overhead per OP cache entry. We also discuss the limitations of the heuristic: it does not provide a formal approximation ratio and does not guarantee feasibility for every possible topology and capacity setting. Nevertheless, its performance on a broad set of representative topologies (Table 6) demonstrates its practicality. Finally, OPP adapts task-group configurations to new topologies by recomputing task footprints and switch capacities from the new topology and routing paths, and then rerunning the heuristic to obtain an updated task-group partition.

6 EVALUATION

In this section, we use large-scale NS-3 [18] simulations and testbed experiments to evaluate the performance of OPP.

6.1 Large-scale NS-3 Simulations

6.1.1 Setup.

Topologies: We use four widely-used topologies: Fat-tree [15] (Clos), Leaf-spine (Clos), 2D torus [17] (switchless), and 3D torus (switchless). The largest Fat-tree and Leaf-spine are both 64-ary (radix-64 switch), containing 65536 and 2048 hosts, respectively. The largest 2D and 3D torus are 10×10 and $6 \times 6 \times 6$, containing 100 and 216 hosts, respectively. Note that we model each node in 2D and 3D torus as a ToR connected to a single host. The propagation delay of each link is set to 1 µs, and the link speed is set to 400 Gbps. In Fat-tree and Leaf-spine, the maximum base RTT is 12 µs and 8 µs, respectively. In 2D and 3D torus, the maximum base RTT is 24 µs and 22 µs, respectively.

Traffic workload: We summarize the workload construction here and provide the full construction in Appendix E.1. We construct the workload by deriving a set of service flows from the communication patterns of distributed training and inference jobs. For Clos topologies (Fat-tree and Leaf-spine), we model a distributed training job where each host represents one GPU. The modeled training job uses hybrid parallelism with TP, PP, EP, and DP. We set $PP = 8$, $TP = 8$, $EP = 64$, and $ETP = 1$, and derive the corresponding DP and EDP sizes from the total number of GPUs. Service flows are generated from inter-GPU communication within DP, EP, EDP, and PP groups, while communication within TP groups is modeled as intra-node/NVLink communication and excluded. For a 32-ary Fat-tree (8,192 GPUs), we obtain $DP = 128$, $EDP = 16$, and 66 service flows per GPU.

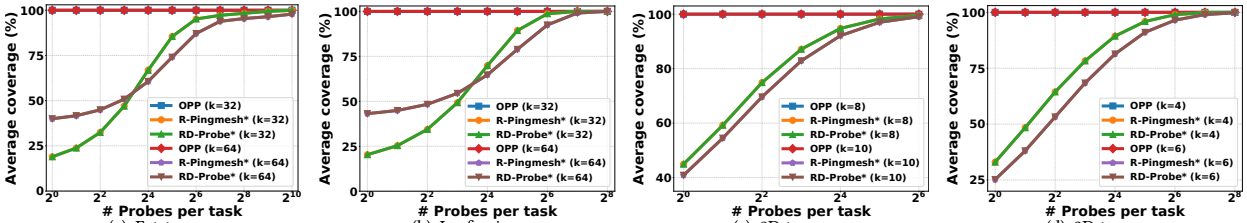
For switchless topologies (2D/3D torus), we model a distributed inference job using only EP. The EP group includes all GPUs in the topology, and we generate all-to-all EP service flows among them. For an 8×8 2D torus, this gives 63 service flows per GPU.

The resulting service-flow set is used to generate periodic probe tasks, each of which injects probes every probe period. We set both probe and ACK packets to 100 bytes.

Routing and load balancing: We use shortest-path routing as the routing policy. For LB, we adopt random packet spraying. Specifically, when a switch forwards a packet, it computes a hash over the flow ID and the current time, and uses the hash result to uniformly select an egress port from the set of equal-cost ports.

Link-level reordering: We do not introduce an in-network reordering mechanism such as ConWeave [47], as such mechanisms are not widely deployed in production networks.

Recirculation configuration: To simulate packet recirculation in NS-3, we add a recirculation port to each switch and configure the pipeline latency. We set the recirculation port speed to 400 Gbps, matching a representative port rate of current commodity switches [48–50]. For the pipeline latency, measurements on a Tofino switch running OPP show ingress and egress latencies of 387 ns and 269 ns, respectively. We approximate these values in NS-3 as 400 ns and 300 ns.



(a) Fat-tree. (b) Leaf-spine. (c) 2D torus. (d) 3D torus.

Figure 6: Coverage *vs.* host overhead on different topologies.

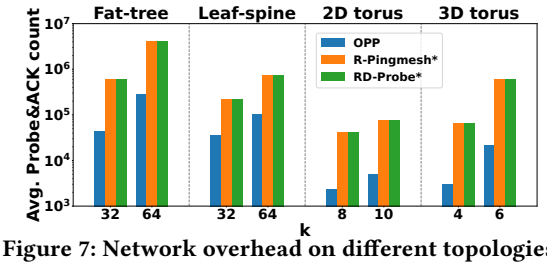


Figure 7: Network overhead on different topologies.

Failures: We inject 4 typical failures listed in Table 1 to evaluate OPP: (1) ACL drop rule misconfiguration; (2) forwarding loop; (3) link flapping; (4) congestion. We specify failure configuration in corresponding experiments.

Parameter settings: For concurrency control, we set $m = m_3 = 1$ and $m1 = m_2 = 10$, applied consistently across all four simulation topologies, with the OP cache capacity configured to exactly match the required concurrency. We set the default USM sampling rate p to 10%. We set T_{recirc}, T_{expire}, and T_{qdelay} to 1000, 10 ms, and 100 µs, respectively. Finally, we set the duration of a single probe period to 10 ms.

Baselines: We compare OPP against R-Pingmesh* [6] and RD-Probe* [16]. We denote these systems with * to indicate that we have introduced minor modifications to adapt them to service tracing under packet spraying.

- **R-Pingmesh* (Ping + Traceroute):** R-Pingmesh* utilizes Ping and Traceroute for service tracing. Upon detecting packet loss, abnormal latency, or unexpected TTL for a Ping probe, R-Pingmesh* initiates Traceroute. For each possible TTL, it injects a number of Traceroute probes equivalent to the number of Ping probes. For congested links, R-Pingmesh* identifies them by analyzing the RTT difference between Traceroute probes returned from adjacent switches. For loops, R-Pingmesh* identifies them if a switch returns a Traceroute probe with an unexpected TTL. For packet loss, as a dropped Traceroute probe cannot reveal its path, R-Pingmesh* employs a heuristic. If Traceroute probes for a specific TTL are lost, R-Pingmesh* identifies the faulty switch by selecting the one with the fewest Traceroute probes (choosing randomly if multiple switches have the same minimal count).
- **RD-Probe* (Ping + Mirror):** While RD-Probe primarily targets link and switch coverage, we repurpose its combination of Ping and mirror to enable service tracing. Leveraging mirrored probes to reconstruct forwarding paths and per-hop latency, RD-Probe* can directly pinpoint loops, congested links, and faulty links.

6.1.2 Simulation Results.

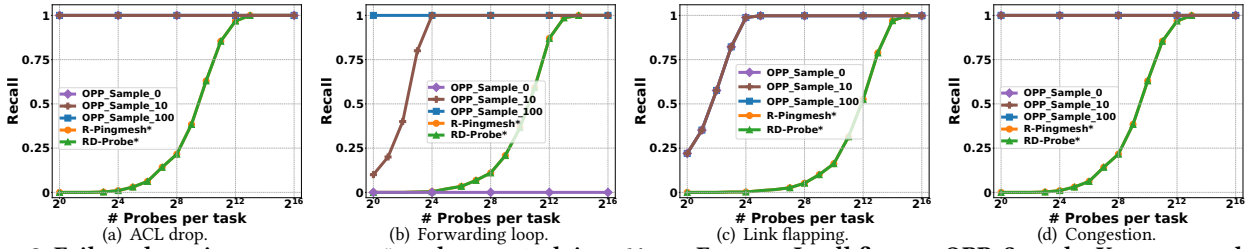
We present the selected simulation results below.

OPP achieves full path coverage with minimal host overhead. Figure 6 illustrates the average path coverage as the number of probes sent by each probe task varies. The average path coverage is computed as the mean value across all individual tasks. The results demonstrate that OPP achieves full path coverage with just a single probe per task, incurring minimal host overhead, regardless of the network topology. In contrast, R-Pingmesh* and RD-Probe* require over 1024 and 256 probes to achieve 99% coverage on 64-ary Fat-tree and $6 \times 6 \times 6$ 3D torus, respectively. This efficiency (99.90% overhead reduction) stems from OPP shifting the responsibility of probe generation and path exploration from the host CPU to the switch ASIC. Note that R-Pingmesh* and RD-Probe* exhibit identical performance, as they both rely on host-generated Ping probes for path coverage.

OPP imposes minimal network overhead. Figure 7 illustrates the average number of probes and ACKs received by a switch when OPP and the baselines achieve over 99% average path coverage. The results reveal that OPP imposes far less network overhead compared to the baselines. Specifically, for a 64-ary Fat-tree, a switch in OPP receives only 2.8×10^5 probe and ACK packets on average, a significant reduction from the 4.2×10^6 packets in R-Pingmesh* and RD-Probe*. This efficiency (93.33% overhead reduction) stems from OPP's in-network duplication and deduplication mechanisms, which guarantee that each distinct probe/ACK traverses every equal-cost link exactly once.

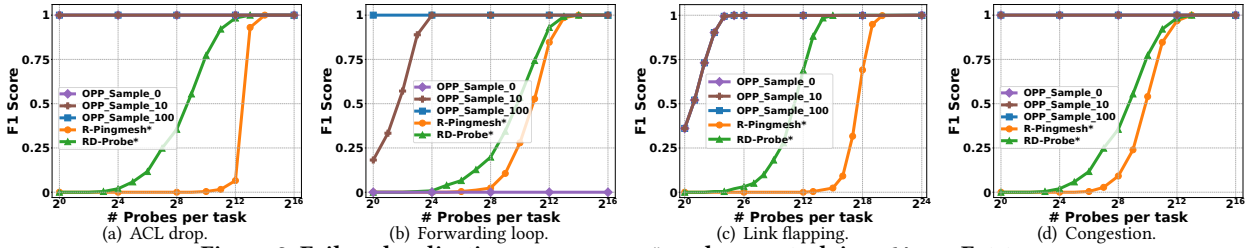
OPP detects network failures efficiently and accurately. Figure 8 illustrates the per-flow failure detection accuracy as the number of probes sent by each probe task varies. For each failure type, we simulate failure events on the uplinks from Agg switches to Core switches in a 64-ary Fat-tree topology. The detailed configuration is as follows: for ACL drop rule misconfiguration, we configure the Agg switch to drop all packets of 10 specific flows destined for the uplink; for forwarding loops, we configure the Agg switch's unicast routing table so that a probe destined for the uplink is forwarded correctly with 50% probability and misdirected to a ToR switch with the remaining 50% probability, while the native multicast table is correctly configured; for link flapping, we introduce a 20% packet drop probability on the uplink; for congestion, we impose a 500 µs queuing delay on the uplink. Here, we report the recall rate (reflecting the proportion of flows that successfully detect failures), as neither OPP nor the baselines generate false positives in this configuration.

Results demonstrate that OPP achieves full accuracy with significantly lower probe overhead compared to the baselines. Specifically, for congestion, R-Pingmesh* and RD-Probe* achieve near-full accuracy using 2^{12} probes, whereas OPP requires only a single probe.



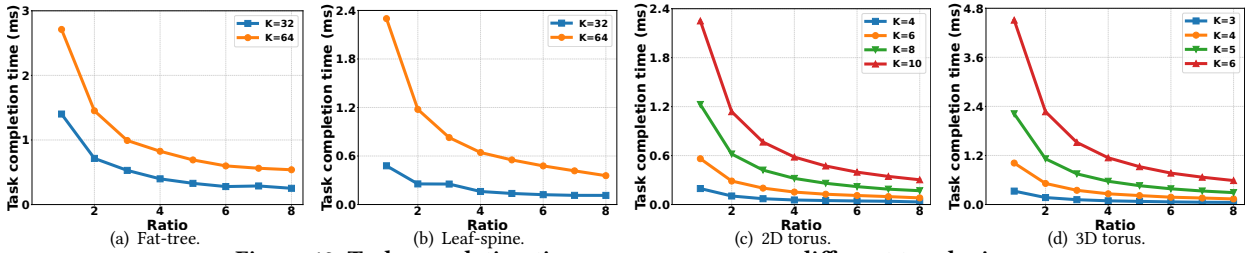
(a) ACL drop. (b) Forwarding loop. (c) Link flapping. (d) Congestion.

Figure 8: Failure detection accuracy *vs.* # probes per task in a 64-ary Fat-tree. In all figures, OPP_Sample_X corresponds to a sampling rate of X%.



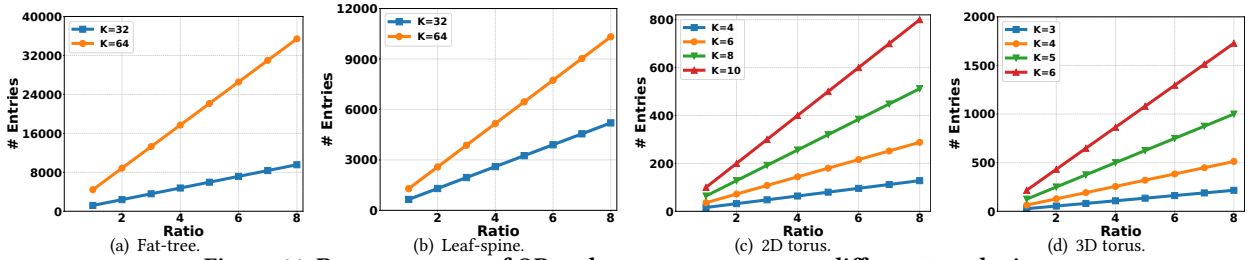
(a) ACL drop. (b) Forwarding loop. (c) Link flapping. (d) Congestion.

Figure 9: Failure localization accuracy *vs.* # probes per task in a 64-ary Fat-tree.



(a) Fat-tree. (b) Leaf-spine. (c) 2D torus. (d) 3D torus.

Figure 10: Task completion time *vs.* concurrency on different topologies.



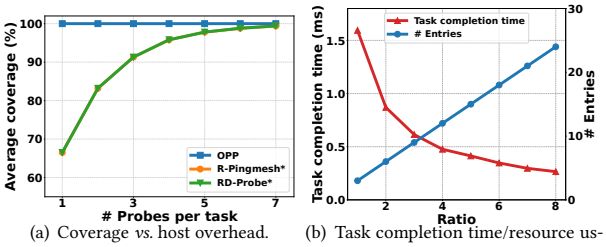
(a) Fat-tree. (b) Leaf-spine. (c) 2D torus. (d) 3D torus.

Figure 11: Resource usage of OP cache *vs.* concurrency on different topologies.

This efficiency (99.98% overhead reduction) arises from OPP's ability to achieve full path coverage with minimal overhead. Note that R-Pingmesh* and RD-Probe* remain identical in performance, given that failure detection is inherently determined by path coverage.

Results further show how USM sampling affects the detection latency of loops. Specifically, when the USM sampling rate is 100%, all probes execute USM, so one probe per task is sufficient to detect the loop. When the USM sampling rate is 10%, after each task sends one probe, only 10% of probe tasks whose probes are selected for USM can detect the loop. This fraction (recall) increases linearly as more probes are sent. When the USM sampling rate is 0%, all probes use native multicast, and OPP fails to detect loops (§ 4.3). Since OPP is configured by default to send one probe for each task in every probe period, the above results also imply that the worst-case detection latency of unicast-specific failures is inversely proportional to the USM sampling rate. The detection of other failures is not affected by the sampling rate.

OPP localizes network failures efficiently and accurately. Figure 9 illustrates the per-flow failure localization accuracy as the number of probes sent by each probe task varies. The failure configuration remains the same as above. Here, we report the F1 score, as R-Pingmesh* relies on a heuristic method for packet loss localization, which is susceptible to false positives. Results demonstrate that OPP achieves full accuracy with significantly lower probe overhead compared to the baselines. Specifically, for ACL drop rule misconfiguration, R-Pingmesh* and RD-Probe* achieve near-full accuracy using 2^{14} and 2^{12} probes, whereas OPP requires only a single probe. This efficiency (99.99% overhead reduction) stems from OPP leveraging OP cache to directly obtain per-hop visibility. Note that R-Pingmesh* and RD-Probe* diverge in performance, as the methods employed for acquiring failure localization information (*i.e.*, Traceroute and packet mirroring) exhibit distinct characteristics. As expected, the results also show that the USM sampling rate only affects OPP's localization latency of loops.



(a) Coverage *vs.* host overhead. (b) Task completion time/resource usage *vs.* concurrency.

Figure 12: Testbed experimental results.

OPP guarantees real-time service tracing with a negligible memory footprint. Figure 10 and Figure 11 illustrate the total completion time of all probe tasks within a probe period and the corresponding number of required entries in OP cache, as the concurrency control parameters (m, m_1, m_2, and m_3) are scaled by a ratio value. Specifically, for a 64-ary Fat-tree, OPP completes all tasks in just 5 ms with 4426 OP cache entries. Based on our switch implementation (§ F.1), where each entry consumes 42 bytes, this translates to a negligible 181 KB SRAM per switch to enable real-time service tracing. The results also highlight a roughly inverse relationship between completion time and the number of entries. This is expected, as the number of required entries scales linearly with the number of concurrent in-flight probes, which is inversely proportional to the total completion time.

Additional results: Appendix E.2 provides additional simulation results, and Appendix F.2 reports the corresponding testbed results: the native-multicast/USM comparison shows that USM is needed to expose unicast-specific failures and quantifies its delay and recirculation overheads, while the peak-rate analysis shows that USM-induced recirculation remains within the recirculation-port capacity under stress tests.

6.2 Testbed Experiments

Next, we present testbed settings and selected results. Refer to Appendix F for implementation details and other results.

Topology: Our testbed is a Leaf-spine with 2 leaves, 2 spines, and 4 servers. All devices are interconnected via 100Gbps links, with servers using Mellanox CX5 NICs. The base RTT is around 8 μs within a rack and 11 μs across racks.

Traffic loads: We model an all-to-all workload on the testbed. By instantiating the all-to-all operation 32 times, the number of service flows per server is $(4-1)\cdot 32 = 96$. The other settings are consistent with the simulation, except that the concurrency parameters are set to $m_1 = m_2 = 1$ and the USM sampling rate is set to 100%.

OPP achieves full path coverage with minimal host overhead. Figure 12(a) illustrates the average path coverage as the number of probes sent by each probe task varies. Results demonstrate that OPP achieves full path coverage with a single probe per task, consistent with our simulation findings.

OPP guarantees real-time service tracing with a negligible memory footprint. Figure 12(b) illustrates the total completion time of all probe tasks within a probe period and the corresponding number of required entries in OP cache, as the concurrency control parameters (m_1 and m_2) are scaled by a ratio value. Results demonstrate that OPP completes all tasks in just 2 ms with 3 entries.

Limitations of testbed scale. Our hardware testbed is limited to a 2-leaf/2-spine topology with 4 servers. Consequently, testbed results mainly validate OPP's hardware feasibility. The experiments confirm that the P4 switch pipeline and DPDK-based host agent can correctly implement the OP cache, in-network duplication/deduplication, ACK aggregation, and recirculation-based USM, while achieving full path coverage with low probe overhead and performing end-to-end failure detection/localization over real 100 Gbps links.

However, the current setup does not exercise multi-PoD Clos behavior, probe duplication at non-leaf switches with many ports, large-scale coordination across many servers, or the recirculation pressure induced by thousands of concurrent probe tasks. Our scalability claims, including 10-ms-level service tracing in the 64-ary Fat-tree and the corresponding SRAM and recirculation overheads, are therefore supported by NS-3 simulations rather than testbed measurements. Evaluation on a larger testbed remains future work.

7 RELATED WORK

Active probing: These solutions sense network quality by injecting probes into the network. Typical solutions include Pingmesh [2], NetBouncer [4], and more [3, 5, 7, 36, 51–54]. Among them, Pingmesh is the first network-wide system that monitors host-to-host connectivity and latency for failure localization. RD-Probe [16] first assesses link/switch coverage by mirroring probes on switches and then enhances it by injecting targeted, deterministic probes to cover the remaining links. R-Pingmesh [6] extends Pingmesh to RoCE networks, with an added focus on service tracing. However, these solutions rely on either source routing (*e.g.*, IP-in-IP) or ECMP, making them inadequate for service tracing under packet spraying.

Passive measurement: These solutions passively measure network traffic to analyze network quality through built-in switch features or at end-hosts. Typical solutions include INT [34, 55–63], sketches [20, 64–74], sampling [75–80], and more [26, 81–89]. Among them, INT uses switch ASIC to encapsulate per-packet per-switch metadata into packet header, including switch/link ID, timestamp, and more. OPP is orthogonal to and compatible with these solutions.

8 CONCLUSION

In this paper, we propose OPP, the first diagnostic primitive designed for service tracing under packet spraying. OPP enables accurate fault localization by leveraging OP cache and in-network duplication/deduplication to ensure full path coverage with minimal overhead. To support network-wide service tracing, OPP employs a concurrency control mechanism to regulate in-flight probes, striking a balance between SRAM constraints and probe timeliness to ensure scalability. Extensive simulation and testbed experiments demonstrate that OPP achieves 10-ms-level service tracing with full coverage and minimal probe overhead, while requiring only 181 KB of SRAM.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable suggestions that helped improve the paper. This work is supported by the National Key Research and Development Program of China under Grant No. 2024YFB2906602, and in part by the National Natural Science Foundation of China (NSFC) (No. 62372009).

REFERENCES

[1] Jon Postel. Internet control message protocol. Technical report, 1981.

[2] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[3] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. {deTector}: a topology-aware monitoring system for data center networks. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 55–68, 2017.

[4] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. {NetBouncer}: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, 2019.

[5] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. {SIMON}: A simple and scalable method for sensing, inference and measurement in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 549–564, 2019.

[6] Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, et al. R-pingmesh: A service-aware roce network monitoring and diagnostic system. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 554–567, 2024.

[7] Wei Liu, Kun Qian, Zhenhua Li, Tianyin Xu, Yunhao Liu, Weicheng Wang, Yun Zhang, Jiakang Li, Shuhong Zhu, Xue Li, et al. Skeletonhunter: Diagnosing and localizing network failures in containerized large model training. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 527–540, 2025.

[8] Christian Hopps. Analysis of an equal-cost multi-path algorithm. Technical report, 2000.

[9] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *2013 proceedings ieee infocom*, pages 2130–2138. IEEE, 2013.

[10] Wenxue Li, Xiangzhou Liu, Yunxuan Zhang, Zihao Wang, Wei Gu, Tao Qian, Gaoxiong Zeng, Shoushou Ren, Xinyang Huang, Zhenghang Ren, et al. Revisiting rdma reliability for lossy fabrics. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 85–98, 2025.

[11] Nvidia infiniband adaptive routing technology - accelerating hpc and ai applications. docs.nvidia.com/networking-ethernet-software/nvue-reference/Set-and-Unset-Commands/Adaptive-Routing/.

[12] Tommaso Bonato, Abdul Kabbani, Ahmad Ghalayini, Michael Papamichael, Mohammad Dohadwala, Lukas Gianinazzi, Mikhail Khalilov, Elias Achermann, Daniele De Sensi, and Torsten Hoefler. Reps: Recycled entropy packet spraying for adaptive load balancing and failure mitigation. *arXiv preprint arXiv:2407.21625*, 2025.

[13] Arjun Singhvi, Nandita Dukkipati, Prashant Chandra, Hassan MG Wassel, Naveen Kr Sharma, Anthony Rebello, Henry Schuh, Praveen Kumar, Behnam Montazeri, Neelesh Bansod, et al. Falcon: A reliable, low latency hardware transport. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 248–263, 2025.

[14] Marco Ferrante and Monica Saltalamacchia. The coupon collector's problem. *Materials matematics*, pages 0001–35, 2014.

[15] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

[16] Rui Ding, Xunpeng Liu, Shibo Yang, Qun Huang, Baoshu Xie, Ronghua Sun, Zhi Zhang, and Bolong Cui. Rd-probe: Scalable monitoring with sufficient coverage in complex datacenter networks. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 258–273, 2024.

[17] Narasimha R Adiga, Matthias A Blumrich, Dong Chen, Paul Coteus, Alan Gara, Mark E Giampapa, Philip Heidelberger, Sarabjeet Singh, Burkhard D Steinmacher-Burow, Todd Takken, et al. Blue gene/l torus interconnection network. *IBM Journal of Research and Development*, 49(2.3):265–276, 2005.

[18] George F Riley and Thomas R Henderson. The ns-3 network simulator. In *Modeling and tools for network simulation*, pages 15–34. Springer, 2010.

[19] Source code of OPP. https://github.com/omni-path/OmniPath-Ping.

[20] Cisco managing flows. www.cisco.com/c/en/us/td/docs/net_mgmt/xnc/nexus_data_broker/deploy_config/2-0/b_Nexus_Data_Broker_Configuration_Guide_20/b_Nexus_Data_Broker_Configuration_Guide_20_chapter_0101.pdf.

[21] Weihao Jiang, Wenli Xiao, Yuqing Yang, Peirui Cao, and Shizhen Zhao. Orderlock: A new type of deadlock and its implications on high performance network protocol design. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 575–591, 2025.

[22] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.

[23] Yu-Wei Eric Sung, Carsten Lund, Mark Lyn, Sanjay G Rao, and Subhabrata Sen. Modeling and understanding end-to-end class of service policies in operational networks. *ACM SIGCOMM Computer Communication Review*, 39(4):219–230, 2009.

[24] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. An internet-wide analysis of traffic policing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 468–482, 2016.

[25] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 214–226. 2019.

[26] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.

[27] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. *ACM SIGCOMM Computer Communication Review*, 32(4):3–16, 2002.

[28] Use parity errors troubleshooting guide. www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/116135-trouble-6500-parity-00.html.

[29] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[30] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 407–420, 2017.

[31] Yanfang Le, Rong Pan, Peter Newman, Jeremias Blendin, Abdul Kabbani, Vipin Jain, Raghava Sivaramu, and Francis Matus. Strack: A reliable multipath transport for ai/ml clusters. *arXiv preprint arXiv:2407.15266*, 2024.

[32] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. {Multi-Path} transport for {RDMA} in datacenters. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, pages 357–371, 2018.

[33] M. Foschiano et al. Cisco systems' encapsulated remote switch port analyzer (ERSPAN), 2015.

[34] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, Lawrence J Wobker, et al. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, volume 15, pages 1–2, 2015.

[35] Cisco nexus 9000 series nx-os programmability guide, release 9.2(x). www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/92x/programmability/guide/b-cisco-nexus-9000-series-nx-os-programmability-guide-92x/b-cisco-nexus-9000-series-nx-os-programmability-guide-92x_chapter_0100001.html#id_95566.

[36] Shixian Guo, Kefei Liu, Yulin Lai, Yangyang Bai, Ziwei Zhao, Songlin Liu, Jianghang Ning, Gen Li, Jianwei Hu, Yongbin Dong, et al. Bytetracker: An agentless and real-time path-aware network probing system. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 541–553, 2025.

[37] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. Per-packet load-balanced, low-latency routing for clos-based data center networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 49–60, 2013.

[38] William Simpson. Ip in ip tunneling. Technical report, 1995.

[39] Dave Katz and David Ward. Bidirectional forwarding detection (bfd). Technical report, 2010.

[40] Cisco Systems. ASR9000 Multicast Troubleshooting Document. https://community.cisco.com/kxiwq67737/attachments/kxiwq67737/4441-docs-service-providers/4115/1/91691-asr9k-multicast-troubleshooting-external.pdf.

[41] Intel Corporation. P4_16 Intel Tofino Native Architecture – Public Version. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf, 2021.

[42] Haifeng Sun, Bing Liu, Taixu Tian, Jinbo Sun, Jintao He, Qun Huang, Luyou He, Xuan Wang, Feng Gao, Liguo Wang, Xiangcan Xu, Junyi Guo, Xiaoping Zhu, and Yongqiang Yang. Rearchitecting programmable networks for in-network computing: From hardware to language. In *Proceedings of the European Conference on Computer Systems (EuroSys '26)*, pages 1094–1110, 2026.

[43] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using trio – juniper networks' programmable chipset – for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 633–648, 2022.

[44] Broadcom. Broadcom Delivers High Performance Data Plane Programmability with New Trident 3 Generation of 10/25/100G Ethernet Switches. https://investors.broadcom.com/node/12056/pdf, 2017.

[45] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*, pages 1–14, 2023.

[46] Google tpu v4. cloud.google.com/tpu/docs/v4.

[47] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. Network load balancing with in-network reordering support for rdma. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 816–831, 2023.

[48] Arista Networks. Arista 7060X5 Series 400/800G Data Center Switches Data Sheet. https://www.arista.com/assets/data/pdf/Datasheets/7060X5-Datasheet.pdf.

[49] Cisco. Cisco Nexus 9300-GX2 Series Fixed Switches Data Sheet. https://www.cisco.com/site/us/en/products/collateral/networking/switches/nexus-9000-series-switches/nexus-9300-gx2-series-fixed-switches-data-sheet.html.

[50] NVIDIA. NVIDIA Spectrum-3 SN4000 Switch Systems Hardware User Manual. https://docs.nvidia.com/networking/display/sn4000/specifications.

[51] Aijay Adams, Petr Lapukhov, and J Hongyi Zeng. Netnorad: Troubleshooting networks via end-to-end probing. *Facebook White Paper*, 2016.

[52] Shunmin Zhu, Jianyuan Lu, Biao Lyu, Tian Pan, Chenhao Jia, Xin Cheng, Daxiang Kang, Yilong Lv, Fukun Yang, Xiaobo Xue, et al. Zoonet: a proactive telemetry system for large-scale cloud networks. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, pages 321–336, 2022.

[53] Amogh Dhamdhere, Renata Teixeira, Constantine Dovrolis, and Christophe Diot. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proceedings of the 2007 ACM CoNEXT conference*, pages 1–12, 2007.

[54] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in {RDMA} servers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 15–29, 2023.

[55] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 662–680, 2020.

[56] Jonatan Langlet, Ran Ben Basat, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. Direct telemetry access. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 832–849, 2023.

[57] Tal Mizrahi, Gidi Navon, Giuseppe Fioccola, Mauro Cociglio, Mach Chen, and Greg Mirsky. Am-pm: Efficient network telemetry using alternate marking. *IEEE Network*, 33(4):155–161, 2019.

[58] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with {SwitchPointer}. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 453–456, 2018.

[59] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 71–85, 2014.

[60] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *Proceedings of the 1st ACM SIGCOMM symposium on software defined networking research*, pages 1–7, 2015.

[61] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with {PathDump}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 233–248, 2016.

[62] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. *ACM SIGCOMM Computer Communication Review*, 44(4):3–14, 2014.

[63] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: Intent-driven network traffic monitoring. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 295–308, 2020.

[64] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. {FlowRadar}: A better {NetFlow} for data centers. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, pages 311–324, 2016.

[65] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[66] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 207–222, 2021.

[67] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 404–421, 2020.

[68] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[69] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. {LightGuardian}: A {full-visibility}, lightweight, in-band telemetry system using sketchlets. In *18th USENIX symposium on networked systems design and implementation (NSDI 21)*, pages 991–1010, 2021.

[70] Hao Zheng, Chengyuan Huang, Xiangyu Han, Jiaqi Zheng, Xiaoliang Wang, Chen Tian, Wanchun Dou, and Guihai Chen. μmon: Empowering microsecond-level network monitoring with wavelets. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 274–290, 2024.

[71] Kaicheng Yang, Yuhan Wu, Ruijie Miao, Tong Yang, Zirui Liu, Zicang Xu, Rui Qiu, Yikai Zhao, Hanglong Lv, Zhigang Ji, et al. Chamelemon: Shifting measurement attention as network state changes. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 881–903, 2023.

[72] Yinda Zhang, Peiqing Chen, and Zaoxing Liu. Octosketch: Enabling real-time, continuous network monitoring over multiple cores. In *NSDI*, 2024.

[73] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. Toward {Nearly-Zero-Error} sketching via compressive sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 1027–1044, 2021.

[74] Rui Ding, Shibo Yang, Xiang Chen, and Qun Huang. Bitsense: Universal and nearly zero-error optimization for sketch counters with compressive sensing. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 220–238, 2023.

[75] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. Technical report, 2001.

[76] Vyas Sekar, Michael K Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G Andersen. csamp: A system for network-wide flow monitoring. 2008.

[77] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

[78] Benoit Claise. Cisco systems netflow services export version 9. Technical report, 2004.

[79] Nick G Duffield and Matthias Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM transactions on networking*, 9(3):280–292, 2002.

[80] Pavlos Nikolopoulos, Christos Pappas, Katerina Argyraki, and Adrian Perrig. Retroactive packet sampling for traffic receipts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(1):1–39, 2019.

[81] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 357–371, 2018.

[82] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 129–143, 2016.

[83] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 419–435, 2018.

[84] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and TS Eugene Ng. Closed-loop network performance monitoring and diagnosis with {SpiderMon}. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 267–285, 2022.

[85] Shicheng Wang, Menghao Zhang, Xiao Li, Qiyang Peng, Haoyuan Yu, Zhiliang Wang, Mingwei Xu, Xiaohe Hu, Jiahai Yang, and Xingang Shi. Hawkeye: Diagnosing rdma network performance anomalies with pfc provenance. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 481–495, 2025.

[86] Chenxu Wang, Xumiao Zhang, Runwei Lu, Xianshang Lin, Xuan Zeng, Xinlei Zhang, Zhe An, Gongwei Wu, Jiaqi Gao, Chen Tian, et al. Towards llm-based failure localization in production-scale networks. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 496–511, 2025.

[87] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, et al. Evolution of aegis: Fault diagnosis for {AI} model training service in production. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 865–881, 2025.

[88] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. Packetscope: Monitoring the packet lifecycle inside a switch. In *Proceedings of the Symposium on SDN Research*, pages 76–82, 2020.

[89] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research*, pages 61–74, 2017.

[90] Huimin Luo, Jiao Zhang, Mingxuan Yu, Yongchen Pan, Tian Pan, and Tao Huang. Network load balancing with no out-of-order packet for roce. *IEEE Network*, 2024.

[91] Dpdk. www.dpdk.org.

[92] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[93] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 595–612, 2017.

[94] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.

[95] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithmics*, 27, 2022.

[96] Zeromq. zeromq.org.

[97] Hao Zheng, Xin Yan, Wenbo Li, Jiaqi Zheng, Xiaoliang Wang, Qingqing Zhao, Luyou He, Xiaofei Lai, Feng Gao, Fuguang Huang, Wanchun Dou, Guihai Chen, and Chen Tian. When p4 meets run-to-completion architecture. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1487–1505. USENIX Association, 2025.

[98] Broadcom. Trident3-X4 / BCM56470 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56470-series.

[99] Juniper Networks. Configuring RPM Timestamping on MX, M, T, and PTX Series Routers and EX Series Switches. https://www.juniper.net/documentation/us/en/software/junos/flow-monitoring/topics/task/rpm-timestamps-configuring.html.

[100] Chris Misa. Cedar: A Reconfigurable Data Plane Telemetry System. Technical report, University of Oregon, 2020.

[101] Chris Misa, Walt O'Connor, Ramakrishnan Durairajan, Reza Rejaie, and Walter Willinger. Dynamic Scheduling of Approximate Telemetry Queries. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 701–717. USENIX Association, 2022.

[102] Mana Atarod. IRIS: An Interactive Programmable Fine-Grained Network Telemetry System. Master's thesis, University of Oregon, 2025.

[103] Juniper Networks. Juniper Apstra 5.0 User Guide. https://www.juniper.net/documentation/us/en/software/apstra5.0/apstra-user-guide/topics/topic-map/evpn-support-addendum.html.

[104] NVIDIA Corporation. NVIDIA Cumulus Linux 4.3 User Guide. https://docs.nvidia.com/networking-ethernet-software/cumulus-linux-43/Monitoring-and-Troubleshooting/Network-Troubleshooting/SPAN-and-ERSPAN.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A PROBE FORMAT

Emulating service flow headers: To monitor a service flow's forwarding behavior, OPP configures probes with the flow's header fields. As most service flows utilize the Reliable Connection (RC) transport mode in RoCE networks [90], we format our probes using the RoCE RC packet format as a representative example. To ensure comprehensive coverage, probes iteratively carry different RC Opcodes (e.g., SEND, WRITE) as well as extended header fields such as RETH or AETH across a probe period. Note that the 5-tuple, essential for routing and ACL lookups, is configured to exactly match the active service flow, while other fields, like Packet Sequence Number (PSN) and Partition Key (P_Key), are set to valid, static values that do not affect forwarding behavior.

Differentiating probes from service flows: To prevent interference with service traffic, OPP assigns a predefined non-zero value to the reserved bits in the Base Transport Header (BTH), which are set to zero by default, as a unique identifier. This allows network switches to distinguish probes from standard packets upon receipt.

Probe injection and modification: The probe agent injects probes into the network using a user-space packet generator (e.g., DPDK [91]). To avoid disrupting the destination QP at the remote NIC, OPP reserves a specific UDP port on all hosts. At the egress pipeline of the destination ToR switch, both the source and destination UDP ports are modified to this reserved port, and the UDP checksum is set to zero to disable validation. This ensures the destination host receives the probe as a standard UDP packet, preventing interference with the service flow's QP. The modification is performed only at the final switch, preserving the probe's original header for accurate service tracing.

B SWITCH-SIDE PROCESSING

B.1 Probe Processing

Upon receiving a probe in the ingress pipeline, the switch first calculates the entry index using `Token_ID_P`. Then, the switch queries and updates the corresponding OP cache entry. Based on the cache state, the switch determines whether to perform probe duplication or deduplication. The detailed processing logic follows.

Probe freshness check. The switch determines the probe's status by comparing `Token_Seq_No_P` with `Token_Seq_No_C` and checking `Is_Expired_C`.

- If `Token_Seq_No_P` > `Token_Seq_No_C`, it is a new probe (proceed to Case 1).
- If `Token_Seq_No_P` = `Token_Seq_No_C` and `Is_Expired_C` = 0, it is a duplicate probe (proceed to Case 2).
- Otherwise (outdated probe), the probe is dropped immediately.

Case 1: entry initialization and probe duplication. For the first arrival of a new distinct probe, the switch initializes a new entry by:

- Set `Flow_ID_C` using the probe's UDP header, BTH, and `Source_QPN_P`.
- Set the bit for the ingress port to 1 after resetting `Input_Ports_C` to 0.
- Set `Output_Ports_C` to a bitmap encoding all equal-cost ports, obtained from the routing table.
- Set `Maximum_QDelay_C` and `Congested_Link_C` to `Previous_Hop_QDelay_P` and `Previous_Hop_Link_P`, respectively.
- Set `Entry_Expiration_Time_C` to the current timestamp plus `Remaining_Hop_Count_P` $\cdot\ T_{expire}$ (T_{expire} is a fixed value, *e.g.*, 10 ms), used to detect loss (§ 4.5).
- Set `Remaining_Hop_Count_C` to `Remaining_Hop_Count_P`, used to detect loops (§ C).
- Reset other fields to zero.

Then, the switch duplicates the probe to all its equal-cost ports through simulated multicast, and the workflow terminates.

Case 2: entry update and probe deduplication. For a subsequent probe associated with the current entry, the switch updates the state by:

- Set the bit for the ingress port to 1 in `Input_Ports_C`.
- If `Previous_Hop_QDelay_P` exceeds `Maximum_QDelay_C`, update both `Maximum_QDelay_C` and `Congested_Link_C` to `Previous_Hop_QDelay_P` and `Previous_Hop_Link_P`, respectively.
- If `Remaining_Hop_Count_C` does not equal `Remaining_Hop_Count_P`, set `Is_Loop_C` to 1 (for loop detection, § C).

Then, the switch drops the probe (deduplication), and the workflow terminates.

Upon dequeuing into the egress pipeline (if not dropped), the switch sets `Previous_Hop_QDelay_P` to the probe's queueing delay, sets `Previous_Hop_Link_P` to the egress link ID (switch ID and port number), and decrements `Remaining_Hop_Count_P` by 1 for next-hop processing.

B.2 ACK Processing

Upon receiving an ACK in the ingress pipeline, the switch first calculates the entry index using `Token_ID_A`. Then the switch queries and updates the corresponding OP cache entry. Based on the cache state, the switch then determines whether to perform ACK duplication or deduplication. The detailed processing logic follows.

Step 1: ACK match check. The switch verifies if the ACK matches the entry by checking if `Token_Seq_No_A` equals `Token_Seq_No_C` and `Is_Expired_C` is 0. If matched, proceed to Step 2; otherwise, the ACK is outdated, so the switch drops it and terminates the workflow.

Step 2: result aggregation and ACK deduplication. The switch updates the entry to aggregate measurement results:

- Set the bit for the ACK's ingress port to 0 in `Output_Ports_C`, indicating receipt of the ACK from that port.
- Increment `Equal_Cost_Path_Count_C` by `Equal_Cost_Path_Count_A`.
- If `Maximum_QDelay_A` exceeds `Maximum_QDelay_C`, update `Maximum_QDelay_C` and `Congested_Link_C` to `Maximum_QDelay_A` and `Congested_Link_A`, respectively.
- If `Faulty_Link_C` is 0 but `Faulty_Link_A` is not, update `Faulty_Link_C` to `Faulty_Link_A`; if both are non-zero, update `Faulty_Link_C` to `Faulty_Link_A` with 50% probability.
- If `Is_Loop_A` is 1, update `Is_Loop_C` to 1.

If `Output_Ports_C` is 0, indicating all expected ACKs are received, proceed to Step 3; otherwise, the switch drops the ACK (deduplication), and the workflow terminates.

Step 3: result propagation and ACK duplication. To propagate measurement results, the switch duplicates the ACK to all ports encoded in `Input_Ports_C` through multicast. It prepares the ACK by setting `Equal_Cost_Path_Count_A`, `Maximum_QDelay_A`, `Congested_Link_A`, `Faulty_Link_A`, and `Is_Loop_A` to their corresponding `_C` fields. Using the mirror feature, the switch forwards the ACK deterministically (not randomly) to each port encoded in `Input_Ports_C` sequentially, as network quality on ACK paths is not a concern.

C FORWARDING LOOP LOCALIZATION

While OPP effectively localizes other failures that are co-located with the root causes (*e.g.*, link flapping), loops present a unique challenge due to the spatial divergence between the failure location and the root cause. Specifically, due to the probe deduplication mechanism, a looping probe is discarded at the loop closure switch where the loop closes (*i.e.*, upon its second arrival). Consequently, the immediately upstream switch reports a faulty link, masking the actual switch responsible for the misconfigured routing. OPP addresses this ambiguity through a two-phase diagnosis process:

Phase 1: loop detection. To distinguish loops from other failures, OPP leverages the `Remaining_Hop_Count_C` and `Is_Loop_C` fields. Under shortest-path routing, the loop closure switch will encounter a probe whose `Remaining_Hop_Count_P` deviates from the `Remaining_Hop_Count_C` recorded during initialization (§ 4.3). Upon detecting this mismatch, the switch sets the `Is_Loop_C` flag to 1. This status is subsequently conveyed to the agent via an ACK (§ 4.4), explicitly identifying the root cause as a routing misconfiguration.

Phase 2: routing misconfiguration localization. If the faulty link is an unexpected link originating from an on-path switch, OPP directly attributes it to a routing misconfiguration on that switch. Otherwise, to pinpoint the specific misconfigured switch, OPP re-injects the original probe into the network. Prior to re-injection, OPP configures ACLs to authorize probe processing only on switches along the expected path for the specific `Token_ID_P`. Consequently, if a misconfigured switch forwards the probe to an unauthorized switch, the receiving switch immediately drops the probe. This drop triggers a failure report for the connecting link, thereby explicitly identifying the upstream switch as the source of the misconfiguration.

Note that: (1) OPP requires no more prior knowledge than Traceroute or packet mirroring, as pinpointing a specific misconfigured switch inherently necessitates the expected path (ground truth); (2) ACL configurations can be strictly confined to switches along the expected path via a dedicated operational mode (indicated by a 1-bit flag in packet header), in which switches suppress default probe processing unless the corresponding `Token_ID_P` is explicitly authorized; (3) this approach extends to detecting other routing anomalies that induce deviations in `Remaining_Hop_Count_C`, such as suboptimal routing.

Potential index collisions. Routing misconfigurations may direct probes to unexpected switches, creating a risk where the probe's `Token_ID_P` indexes an entry already occupied by a legitimate probe, resulting in collisions and probe failures. Next, we demonstrate that such collisions are eliminated under our target topologies.

For Clos topologies (*e.g.*, Fat-Tree, § D.1), `Token_ID_P` explicitly encodes path features, including the Source ToR ID and Destination PoD ID. This allows switches to validate routing correctness by verifying these features against the local topology state (*e.g.*, ingress port, PoD/switch ID). Consequently, probes arriving at unexpected switches are immediately dropped. This mechanism not only prevents collisions but also enables the immediate localization of routing misconfigurations, obviating the need for ACL-based diagnosis. This logic extends to Leaf-Spine topologies as well (§ D.2).

For switchless topologies (§ D.3), characterized by smaller scales yet lacking explicit path features, the configuration of `Token_ID_P` and its mapping to indices are designed to be inherently collision-free.

Moreover, transient collisions caused by routing misconfigurations do not constitute a critical threat. OPP can temporarily freeze the conflicting `Token_ID` to maintain its fault localization capabilities, resuming normal operation once the failure is resolved.

D TASK GROUP CONFIGURATION

In this section, we first present task group configurations specialized for three widely used topology classes: Fat-tree, Leaf-spine, and switchless topologies. Then, we formulate the problem for arbitrary topologies, reduce it to a hypergraph partitioning view, and solve it with a heuristic.

D.1 Fat-tree

We analyze the configuration of the task group for a 3-tier k-ary Fat-tree topology. In the simplest configuration, each task group aggregates all probes originating from a specific source ToR. The corresponding `Token_ID` is a 2-tuple ⟨source ToR ID, token number⟩, where the token number ranges from 0 to $m-1$. This allows a concurrency of m in-flight probes per source ToR.

However, this design is inefficient in terms of both space and time. In a k-ary Fat-tree with $\frac{k^2}{2}$ ToR switches, a worst-case scenario assumes all concurrent in-flight probes ($m \cdot \frac{k^2}{2}$) from every ToR switch target a single destination ToR switch, requiring each switch to reserve $m \cdot \frac{k^2}{2}$ entries in OP cache. This is highly space-inefficient, as probes are unlikely to be so concentrated: under uniform traffic distribution, only m entries are needed, significantly fewer than $m \cdot \frac{k^2}{2}$. Additionally, for large-scale topologies where k is 64 or greater, the concurrency limit m must be small (*e.g.*, 1 or 2) to keep memory usage manageable, which restricts probe concurrency. This limitation hampers OPP's ability to support efficient and real-time service tracing.

Consider a network with a round-trip time (RTT), including host processing delay, of 10 µs. In a $k = 64$ Fat-tree topology, each ToR switch supports $\frac{k}{2} = 32$ servers, each with 400 active service flows to trace.[1] With a concurrency limit of $m = 2$ probes per ToR switch, a probe period requires approximately $32 \cdot 400 \cdot 10\,\mu s \div 2 = 64$ ms.

[1] 400 is justified, as expert parallelism for inference at a scale of 320 [92] generates over 300 active service flows.

Such a probe period is inadequate for real-time service tracing, as systems like R-Pingmesh trace every 10 ms.

To mitigate this inefficiency, we take path features into consideration, configuring each task group to aggregate all probes originating from a specific source ToR and destined for a specific destination PoD, while defining `Token_ID` as a 3-tuple ⟨ source ToR ID, destination PoD ID, token number ⟩, with fields specified as follows:

- *Source ToR ID:* Ranges from 0 to $\frac{k^2}{2}$. A value of 0 indicates the probe's destination is within the same source ToR switch; a value $i > 0$ indicates the probe originates from the i-th ToR switch but targets a different ToR switch.
- *Destination PoD ID:* Ranges from 0 to k. A value of 0 indicates the probe's destination host is within the same source PoD; a value $j > 0$ indicates the probe targets the j-th PoD but does not originate from it.
- *Token number:* Varies based on the other fields for fine-grained concurrency control:
  - Intra-rack (Source ToR ID = 0): Ranges from 0 to $m_1 - 1$.
  - Intra-PoD, non-intra-rack (Source ToR ID ≠ 0, Destination PoD ID = 0): Ranges from 0 to $m_2 - 1$.
  - Inter-PoD (both IDs non-zero): Ranges from 0 to $m_3 - 1$.

This configuration allows OPP to support $(m_1 + m_2 + m_3 \cdot (k-1))$ concurrent in-flight probes per source ToR switch: m_1 intra-rack, m_2 intra-PoD, and m_3 inter-PoD per destination PoD.

We analyze the OP cache entry requirements per tier in the new configuration. In a worst-case scenario:

- ToR switch: m_1 (intra-rack) + $m_2 \cdot \frac{k}{2}$ (intra-PoD) + $m_3 \cdot (\frac{k^2}{2} + k)$ (inter-PoD) entries.
- Agg switch: $m_2 \cdot \frac{k}{2}$ (intra-PoD) + $m_3 \cdot k^2$ (inter-PoD) entries.
- Core switch: $m_3 \cdot \frac{k^3}{2}$ (inter-PoD) entries.

The entry requirements for ToR and Agg switches are comparable to the simple design and acceptable. However, the demand on core switches scales cubically with k, becoming prohibitive at $k = 64$ (requiring 131072 entries).

We address this scalability bottleneck by eliminating OP cache deployment on Core switches. This design leverages a fundamental property of Clos topologies, where a Core switch has a unique downlink to any specific destination host [4, 93]. Consequently, once the source Agg switch selects a Core switch, the output port of the Core switch (connected to the destination Agg switch) is implicitly determined. This determinism allows us to abstract out the core tier and treat the ⟨ Src-Agg → Core → Dst-Agg ⟩ segment as a single logical link. Although this design slightly reduces failure localization granularity, this trade-off is acceptable for most operational scenarios.

We analyze the throughput gains of this new design. Setting $m_3 = \frac{m}{2}$ (where m is the concurrency in the simple design) for a fair comparison, the concurrency per ToR switch increases from m to $m_1 + m_2 + m_3 \cdot (k-1)$, roughly a $\frac{k}{2}$-fold improvement. Thus, a 64 ms probe period (for $k = 64$) reduces to 64 ms/32 = 2 ms, enabling real-time service tracing.

The entry index for `Token_ID` = $\langle x, y, z \rangle$ is calculated as follows:

- *ToR switch:*

$$\begin{cases} z & \text{if } x = 0 \\ z + m_1 + m_2(x - \beta) & \text{if } x \neq 0, y = 0 \\ z + m_1 + m_2 \cdot \frac{k}{2} + m_3(x-1) & \text{if } x, y \neq 0, \rho = 1 \\ z + m_1 + m_2 \cdot \frac{k}{2} + m_3(\frac{k^2}{2} + y - 1) & \text{if } x, y \neq 0, \rho = 0 \end{cases}$$

  where ToR switch IDs are contiguous within each PoD, and β denotes the minimum ToR ID in the PoD of switch x. The indicator ρ is set to 1 if the switch is the destination ToR, and 0 otherwise.
- *Agg switch:*

$$\begin{cases} z + m_2(x - \beta) & \text{if } x \neq 0, y = 0 \\ z + m_2 \cdot \frac{k}{2} + m_3(x-1) & \text{if } x, y \neq 0, \rho = 1 \\ z + m_2 \cdot \frac{k}{2} + m_3(\frac{k^2}{2} + (x-\beta)k + y - 1) & \text{if } x, y \neq 0, \rho = 0 \end{cases}$$

  The indicator ρ is set to 1 if the switch is the destination Agg, and 0 otherwise.

D.2 Leaf-spine

For a Leaf-spine topology with k leaf/ToR switches, we configure each task group to aggregate all probes originating from a specific source ToR and destined for a specific destination ToR, while defining `Token_ID` as ⟨ source ToR switch ID, destination ToR switch ID, token number ⟩, with fields specified as follows:

- *Source ToR ID*: Ranges from 1 to k, where the value i indicates that the probe originates from the i-th ToR.
- *Destination ToR ID*: Ranges from 0 to k; the value $j = 0$ indicates that the probe's destination is under the source ToR, while $j > 0$ indicates that the probe targets the j-th ToR.
- *Token number*: For intra-rack probe tasks, the token number ranges from 0 to $m_1 - 1$; for inter-rack probe tasks, the token number ranges from 0 to $m_2 - 1$.

Through this `Token_ID` allocation method, we allow up to $m_1 + m_2 \cdot (k-1)$ concurrent in-flight probes within a single ToR, and use up to $m_1 + 2 \cdot m_2 \cdot k$ entries on each ToR switch.

On a ToR switch, given a `Token_ID` = $\langle x, y, z \rangle$, the entry index can be calculated using the following formula:

$$\begin{cases} z & \text{if } y = 0 \\ z + m_1 + m_2 \cdot (x-1) & \text{if } y \neq 0, \rho = 1 \\ z + m_1 + m_2 \cdot (k + y - 1) & \text{if } y \neq 0, \rho = 0 \end{cases}$$

The indicator ρ is set to 1 if the switch is the destination ToR, and 0 otherwise. For Spine switches, due to the same considerations as for core switches in the Fat-tree topology, we do not maintain OP cache.

D.3 Switchless Topologies

For a switchless topology with d nodes, each abstracted as a ToR switch connected to one host, we adopt the simplest configuration, where each task group aggregates all probes originating from a specific source ToR. The corresponding `Token_ID` is a 2-tuple ⟨source ToR ID, token number⟩, where the source ToR ID ranges from 0 to $d - 1$ and token number from 0 to $m - 1$. This allows m concurrent in-flight probes per ToR switch. The required OP cache entries per ToR are $m \cdot d$, with the entry index for $\langle x, y \rangle$ given by $y + m \cdot x$. As d is typically small (e.g., under 100 [46]), a larger m (e.g.,

$O(10)$) ensures sufficient throughput for real-time service tracing without violating memory constraints.

Switchless topologies are often used within racks, interconnected by a Clos topology for larger systems like superPoDs. Our concurrency control can extend to such hybrid topologies by managing intra-rack and inter-rack probes separately using different `Token_ID` definitions, though further details are beyond this paper's scope.

D.4 General Formulation

The topology-specialized configurations above exploit structural properties of specific topologies. To support more general topologies, we now formulate the task group configuration problem as follows.

Let $\mathcal{V}$ be the set of switches, where switch $v \in \mathcal{V}$ has OP cache entry capacity C_v. Let $\mathcal{T}$ be the set of probe tasks. Each probe task $\tau \in \mathcal{T}$ is represented as $\tau = (s_\tau, d_\tau, D_\tau)$, where s_τ and d_τ denote the source and destination ToR switches, respectively, and $D_\tau \subseteq \mathcal{V}$ denotes the task footprint, *i.e.*, the union of all switches that appear on any forwarding path from s_τ to d_τ allowed by the routing policy. Since hosts do not maintain OP cache and thus do not affect the task footprint, we omit hosts and model probe tasks at the ToR-to-ToR granularity. For switchless topologies, we use the same notation by modeling each node as an abstract ToR switch connected to a host.

To keep OP cache usage within capacity, OPP partitions $\mathcal{T}$ into task groups and assigns each group a concurrency quota. Formally, let $\mathcal{G} = \{G_1, G_2, \ldots, G_r\}$ be a partition of $\mathcal{T}$:

$$\mathcal{T} = G_1 \dot\cup G_2 \dot\cup \cdots \dot\cup G_r,$$

where group G_i is assigned an integer concurrency quota μ_i. For a task group G_i, define its footprint as

$$D(G_i) = \bigcup_{\tau \in G_i} D_\tau.$$

At any time, at most μ_i probes from G_i can be in flight, and thus G_i consumes at most μ_i entries on each switch in $D(G_i)$. Therefore, any valid task group configuration must satisfy the per-switch capacity constraint:

$$\forall v \in \mathcal{V}, \quad \sum_{i=1}^{r} \mu_i \cdot \mathbf{1}[v \in D(G_i)] \le C_v.$$

A task group's concurrency quota must be enforced by the probe agents that launch its probes. We use a *control domain* to denote a set of probe agents that can coordinate this quota enforcement. Control domains induce a partition of the set of probe tasks:

$$\mathcal{T} = \mathcal{T}_1 \dot\cup \mathcal{T}_2 \dot\cup \cdots \dot\cup \mathcal{T}_q.$$

Each task group must be contained within one control domain:

$$\forall G_i \in \mathcal{G}, \quad \exists h \in \{1, \ldots, q\} \text{ s.t. } G_i \subseteq \mathcal{T}_h.$$

In the three topology-specialized task group configurations described above, each control domain consists of all probe tasks originating from the same source ToR:

$$\mathcal{T}_x = \{\tau \in \mathcal{T} : s_\tau = x\}.$$

This design confines quota enforcement to a ToR-local control domain, allowing probe agents under the same ToR to coordinate probe launches with low overhead.

The timeliness of a task group is determined by how many rounds are required to complete all its probe tasks. Assuming unit probe delay, a group G_i with quota μ_i requires $\lceil |G_i|/\mu_i \rceil$ rounds. Thus, the natural objective is to minimize the maximum number of rounds required for completion across task groups:

$$\min_{\mathcal{G}, \{\mu_i\}} \max_{G_i \in \mathcal{G}} \left\lceil \frac{|G_i|}{\mu_i} \right\rceil.$$

Jointly optimizing the task group partition $\mathcal{G}$ and the quotas $\{\mu_i\}$ is computationally expensive, because the capacity consumed by a group on each switch depends on both its footprint and its quota. We therefore simplify the problem by restricting the concurrency quota of each group to one, *i.e.*, $\mu_i = 1$, and refer to such groups as unit-quota task groups. This simplification does not degrade timeliness: any group G_i with quota μ_i can be split into μ_i nearly equal-sized groups with unit quota, each requiring at most $\lceil |G_i|/\mu_i \rceil$ rounds for completion, which matches the number of rounds required by the original group. Each new group's footprint is contained in $D(G_i)$, so the per-switch OP cache entry consumption does not increase.

Under this simplification, the problem reduces to minimizing the maximum group size:

$$\begin{aligned} \min \quad & \max_{G \in \mathcal{G}} |G| \\ \text{s.t.} \quad & \mathcal{G} \text{ partitions } \mathcal{T}, \\ & \forall G \in \mathcal{G},\ \exists h \in \{1, \ldots, q\} \text{ s.t. } G \subseteq \mathcal{T}_h, \\ & \forall v \in \mathcal{V},\ \textstyle\sum_{G \in \mathcal{G}} \mathbf{1}[v \in D(G)] \le C_v. \end{aligned}$$

D.5 Hypergraph View and Merge Heuristic

However, solving the optimization above is difficult because the number of possible task group partitions grows combinatorially with the number of tasks. We instead map it to a constrained hypergraph partitioning problem and solve it with a merge heuristic inspired by the coarsening strategy of the widely used KaHyPar partitioner [94, 95].

Hypergraph view: Let $\mathcal{H} = (\mathcal{U}, \mathcal{E})$ be the constructed hypergraph, where $\mathcal{U}$ is the vertex set and $\mathcal{E}$ is the hyperedge set. We create one vertex $u_\tau \in \mathcal{U}$ for each probe task $\tau \in \mathcal{T}$:

$$\mathcal{U} = \{u_\tau : \tau \in \mathcal{T}\}.$$

For each switch $v \in \mathcal{V}$, we create one hyperedge $e_v \in \mathcal{E}$ that connects the vertices corresponding to tasks whose footprints include v:

$$e_v = \{u_\tau \in \mathcal{U} : v \in D_\tau\}, \qquad \mathcal{E} = \{e_v : v \in \mathcal{V}\}.$$

For a vertex partition $\Pi = \{U_1, \ldots, U_p\}$, whose parts are called blocks, the corresponding unit-quota task group of block U_a is

$$G(U_a) = \{\tau : u_\tau \in U_a\}.$$

Each block must remain within one control domain (control-domain constraint):

$$\forall U_a \in \Pi, \quad \exists h \in \{1, \ldots, q\} \text{ s.t. } G(U_a) \subseteq \mathcal{T}_h.$$

The footprint of block U_a is defined as the footprint of its corresponding task group:

$$D(U_a) = D(G(U_a)) = \bigcup_{\tau : u_\tau \in U_a} D_\tau.$$

The connectivity of hyperedge e_v under partition Π is

$$\lambda_\Pi(e_v) = |\{U_a \in \Pi : U_a \cap e_v \neq \emptyset\}|.$$

Table 6: Capacity comparison under the same maximum group size B.

Topology	Topology-specialized config	# Switches	# Hosts	B	$\lvert\mathcal{T}\rvert$	C_{base}	C_{spec}	C_{heur}	Runtime
32-ary Fat-tree	$m_1 = m_2 = m_3 = 1$	1,280	8,192	16	261,632	16,384	1,041	1,024	3.99s
Leaf-spine (512 leaf, 256 spine)	$m_1 = m_2 = 1$	768	131,072	1	261,632	261,632	1,025	1,024	0.11s
2D torus (16×16)	$m = 43$	256	256	6	65,280	11,008	11,008	2,048	1.65s
3D torus ($8 \times 8 \times 8$)	$m = 31$	512	512	17	261,632	15,872	15,872	4,096	10.15s
Dragonfly (129 groups, 16 switches/group, 8 global links/switch)	N/A	2,064	16,512	20	4,258,032	214,656	N/A	4,096	36.93s
Dragonfly+ (33 groups, 32 leaf and 32 spine/group)	N/A	2,112	67,584	20	1,114,080	55,968	N/A	4,096	58.96s

Algorithm 1 Merge heuristic for a fixed block-size bound B.

Require: Probe tasks $\mathcal{T}$, where each $\tau \in \mathcal{T}$ has footprint D_τ; control-domain partition $\{\mathcal{T}_h\}_{h=1}^{q}$; switch capacities $\{C_v\}_{v\in\mathcal{V}}$; block-size bound B
Ensure: A feasible block partition Π representing unit-quota task groups, or Unresolved

Create one vertex u_τ for each task $\tau \in \mathcal{T}$
for all $v \in \mathcal{V}$ **do**
  $e_v \leftarrow \{u_\tau : v \in D_\tau\}$
end for
$\Pi \leftarrow \{\{u_\tau\} : \tau \in \mathcal{T}\}$ ▷ Singleton partition
for all $\{u_\tau\} \in \Pi$ **do**
  $D(\{u_\tau\}) \leftarrow D_\tau$
end for
for all $v \in \mathcal{V}$ **do**
  $\lambda_\Pi(e_v) \leftarrow |\{U \in \Pi : U \cap e_v \neq \emptyset\}|$
end for
while $\exists v \in \mathcal{V}$ such that $\lambda_\Pi(e_v) > C_v$ **do**
  $(U_a^\star, U_b^\star, \Delta^\star, A^\star) \leftarrow (\text{None}, \text{None}, 0, \infty)$
  for all block pairs (U_a, U_b) sharing overloaded hyperedges **do**
    if $\exists h$ s.t. $G(U_a) \cup G(U_b) \subseteq \mathcal{T}_h$ **and** $|U_a| + |U_b| \leq B$ **then**
      $S \leftarrow D(U_a) \cap D(U_b)$
      $\Delta \leftarrow \sum_{v\in S:\lambda_\Pi(e_v)>C_v} (\lambda_\Pi(e_v) - C_v)$ ▷ priority score
      $A \leftarrow |D(U_a) \cup D(U_b)|$ ▷ merged footprint size
      if $\Delta > \Delta^\star$ **or** $(\Delta = \Delta^\star$ **and** $A < A^\star)$ **then**
        $(U_a^\star, U_b^\star, \Delta^\star, A^\star) \leftarrow (U_a, U_b, \Delta, A)$
      end if
    end if
  end for
  if $U_a^\star = \text{None}$ **then**
    return Unresolved
  end if
  $U_{ab} \leftarrow U_a^\star \cup U_b^\star$
  $D(U_{ab}) \leftarrow D(U_a^\star) \cup D(U_b^\star)$
  $\Pi \leftarrow (\Pi \setminus \{U_a^\star, U_b^\star\}) \cup \{U_{ab}\}$
  for all $v \in D(U_a^\star) \cap D(U_b^\star)$ **do**
    $\lambda_\Pi(e_v) \leftarrow \lambda_\Pi(e_v) - 1$
  end for
end while
return Π

In this construction, $\lambda_\Pi(e_v)$ is exactly the number of task groups consuming OP cache entries on switch v. Therefore, the switch-capacity constraint becomes

$$\lambda_\Pi(e_v) \leq C_v, \quad \forall v \in \mathcal{V}.$$

Under this mapping, the objective becomes minimizing the maximum block size.

Merge heuristic (Algorithm 1): The heuristic takes a block-size bound B as input. The goal is to find a feasible hypergraph partition in which every block has size at most B while satisfying the switch-capacity and control-domain constraints.

The heuristic starts from the singleton partition, where each vertex (*i.e.*, task) forms its own block. This partition gives the smallest possible block size, but may violate switch-capacity constraints because many hyperedges can have connectivity larger than C_v. Merging two blocks reduces the connectivity of any hyperedge that intersects both blocks by one, thereby reducing the number of OP cache entries required on the corresponding switch. However, merging also increases the block size, which increases the number of rounds needed for completion and degrades probe timeliness. This motivates the merge strategy: it prioritizes merges that relieve overloaded hyperedges (*i.e.*, those with $\lambda_\Pi(e_v) > C_v$), while keeping the merged blocks small and their footprints compact.

In detail, the heuristic repeatedly applies legal merges: the two blocks must belong to the same control domain, the merged block must have size at most B, and their footprints must share at least one hyperedge whose connectivity currently exceeds capacity. At each iteration, the heuristic prioritizes merges that reduce the connectivity of more severely overloaded hyperedges. When multiple candidate merges have the same priority, it favors the one with the smallest merged footprint, keeping the resulting block compact. The procedure succeeds once all capacity constraints are satisfied, yielding a feasible partition under block-size bound B; if no legal merge remains, the heuristic fails to find a feasible partition for the given bound B.

Experimental results (Table 6): For each topology, we generate a set of probe tasks following the all-to-all communication pattern at ToR granularity, *i.e.*, one task from each ToR to every other ToR. For all experiments, control domains are defined by source ToR, consistent with the topology-specialized configurations described above:

$$\mathcal{T}_x = \{\tau \in \mathcal{T} : s_\tau = x\}.$$

Thus, blocks may only be formed by merging tasks that originate from the same ToR. We run the heuristic with block-size bound B (maximum allowed task group size) and a uniform per-switch capacity budget C_{heur} under which a feasible partition is found, as listed in Table 6. We find that the heuristic finishes within one minute in all cases. For comparison, we compute the per-switch capacity demand of two alternative methods under the same bound B: (1) a topology-oblivious baseline (denoted by C_{base}) and (2) a topology-specialized configuration (if applicable) whose parameters are chosen to meet B (denoted by C_{spec}).

The topology-oblivious baseline partitions probe tasks into unit-quota task groups without using topology information, while respecting the same control-domain constraint. It therefore partitions each control domain $\mathcal{T}_x$ independently. Under the same maximum group size B, each control domain $\mathcal{T}_x$ contributes $\lceil |\mathcal{T}_x|/B \rceil$ groups. Since any switch may be covered by all groups in the worst case, the per-switch capacity demand equals the total number of groups:

$$C_{\text{base}} = \sum_x \left\lceil \frac{|\mathcal{T}_x|}{B} \right\rceil.$$

Compared with this baseline, the heuristic reduces the per-switch capacity demand by 3.88× to 255.5× across the evaluated topologies.

Compared with topology-specialized configurations, the heuristic achieves similar capacity demand on Fat-tree and Leaf-spine, because these topology-specialized configurations already group

tasks according to topology-specific path structures, such as source ToR and destination PoD/ToR. On 2D/3D torus, however, the heuristic provides a larger reduction in per-switch capacity demand. This is because the topology-specialized configuration for switchless topologies intentionally uses a simple source-based grouping, which is essentially equivalent to the topology-oblivious baseline, since such topologies are typically small. Consequently, it does not exploit topology-specific structure as effectively as the heuristic, leading to higher per-switch capacity demand.

D.6 Practical Considerations

Realizing heuristic-generated groups: The heuristic outputs a vertex partition Π, where each block U corresponds to one unit-quota task group G. To realize this partition in OPP, we assign a distinct `Token_ID` to each group. For a group G, this token is carried by the only in-flight probe allowed for the group and corresponds to one OP cache entry on every switch in the group footprint $D(G)$. Each switch $v \in D(G)$ allocates one local OP cache entry and binds it to the token of group G. To support this binding, each switch maintains an exact-match table that maps each `Token_ID` to its local OP cache entry index. For a switch v, the control plane installs entries in this table only for tokens of groups whose footprints include v. Therefore, the number of exact-match entries on switch v is strictly bounded by the OP cache capacity constraint C_v. As a result, this mapping adds only a small SRAM overhead: with 16-bit `Token_ID` and 16-bit entry index, the SRAM usage of the exact-match table is less than 40 bits per OP cache entry. At runtime, when a probe carrying a `Token_ID` arrives at switch v, the switch looks up the exact-match table to obtain the corresponding OP cache entry index and then accesses that entry.

Note that this realization is intended for task groups generated by the heuristic. For the topology-specialized configurations described above, the `Token_ID` field already carries topology-specific path features, such as source ToR, destination PoD, or destination ToR. Therefore, each switch can directly compute the corresponding OP cache entry index from `Token_ID` using the topology-specific formulas (see Appendix D.1-D.3), without maintaining an additional exact-match table.

Extension to heterogeneous probe delays: The formulation in Appendix D.4 assumes unit probe delay, so the timeliness objective is expressed by the maximum group size. If probe tasks have heterogeneous delays, we replace the group size $|G|$ with a weighted group size

$$W(G) = \sum_{\tau \in G} w_\tau, \qquad w_\tau = \frac{\delta_\tau}{\delta_{\text{ref}}},$$

where δ_τ is the probe delay of task τ and δ_{ref} is a reference probe delay. The objective then becomes minimizing $\max_{G \in \mathcal{G}} W(G)$. In the merge heuristic, the block-size bound $|U_a|+|U_b| \le B$ is replaced by the weighted block-size bound $W(G(U_a)) + W(G(U_b)) \le B$. All other merge rules, scoring, and constraints remain unchanged.

Generality and practical limitations: The general formulation in Appendix D.4 is topology-agnostic: given task footprints and per-switch OP cache capacities, it applies to arbitrary topologies. Solving this formulation optimally, however, can be computationally expensive at large scale. Our merge heuristic provides a practical solver for this formulation, although we do not claim a formal approximation ratio or guaranteed feasibility for every possible topology and capacity setting. Nevertheless, the results in Table 6 show that the heuristic works efficiently across representative topology classes, including Clos topologies, switchless topologies, Dragonfly, and Dragonfly+.

Adapting to new topologies and task changes: For a new topology, OPP can recompute the task group partition by rerunning the heuristic with the updated task footprints and switch capacities. When the set of probe tasks changes, OPP incrementally assigns each new task to a feasible existing group within the same control domain in the short term, choosing the group that minimizes the increase in the maximum group size. Over longer timescales, OPP periodically reruns the heuristic to recompute a globally refined grouping.

E SIMULATION EXPERIMENTS

In this section, we first present the remaining simulation settings of OPP, followed by the remaining simulation results.

E.1 Other Simulation Settings

Traffic workload: We construct the workload by deriving a set of service flows from the communication patterns of distributed training and inference jobs. On Clos topologies, including Fat-tree and Leaf-spine, we model a distributed training job where each host represents one GPU, and derive the corresponding service-flow set. The modeled training job uses hybrid parallelism with Tensor Parallelism (TP), Pipeline Parallelism (PP), Expert Parallelism (EP), and Data Parallelism (DP). The PP size is shared by dense and MoE layers and is set to $PP = 8$. For dense layers, we set the TP size to $TP = 8$. The dense DP size is determined by the total number of GPUs in the topology:

$$DP = \frac{N_{\text{GPU}}}{TP \cdot PP},$$

where N_{GPU} denotes the total number of GPUs. We assign GPU indices according to a topology-aware placement, so consecutive indices correspond to nearby GPUs in the physical topology. Each GPU in the dense-layer parallel configuration is indexed as (pp, dp, tp), where $0 \le pp < PP$, $0 \le dp < DP$, and $0 \le tp < TP$ denote its PP, DP, and TP indices, respectively. The corresponding GPU index is

$$g = (pp \cdot DP + dp) \cdot TP + tp.$$

For MoE layers, we set the EP size to $EP = 64$ and the Expert Tensor Parallelism (ETP) size to $ETP = 1$. Therefore, tensor parallelism is not applied within each expert, and each expert remains unpartitioned and resides on a single GPU. The Expert Data Parallelism (EDP) size is

$$EDP = \frac{DP \cdot TP}{EP}.$$

A GPU can also be indexed in the MoE-layer parallel configuration as (pp, edp, ep). Since $ETP = 1$, this MoE-layer index maps to the GPU index:

$$g = (pp \cdot EDP + edp) \cdot EP + ep.$$

The mapping from a dense-layer index (pp, dp, tp) to the MoE-layer index (pp, edp, ep) keeps pp unchanged and sets

$$edp = \left\lfloor \frac{dp \cdot TP + tp}{EP} \right\rfloor, \quad ep = (dp \cdot TP + tp) \bmod EP.$$

We map parallel groups to the physical topology as follows. For dense layers, GPUs within the same TP group, *i.e.*, GPUs with the same (pp, dp) and different tp indices, are placed physically closest, with each TP group occupying 8 consecutive GPUs. We model TP communication as intra-node/NVLink communication and exclude it from the service-flow set. For dense DP, GPUs with the same (pp, tp) and different dp indices form a DP group. We generate DP service flows within each DP group using a halving-doubling communication pattern: for $s = 0, 1, \ldots, \log_2(DP) - 1$, GPU (pp, dp, tp) sends data to GPU $(pp, dp \oplus 2^s, tp)$, indicating a service flow from GPU (pp, dp, tp) to GPU $(pp, dp \oplus 2^s, tp)$.

For MoE layers, an EP group consists of 64 consecutive GPUs with the same (pp, edp) and different ep indices. We generate all-to-all EP service flows within each EP group: GPU (pp, edp, ep) sends data to GPU (pp, edp, ep') for all $ep' \neq ep$, indicating a service flow from GPU (pp, edp, ep) to GPU (pp, edp, ep'). To remain consistent with the intra-node/NVLink approximation, we exclude EP service flows whose two endpoints belong to the same TP group. For EDP, GPUs with the same (pp, ep) and different edp indices form an EDP group. We generate EDP service flows within each EDP group using a ring communication pattern: GPU (pp, edp, ep) sends data to GPU $(pp, (edp+1) \bmod EDP, ep)$, indicating a service flow from GPU (pp, edp, ep) to GPU $(pp, (edp + 1) \bmod EDP, ep)$.

For PP, GPUs with the same dense-layer indices (dp, tp) and different pp indices form a PP group. Because (dp, tp) is mapped one-to-one to (edp, ep) within each PP stage, these GPUs also share the same MoE-layer indices (edp, ep). We generate PP service flows only between adjacent PP stages: GPU (pp, dp, tp) sends data to GPUs $(pp - 1, dp, tp)$ and $(pp + 1, dp, tp)$ whenever those stages exist, indicating service flows from GPU (pp, dp, tp) to its adjacent PP-stage GPUs.

Consequently, for a 32-ary Fat-tree, $N_{\text{GPU}} = 8192$, $DP = 128$, and $EDP = 16$. The number of service flows per GPU is calculated as

$$\underbrace{2}_{\text{PP}} + \underbrace{(EP - TP)}_{\text{EP}} + \underbrace{\log_2(DP)}_{\text{DP}} + \underbrace{1}_{\text{EDP}} = 2 + 56 + 7 + 1 = 66.$$

On switchless topologies, including 2D and 3D torus, we model a distributed inference job using EP only. We set $EP = N_{\text{GPU}}$, so the EP group includes all GPUs in the topology. We generate all-to-all EP service flows among them. Consequently, each GPU has $(N_{\text{GPU}} - 1)$ service flows. For an 8×8 2D torus, $N_{\text{GPU}} = EP = 64$, and the number of service flows per GPU is $64 - 1 = 63$.

The resulting service-flow set is used to generate periodic probe tasks, each of which injects probes every probe period. We set both probe and ACK packets to 100 bytes, larger than the minimum 64-byte packet size, so that the reported recirculation bandwidth intentionally overestimates the actual load and serves as an upper bound.

Table 7: Extra residence time on different topologies.

Topology	Avg. (µs)	P50 (µs)	P95 (µs)	P99 (µs)
Fat-tree ($k = 64$)	90.54	85.64	141.82	178.31
Leaf-spine ($k = 64$)	89.77	85.64	142.51	171.29
2D torus ($k = 10$)	1.52	1.40	3.51	5.62
3D torus ($k = 6$)	2.44	2.11	6.32	9.83

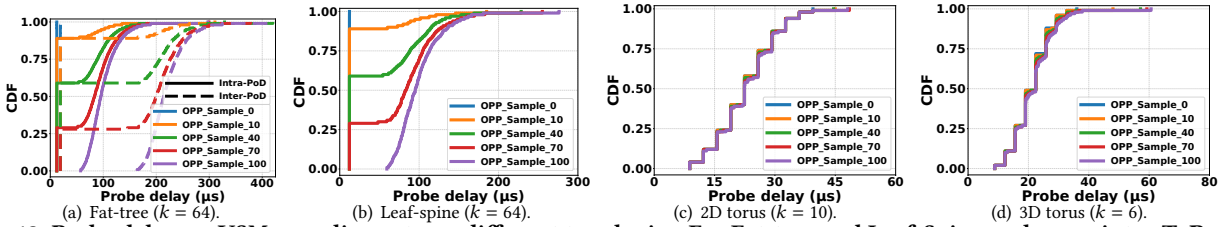
E.2 Other Simulation Results

USM sampling controls extra probe delay. Figure 13 illustrates the CDF of OPP's probe delay under different USM sampling rates. Probe delay is defined as the time from sending a probe to receiving its corresponding ACK. Overall, the USM sampling rate serves as an effective knob for controlling probe delay. On Fat-tree ($k = 64$) and Leaf-Spine ($k = 64$), probe delay is highly sensitive to the sampling rate, whereas on 2D Torus ($k = 10$) and 3D Torus ($k = 6$), the impact is limited. This difference stems from the number of egress ports that a USM probe must explore: on Fat-tree and Leaf-Spine, a USM probe needs to cover 32 uplink ports via random packet spraying, resulting in approximately 130 recirculations on average; on 2D/3D Torus, it only needs to cover 2 or 3 ports.

To validate this explanation, we further analyze the extra switch residence time of USM probes that trigger at least one duplication. We define extra switch residence time as the time between the departure of the first duplicated probe and the departure of the last duplicated probe from the switch. As shown in Table 7, the average extra switch residence time reaches 90 µs on Leaf-Spine and Fat-tree, substantially higher than 1.52/2.44 µs on 2D/3D Torus, confirming that the extra probe delay mainly comes from repeated recirculations during USM.

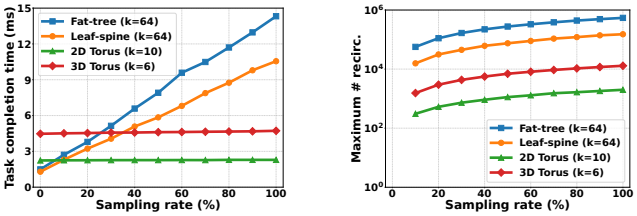
This USM-induced extra delay increases probe task completion time and can therefore increase the minimum feasible probe period, affecting the timeliness of service tracing. To control this overhead, OPP uses USM sampling so that only a sampled subset of probes executes USM. Specifically, under the default setting with a 10% USM sampling rate, the p95 and p50 inter-ToR probe delays on Leaf-Spine ($k = 64$) drop to 94.4 µs and 12.2 µs, respectively, compared with 152 µs and 95.1 µs under a 100% sampling rate. Importantly, this sampling knob affects detection latency only for unicast-specific failures: failures observable through native multicast remain exposed in every period, whereas unicast-specific failures can be exposed only by USM probes.

USM sampling controls task completion time to meet probe-period targets. Figure 14(a) illustrates the total completion time of all probe tasks within a probe period under different USM sampling rates. Since a new probe period can start only after all probe tasks in the current period complete, this completion time determines the minimum feasible probe period. We observe that the USM sampling rate provides direct control over this completion time: increasing the sampling rate increases the completion time approximately linearly, while reducing the sampling rate shortens it accordingly. This trend is expected, because probes that execute USM incur extra delay compared with probes using native multicast. Therefore, increasing the fraction of USM probes proportionally increases the total completion time. The increase is much less pronounced on 2D/3D Torus, where USM introduces only negligible extra switch residence time.



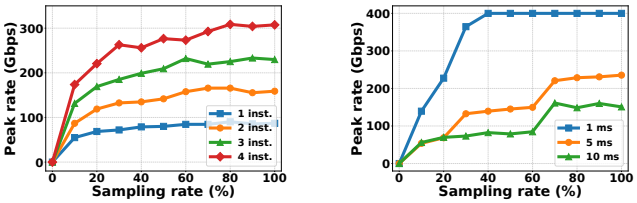
(a) Fat-tree ($k = 64$). (b) Leaf-spine ($k = 64$). (c) 2D torus ($k = 10$). (d) 3D torus ($k = 6$).

Figure 13: Probe delay *vs.* USM sampling rate on different topologies. For Fat-tree and Leaf-Spine, only non-intra-ToR probes are included (intra-ToR probes do not duplicate and are unaffected by USM).



(a) Task completion time. (b) Maximum # recirculations.

Figure 14: Overhead *vs.* USM sampling rate.



(a) Multiple concurrent instances. (b) Tightened probe periods.

Figure 15: Peak rate of the recirculation port.

These results show that the USM sampling rate can be used as an effective knob to satisfy different probe-period requirements. For example, on a 64-ary Fat-tree under the default concurrency settings ($m_1 = m_2 = 10, m_3 = 1$), OPP can meet a 10 ms probe-period target with a USM sampling rate up to 60%. When the target probe period is tightened to 5 ms, OPP can still satisfy the requirement by reducing the USM sampling rate to 30%. This reduction lowers the completion time, while its impact on detection latency is limited to unicast-specific failures.

USM sampling controls recirculation overhead to meet port-capacity constraints. Figure 14(b) illustrates the maximum per-switch recirculation count across all switches within a probe period under different USM sampling rates. As expected, the maximum recirculation count grows approximately linearly with the sampling rate, making the recirculation load predictable and directly controllable. Specifically, for a 64-ary Fat-tree with a 60% USM sampling rate, the maximum per-switch recirculation count reaches 3.3×10^5. Given a probe size of 100 bytes and a total task completion time of roughly 10 ms (Figure 14(a)), this translates to an average bandwidth of 26.4 Gbps. These results show that the USM sampling rate can be used as an effective knob to keep recirculation overhead within the capacity of recirculation ports. As discussed above, reducing the sampling rate lowers recirculation overhead, while its impact on detection latency is limited to unicast-specific failures.

USM sampling keeps peak recirculation rate within port capacity under stress. To evaluate whether OPP can sustain recirculation under stress, we measure the peak recirculation-port rate across all switches under two demanding settings: multiple concurrent diagnostic instances and tightened probe periods. We measure the peak rate over 700 ns windows, matching the modeled per-recirculation pipeline latency, *i.e.*, the sum of the ingress pipeline latency (400 ns) and the egress pipeline latency (300 ns). Therefore, the peak rate accurately reflects the number of USM probes concurrently undergoing recirculation within the switch. We focus on the 64-ary Fat-tree topology, which incurs the heaviest recirculation load among all evaluated topologies.

Figure 15(a) shows the peak recirculation rate under 1 to 4 concurrent diagnostic instances and different USM sampling rates, where each instance uses the default configuration and probe-task set. As expected, the peak rate increases with the number of instances because more USM probes may recirculate concurrently. Even under the most demanding setting of 4 concurrent instances with a 100% USM sampling rate, the peak rate reaches only 308 Gbps, remaining below the 400 Gbps recirculation-port capacity. USM sampling further provides direct control over this rate: reducing the sampling rate to 10% keeps the peak rate below 200 Gbps.

We also observe that increasing the USM sampling rate does not always strictly increase the measured peak rate. This is because the peak rate is ultimately bounded by the number of concurrent USM probes allowed by OPP's concurrency parameters. USM probes experience longer delays than native-multicast probes, so native-multicast probes may finish earlier and be replaced by newly injected probes, some of which may also be selected for USM. In this case, the number of concurrent USM probes can approach the concurrency bound even at a lower sampling rate. Thus, the sampling rate changes the distribution and likelihood of concurrent USM probes, while the worst-case concurrency remains bounded by the configured concurrency parameters.

Figure 15(b) shows the peak recirculation rate for a single diagnostic instance under different probe periods and USM sampling rates, achieved by reconfiguring OPP's concurrency parameters. As expected, the peak rate increases as the probe period becomes shorter. Under a 5 ms probe period, the recirculation port can still support a 100% USM sampling rate, with a peak rate of only 235 Gbps. Even under a 1 ms probe period, OPP can keep the peak rate within the 400 Gbps port capacity by reducing the USM sampling rate to 30%, yielding a peak rate of 364 Gbps.

Even when the instantaneous recirculation-rate demand approaches the recirculation-port capacity, the recirculation queue does not grow unbounded. The rate demand is determined by the number of probes concurrently executing USM, which is bounded by OPP's concurrency parameters. Therefore, transient overload can only queue a bounded number of probes before the concurrent USM probes drain from the switch.

Table 8: Resources used by OPP.

Resource	Usage	Percentage
Exact Match Input xbar	161	10.48%
VLIW Instructions	18	4.69%
SRAM	33	3.43%
TCAM	2	0.69%
Stateful ALU	11	22.92%

Overall, these results show that, even under extreme conditions, OPP can keep the peak recirculation rate within port capacity by configuring the USM sampling rate.

F TESTBED EXPERIMENTS

In this section, we first present the implementation details of the OPP prototype, followed by the remaining evaluation results.

F.1 OPP Implementation

Switch implementation: We have implemented the switch logic of OPP on a Tofino switch in P4. At a high level: (1) we leverage registers to construct OP cache, employing register action primitives to update and query OP cache; (2) we utilize the recirculation port and mirror features to simulate multicast operations. Given that the workflow of OPP is straightforward and clearly pipelined, we omit detailed discussion of the P4 implementation. The resource usage of OPP with OP cache consisting of 4096 entries is presented in Table 8, where each entry consumes 42 bytes in total. As shown in Figure 11, the entry usage of OPP is 4426 in a 64-ary Fat-tree under default setting, representing a negligible SRAM usage of less than 4%.

Host implementation: We implement a custom probe agent based on DPDK [91]. This agent manages the generation of probes and ACKs, while strictly enforcing our concurrency control mechanism to regulate in-flight probes. Specifically, we integrate a ZeroMQ [96] module into the agent to handle control-plane coordination (§ 5.2): it exchanges the tokens during each probe period and enforces synchronization barriers after each probe period via the out-of-band management network. Furthermore, the agent incorporates failure localization logic to pinpoint failures in real-time by analyzing the aggregated results carried by ACKs (§ 4.6).

F.2 Other Testbed Results

OPP imposes minimal network overhead. Table 10 illustrates the average number of probes and ACKs received by a switch when OPP and the baselines achieve over 99% average path coverage. The results reveal that OPP imposes far less network overhead compared to the baselines, consistent with our simulation findings.

OPP detects network failures efficiently and accurately. Figure 16 illustrates the failure detection accuracy as the number of probes sent by each probe task varies. The failure configuration remains consistent with the simulation setup, except that the failure location is shifted to the uplink between the leaf/ToR and the spine switch. Results demonstrate that (1) OPP achieves full accuracy with significantly lower probe overhead compared to the baselines, and (2) USM helps OPP detect loops, both consistent with our simulation findings.

OPP localizes network failures efficiently and accurately. Figure 17 illustrates the failure localization accuracy as the number of probes sent by each probe task varies. The failure configuration remains the same as above. Results demonstrate that (1) OPP achieves full accuracy with significantly lower probe overhead compared to the baselines, and (2) USM helps OPP localize loops, both consistent with our simulation findings.

Table 9: Extra residence time on testbed.

Topology	Avg. (µs)	P50 (µs)	P95 (µs)	P99 (µs)
Testbed	1.31	0.66	3.28	4.59

Table 10: Testbed experimental results.

System	Avg. Probe&ACK	Avg. Recirc.	Max. Recirc.
OPP	576	256	287
R-Pingmesh*	2688	N/A	N/A
RD-Probe*	2688	N/A	N/A

Table 11: Peak recirculation rate.

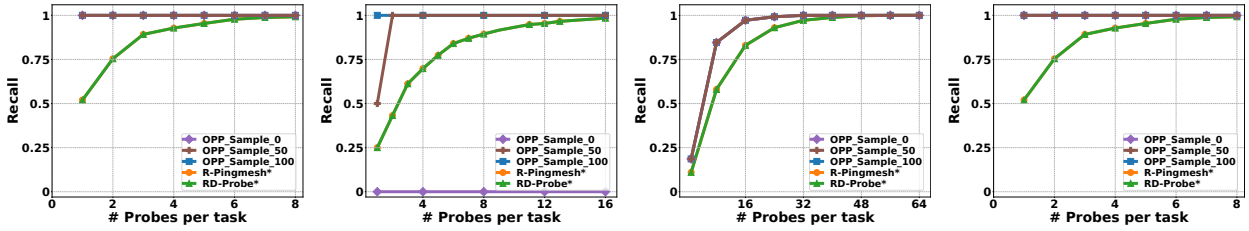
Concurrent instances	Peak rate (Gbps)	Probe period	Peak rate (Gbps)
1 instance	2.4	2 ms period	2.4
2 instances	4.9	1 ms period	4.9
3 instances	7.3	–	–
4 instances	9.8	–	–

OPP masks most USM-induced delay in testbed measurements. Figure 18(a) presents the CDF of inter-ToR probe delay with 100% and 0% USM sampling rates. Surprisingly, the two distributions almost completely overlap, except in the tail. The probe delay is not constant because the measurement includes noise from software timestamping and CPU packet processing. This result is counterintuitive because an inter-ToR probe that executes USM must be duplicated at the upstream ToR switch and forwarded to two egress ports.

We find that this is because OPP can partially mask the delay introduced by USM. Specifically, the first duplicated probe can continue independently until its ACK returns to the downstream ToR switch. After the first duplicated probe reaches the downstream ToR switch, it is forwarded to the destination host, and the corresponding ACK then returns to the same downstream ToR switch. If the second duplicated probe has also arrived at the downstream ToR switch before this ACK returns, the ACK can proceed without additional waiting. Therefore, the probe delay remains unaffected as long as the extra switch residence time at the upstream ToR switch is shorter than the interval between the arrival of the first duplicated probe at the downstream ToR switch and the return of its ACK to that switch. Only when the extra residence time exceeds this interval does the excess time contribute to probe delay. We measure this interval to be approximately 4 μs on our testbed. Table 9 reports the distribution of extra switch residence time at the upstream ToR switch for USM probes. Only a very small fraction of USM probes experience extra residence time above 4 μs, explaining why USM mainly affects the tail of the probe-delay distribution.

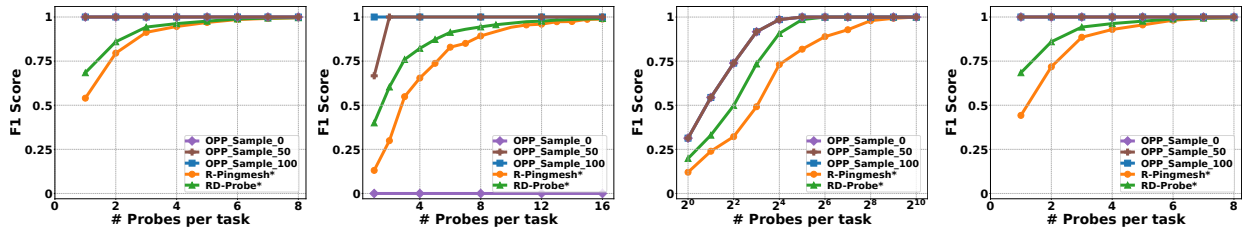
Consequently, the task completion time also changes negligibly with the USM sampling rate, as shown in Figure 18(b). The remaining small fluctuations are mainly caused by noise from software timestamping and CPU packet processing.

Recirculation port can accommodate USM. Table 10 reports the average and maximum per-switch recirculation counts induced by USM within one probe period. Table 11 further reports the peak recirculation rate under concurrent diagnostic instances and tightened probe periods, following the same experimental methodology



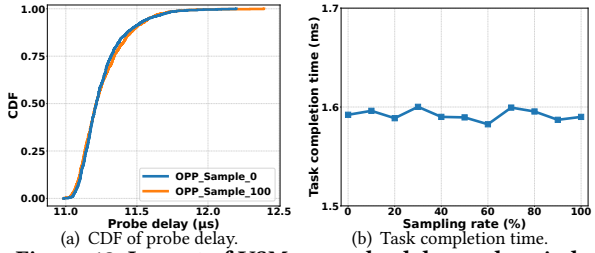
(a) ACL drop. (b) Forwarding loop. (c) Link flapping. (d) Congestion.

Figure 16: Failure detection accuracy *vs.* # probes per task on the testbed.



(a) ACL drop. (b) Forwarding loop. (c) Link flapping. (d) Congestion.

Figure 17: Failure localization accuracy *vs.* # probes per task on the testbed.



(a) CDF of probe delay. (b) Task completion time.

Figure 18: Impact of USM on probe delay and period.

as Figure 15. Together, these results show that the USM-induced recirculation overhead remains within the capacity of recirculation ports.

G SUPPORT FOR HOST-BASED LB

Extending OPP to support host-based LB requires only a single modification to the unicast-based simulated multicast mechanism. Specifically, the switch must explicitly randomize the entropy field in the probe header during each recirculation pass to forward probes to different equal-cost ports, thereby ensuring full path coverage. Efficiency can be further improved if the switch's hashing algorithm is exposed to OPP. In such cases, the agent can pre-compute a specific set of entropy values that cover all equal-cost ports for a given flow and embed them into the probes. By utilizing these deterministic values instead of random entropy, OPP eliminates the redundancy inherent in random port selection, thereby minimizing recirculation overhead.

H OPP ON COMMODITY SILICON

In this section, we explicitly present the switch-programming capabilities required by OPP and map these capabilities to representative commodity switch-silicon families, including Intel Tofino [41], Huawei NP [42, 97], Juniper Trio [43], and Broadcom Trident [98].

Core primitives and capabilities: OPP relies on three core dataplane primitives: (P1) OP cache access and update; (P2) unicast-based simulated multicast (USM) to detect unicast-specific failures; (P3) OP cache entry expiration for failure detection. At the switch-programming level, these primitives translate into five required capabilities. P1 requires stateful memory read-modify-write (RMW) for OP cache access and update. Bitmap manipulation is not a standalone requirement: OPP's bitmap operations can be implemented as RMW operations over stateful registers or equivalent direct-access memory, and are therefore covered by stateful memory RMW. P2 requires repeated lookups of the unicast routing table and packet replication to generate duplicate probes. P3 requires entry scanning to traverse OP cache entries and timestamping to determine whether an entry has expired. Table 5 summarizes how these capabilities map to representative commodity switch-silicon families, and we discuss each family in detail below.

Intel Tofino: On Intel Tofino, P1 is implemented by using stateful ALUs to perform stateful memory RMW on stateful register arrays [41]. Due to Tofino's pipeline architecture, each register can be accessed only once per packet per pipeline pass. Thus, P2 is implemented using recirculation [41] for repeated lookups of the unicast routing table and mirroring [41] for packet replication. P3 is implemented using a packet generator [41] to create scanner packets and recirculating these packets to perform entry scanning, expiration checks, and ACK generation, with precise timestamps obtained from intrinsic metadata [41].

Huawei NP: On Huawei NP, P1 can be implemented using memory-access engines to perform stateful memory RMW on stateful register arrays [42]. For P2, Huawei NP's multi-threaded run-to-completion (RTC) architecture enables repeated lookups of the unicast routing table through repeated memory accesses during single-packet processing, thereby eliminating the need for packet recirculation [42]. Packet replication can be implemented by duplicating the current packet and scheduling the duplicate to a new RTC thread. P3 can be implemented using background timer threads [42] for entry scanning, expiration checks, and ACK generation, with precise timestamps obtained from intrinsic metadata.

Juniper Trio: Juniper Trio follows a similar multi-threaded RTC design. On Trio, P1 can be implemented using RMW engines to

perform stateful memory RMW on the Shared Memory System [43]. For P2, repeated lookups of the unicast routing table can be implemented through repeated memory accesses during single-packet processing [43], while packet replication can be implemented by leveraging the packet-creation capability of Packet Processing Engines (PPEs) [43]. P3 can be implemented using background timer threads [43] for entry scanning, expiration checks, and ACK generation, with precise timestamps obtained from intrinsic metadata [99].

Broadcom Trident: On Broadcom Trident, P1 can be implemented using BroadScan ALU/reducer operators to perform stateful memory RMW on BroadScan flow-table entries [100–102]. P2 can be implemented using recirculation [103] for repeated lookups of the unicast routing table and mirroring [104] for packet replication. For P3, Broadcom Trident can support entry scanning and expiration checks using BroadScan's telemetry-specific hardware processes for aging out flow-table entries and exporting telemetry reports [100, 102], with precise timestamps obtained from intrinsic metadata [44]. Public documentation does not indicate that Trident provides a general-purpose data-plane packet generator for constructing OPP-specific ACK packets upon entry expiration. This capability, however, is not strictly required, because the control plane can receive the expiration report and inject the corresponding ACK back into the data plane.

Summary of deployment requirements: Overall, deploying OPP requires five switch-programming capabilities in the data plane. First, stateful memory RMW, which also covers bitmap manipulation, is fundamental because it supports OP cache access and update. Second, USM requires repeated lookups of the unicast routing table and packet replication. Repeated table lookups can be implemented through packet recirculation or through architectures that naturally support repeated memory accesses during single-packet processing. Packet replication can be implemented through mirroring or switch-specific packet-creation mechanisms, such as Trio's PPEs. Third, OP cache expiration requires entry scanning and timestamping. Entry scanning can be implemented through packet-driven scanning, background timer threads, or hardware telemetry processes such as BroadScan's aging/export mechanism, while timestamping can be obtained from intrinsic metadata.