RESEARCH-ARTICLE

PerfMon: Performance Monitoring of Host Network Stack

RANJITHA K, Indian Institute of Technology Hyderabad, Sangareddy, TG, India

ANKIT SHARMA, Indian Institute of Technology Hyderabad, Sangareddy, TG, India

MALSAWMSANGA SAILO, Indian Institute of Technology Hyderabad, Sangareddy, TG, India

ARUN SIDDARDHA, Indian Institute of Technology Hyderabad, Sangareddy, TG, India

AMRIT KUMAR, Indian Institute of Technology Hyderabad, Sangareddy, TG, India

PRAVEEN TAMMANA, Indian Institute of Technology Hyderabad, Sangareddy, TG, India

View all

Open Access Support provided by:

IBM Research

Indian Institute of Technology Hyderabad

Published: 19 November 2025

Citation in BibTeX format

SoCC '25: ACM Symposium on Cloud Computing
November 19 - 21, 2025
Online, USA

Conference Sponsors:
SIGOPS
SIGMOD

PerfMon: Performance Monitoring of Host Network Stack

Ranjitha K
IIT Hyderabad
India

Ankit Sharma
IIT Hyderabad
India

Malsawmsanga Sailo
IIT Hyderabad
India

Arun Siddardha*
IIT Hyderabad
India

Amrit Kumar*
IIT Hyderabad
India

Praveen Tammana
IIT Hyderabad
India

Pravein Govindan Kannan
IBM Research
India

Priyanka Naik
IBM Research
India

Abstract

Modern cloud applications are refactored into microservices, which are deployed as containers across multiple servers. An end-user request often triggers several remote procedure calls (RPCs) between these microservices. RPC latency anomalies caused by packet-processing delays (bottlenecks) in the host network stack are common. Bottlenecks at a few network components can compound across services, causing SLA violations for many requests.

Diagnosing RPC latency anomalies is challenging because many host-level components can contribute to the delay. Identifying the bottleneck component is a crucial first step. However, it often takes significant manual effort and expertise to find the bottleneck component due to the lack of visibility on per-component processing time. In this paper, we present PerfMon, a lightweight system designed to monitor the performance of components in the host stack that automatically identifies the bottleneck component. We develop PerfMon using eBPF technology and evaluate it on a Kubernetes-managed cluster of bare metal servers. Our evaluation demonstrates that PerfMon introduces minimal monitoring overheads while accurately identifying the bottleneck components.

CCS Concepts

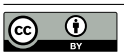
• **Networks** → **Cloud computing**; **Network monitoring**.

Keywords

network stack monitoring, RPC performance monitoring

ACM Reference Format:
Ranjitha K, Ankit Sharma, Malsawmsanga Sailo, Arun Siddardha, Amrit Kumar, Praveen Tammana, Pravein Govindan Kannan, and Priyanka Naik. 2025. PerfMon: Performance Monitoring of Host Network Stack. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3772052.3772244

*Work done while at IIT Hyderabad



SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/2025/11
https://doi.org/10.1145/3772052.3772244

1 Introduction

Cloud applications are refactored into multiple microservices distributed across physical or virtual nodes. These applications generally have one public-facing or front-end service that handles user requests by communicating with several other microservices using remote procedure calls (RPC). Typically, a user request comprises multiple RPCs, with each RPC traversing a complex path that includes multiple entities - *the source and destination end-host nodes, and the underlying network*. Within an end-host node, the RPC traverses multiple components in the host networking stack - *the container network stack, the host network stack in the root namespace, and the network interface stack* (Fig. 1).

Request latency anomalies (user requests taking longer to finish) are common in cloud applications [9, 15] due to their distributed nature. Many recent works [9, 15, 21, 34, 37, 39] have shown that sporadic transient packet-processing delays within the host network stack can increase request completion time by tens of milliseconds. Moreover, delays at a few components can compound, eventually leading to SLA violations on the completion time of many user requests. Debugging such SLA violations is both challenging and time-consuming, as there are multiple components along the request path to suspect. The challenge is further compounded by the transient and sporadic nature of these delays. Therefore, accurately identifying the bottleneck component, especially within the host stack, is a crucial first step toward diagnosing the issue. In practice, due to a lack of fine-grained and end-to-end visibility, the blame game across the teams (application, server, network) continues until the respective team is proven innocent [37], which takes a lot of time before the issue is assigned to the appropriate team. It is crucial to have a performance monitoring system that provides end-to-end visibility and automatically identifies the bottleneck component causing RPC latency anomalies.

The existing performance monitoring systems operate either at the application level or at the host level. Application-level monitoring systems [5, 13, 19, 35, 43, 45, 50] perform distributed tracing and provide the time spent at each microservice. While this helps identify which RPCs experienced delays, it lacks the granularity needed to pinpoint the specific component within the host stack responsible for those delays. On the other hand, end-host tracing approaches [11, 12, 16, 22, 33, 39] provide deep visibility into the host network stack. However, they are primarily designed for after-the-fact analysis rather than the continuous monitoring needed to detect transient and sporadic delays. Always-on trace collection imposes significant processing and storage overhead, making it

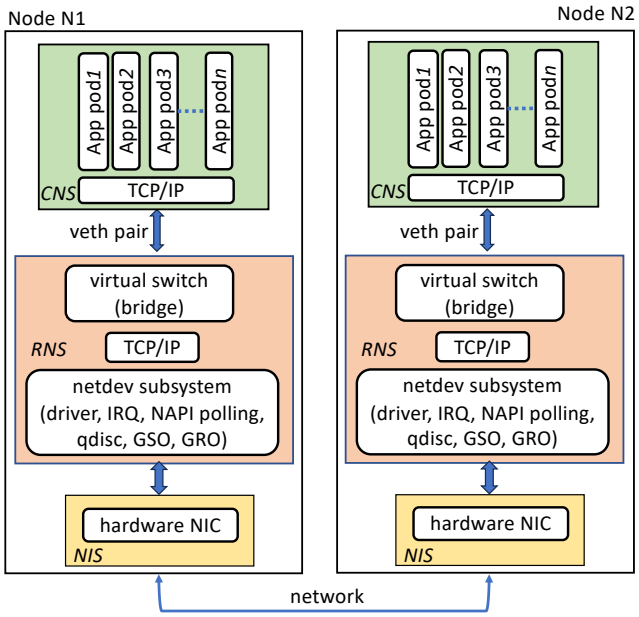


Figure 1: An RPC request and response traverse multiple entities - N1, N2, and network. Within each node, it traverses host-level components - CNS (container namespace), RNS (Root namespace), and NIS (network interface space).

impractical for real-time use in production environments. One alternative is to sample or monitor periodically, but it is hard to know when to collect traces and how long, so they often fail to detect sporadic and transient events causing bottlenecks (more in §2).

We propose PerfMon, a system that complements the existing application-level monitoring systems with fine-grained performance monitoring of the components within the host network. PerfMon enables the identification of the bottleneck component causing RPC latency anomalies, significantly accelerating the debugging process. The key idea is to continuously monitor the processing time of each component in the host stack and report bottleneck component details when the processing time takes longer than expected. This approach keeps the resource overheads low while accurately detecting abnormal events. We realize this idea using lightweight eBPF programs deployed at various hook points in the Linux kernel.

While developing PerfMon, we address three main research challenges.

C1. Identifying and instrumenting the relevant components. The first challenge is determining which component timestamps to collect within the host stack and how to maintain them to accurately identify the bottleneck component. We address this by understanding the components a packet traverses in the host network stack and instrumenting these components to compute their processing time. PerfMon continuously monitors the processing time of packets in both directions (data and acks) and raises alerts whenever abnormal delays are detected (details in §4.1).

C2. Balancing detection accuracy and overhead. The second challenge is ensuring that PerfMon's abnormal delay detection approach is both accurate and efficient. PerfMon detects a latency anomaly when the observed processing time deviates from the expected time using a predefined threshold. Arriving at the right threshold is challenging due to the heterogeneity in deployments (multiple instances of each service are deployed across nodes in a cluster with multiple RPC types and different component-specific processing times). A threshold value that is too low can increase the data collection overhead and false positives, whereas a high threshold can miss latency anomalies, increasing false negatives. We address this by carefully profiling application RPCs under various workloads and deriving thresholds that strike a balance between overhead and accuracy (details in §4.2)

C3. Distinguishing anomaly from noise. The third challenge is distinguishing transient and sporadic latency anomalies from noise (*e.g.*, jitter). Otherwise, PerfMon could raise false positives, producing alerts that are not relevant. We address this problem using a smoothing factor that absorbs the noise while detecting abnormal delays and reports only relevant alerts representing latency anomaly events (details in §5).

The key contributions of this paper are:

- We design PerfMon, a performance monitoring system that enables automatic identification of bottleneck components within the host network stack responsible for RPC latency anomalies.
- We prototype PerfMon and deploy it on a Kubernetes-managed server cluster running a microservice-based application from DeathStarBench [14], an open source benchmark suite. PerfMon prototype is available at [1].
- We evaluate PerfMon running on the Kubernetes cluster and observe that it introduces minimal monitoring overheads while accurately detecting abnormal delay events with a low false positive rate. At 40% application load (representing the baseline average load), the P999 request completion time increases by only 0.9%. Additionally, while scaling with a large number of concurrent flows (one million concurrent flows), PerfMon incurs only 3% drop in throughput.

2 Background and Motivation

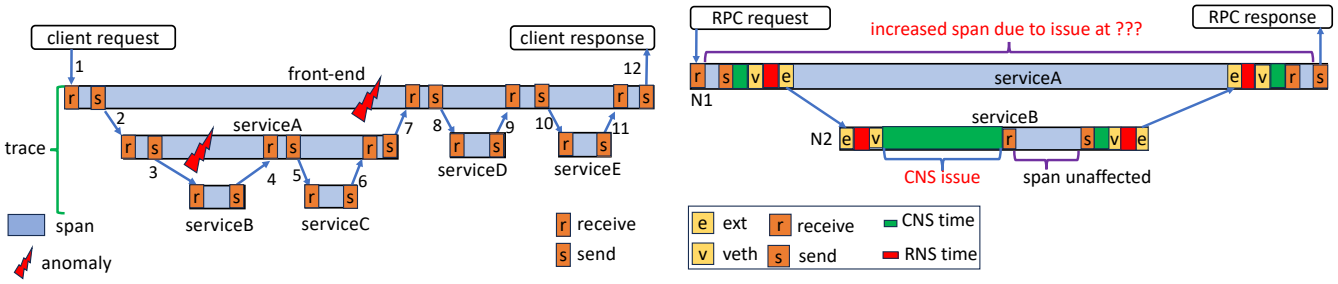
2.1 SLA violation due to slow RPCs

User requests to microservice-based distributed applications generally go to a front-end microservice, which then invokes multiple backend microservices recursively via remote procedure calls (RPCs). Each RPC request and response traverses multiple end-host nodes and network devices along its path.

Host-level components. Within each end-host node, an RPC request traverses multiple components in the host network stack at both the sender and the receiver, as shown in Fig. 1. The components include: (1) the root network stack (RNS) in the host's root namespace; (2) the container network stack (CNS) within the container namespace; and (3) the network interface stack (NIS) associated with the physical/virtual NIC.

User requests taking longer to finish, leading to an SLA violation, often result from one or more of the underlying RPCs experiencing unexpected delays. Operators often struggle to debug such SLA violations because a bottleneck at any of the host-level components

[1] https://github.com/networked-systems-iith/PerfMon-SoCC



(a) Application trace with spans of associated RPCs

(b) Trace with spans of host-level components

Figure 2: Application trace (left) lacks visibility into the host stack. Trace with component-level span (right) is desired.

(either at the sender or the receiver), or a bottleneck in the underlying network or in the application, could lead to high RPC delays. Application tracing tools like Jaeger [13] provide the span and trace of user requests (Fig. 2a), where the span is the time a single microservice takes to execute an RPC, and the trace contains spans of all RPCs invoked on behalf of the user request. This helps identify issues at the application level and RPCs that took longer to finish. However, application tracing tools [5, 13, 19, 35, 43, 45, 50] lack visibility into delays originating from the host stack and the network. Conversely, bottlenecks at the network level are diagnosed using network telemetry tools [38, 40, 42, 44, 48, 51]. In this paper, we focus on localizing bottlenecks in the end-host stack, which happen easily but are difficult to mitigate and troubleshoot.

2.2 Host-level bottlenecks inflate RPC latency

Table 1 list the events at host-level components leading to RPC latency anomalies. The two leading causes for bottlenecks are (1) unexpected CPU scheduling due to interference and (2) traffic bursts.

Unexpected CPU scheduling. CPU interference causes unexpected CPU scheduling, leading to CPU starvation (due to HOL blocking at the scheduler's ready queue), further causing buffer build-up at different buffers (such as NIX Rx buffer, CNS socket buffer). A recent work, nSight [39], observed that unexpected CPU scheduling and HOL blocking can introduce up to 5ms delay. Similarly, unexpected CPU interference from system calls delayed the ksoftirq thread, slowing reads from the NIC Rx buffer and delaying RPC packets up to tens of milliseconds [9, 15].

Incast and microburst. A sudden burst of incoming or outgoing requests can cause buffer buildup at the NIC, RNS, and CNS. An incast of many requests to an application pod from other pods causes buffer buildup at the NIC Rx [31] and CNS socket buffer [53], eventually inflating RPC latency. At RNS, a high traffic rate or burst from one or more outgoing flows causes buffer buildup (forming a standing queue at QDisc) [2, 53].

2.3 Identifying bottleneck component

Need for fine-grained host-stack visibility. Assume that serviceA and serviceB from Fig. 2a are deployed on node N1 and N2, respectively, as shown in Fig. 1. Consider that the RPC from serviceA to serviceB (3-4) is delayed due to a host-level bottleneck at N2. Consequently, the associated user request observed an SLA violation. With the existing application tracing approaches, we can identify slow RPCs (Fig. 2a) among all RPCs invoked. However, they do not provide the fine-grained visibility necessary to identify the bottleneck network stack component inflating RPCs. More specifically, serviceB's span captures the request processing time at the application level but does not include the time spent in N2's CNS, RNS, and NIS. On the other hand, serviceA's span includes (1) the time spent in CNS, RNS, and NIS at both N1 and N2; (2) the time spent in the underlying network; and (3) serviceB's span. However, it lacks the fine-grained visibility necessary to identify the bottleneck component at N2 (*e.g.*, CNS issue in Fig. 2b).

Given an RPC latency anomaly as discussed above (*e.g.*, service A to B in Fig. 2a), identifying the entity (sender, receiver, or network) and associated bottleneck component (CNS, RNS, or NIS) is the first and most crucial step toward fixing the root cause. In practice, this is often a tedious and contentious process. Most often, application teams blame the server or network teams for delays [37], and the

Table 1: Issues at host-level components

Bottleneck component	Reason for request latency anomaly	Details
NIS	Incast [31, 37]	When multiple senders send traffic to the same receiver, packets get queued at the NIC Rx buffer as the end-host network stack struggles to process the packets.
RNS	Unexpected interference at CPU core [9, 15]	Unexpected CPU interference from system calls delays scheduling netdev_subsystem processes (*e.g.* ksoftirq thread) slowing reads from NIC Rx buffers, causing delays up to 100s of milliseconds.
CNS	CPU polling hang [37]	Due to CPU starvation, the application core reads the data from the TCP receive buffer slowly.
	Burst [53]	A microburst of requests to an application pod fills the TCP receive buffer.

blame game persists until each team proves it is not at fault. A significant amount of time is wasted just assigning responsibility to the correct team before debugging can even begin. Streamlining this process requires having a tool that can automatically identify the entity and the component responsible for the issue. With such a tool, engineers can immediately begin focused debugging by collecting and analyzing component-specific traces (*e.g.*, perf record [8]) instead of spending many man-hours on collecting and analyzing traces from all components across entities, as discussed in blogs [9, 15] and prior research work [37, 39].

2.4 Challenges in identifying bottleneck component

Two features make bottleneck component identification challenging

F1: Bottlenecks happen at any host-level component. The bottleneck component could be in the RPC's sender host network stack, the receiver host network stack, or both. Finding *which* component in the host network stack is causing SLA violation is non-trivial [9, 15, 21, 34] because it requires fine-grained instrumentation capable of monitoring processing time at each component across multiple hosts.

F2: Bottleneck events are sporadic and transient. Events that cause bottlenecks are usually sporadic and transient. **(i) Sporadic** because they occur randomly at unpredictable times [15]. Any RPC and associated packets processed during the bottleneck event observe performance degradation. Thus, the monitoring system must capture performance data of every RPC at all components. **(ii) Transient** because they disappear quickly [9]. Therefore, the monitoring system should maintain fine-grained information (*e.g.*, component, timestamp) of abnormal delays so that the bottleneck component causing an SLA violation can be identified accurately.

2.5 Existing solutions fall short

Solutions based on end-to-end latency monitoring are insufficient. Distributed application-level tracing tools such as Jaeger [13], jcallgraph [45], and xtrace [35], measure the time a user request spends at each microservice (*i.e.*, span), providing visibility at the application level, but lack component-level processing time visibility. An alternative approach is to monitor TCP statistics [27, 28, 46, 50, 53] of application traffic or probe traffic and identify the problematic entity, such as the application, the host stack, and the network. However, this approach would not provide the necessary visibility to identify the bottleneck component inside the host stack (**F1**).

Solutions based on statistics collected at coarse time scales are inaccurate. As mentioned earlier, events that cause bottlenecks are sporadic and transient. Therefore, identifying a bottleneck component requires monitoring the performance of every RPC at all times for all components and capturing logs relevant to latency anomaly at each component. Existing fault localization works [28, 53] monitor TCP connection performance and collect TCP statistics periodically at coarse timescales to keep the monitoring overhead low. These statistics are insufficient as they often fail to capture sporadic and transient anomaly events (**F2**).

Solutions based on continuous trace collection have high overheads or require specific hardware. One common approach is to manually collect and analyze pcap-like traces at each component's ingress or egress at both the sender host and the receiver host [9, 15, 21, 34]. This is followed by collecting and analyzing host-level traces [11, 12, 16, 22, 25, 33, 39] (*e.g.*, perf, kprobes, intel-pt) to find the root cause. This approach provides deep visibility into host-level events, and such tools are well suited for after-the-fact root cause analysis, but not for always-on monitoring because of their high processing and storage overheads. For instance, NSight [39] collects traces using hardware profiling tool called intel_pt [25], which requires 1GB of storage space for 1 minute. One alternative is to offload monitoring computation [37] to specialized hardware such as smartNICs, but this hinders the deployment because it requires upgradation to specific hardware.

3 Design

We design PerfMon to identify the bottleneck component accurately while keeping the compute and storage overheads low. This paper focuses on RPCs using TCP as the underlying transport protocol, which is the majority case in practice. One approach is to capture fine-grained per-packet logs of TCP connections at every host, such as RTT and per-component timestamps in both directions and analyze the time spent at each component. Moreover, continuous RTT monitoring of every RPC's TCP packet gives a good estimate of end-to-end delay [32] as it covers the time spent in the entities (*i.e.*, host network stack, and the underlying network). This approach can accurately record transient and sporadic bottleneck events, but collecting, maintaining, and analyzing per-packet logs has high compute and storage overheads. Also, only a few packet logs are relevant to the bottleneck event, whereas the rest add significant overhead without contributing useful diagnostic information.An alternative is to collect only relevant logs. But it is extremely challenging to know *when to start collecting logs and how long* because of the sporadic and transient nature of the events.

PerfMon addresses this problem based on the design principle that *capture relevant logs at the right time* following distributed monitoring. The key idea is to continuously monitor per-packet RTT and component-level processing time and capture relevant logs only when they deviate from normal behavior; that is, they are found to be abnormal. This approach keeps the resource overheads low while accurately detecting transient and sporadic delay events. We realize this idea using lightweight eBPF programs deployed at various hook points in the Linux kernel. eBPF allows safe and secure extensions without changing the kernel source code or loading kernel modules. The ability to run programs in kernel space enables massive efficiency gains of up to 10x [26] faster execution when compared to running them in user space, making it an ideal choice for monitoring and observability.

3.1 PerfMon

PerfMon system overview. PerfMon comprises three components: (1) a Data Plane Agent (DPA); (2) a Control Plane Agent (CPA); and (3) a Central Controller (CC), as shown in Fig. 3. The DPA is deployed across eBPF hookpoints, which monitor per-packet RTT and component-level timestamps and raise alerts for CPA

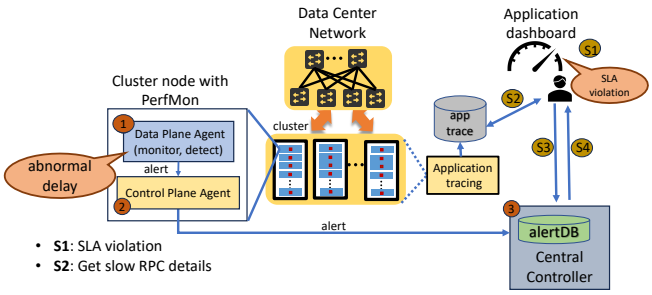


Figure 3: PerfMon system overview

whenever latency anomalies are detected. While designing DPA, we address the challenges C1, C2, and C3, as mentioned in §1, and incorporate the proposed solutions. The CPA shares the time and location of each latency anomaly with the central controller. Finally, the central controller runs queries on the database and pinpoints the bottleneck component (*i.e.*, CNS, RNS, NIS).

Example workflow. S1: From the dashboard, assume that an operator observes a user request takes longer to finish (SLA violation). S2: The operator retrieves the application-level trace for this user request, which contains details of all RPCs invoked and the time each RPC takes. S3: Next, the operator identifies the slow RPCs (those took longer than expected) and requests the central controller to find if there is any problematic network stack-level component in the slow RPC path. S4: The central controller queries the database, analyzes alerts from the components in the RPC path, and identifies the entity and associated components potentially affecting the RPC completion time.

Root cause diagnosis. Finding the bottleneck's cause is not in the scope of this paper. However, we highlight how identifying the bottleneck component (the outcome of this paper) helps to automate and find the root cause. Once the bottleneck component is identified, we can instruct the host (*i.e.*, entity) to enable trace collection only at the problematic component (instead of all components at the sender and receiver) using either eBPF kprobes [17] or CPU hardware profilers like intel_pt [25] or other tracing tools [12, 22]. The collected traces have process IDs, call traces, and per-call latency distributions, which can be correlated to find the cause [9, 15].

4 Data Plane Agent (DPA)

PerfMon's DPA is designed to perform the following tasks: (1) monitors the performance of TCP connections (associated with each RPC) across host-level network components (§4.1), (2) detects abnormal delays and reports the occurrence of abnormal delays to a control plane agent (§4.2).

4.1 Performance monitoring

RTT monitoring gives a good estimate of end-to-end delay [32]. However, this is insufficient to find the bottleneck component in the host stack, causing high RTT delays. The bottleneck component could be one or more host-level components such as CNS, RNS, and NIS. Therefore, per-component monitoring and necessary instrumentation are essential for accurately identifying the bottleneck component (**C1** in §1). To do so, we track the set of components that a packet traverses in a typical Linux-based host network stack and identify five types of per-component timestamps sufficient to monitor.

To understand this better, we break down the host components between the two nodes, N1 and N2. Fig. 4a shows the data exchange between N1 and N2 in a TCP-based RPC. The *data (D)* packet from the application microservice A traverses the N1's CNS and enters RNS via the veth interface associated with A's pod. From the RNS, N1 is left via the ext interface (the network interface that connects N1 to other cluster nodes). It traverses the underlying network, followed by RNS, CNS, and application microservice B at N2. It comprises (i) *data (D)* from N1 to N2; (ii) *data+ack (DA)* from N2 to N1 and; (iii) *ack (A)* from N1 to N2. The *data+ack (DA)* packet from N2's CNS follows the reverse path whereas *ack (A)* packet traverses the path same as *data (D)*.

We instrument the host network stack to capture five types of timestamps that are sufficient to identify the bottleneck component:

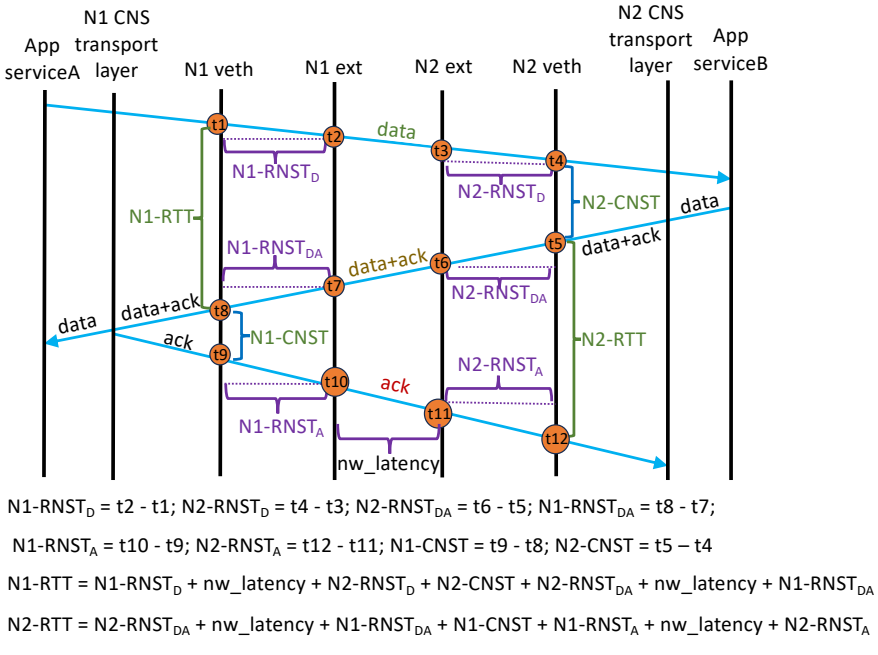
- RTT: The time between the outgoing data packet and corresponding incoming acknowledgment observed at the pod's veth interface. (*e.g.*, N1-RTT and N2-RTT in Fig. 4a).
- RNST_D: The time spent in RNS by *data* packet (*e.g.*, N1-RNST_D in Fig. 4a).
- RNST_DA: The time spent in RNS by *data+ack* packets (*e.g.*, N1-RNST_DA in Fig. 4a).
- RNST_A: The time spent by the *ack* packet in RNS (*e.g.*, N1-RNST_A in Fig. 4a).
- CNST: The time elapsed between an incoming data packet and corresponding outgoing acknowledgement observed at the veth interface (*e.g.*, N1-CNST and N2-CNST in Fig. 4a).

Cause affect analysis. As shown in Fig. 4b, an abnormal delay at any of the above monitored times affects the RTT observed at N1 and N2. For example, high N1-RTT could be due to an increase in one or more N1-RNST_D, N2-RNST_D, N2-CNST, N2-RNST_DA, N1-RNST_DA. The causality graph depicting the cause-and-effect relationship between the above-mentioned monitored times and the RTT observed at N1 and N2 is also shown in Fig. 4b. Since the per-component delay information collected at one node is not locally visible to the other node, the information must be either shared between the nodes or reported to a central entity. We adopt the latter approach, using a central controller (CC) because of its ease of managing and abstracting the underlying primitives. The CC takes care of identifying the bottleneck entity and the component (details in §6)

To differentiate between the time spent in RNS by an outgoing data packet (D) and an incoming data packet (DA), we separate the RNST into RNST1 (sum of RNST_D and RNST_DA) and RNST2 (sum of RNST_DA and RNST_A). To summarise, PerfMon collects and monitors RTT, RNST1, RNST2, and CNST on each end-host node. In this work, we do not monitor the time spent in NIS explicitly; instead, we treat NIS as part of the underlying network. Using smartNICs for capturing NIC timestamps is a promising future research direction.

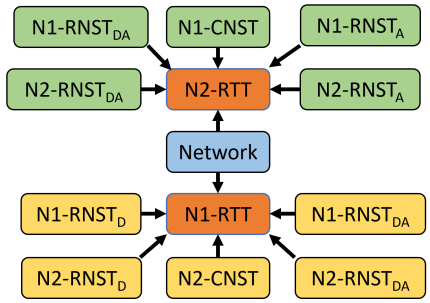
4.2 Capturing latency anomalies

Latency anomaly detection. Continuously collecting and maintaining all per-packet timestamps at each component helps to



(a) RPC performance monitoring

Entity	Component with abnormal delay	Affected RTT
N1	N1-RNST$_D$	N1-RTT
	N1-RNST$_{DA}$	N1-RTT, N2-RTT
	N1-RNST$_A$	N2-RTT
	N1-CNST	N2-RTT
N2	N2-RNST$_D$	N1-RTT
	N2-RNST$_{DA}$	N2-RTT, N1-RTT
	N2-RNST$_A$	N2-RTT
	N2-CNST	N1-RTT



(b) Cause and affect

Figure 4: Abnormal delays at different components inflate RTT observed at different entities

achieve good accuracy but has huge computation and storage overheads (**C2** in §1). To keep the overheads low, we design PerfMon's DPA to monitor per-component packet processing time and detect latency anomalies. This will significantly reduce the collection and storage overheads without compromising detection accuracy. Specifically, we design the DPA to monitor and detect deviations in per-component processing times. Upon detecting an anomaly, the DPA captures the component details and the time of occurrence of the bottleneck event so that the SLA violations observed at the RPC level can be reasoned by correlating with the problematic host-level component.

To realize our light-weight approach, we design our detection algorithm with the following two key features. First, we maintain the time spent values at the connection level (*i.e.*, flow level) instead of at the packet level so that all packets of a connection share the state. Also, we observed that multiple RPCs from one service to another share the same connection that stays for longer durations. This means the state per service instance (say A) is proportional to the number of instances of the other service (say B) it is talking to. For each connection, we maintain f-RTT, f-RNST1, f-CNST, and f-RNST2 as a moving average of time spent by individual packets, *i.e.*, RTT, RNST1, CNST, and RNST2, respectively.

Second, to keep the detection algorithm's computation light-weight and amenable to implementation at an eBPF hook point, we do a threshold-based check on per-component processing times. The main challenge is concerned with deriving the right threshold; a threshold value that is too low can lead to too many false alerts and associated overheads, whereas a threshold value that is too high may fail to capture latency anomalies (false negatives). An ideal threshold should strike a balance. We carefully derive the threshold in two steps. First, we derive a baseline processing time (B) for each component by profiling the application under various workloads. Next, we derive the threshold (T) higher than the baseline by Δ. The design choices and their importance are as follows.

(1) Deriving baseline. For a cloud-native application managed by a Kubernetes-like cluster, deriving a baseline for the expected processing time of each host network stack component is challenging. This is because the application comprises multiple services, where each service has multiple instances deployed across nodes in the cluster. Some RPCs are made across nodes (inter-node), and some within the same node (intra-node). To address this, we propose a topology-based categorization approach. We classify the RPCs into two categories: inter-node and intra-node, and derive separate baselines (and corresponding thresholds) for each component within these categories. This approach is based on the observation that the layer-4 RTT is primarily influenced by delays in the network path comprising both the host network stack and the underlying network, and is relatively unaffected by application-layer processing time. To be concrete, inspired by a recent work [47], we profile the application at 40% workload (representing typical workload condition) and find the P99 values for RTT, RNST1, CNST, and RNST2 for all RPCs. Then, we categorize the RPCs into two buckets (inter-node and intra-node), and the average of each bucket is considered the baseline value.

(2) Deriving threshold (T). In general, a threshold T is derived from the baseline B [29] following the equation, $T = B + \Delta$, where Δ is chosen based on the requirements specific to anomaly detection. Since our goal is to detect RPC latency anomalies in the order of tens of milliseconds (50ms-100ms) that potentially cause SLA violations, we adopt a percentile-based thresholding approach [47]. Specifically,

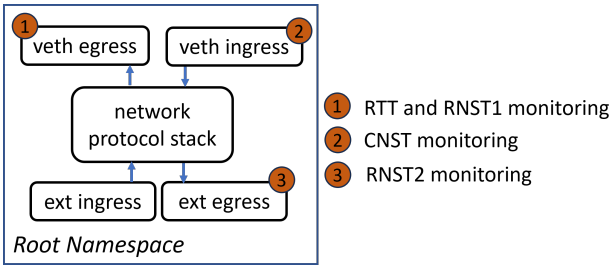


Figure 5: f-RTT, f-RNST1, f-CNST and f-RNST2 are monitored at three different hook points

we define Δ as a multiple of the baseline value calculated as the average of P99 values in the previous step. We believe that this strategy is well-suited for identifying the class of latency anomalies we target (in the order of 50-100ms). To derive the threshold for each component timestamp (RTT, RNST1, CNST, and RNST2), one naive approach is to scale all the baseline values by a constant factor X. However, since CNST and RNST values are relatively much smaller than the RTT, this approach can lead to generating many false alerts for CNST and RNST without any RTT alerts, leading to too many false positives. As an alternative, we adopt to scale component thresholds proportionally based on their contribution to RTT (Eq. (1)). This ensures CNST and RNST anomalies are flagged only when their contribution to the overall RTT is significant, that is, when they are likely to affect the RPC latency, thus keeping the false positive rate low. In our prototype, we empirically select the scaling factor X that minimizes the number of alerts (see Fig. 12b) at 40% application load [30]. However, operators can also tune X based on the application's latency tolerance, such as acceptable delay magnitude and duration.

$$\begin{aligned} T_{\text{RTT}} &= X * RTT \\ k = \lfloor RTT/RNST1 \rfloor \quad and \quad T_{\text{RNST1}} &= X * k * RNST1 \\ l = \lfloor RTT/CNST \rfloor \quad and \quad T_{\text{CNST}} &= X * l * CNST \\ m = \lfloor RTT/RNST2 \rfloor \quad and \quad T_{\text{RNST2}} &= X * m * RNST2 \end{aligned} \tag{1}$$

Handling change in threshold. The preset threshold used in PerfMon depends on the underlying hardware, operating system, and the average workload; thus, it can change if any of these factors vary. For instance, if the baseline average application workload changes or if the operator upgrades the underlying hardware/OS/software, leading to a change in earlier agreed SLA on request latency, then PerfMon requires reprofiling and updating the threshold. The operator can plan to reprofile either during the planned maintenance window or during non-peak hours, minimising the impact on application performance. Further, DPA can be configured to use the newly updated threshold.

4.3 Detection algorithm and Implementation

eBPF hookpoints. PerfMon attaches eBPF programs for monitoring and anomaly detection at two hook points in the host: 1) microservice's virtual ethernet (veth) interface ingress and egress, and 2) ext's ingress and egress, where ext is the interface that connects the host to other cluster nodes. As shown in Fig. 5, f-RTT and f-RNST1 are monitored at veth egress, f-CNST at veth ingress, and f-RNST2 at ext egress.

Algorithm 1 DPA pseudo code for monitoring and detecting abnormal packet-processing delay events

Input:
  f, RTT, RNST1, CNST, RNST2
  f = (srcIP, srcPort, dstIP, dstPort)
  α - smoothening factor, varies from (0,1)
Output:
  alert for the monitored data
  $\text{alert}_{\text{RTT}}$ = (f, 1, RTT, seqNo, ackNo)
  $\text{alert}_{\text{RNST1}}$ = (f, 2, RNST1, seqNo, ackNo)
  $\text{alert}_{\text{CNST}}$ = (f, 3, CNST, seqNo, ackNo)
  $\text{alert}_{\text{RNST2}}$ = (f, 4, RNST2, seqNo, ackNo)

begin:
(1) At veth egress:
for each data+ack packet for flow f **do**
  f-RTT = α * f-RTT + (1 − α) * RTT
  f-RNST1 = α * f-RNST1 + (1 − α) * RNST1
  if f-RTT > T_{RTT} **then**
    send_event ($\text{alert}_{\text{RTT}}$)
  end if
  if f-RNST1 > T_{RNST} **then**
    send_event ($\text{alert}_{\text{RNST1}}$)
  end if
end for

(2) At veth ingress:
for each ack packet for flow f **do**
  f-CNST = α * f-CNST + (1 − α) * CNST
  if f-CNST > T_{CNST} **then**
    send_event ($\text{alert}_{\text{CNST}}$)
  end if
end for

(3) At ext egress:
for each ack packet for flow f **do**
  f-RNST2 = α * f-RNST2 + (1 − α) * RNST2
  if f-RNST2 > T_{RNST} **then**
    send_event ($\text{alert}_{\text{RNST2}}$)
  end if
end for
end

Detection algorithm. Algorithm 1 shows the pseudo-code of PerfMon's detection of latency anomalies at node N1. Upon exceeding the pre-defined threshold (as discussed in §4.2), an alert comprising five fields: 4-tuple flow-id (f), alertType (1 to 4), abnormal packet processing time (one of RTT, RNST1, RNST2, CNST), sequence number (seqNo), and acknowledgment number (ackNo), is sent to the CPA.

5 Control plane agent (CPA)

A burst of packets can arrive at a component during the bottleneck period, where per-component processing time for these packets exceeds the respective threshold. It is crucial to not to overwhelm

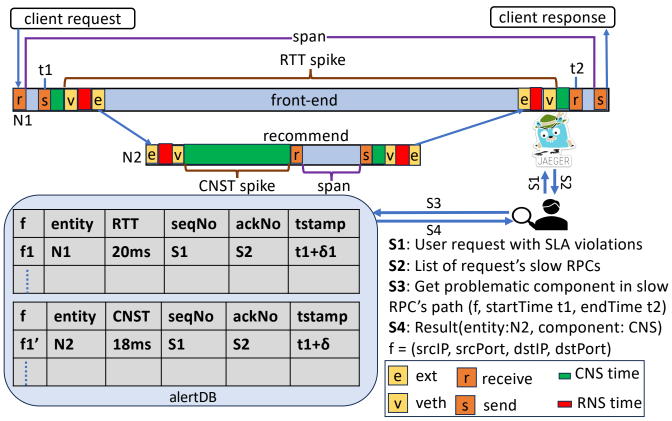


Figure 6: Identifying problematic component

the CPA with too many alerts. At the same time, the alerts due to a bottleneck event should be differentiated from random alerts due to noise (**C3** mentioned in §1). We address this by introducing a temporary buffer at the CPA, which buffers alerts from the DPA for a short time. More specifically, the CPA starts a timer on receiving an alert, if the timer is not running. If the number of alerts in a window exceeds a pre-defined threshold, say T, the alerts are sent to the controller; otherwise, they are ignored. T is inversely proportional to the smoothing factor α used in per-connection processing time (*i.e.*, RTT, RNST1, CNST and RNST2) computation in the DPA (see Algorithm1). That is, for higher α values (*e.g.*, $0.6 \leq \alpha \leq 1$), as the average smooths out without immediately reflecting on RTT/RNST1/CNST/RNST2 violations, there are fewer random alerts. On the other hand, for lower α values (*e.g.*, $0 \leq \alpha \leq 0.4$), more random alerts are sent to the CPA. Fig. 7 depicts this relationship between α, T, and the alerts generated. Based on the α value, T is configured such that the classification of random and relevant alerts is accurate. We set T to 10 as suggested in [27].

6 Central controller (CC)

This section explains how the CC identifies the bottleneck component causing increased RPC completion time (*i.e.*, slow RPC).

The CC receives alerts from CPA agents running on each cluster node and adds the alerts to the alertDB that are later queried for bottleneck identification. As shown in Fig. 6, on observing high delays for end-user requests (S1), the operator retrieves the associated application trace (like Jaeger) and the details of slow RPC (S2). The operator then queries the alertDB and retrieves alerts overlapping with the slow RPC period (S3). CC then executes its bottleneck identification algorithm (discussed below) on the retrieved alerts to identify the bottleneck component responsible for slow RPC (S4).

Bottleneck identification. To localize the source of an RPC latency, CC retrieves the potential bottleneck components (bottleneck candidates) and timestamps from the alertDB. CC identifies the bottleneck component from the bottleneck candidates using a candidate selection algorithm based on the relative importance of each component [41] in affecting the RTT. Per-component relative importance is computed based on each component's contribution towards RTT deviation (from the threshold). The component with the maximum contribution is given the highest importance and identified as the candidate bottleneck component. More specifically, consider $C_1, C_2, ..., C_n$ as the candidate bottleneck components and $t_1, t_2, ..., t_n$ as the time spent at the respective components. We represent RTT as the sum of time spent at each component (Eq. 2).

$$RTT = \sum t_i \tag{2}$$

We then determine the RTT deviation (D_{RTT}) and per-component deviation (D_i) as the deviation of the observed value from the threshold value (T_i)(Eq. 3).

$$\begin{aligned} D_{RTT} &= RTT - T_{RTT} \\ D_i &= t_i - T_i \end{aligned} \tag{3}$$

Per-component relative importance, X_i is computed based on the contribution towards D_{RTT} (Eq. 4).

$$X_i = D_i / D_{RTT} \tag{4}$$

The component that contributes the maximum towards RTT deviation is identified as the candidate bottleneck component (Eq. 5).

$$\begin{aligned} &\text{if } X_i = \max(X_1, X_2, ..., X_n) \text{ then,} \\ &bottleneck\_component = C_i \end{aligned} \tag{5}$$

Example. Consider N1-RTT violation caused by a bottleneck in N2-CNST and N1-RNST1, resulting in an increase in end-user request delay. CC identifies N2-CNST and N1-RNST1 as the candidate bottleneck components.

$$\begin{aligned} C &= \{N2\text{-}CNST, N1\text{-}RNST1\} \\ RTT &= t_{N2\text{-CNST}} + t_{N1\text{-RNST1}} \end{aligned} \tag{6}$$

With an RTT deviation (D_{RTT}) of 50ms and per-component deviation of 35ms ($D_{N2\text{-CNST}}$) and 15ms ($D_{N1\text{-RNST1}}$), CC computes the relative importance and candidate selection as below.

$$\begin{aligned} X_{N2\text{-CNST}} &= 35/50 = 0.7 \\ X_{N1\text{-RNST1}} &= 15/50 = 0.3 \\ X &= max(0.7, 0.3) = 0.7 \\ bottleneck\_component &= N2 - CNST \end{aligned} \tag{7}$$

7 Identifying bottleneck component using PerfMon

This section explains how PerfMon's bottleneck component identification helps to diagnose the root cause of RPC latency anomalies observed for an application deployed in a Kubernetes cluster. We deploy *hotelReservation* application from DeathStarBench, an open source benchmark suite [14, 36], on a Kubernetes cluster with three bare-metal servers (master, worker1, and worker2). We send user requests to hotel recommendations using a workload generator from the benchmark suite [14]. Each user request passes through three services: *frontend*, *recommendation*, and *profile*. Using stress-ng [23] tool, we inject a fault that throttles CPU for *recommendation* container for every 30 seconds. When a user request overlaps with the CPU throttling period, the completion time of *GetRecommendation* RPC from the frontend shoots up to ≈100ms, as shown in Fig. 8.

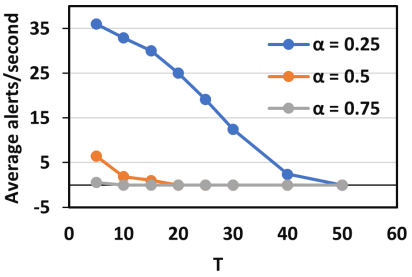


Figure 7: Alerts rate for different α and T values

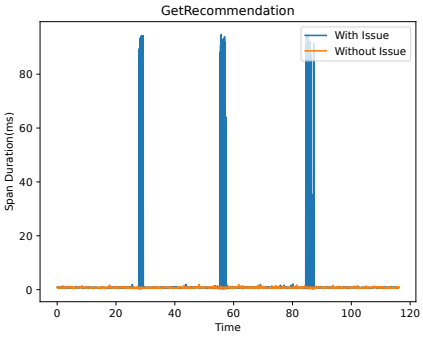


Figure 8: GetRecommendation span shows latency anomalies

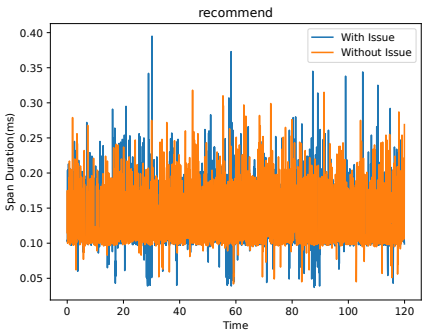


Figure 9: Recommend span does not capture abnormal delays by CNS component

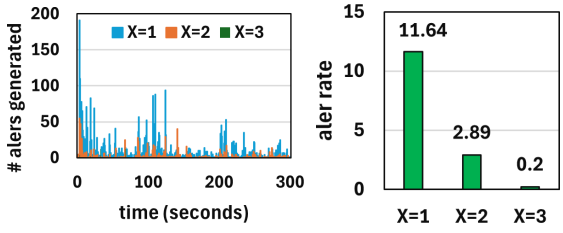
The span of *recommend* microservice, as shown in Fig. 9, does not detect any deviation, which indicates the delays are not from the application level. Next, we query PerfMon's central controller (§6) to find possible bottlenecks in the host stack during the slow RPC period. It responds with CNS as the bottleneck component and worker node 2 as the entity. This way PerfMon helps to quickly narrow down and look at the potential issues at a specific component which saves a lot of time. To find the cause for the bottleneck, the operator can check for the possibility of socket buffer build-up by monitoring the socket's send and receive queue using *lsof -T q* [6]) command from the pod's namespace. The most popular root causes for socket buffer build-up are a sudden increase in requests (incast) or CPU contention in the pod (Table 1). Finally, the cause can be determined by collecting and analyzing low-level logs, such as the pod's interface statistics for the incast case and perf record [8] for the CPU contention case.

8 Evaluation

We evaluate PerfMon against the following questions:

- Performance overhead: What is the impact of PerfMon on application's request completion time (RCT) and requests completed per second (RCS)?
- Scalability: What is the impact of PerfMon always-on monitoring on throughput at high packet rate?
- Effectiveness: Does PerfMon detect abnormal transient sporadic delay events and capture alerts relevant for identifying the problematic component?

Implementation. PerfMon comprises three modules: (1) Data plane module, (2) Control plane module, and (3) Controller module. The data plane module comprises four eBPF programs attached to one of the TC hook points (shown in Fig. 5) that track the timestamps as discussed in Fig. 4a. We enable receive flow steering (RFS)[20] to steer packets to the CPU on which the receiving application is running and use per-CPU LRU (least recently used) hash maps to store the monitored timestamps. We implement the eBPF program on Linux kernel version 5.15.0-86-generic using ≈1200 LOC. The control plane module is implemented in the C language (GLIBC 2.35) using ≈850 LOC. The central controller is implemented in Python3.10. We implement a UDP server at the controller to receive flow reports from the control plane agent. The central database is MongoDB (version 7.0.2), containing four tables. Each table stores the RTT, RNST1, CNST, and RNST2 violation reports received from the control plane agent. To compare PerfMon with TCP-stats-based periodic monitoring, we also implement TCP-stats-based periodic monitoring in python3.7 comprising 150 LOC. The program periodically monitors the per-flow RTT from each service pod using Linux *ss*[3] command.



(a) Alerts generated **(b) Average alert rate**

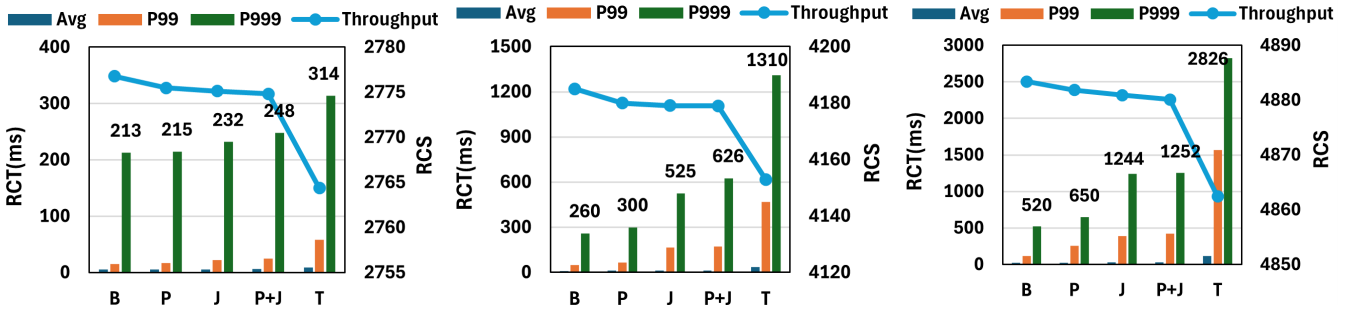
Figure 10: Threshold scaling factor X=3 has the least alert rate.

8.1 Impact on application performance

Objective. To understand the PerfMon's impact on application performance and its compute and memory overheads.

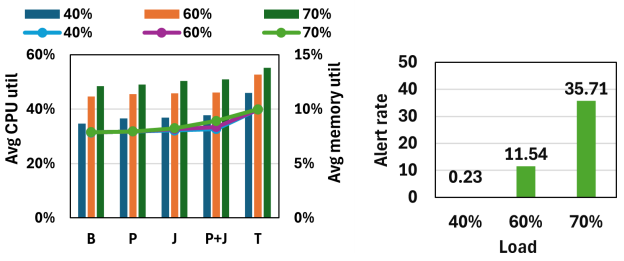
Experiment setup. We evaluate PerfMon in a Kubernetes cluster with three bare metal servers, one as a master and the other two as workers. The servers are connected to a top-of-rack switch via a 1 Gbps link. Each server has AMD EPYC7262 8-core 3.20GHz CPU and 64 GB RAM. We deployed the *hotelReservation* application from DeathStarBench open source benchmark suite [14, 36]. Flannel [4] is used as the CNI. PerfMon's DPA and CPA are deployed on the worker nodes, and PerfMon's controller and database are deployed on the master node. Workloads are generated using wrk2 [24], an HTTP benchmarking tool.

Deriving threshold. To find the threshold value for our setup, we profile the *hotelReservation* application. We collect per-packet RTT for each RPC flow and derive the inter-node and intra-node baseline as the average P99 values of all the inter-node and intra-node RPC flows. As discussed in §4.2, we use the contribution-based scaling approach at 40% application load (as suggested in [30]) and find the scaling factor X that has alert rate ≤ 1. Fig. 10 shows the



(a) RCT and RCS at 40% load (b) RCT and RCS at 60% load (c) RCT and RCS at 70% load

Figure 11: Average, P99, and P999 application RCT and average RCS for different loads.



(a) CPU and memory utilization (b) Alert rate from DPA

Figure 12: For varying application load, (a) utilization and (b) alert rate

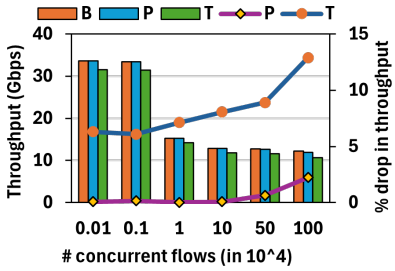


Figure 13: Impact of PerfMon always-on monitoring on application throughput with increasing concurrent connections

alerts generated by DPA and the alert rate for different X values. Based on the results, we set X to 3 as it has the lowest alert rate and derive the threshold as described in §4.2.

Performance metrics and experiment. We study the impact of PerfMon on application performance in terms of: (1) the number of application requests completed per second (RCS), (2) request completion time (RCT), (3) memory utilization, and (4) CPU utilization. We vary application workload (40%, 60%, and 70%) and conduct experiments for five different systems: (1) Benchmark application without any monitoring (B); (2) Benchmark with PerfMon (P); (3) Benchmark with only application-level tracing using Jaeger (J); (4) Benchmark with PerfMon and Jaeger (P+J). and (5) Benchmark with TCP-stats-based monitoring (T).

Results. Fig. 11 shows the RCT and RCS of all five systems at different workloads. Compared to the benchmark (B), we observe that PerfMon (P) overhead is minimal as it slightly increases the P999 latency by 0.9% for 40% load. This increase in latency is attributed to PerfMon's monitoring and processing overheads. However, as we increase the workload, P999 latency increases by 15%, and 25% for 60%, and 70% workloads, respectively. The increase is higher at 60% and 70% load because as the load increases, more alerts are generated by PerfMon's DPA, which is evident from Fig. 12b. At high load, packets are queued at the application socket buffers, exceeding CNST and RTT thresholds, followed by alerts. On the other hand, T increases both P99 and P999 latency which is reflected also as a drop in RCS. When compared with B, T increases P999 latency 46%, 400%, and 440% respectively, for 40, 60, and 70% workload.

From Fig. 12a, we observe that average CPU utilization increases as the load increases for all the systems. It is worth noting that the CPU utilization remains almost the same for B and P across different loads. However, TCP-stats-based monitoring increases the CPU utilization by 32% for 40% load. On the other hand, B and P have similar average memory utilization across workloads; P increase the memory utilization by a meagre 1% accounting to the eBPF maps for storing monitored component timestamps. Whereas J, P+J, and T requires more memory. J and P+J increases memory utilization by ≈ 4% due to Jaeger's memory overheads for trace collection and T increases the memory utilization by 26%.

8.2 Scalability

Objective. To study the impact of PerfMon DPA (always-on per-packet monitoring) at high packet rates (40Gbps) and with many concurrent flows.

Experimental setup. The experimental setup comprises a Kubernetes cluster with two nodes. Each node is a bare metal server equipped with two Intel(R) Xeon(R) Silver 4316 20-core 2.30GHz CPU with 256 GB RAM and a 40 Gbps Netronome Agilo SmartNIC [10]. These nodes are directly connected using QSFP40Gb Ethernet copper cable. We use *ntttcp*[7] to generate many concurrent flows (1 million concurrent flows). We deployed *ntttcp* client on the master node to generate flows. *ntttcp* server with PerfMon monitoring deployed on the worker node acts as the DUT (device under test).

Performance metrics and experiment. We study the impact of PerfMon's always-on per-packet monitoring using application throughput as the performance metric. We perform the experiment

by varying the number of concurrent connections using *ntttcp*. Our experiment comprises 5 iterations. Each iteration consists of 5 runs, each lasting 120 seconds.

Results. Fig. 13 shows the application throughput achieved without any monitoring (benchmark B), with PerfMon monitoring (P), and with always-on TCP-stats monitoring (T). The secondary Y-axis shows the drop in throughput compared with the baseline case. We can observe that PerfMon causes minimal impact on application throughput with increasing concurrent connections. With one million concurrent connections, PerfMon monitoring reduces the application throughput by ≈3%. This drop is negligible for an always-on per-packet monitoring. On the other hand, T reduces the throughput by ≈13% adversly affecting application performance.

8.3 Effectiveness

Objective. To study PerfMon's effectiveness in detecting transient and sporadic events causing bottlenecks in the host.

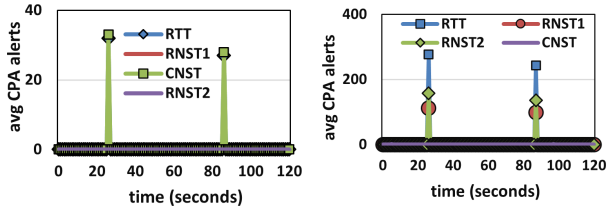
Setup and metrics. The experiment setup is the same as the one described previously in 8.1. We evaluate PerfMon's effectiveness in terms of its ability to (1) detect bottleneck components, (2) minimize random alerts, and (3) detect transient and sporadic bottleneck events. We use *tc-netem* [1] to inject processing delays in CNS (pod interface *eth0*) and RNS (Linux bridge *cni0*). We consider FPR (false positive rate), FNR (false negative rate), and accuracy as the metrics. From the literature and blogs [9, 15, 39], we understand that transient and sporadic bottleneck events can cause processing delays varying from 10ms to 100ms and can last for 30ms to 50ms. Hence, we inject 10ms processing delays (fault) randomly at CNS and RNS, each lasting for 40 ms in our experiment run for 120 seconds.

8.3.1 Detecting bottleneck component. Fig. 14 shows the DPA and CPA alerts generated by PerfMon with synthetic fault injection at CNS and RNS. As shown in Fig. 14, we inject processing delays (fault) twice during the 120-second run, and PerfMon detects and captures both the delay events. CNS fault in the pod inflates the RTT and CNST of ingress RPC request packets destined for the pod. Hence the number of CNST and RTT alerts are almost equal (Fig. 14a). However, with the RNS fault in the node, the number of alerts is higher (Fig. 14b) as the RNS fault affects multiple RPCs. Also, the number of RTT alerts is almost double that of RNST alerts because RNS fault inflates the RTT of both RPC requests and responses (packets exchanged in both directions). That is, the RNS fault inflates RNST1 of RPC response packets (egress packets from the node) and RNST2 of RPC request packets (ingress packets to the node), thus inflating RTT of both request and response.

8.3.2 Minimize random alerts. Table 2 shows the FPR and FNR of DPA alerts (alerts with noise) and CPA alerts (alerts after noise reduction). CPA reduces the FPR by 80% and 73% for CNS and RNS faults, respectively. This shows that CPA successfully ignores random alerts (noise) from relevant alerts, which minimizes false positives and reduces PerfMon's compute and storage overhead (which will otherwise be spent on false positives). As PerfMon detects both the injected faults, FNR becomes zero for CNS and RNS. Also, the average alert rate shows the number of packets affected by the fault; it is higher for RNS fault because RNS fault affects multiple RPCs.

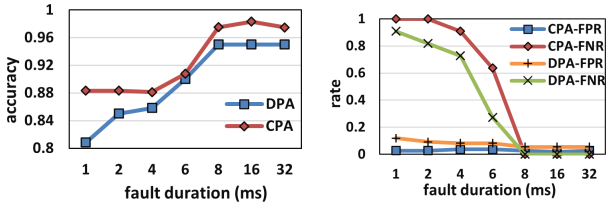
Fault	DPA		CPA		Avg. alert rate
	FPR	FNR	FPR	FNR	
CNS	0.071	0	0.014	0	1.36
RNS	0.056	0	0.015	0	6.87

Table 2: CPA reduces FPR by 80% for CNS faults and by 73% for RNS faults.



(a) CNS fault **(b) RNS fault**

Figure 14: PerfMon's DPA detects bottleneck component and generates alerts

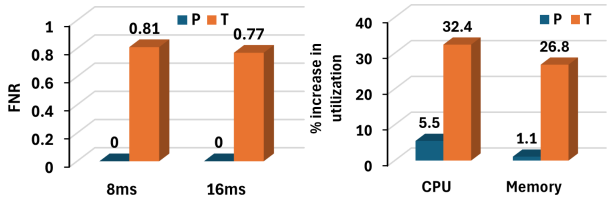


(a) Accuracy **(b) FPR and FNR**

Figure 15: PerfMon's detection accuracy is high for bottleneck events lasting 8ms or more.

8.3.3 Detect transient and sporadic bottleneck events. To understand PerfMon's ability to detect transient and sporadic bottleneck events, we inject processing delays (faults) at CNS randomly, causing a 10ms delay while varying the duration of the fault from 1ms to 32ms. Fig. 15 shows the detection accuracy, FPR, and FNR observed at CPA and DPA. We observe that PerfMon's detection accuracy increases with fault duration (Fig. 15a). With faults lasting 8ms or more, PerfMon has high accuracy with FNR close to zero (Fig. 15b). But for shorter fault durations (1ms and 2ms), PerfMon's accuracy is low because the fault affects only a few packets, which gets smoothened without inflating the EWMA for larger α values (as discussed in §5). Consequently, the DPA either fails to generate or generates fewer alerts, leading to higher DPA-FNR. Further, CPA's noise reduction identifies these alerts as noise, leaving CPA-FNR at 1. As the fault duration increases, PerfMon's FNR decreases. Specifically, PerfMon FNR falls to zero for transient faults lasting as low as 8ms.

8.3.4 Comparison with TCP-stats-based monitoring. We compare PerfMon's effectiveness in detecting transient and sporadic faults with that of TCP-stats-based always-on monitoring. We inject a CNST delay of 10ms and set the fault duration to 8 and 16 milliseconds and compare the effectiveness using FNR as the metric. In the previous experiment, we already observed that PerfMon



(a) FNR (b) CPU and memory utilization

Figure 16: Comparison between PerfMon (P) and TCP-stats based monitoring (T)

successfully captures all the faults when the fault lasts for 8ms or more (Fig. 16a). However, on the contrary, with TCP-stats-based monitoring, we observe that when the delay lasts 8ms and 16ms, the FNR is as high as 81% and 79%. This is because TCP-stats-based monitoring does not monitor per-packet latencies and hence is not fine-grained enough to capture transient sporadic faults. Additionally, when compared with PerfMon, always-on TCP-stats-based monitoring requires 6x and 24x more CPU and memory (Fig. 16b).

9 Related Work

Application-level tracing. Application-level tracing tools [5, 13, 19, 35, 43, 45, 50, 52] provide the datapath (microservices) that a request has traversed and the amount of time spent at each microservice. However, they do not provide visibility below the application layer. That is, they can tell which RPC in a given user request is experiencing delays but cannot tell which underlying entity and associated component is causing those delays.

Host-level tracing. End-host tracing tools [11, 12, 16, 22, 33, 39] provide deep visibility into host-level events. For instance, NSight [39] captures the end host system state using a hardware profiling tool called intel_pt [25] and identifies the root cause of delays. Though such tools can identify the problematic component, they suit well for after-the-fact root cause analysis because of their high storage overheads. We observe that running intel_pt for 30 seconds and 60 seconds requires around 440MB and 1GB of storage space, respectively. We envision such tools complement our approach – after identifying the problematic component, component-level traces provided by these tool helps to find the root cause.

NIC-level monitoring. BuffScope[37] is a buffer-based request event monitoring system that provides end-to-end visibility across different entities and also at the end-host stack level. Visibility at the end-host stack is achieved by monitoring various buffer events, and end-to-end visibility is achieved by RPC packet instrumentation, leveraging the capabilities of SmartNICs. Buffscope assumes that the data centre's east-west traffic is unencrypted, which makes packet-level instrumentation less practical to deploy.

Network monitoring. Network monitoring tools [38, 40, 42, 44, 48, 51] provide details of flows subject to congestion at a switch in a data center network space. Since the RPC information from an application trace contains flow information in the cluster address space, this information is insufficient for correlating flow information from different locations, hindering end-to-end visibility.

Host network stack-level monitoring. Host-INT [49], an eBPF-based system, extends INT[18] support to the end-host networking stack. This provides end-to-end visibility as packets traverse switches, end hosts, and application containers. Host-INT collects records for every packet at every host and exports records to a central database. Continuous per-packet record collection, processing, and storage have huge performance and storage overheads.

10 Discussions

Impact of workload variation on preset threshold and alerts. In deployments using orchestrators like Kubernetes, pod autoscaling ensures a predictable maximum workload at each pod as the workload varies. Based on the predictions, it is feasible to derive a preset threshold to detect anomalies. In such cases, higher application load does not lead to latency anomalies; therefore, do not generate alerts. Hence, variation in workload does not require any change in the preset threshold. However, if an increased workload still results in latency anomalies, it indicates an underlying issue. In such scenarios, PerfMon correctly detects the anomaly and generates alerts, which is the intended behavior. PerfMon use 40% load as the baseline workload for deriving the threshold. However, if the operator wishes to change the baseline workload (due to application feature change/optimization/configuration and infrastructure change etc.), then PerfMon requires reprofiling the application at the new baseline load and updating the threshold values.

Impact of software and hardware upgrades on PerfMon threshold. In the cloud, operators often perform periodic updates and maintenance of the operating system, middleware, and the underlying hardware infrastructure. These tasks are often transparent to the tenants and performed without impacting the application SLA. Though an OS upgrade or a hardware upgrade can improve the baseline RNST, CNST, and RTT values, PerfMon need not recompute the threshold because any anomaly at the RPC level that potentially causes an SLA violation is still detected by the preset threshold. However, if the provider and the tenant decide to update the SLA following the upgrade, then PerfMon requires reprofiling and updating the threshold. The provider can plan to reprofile either during the planned maintenance window or during non-peak hours to minimize the impact on the application performance.

11 Conclusion

We present PerfMon, a lightweight performance monitoring system that enables the automatic identification of bottleneck components in the host network stack. PerfMon complements the existing application-level monitoring system and helps to quickly assign the issue to the right team in operations. We developed PerfMon using eBPF technology and tested it by deploying it on a Kubernetes-managed cluster of servers. PerfMon's monitoring overheads are low and detect the bottleneck component with high accuracy.

12 Acknowledgements

We thank the anonymous reviewers for their thoughtful feedback. We thank Anirudh Sivaraman for his valuable input on the earlier draft that helped improve the paper substantially. We also thank Prashanth P S, K Shiv Kumar, and Harish S A for their participation in the discussions and feedback. This work is supported by NM-ICPS TiHAN, PMRF, National Security Council Secretariat (NSCS), ZF Technologies, and Marvell Technologies.

References

[1] 2011. *tc-netem(8) — Linux manual page* . Retrieved "December 2023" from https://man7.org/linux/man-pages/man8/tc-netem.8.html

[2] 2013. *Queueing in the Linux Network Stack* . Retrieved "October 2024" from https://www.coverfire.com/articles/queueing-in-the-linux-network-stack/

[3] 2013. *ss(8) — Linux manual page*. Retrieved "October 2024" from https://man7.org/linux/man-pages/man8/ss.8.html

[4] 2014. *flannel*. https://github.com/flannel-io/flannel

[5] 2014. *Zipkin*. Retrieved "December 2023" from https://zipkin.io/

[6] 2015. *lsof(8) — Linux manual page*. Retrieved "October 2024" from https://man7.org/linux/man-pages/man8/lsof.8.html

[7] 2015. *ntttcp-for-linux*. Retrieved "July 2025" from https://github.com/microsoft/ntttcp-for-linux

[8] 2015. *perf-record(1) — Linux manual page*. Retrieved "October 2024" from https://man7.org/linux/man-pages/man1/perf-record.1.html

[9] 2015. *The story of one latency spike*. Retrieved "December 2023" from https://blog.cloudflare.com/the-story-of-one-latency-spike/

[10] 2016. *NETRONOME*. Retrieved "December 2023" from https://www.netronome.com/products/agilio-cx/

[11] 2017. *Uprobe-tracer: Uprobe-based Event Tracing*. https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt

[12] 2017. *ftrace - Function Tracer*. https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[13] 2017. *Jaeger: open source, end-to-end distributed tracing*. https://www.jaegertracing.io/

[14] 2019. *DeathStartBench*. Retrieved "December 2023" from https://github.com/delimitrou/DeathStarBench

[15] 2019. *Debugging network stalls on Kubernetes*. https://github.blog/2019-11-21-debugging-network-stalls-on-kubernetes/

[16] 2019. *Kernel Probes (Kprobes)*. Retrieved "December 2023" from https://www.kernel.org/doc/Documentation/kprobes.txt

[17] 2020. *Dynamically program the kernel for efficient networking, observability, tracing, and security*. Retrieved "October 2025" from https://ebpf.io/

[18] 2020. *In-band Network Telemetry (INT) Dataplane Specificationl*. Retrieved "December 2023" from https://p4.org/p4-spec/docs/INT_v2_1.pdf

[19] 2020. *OpenTelemetry*. https://opentelemetry.io/

[20] 2020. *Scaling in the Linux Networking Stack¶* . Retrieved "February 2025" from https://docs.kernel.org/networking/scaling.html

[21] 2020. *Want to Debug Latency?* Retrieved "October 2024" from https://rakyll.medium.com/want-to-debug-latency-7aa48ecbe8f7

[22] 2022. *perf - Performance analysis tools for Linux*. https://man7.org/linux/man-pages/man1/perf.1.html

[23] 2022. *stress-ng - a tool to load and stress a computer system*. Retrieved "December 2023" from https://manpages.ubuntu.com/manpages/jammy/man1/stress-ng.1.html

[24] 2022. *wrk2*. Retrieved "December 2023" from https://github.com/delimitrou/DeathStarBench/tree/master/wrk2

[25] 2023. *perf-intel-pt(1) — Linux manual page*. Retrieved "December 2023" from https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html

[26] 2024. *The State of eBPF*. Retrieved "November 2024" from https://www.linuxfoundation.org/hubfs/eBPF/The_State_of_eBPF.pdf

[27] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically finding the cause of packet drops. In *USENIX NSDI*.

[28] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the blame game out of data centers operations with netpoirot. In *ACM SIGCOMM*.

[29] Hong Zhang1 Junxue Zhang1 Wei Bai and Kai Chen1 Mosharaf Chowdhury. 2017. Resilient Datacenter Load Balancing in the Wild. (2017).

[30] Jessica Berg, Fabian Ruffy, Khanh Nguyen, Nicholas Yang, Taegyun Kim, Anirudh Sivaraman, Ravi Netravali, and Srinivas Narayana. 2021. Snicket: Query-driven distributed tracing. In *ACM HotNets Workshop*.

[31] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *ACM SIGCOMM*.

[32] Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, and Jennifer Rexford. 2020. Measuring TCP round-trip time in the data plane. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*.

[33] Mathieu Desnoyers and Michel R Dagenais. 2006. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*. Citeseer.

[34] Liz Fong-Jones and Adam Mckaig. 2018. Resolving Outages Faster with Better Debugging Strategies. (2018).

[35] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. 2007. X-trace: A pervasive network tracing framework. In *USENIX NSDI*.

[36] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*.

[37] Kaihui Gao, Chen Sun, Shuai Wang, Dan Li, Yu Zhou, Hongqiang Harry Liu, Lingjun Zhu, and Ming Zhang. 2022. Buffer-based end-to-end request event monitoring in the cloud. In *USENIX NSDI)*.

[38] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*.

[39] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. 2022. How to diagnose nanosecond network latencies in rich end-host stacks. In *USENIX NSDI*.

[40] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *USENIX NSDI*.

[41] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance monitoring and root cause analysis for cloud-hosted web applications. In *Proceedings of the 26th International Conference on World Wide Web*.

[42] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. 2018. BurstRadar: Practical real-time microburst monitoring for datacenter networks. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*.

[43] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An end-to-end performance tracing and analysis system. In *ACM SOSP*.

[44] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. 2021. Debugging Transient Faults in Data Centers using Synchronized Network-wide Packet Histories.. In *USENIX NSDI*. 253–268.

[45] Haifeng Liu, Jinjun Zhang, Huasong Shan, Min Li, Yuan Chen, Xiaofeng He, and Xiaowei Li. 2019. Jcallgraph: tracing microservices in very large scale container cloud platforms. In *Cloud Computing–CLOUD 2019: 12th International Conference, Held as Part of the Services Conference Federation, SCF 2019, San Diego, CA, USA, June 25–30, 2019, Proceedings 12*.

[46] Matt Mathis, John Heffner, Peter O'neil, and Pete Siemsen. 2008. Pathdiag: automated TCP diagnosis. In *PAM*.

[47] Shicong Meng, Arun K Iyengar, Isabelle M Rouvellou, and Ling Liu. 2013. Volley: Violation likelihood based state monitoring for datacenters. In *IEEE ICDCS*.

[48] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*.

[49] Tomasz Osiński and Carmelo Cascone. 2021. Achieving end-to-end network visibility with host-int. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. 140–143.

[50] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[51] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and TS Eugene Ng. 2022. Closed-loop network performance monitoring and diagnosis with {SpiderMon}. In *USENIX NSDI*.

[52] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. Microrca: Root cause localization of performance issues in microservices. In *IEEE/IFIP NOMS*.

[53] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. 2011. Profiling network performance for multi-tier data center applications. In *USENIX NSDI*.