Understanding and Profiling NVMe-over-TCP Using ntprof

Yuyuan Kang and Ming Liu, *University of Wisconsin-Madison*

https://www.usenix.org/conference/nsdi25/presentation/kang

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the
22nd USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Understanding and Profiling NVMe-over-TCP Using `ntprof`

Yuyuan Kang and Ming Liu

University of Wisconsin-Madison

Abstract

NVMe-over-TCP (NVMe/TCP) is an emerging remote storage protocol, increasingly adopted in enterprises and clouds. It establishes a high-performance reliable data channel between clients and storage targets to deliver block I/Os. Understanding and analyzing the protocol execution details and how well storage workloads run atop are pivotal for system developers and infrastructure engineers. However, our community lacks such a profiling utility, whereas existing solutions are ad-hoc, tedious, and heuristic-driven. Realizing it is challenging due to the unpredictable I/O workload profile, intricate system layer interaction, and deep execution pipeline.

This paper presents `ntprof`, a systematic, informative, and lightweight NVMe/TCP profiler. Our key idea is to view the NVMe/TCP storage substrate as a lossless switched network and apply network monitoring techniques. We model each on-path system module as a software switch, equip it with a programmable profiling agent on the data plane, and develop a proactive query interface for statistics collection and analysis. `ntprof`, comprising a kernel module and a user-space utility, allows developers to define various profiling tasks, incurs marginal overhead when co-locating with applications, and generates performance reports based on prescribed specifications. We build `ntprof` atop Linux kernel 5.15.143 and apply it in six cases, i.e., end-to-end latency breakdown, interference analysis, SW/HW bottleneck localization, and application performance diagnostic. `ntprof` is available at https://github.com/netlab-wisconsin/ntprof.

1 Introduction

Storage disaggregation [31, 76, 77, 92, 98] has gained significant traction recently due to independent resource scaling, high utilization, and cost efficiency. It has been increasingly deployed in data centers, enterprise on-premise clusters, and edge clouds. With the rising networking speed and emerging high performance of NVMe drives, a disaggregated SSD can deliver millions of IOPS at tens of microseconds.

The key technology enabler is an NVMe-over-Fabric (NVMe-oF) protocol [43] that delivers high-performance storage accesses between clients (*initiators*) and remote NVMe subsystems (*targets*). An initiator submits block I/O requests–encapsulated into specialized fabric packets–and delivers them to the remote side through an established protocol channel. A target receives packets from the network transport, converts them to NVMe commands, reads/writes from the SSD, and sends the response back. NVMe/TCP is one such networking fabric, which is increasingly adopted and has enabled many real-world deployments [2, 3, 12, 25, 27, 28].

However, our community lacks a system profiling tool to understand and analyze the execution characteristics of the NVMe/TCP protocol. This makes answering many design, operation, testing, and deployment questions impractical. For example, how well do the storage applications run over NVMe/TCP? How should we design the I/O merging and pacing policies of the block layer? What are the system bottlenecks along the protocol pipeline? How should we reconfigure the NVMe/TCP configurations if the initiator/target processors, networks, or NVMe drives are congested?

Building such a utility is non-trivial due to the workload non-determinism, intricate system layer interaction, and deep execution pipeline. First, storage workloads embody diverse I/O profiles, including block size distribution, read-/write mix ratio, sequential/random access pattern, and I/O concurrency. This issue is further exacerbated by (a) emerging NVMe drives with high queue depth; and (b) dense storage appliances, equipped with dozens of SSDs, cumulatively offering large data volumes (10+ terabytes) and massive bandwidth (100+ GB/s). Second, the NVMe/TCP layer closely interacts with other system modules, including the block layer, TCP/IP networking stack, and NVMe subsystem. Each presents certain execution parallelism (e.g., multi-queue, multi-connection, and multi-data path) with flexible load-balancing policies (to maximize throughput and mitigate superfluous I/O stalls). Third, an I/O has a deep pipeline with processing stages spreading two physical hosts across the CPU core, networking, and NVMe drive (Figure 2).

Developers thus resort to a heuristic top-down analysis strategy to diagnose performance anomalies, dissect

overheads, and explore optimization opportunities in an NVMe/TCP-based disaggregated storage. They combine a sequence of development utilities (from the application and system layer [11, 16, 17, 23, 37, 38, 44, 45, 94] to the infrastructure level [18, 39, 51, 66, 67, 85, 86]), embarrassingly combine them based on empirical observations, and manually synthesize isolated profiling reports from different tools to infer the root causes. The entire synthesis process is tedious and inefficient, driven by a vertical view of experimental results, requiring an esoteric understanding of the system and lots of back-and-force test/verification. More importantly, it is generally challenging to reproduce online interleaved I/O scenarios, which are common in a multi-tenant environment, rendering existing approaches futile for root cause exploration.

In this paper, we design and implement `ntprof` to enable systematic tracing and analyzing of the NVMe/TCP. *Our key insight is viewing the NVMe/TCP subsystem as a lossless switched network–an initiator sends I/O requests, a target returns the responses, and I/O blocks encapsulated as TCP packets are forwarded by different system entities along the I/O path–and apply network monitoring techniques.* Essentially, we make each on-path system component a multi-queue software switch, equip it with a programmable profiling agent on the data plane, and develop a proactive query interface for data collection and analysis. As such, our design resembles prior active network-based profiling systems (like TPP [57, 58]).

`ntprof` consists of four system components. First, it provides a profiling task specification template that allows developers to prescribe targeted I/O workloads, deployment environments, profiler configurations, and reporting statistics, enriching profiling functionalities. Second, we develop a lightweight programmable profiling agent based on the Linux tracepoint mechanism, co-locate with each modeled software switch for statistics monitoring, and take translated specifications as predicates. Third, we introduce an in-band TPP-like profiling query protocol, which injects a special NVMe/TCP command capsule to query and collect statistics periodically. Last, `ntprof` produces an informative performance report (including end-to-end application I/O performance, latency breakdown, throughput bottleneck, protocol running details, etc.) either online or offline through a sequence of steps (such as serialization, calibration, and MapReduce-like processing).

`ntprof`, built atop Linux kernel 5.15.143, comprises a kernel model and a user-space utility at the NVMe/TCP initiator and target sides. We evaluate it in a typical NVMe/TCP setup in CloudLab [40] and demonstrate its effectiveness via six case studies. Specifically, `ntprof` can report the I/O latency breakdown of the deep NVMe/TCP pipeline (§4.2) and dissect the I/O interference issue under mixed workload profiles (§4.5). Next, `ntprof` enables developers to locate the system bottlenecks at both software processing pipelines (§4.3) and the underlying I/O substrate (§4.4). Further, `ntprof` can help with application performance diagnostic when deployed in an NVMe/TCP setting (§4.6 and §4.7). `ntprof` is open-source

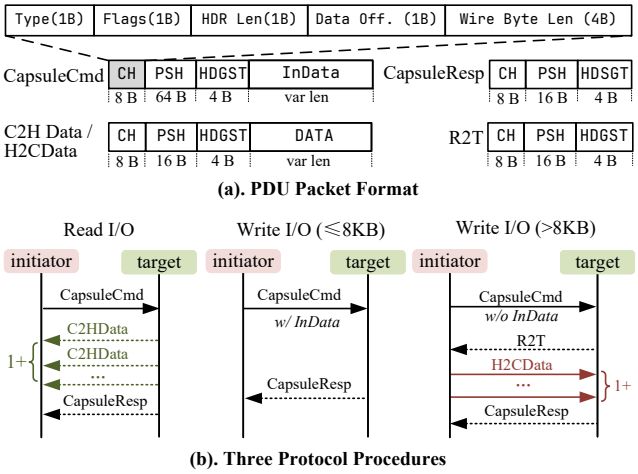


Figure 1: NVMe/TCP protocol details. (a) presents the packet format of five types of PDUs. (b) depicts the protocol procedure of an I/O read and write. InData=In-Capsule Data.

and we will keep working with the community to improve it.

2 Background and Motivation

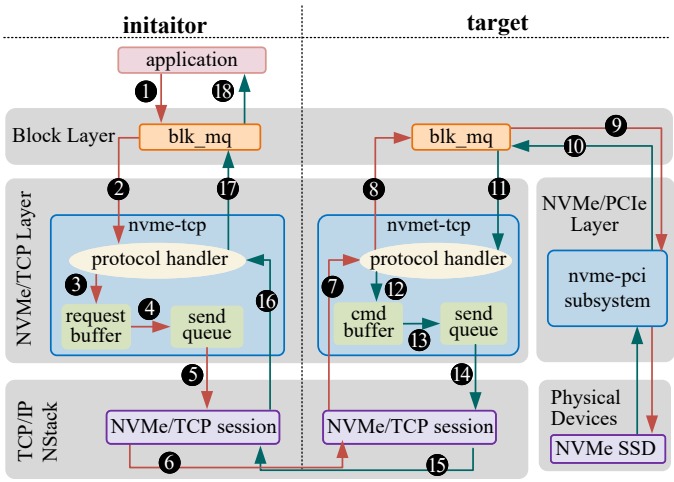
This section provides some background on the NVMe-over-TCP protocol and its Linux implementation. We then discuss the challenges and limitations of existing profiling solutions.

2.1 NVMe-over-TCP Protocol

NVMe-over-TCP (NVMe/TCP) [43] is a high-performance data transfer protocol that defines block-level storage accesses between clients (*initiators*) and remote NVMe subsystems (*targets*) via TCP/IP. An initiator submits block I/O requests–encapsulated into specialized NVMe/TCP packets–and delivers them to the remote side through an established protocol channel. A target receives packets from the network transport, converts them to NVMe commands, reads/writes from the SSD, and sends the response back. NVMe/TCP makes the remote NVMe subsystem behave similarly to a local one.

NVMe/TCP, inherited from the NVMe base protocol [41, 42], constructs reliable bidirectional communication channels (also called NVMe/TCP session) between two sides, where each comprises an initiator-side submission-completion queue pair, a TCP connection, and a target-side NVMe queue pair. The number of channels can be configured to the number of cores of an initiator's processor, which also benefits the multi-queue architecture of an NVMe drive. There is a specialized channel to deliver I/O admission commands.

Protocol Data Unit (PDU) is the basic communication granularity in NVMe/TCP that carries I/O command, data, and control status. As shown in Figure 1-a, it begins with an 8-byte Common Header (CH), followed by a 16 or 64-byte PDU Specific Header (PSH), a 4-byte header digest (HDGST) for cache alignment and integrity check, and variable length in-capsule data (optional). CH defines the PDU type, special flags (e.g., whether digest is present in the header and data), header length, data offset, and the total PDU transmission



Step	Description
❶	Submit a read/write I/O via system calls
❷	Forward the request to NVMe/TCP protocol handler
❸	Generate a CapsuleCmd PDU and insert it into the request buffer
❹	Pop the request from the request buffer and enqueue it to sending queue
❺	Copy the CapsuleCmd or H2CData PDU to the TCP socket buffer
❻	Transmit over the NVMe/TCP session
❼	Read PDUs from the TCP socket buffer
❽	Proccess the PDU, extract an NVMe command, and submit it to the blk_mq
❾	Submit the NVMe command to local NVMe subsystem
❿	Receive the I/O completion from the NVMe subsystem
⓫	Send the block layer response to the protocol handler
⓬	Produce the R2H/C2HData PDU and then insert it into the command buffer
⓭	Pop the response from the command buffer and enqueue it to sending queue
⓮	Dequeue PDUs from the sending queue and copy it to the TCP socket buffer
⓯	Reply over the NVMe/TCP session
⓰	Read a C2HData/R2T/CapsuleResp PDU from the TCP socket buffer
⓱	Perform PDU protocol processing and then produce a block I/O response
⓲	Return an I/O response to the application

Figure 2: The left side depicts the NVMe/TCP system architecture and I/O data path. The right side lists detailed processing stages.

byte. There are five types of PDUs defined in NVMe/TCP for I/O: (a) *CapsuleCmd PDU*, issued from the initiator and transmitting a command capsule that encodes a read/write request in the PSH; (b) *CapsuleResp PDU*, replied from the target and carrying a completion response capsule; (c) *H2CData PDU*(*C2HData PDU*), used for transfer I/O data from an initiator (target) to a target (initiator); (d) *R2T PDU*, a control signal from the target to notify when it is able to accept data from the initiator. The PSH header is not consistent across different PDUs. It encodes the NVMe-oF command capsule submission/completion Queue Entry in CapsuleCmd PDU/-CapsuleResp PDU, carries data in H2CData PDU/C2HData PDU, and stores the R2T PDU's metadata.

NVMe/TCP has three protocol procedures (Figure 1-b). An I/O read sends a CapsuleCmd PDU and receives one or more C2HData PDUs, ending with a CapsuleResp PDU for completion notification. A small write (less than or equal to 8KB) uses CapsuleCmd PDU with the command and data inlined, resulting in a CapsuleResp PDU response from the target, which entails only one round-trip. Large writes require one more admission control phase. First, the initiator submits the write I/O request without data. Upon receiving an R2T PDU, it sends multiple H2CData data transfer PDUs and waits for a CapsuleResp PDU as an acknowledgment. Note that all PDUs are encapsulated as TCP packets and traverse through the corresponding TCP connection.

2.2 NVMe-over-TCP in Linux

NVMe/TCP is first officially released in Linux Kernel 5.0 [1] by Lightbits Labs [26] with host and target drivers, conforming to NVMe over Fabrics 1.1 specifications [43]. It is divided into two kernel modules. One is `nvme-tcp`, staying at the initiator, which (a) instantiates and manages the NVMe/TCP session; and (b) accepts block I/O requests, translates them to NVMe/TCP PDU capsules, and delivers these TCP packets to the remote storage. The other is target-side `nvmet-tcp`, which (a) manages NVMe subsystems of the storage appliance; and (b) executes the NVMe/TCP protocol to facilitate I/O handling. This section describes their system architecture and I/O path based on Linux Kernel 5.15.143 (Figure 2).

`nvme-tcp` (Initiator). Here's its workflow on the submission path of a session. First, applications issue I/O requests via system calls and insert them into the `blk-mq` of the block layer (❶). Next, `nvme-tcp` dequeues a request (❷), traverses the protocol handling, constructs an NVMe/TCP PDU capsule (`struct nvme_tcp_request`), and pushes it into a per-session request buffer (❸). It is a LIFO (Last-In-First-Out) lockless linked list (`struct llist_head req_list`) that accepts block I/Os from all CPU cores. After that, the `nvme-tcp` kernel thread pops a ready-to-transmit request and enqueues it into a per-session LIFO sending queue (`struct list_head send_list`) (❹). Last, these ready capsules are dequeued (❺) and delivered to the corresponding NVMe/TCP session (❻) for transmission via the TCP/IP networking stack. A session is represented as a transport queue (part of `struct nvme_tcp_queue`), associated with a dedicated TCP socket handler and assigned to a specific CPU core. This is realized via either (i) the current kernel thread when the queue occupancy is low (e.g., only one outstanding I/O) or (ii) a deferred tasklet (`nvme_tcp_io_work`) that interleaves the sending and receiving PDU processing to optimize throughput.

The completion path is completely handled by the `nvme_tcp_io_work` routine after the networking stack processing. It reads data from the TCP socket buffer and constructs an NVMe/TCP capsule (⓰). A corresponding PDU handler is triggered based on the PDU type (i.e., C2HData, CapsuleResp, or R2T), which produces a block layer response and returns it to the block layer (⓱) and applications (⓲), or sends an H2CData PDU for data transmission (❸).

`nvmet-tcp` (Target). Storage traffic arrives at the NVMe/TCP session socket, stored in the `nvmet-tcp` transport queue (`struct nvmet_tcp_queue`). Akin to the `nvme-tcp` module, there is a per-session LIFO-based response command buffer (`struct llist_head resp_list`) and a per-session

LIFO-based response sending queue (`struct list_head resp_send_list`). The target workflow is as follows. First, the `nvmet-tcp` kernel thread (`nvmet_tcp_io_work`) reads a batch (up to 64) of PDUs from the TCP socket buffer and activates the PDU handler (❼). As shown in Figure 1-b, a large write without in-capsule data triggers the R2T PDU processing, produces a corresponding response (`struct nvmet_tcp_cmd`), and inserts it into the command buffer (⓬). Otherwise, it processes data payload, extracts the NVMe command (`struct nvme_command`), and delivers it to `blk_mq` (❽). Second, the block layer then interacts with its local NVMe system, submits an IO read/write (❾), accesses the SSD drive, and waits for an I/O completion (❿). Next, similar to Steps 2–4, block I/Os traverse target-side PDU handling (⓫), where the fabricated response capsules are inserted into the command buffer (⓬) and delivered to the send queue (⓭). Finally, the NVMe/TCP session takes response PDUs from the send queue (⓮), performs TCP/IP protocol processing, and sends data back to the initiator (⓯).

2.3 Challenges and Existing Solutions

Challenges. Profiling storage applications running over Linux NVMe/TCP is non-trivial. First, applications exhibit diverse I/O profiles regarding block size, access pattern, and I/O concurrency. Second, the NVMe/TCP layer closely interacts with other system modules on both control and data planes. As shown in Figure 2, it takes I/Os from the block layer, transmits requests/responses via the TCP/IP networking stack, and submits (accepts) NVMe commands to (from) the SSD through the NVMe subsystem. All layers have certain execution parallelism with flexible load-balancing support. Third, each request traverses a long data path between two physical hosts (Figure 2). An I/O can be queued at several locations in a NVMe/TCP setting. There is the block layer [33], where (a) a software staging queue buffers I/Os for flexible scheduling; and (b) a hardware dispatch queue controls the submission rate. Within the TCP/IP networking stack, we have (a) packet pending queue (`sk_write_queue`) to buffer in-network data, (b) traffic control (i.e., `qdisc`), e.g., the TCP Small Queues mechanism (TSQ) limiting the number of transmitted packets to reduce RTT and avoid bufferbloat [80]; (c) driver RX/TX queue, i.e., fixed-sized ring buffer. Further, NVMe employs the multi-queue interface between a host driver and an NVMe controller to carry submitted and completed I/O commands.

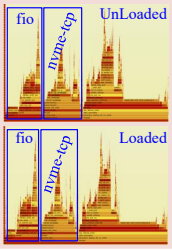
Existing Solutions. There are no structured and systematic profiling utilities in the Linux NVMe/TCP domain for performance diagnostic, overhead analysis, and optimization exploration. Developers today leverage a sequence of development utilities, embarrassingly combine them based on empirical observations, and manually synthesize isolated profiling reports from different tools to infer the root causes.

People usually take a top-down strategy and gradually apply tools from the application layer to the system stack, down to the infrastructure level. For example, one would first

Application Layer *fio*

Metrics	Unloaded	Loaded
Latency (usec)	867.1	1999.6
Throughput (MB/s)	1441.0	625.0

Hot System Layer *Perf*



Top-6 Function Calls Breakdown Sorted by Overhead(%)

Rank	UnLoaded Function Name	%	Loaded Function Name	%
1	get_io_u	24.6	get_io_u	22.7
2	internal_get_*	7.6	__blk_queue*	6.5
3	__blk_queue*	5.9	internal_get_*	6.2
4	try_grab_*	4.1	try_grab_*	3.7
5	__bio_try_*	2.2	bio_iov_iter_*	2.6
6	bio_iov_iter_*	1.9	__bio_try_*	2.5

Infrastructure Layer *iperf3, qperf*

Metrics	Unloaded	Loaded
Latency (usec)	11.6	11.4
Throughput (Gb/s)	88.5	93.8

Figure 3: Analyzing NVMe/TCP via existing solutions. We set up the unloaded and loaded cases on the NVMe/TCP target, where the latter has background I/O streams contending the SSD. In both scenarios, we run the `fio` workload (128KB write, iodepth=10) and show how existing tools from different system layers (`Perf`, `iperf3`, and `qperf`) fail to diagnose the issue.

use application-provided microbenchmarks (like RocksDB's `db_bench` [16] and Lustre's `lnet_selftest` [11]) and tracing utilities to characterize the basic I/O performance and collect running logs. Next, after figuring out anomaly/unexpected execution, she would use system tools (such as `gprof` [45], `JProfiler` [94], `cProfile` [44], `Valgrind` [23], `blktrace` [17], `Oprofile` [37] and `Perf` [38]) to examine CPU usage, memory footprint, thread hotspot, or even some hardware architectural events (like L2/L3 cache misses and retired instructions per cycle) and debug the issue. Last, she might also need to employ low-level infrastructure utilities, e.g., `iperf3` [39], `qperf` [85], `iostat` [18], and `iometer` [66], to validate if the underlying networking fabric and I/O substrate satisfy the deployment assumption. This process is tedious and requires lots of expertise and trials. Worse yet, it is hard to reproduce some interleaved I/O scenarios, especially in a multi-tenant disaggregated storage environment.

We configure an unloaded and loaded NVMe/TCP target and run the same `fio` workload atop (Figure 3). When the SSD is congested (loaded scenario), its I/O throughput drops from 1441.0 MB/s to 625.0 MB/s with average latency increasing from 867.1 µs to 1999.6 µs. However, when using `Perf` to locate the issue, it reports nearly no difference in both cases. For example, the user-space function `get_io_u` consumes the most CPU cycles, i.e., 24.6% and 22.7%, respectively. Kernel functions `__blk_queue_split` and `internal_get_user_pages_fast` rank second and third. When using networking utilities (like `iperf3` and `qperf`), we find that the network pipe between the initiator and target provides enough bandwidth (i.e., 88.5 GB/s and 93.8 GB/s), ruling out the network bottleneck. Thus, using existing tools to pinpoint the root cause is not straightforward.

Therefore, an adequate NVMe/TCP profiling utility should (a) generate different kinds of performance reports for ar-

bitrary I/O workloads (like average/tail latency, throughput, queueing delay); (b) capture I/O data path execution horizontally between distributed hosts and vertically through the system stack; (c) provide diverse and efficient analysis capabilities that serve either online/offline scenarios, identify ephemeral/persistent I/O behaviors, and incur marginal overheads for co-located applications. We target NVMe/TCP-based storage disaggregation due to its wide adoption [2, 3, 12, 25, 27, 28].

3 `ntprof`: An NVMe-over-TCP Profiler

This section describes the design and implementation of `ntprof`. Our system design goals are as follows:

- **Informative.** `ntprof` should provide adequate running statistics about NVMe/TCP processing at the prescribed I/O granularity and profiling scope for protocol understanding, performance diagnostic, and bottleneck localization;
- **Profiling rich.** `ntprof` should provide diverse end-to-end profiling capabilities, accommodating various storage workload profiles (e.g., IO sizes, read/write ratio, access pattern, and concurrency), dissecting deep and pipelined protocol handling, analyzing inter-layer interaction, and reporting characteristics based on the task specification;
- **Lightweight.** `ntprof` should allow online and offline profiling based on the developer's need. Since `ntprof` operates at the data plane and interacts with crucial system components (§ 2.2), it is pivotal to ensure `ntprof` incurs minimal system overheads in terms of memory footprint and CPU cycles with a marginal impact on running I/O applications.

3.1 Key Idea: NVMe/TCP as a Network

`ntprof` treats the NVMe/TCP subsystem as a lossless data network–an initiator sends I/O requests, a target returns the responses, and PDUs encapsulated as TCP packets are forwarded by different system entities along the I/O path. Inspired by the efficacy of network telemetry systems [32, 48, 54, 103], we model the NVMe/TCP setup as a switched networking system and apply network monitoring techniques. Essentially, we make each on-path system component a multi-queue software switch, equip it with a programmable profiling agent on the data plane, and develop a proactive query interface for data collection and analysis. As such, our design behaves similarly to active network-based profiling [57, 58].

In active networks [95, 96], switches, routers, and gateways employ a programmable architecture, and can execute mini-programs (also called "active capsule") carried by network packets. This enables several in-network capabilities, such as application-specific multicast, information fusion, monitoring, and middlebox functionalities (like address translation, load balancing, monitoring, deep packet inspection, and intrusion detection). TPP (tiny packet program) [57, 58] is an active network-based monitoring system. An Ethernet packet, embedding a TPP code segment (based on six load/store instructions), can execute the TPP program on the switch to collect runtime statistics when forwarded. The endhost then develops the data-plane network telemetry using the TPP execution results. `ntprof` employs the same approach to the NVMe/TCP context. We use a special TPP-like I/O request to collect statistics from each on-path system module and reconstruct the profiling report at the end.

Figure 4 presents the overall system architecture of `ntprof`. Programmers define the profiling task and specify its configuration (§3.3). We then translate these specifications into different profiling rules and install them into the corresponding profiling agents (§3.4), co-located with modeled switching components (§3.2). After launching the application, `ntprof` (a) bookkeeps prescribed execution statistics of traversed I/Os at all active entities on the data plane (§3.4); (b) issues a special query command periodically to collect temporary and isolated profiling summary from individual layers (§3.5); (c) reconstructs the entire I/O running profile and performs stipulated analyses (§3.6). `ntprof` works in both offline and online modes. The former targets in-depth analysis and diagnosis based on full profiling data sets, whereas the latter aims for real-time running insights through window-based monitoring.

3.2 Modeling the I/O Path

We model the NVMe/TCP-based storage substrate as a multi-stage Clos network [36, 88], where senders (initiators) and receivers (targets) communicate via a sequence of buffered software switches (Figure 4). Thus, the I/O path of an NVMe/TCP submission request (completion response) consists of a chain of switching queues. We then categorize the queueing model of different software layers using Kendall's notation [60]. In the following expression `A/S/c/K/N/D`, `A` is the arrival process, `S` is the service time distribution, `c` is the number of workers (consumers), `K` is the queueing capacity, `N` is the number of customers (producers), and `D` is the queueing discipline. We summarize three models based on the execution characteristics of different on-path software components.

- The *Central-FCFS* model (formally $G/G/1/Q/X/FCFS$) takes requests from one or several producers ($X \geq 1$) and processes them in the first-come-first-serve (FCFS) order. The per-core software staging queue of the `blk-mq` layer, the initiator-side per-session NVMe/TCP transport queue, and `nvme-pcie` submission queue belong to this;
- The *Split-FCFS* model (formally $G/G/X/Q/1/FCFS$) partitions I/Os to multiple FCFS workers ($X \geq 1$) based on the execution mapping, such as a target-side session transport queue and a `nvme-pcie` completion queue;
- The *Central-PS* model (formally $G/G/1/Q/X/PS$) is a little complex, used to describe the networking stack handling [34]. Even though NVMe/TCP packets traverse each connection in order, the connection-core mapping happens differently, depending on the system setup. For example, one core could concurrently take several sessions ($X \geq 1$)

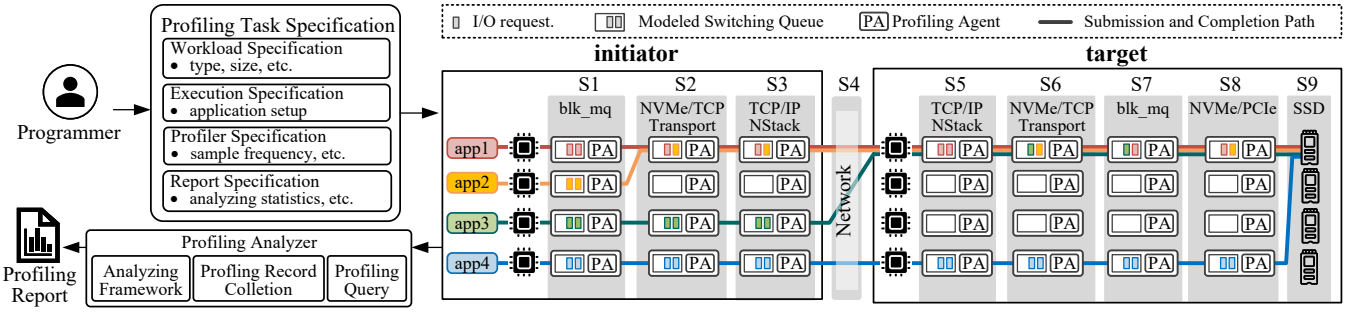


Figure 4: System overview of `ntprof`. PA refers to the Programmable Profiling Agent, attached to each switching queue. "S" refers to a processing stage. The right side of the figure illustrates four example applications. The I/O paths of app1 and app2 overlap at Stage 2 because both are mapped to the same NVMe/TCP queue. App3 shares the same I/O path as app1 at Stage 4, as their TCP connections are handled by the same core on the target side. The I/O path of app4 overlaps with app1 at Stage 8 as both mount the same SSD drive.

and load balances CPU cycles following some processor sharing (PS) discipline. One session usually can not be touched by more than one core at any given time.

These models allow us to abstract the software switch accurately, effectively integrate profiling agents into each layer, and enable more accurate queueing stall estimation.

3.3 Profiling Task Specification

`ntprof` requires a specification template (Figure 4) to define the profiling task, including I/O workloads, running environment setup, profiler configuration, and profiling statistics.

- **Workload Specification** prescribes what I/Os to profile. It tells `ntprof` (a) which NVMe/TCP sessions (i.e., name and mounted address) and (b) what types of I/Os (e.g., I/O read-/write type and I/O sizes) to trace and analyze. We support exact match (size=4KB) and range match (size≥32KB);
- **Execution Specification** defines how the application is deployed and run. This includes (a) application setup, e.g., whether application threads are pinned; (b) NVMe/TCP configuration, e.g., the number and core mapping of transport queues, connections, and NVMe queues; (c) hardware limits of network and storage substrates, e.g., maximum bandwidth of network links and NVMe drives;
- **Profiler Specification** determines the workflow of `ntprof`. First, we support both online and offline modes, where the former reports runtime statistics (like PAPI [19]), and the latter provides an execution summary based on entire tracing. Second, we allow users to specify the sampling frequency (in terms of time epoch or the number of I/Os) and profiling time window, e.g., the start and end time pair being the conventional `YYYY-MM-DD hh:mm:ss` format. Third, `ntprof` requires setting the maximum in-memory buffer size to store the profiling record. When exceeded, it either overwrites the old ones or dumps them on the local drive based on user preference. Last, in the online mode, we provide several programmable primitives to enable more advanced statistical reporting. For example, `MIN|MAX|AVG|DIST` computes the min, max, average, and distinct value of a given profiling range or time window;

Tracepoint	Triggered Condition
`_queue_rq`	When a block I/O enters the NVMe/TCP layer
`_queue_request`	After a block I/O is converted to a CapsuleCmd PDU
`_try_send_cmd_pdu`	When a CapsuleCmd PDU is copied to the TCP skbuff
`_try_send_data_pdu`	When a H2C Data PDU is copied to the TCP skbuff
`_try_send_data`	When a chunk of data is copied to the TCP skbuff
`_done_send_req`	When the entire PDU is copied to the TCP skbuff
`_handle_c2h_data`	After parsing the header of a C2HData PDU
`_recv_data`	After receiving a chunk of C2HData PDUs
`_process_nvme_cqe`	After receiving a CapsuleResp PDU
`_handle_r2t`	When parsing the header of an R2T PDU

Table 1: Tracepoints added in the `nvme-tcp` (initiator) module.

- **Report Specification** lists the format of profiling results. There are two aspects. One is analyzing statistics, including (a) I/O latency distribution, (b) end-to-end throughput, (c) latency breakdown of different layers across the I/O path, and (d) queueing occupancy. The other one is data presentation. `ntprof` allows users to stipulate grouper and aggregator functions to organize execution statistics, such as grouped by the I/O type, size, and session.

3.4 Programmable Profiling Agent

We equip each modeled software switch with a profiling agent that includes a collector capable of gathering I/O running characteristics, and predicates to filter out irrelevant ones.

Statistics collector. We build the collector based on the Linux tracepoint mechanism [10]. A tracepoint is a static instrumentation point in the kernel and exposes a hook to call a function provided by the developers. It can take parameters (Appendix A.1) and enrich the execution logic. This makes collecting detailed low-level statistics at specific points of interest possible. We add tracepoints when a request (a) enters/leaves a software module; (b) traverses essential processing stages (Figure 2). Tables 1/2 list the tracepoints that `ntprof` introduces into the `nvme-tcp` and `nvmet-tcp` modules.

Upon triggering a tracepoint, `ntprof` creates, updates, and seals a profiling record based on the registered function. Each I/O is associated with one record, consisting of request metadata (such as type, size, tag, and command/session ID) and time-series triggered events throughout the lifetime of an I/O (Figure 5). An event log is a 12-byte tuple, including the event name and activating time, which is appended to the profiling

Tracepoint	Triggered Condition
_try_recv_pdu	When nvmet-tcp tries to read data from a skbuff
_done_recv_pdu	After reading the header of a received PDU
_exec_read_req	After receiving a complete read request
_exec_write_req	After receiving a complete write request
_queue_response	When inserting an element into the response queue
_setup_c2h_data_pdu	When starting to make a C2HData PDU
_setup_r2t_pdu	When starting to make an R2T PDU
_setup_response_pdu	When starting to make a CapsuleResp PDU
_try_send_data_pdu	After copying the header of the C2HData PDU
_try_send_r2t	After copying an R2T PDU to the TCP skbuff
_try_send_response	After copying a CapsuleResp PDU to the TCP skbuff
_try_send_data	After copying a chunk of data to the TCP skbuff
_handle_h2c_data_pdu	After reading the header of an H2CData PDU
_try_recv_data	After reading a chunk of data from the TCP skbuff

Table 2: Tracepoints added in the `nvmet-tcp` (target) module.

record. Generally, a record is initialized when the tracepoint `nvme_tcp_queue_rq` happens and completed upon triggering `nvme_tcp_process_ncme_cqe`. The size of a record is I/O-dependent. For example, a read I/O, a write I/O without inline data, and a write I/O with inline data consume at least 207, 183, and 303 bytes, respectively. Appendix A.2 presents the event list of these three cases. In `ntprof`, a profiling record is pinned to a dedicated session that handles the I/O. The collection of records from all sessions forms the entire profiling statistics. Relating to Figure 4, `ntprof` collects the ingress and egress times of the profiled I/O, except for Stage 3-5 and Stage 9, as in the network stack, PDUs are flattened and packed into binary data inside TCP packets. Additionally, `ntprof` also collects the length of the PDU sending queues at Stage 2 and Stage 6. `ntprof` follows the kernel log design and uses an in-memory ring buffer to store records, whose size is configured during initialization.

Predicate. `ntprof` translates profiling specifications (§3.3) into predicates to filter out uninterested I/Os, improving the profiling efficiency. A predicate is a logical expression composed of one or more conditions, often connected by logical operators such as "and". For example, if the programmer specifies capturing 4K read I/Os, the predicate would be: $type = read \wedge size = 4KB$. When a tracepoint is active, the I/O is checked by one or several predicates to determine whether a record update is needed. `ntprof` merges predicates from one I/O stream and allows online editing.

3.5 Profiling Query Protocol

`ntprof` views the NVMe/TCP-based storage substrate as a networking system and develops an in-bind query protocol to obtain statistics at runtime. Akin to TPP [58], we design a special NVMe/TCP command capsule (called `ProbCmd`), carrying a small program segment that states what and where to query via basic load/store/reset primitives. We then introduce a response capsule (`ProbResp`) to return the results. `ProbCmd` and `ProbResp` follow the existing PDU format (Figure 1), consisting of an 8-byte command header and other corresponding headers. The data fields are restructured as a sequence of "query instructions" in the format of `[Query_Opcode] [Software_Switch_ID] [Statistics_Type] [Parameters]`. These two commands

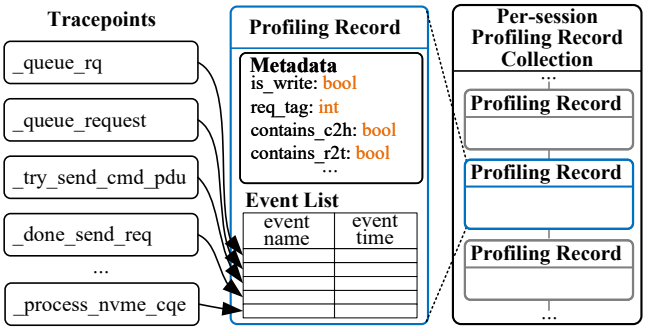


Figure 5: Tracepoints, profiling records, and per-session profiling record collection in the `nvme-tcp` layer.

specify the NVMe/TCP session, not the actual logical block address (LBA), and traverse the same path as read/write I/Os. Our protocol works as follows across different entities:

- **Requester.** The `ntprof` host utility, co-located with storage applications, fabricates the query request and submits it to the block layer. We support both frequency-based (per X I/Os) and time-based operating modes (per Y seconds). Upon receiving the responses, it then extracts the query results and delivers them to the profiling analyzer;
- **Executor.** When receiving the probe command, the software switch extracts the embedded program, teases out belonging instructions (based on the switch ID), and executes them sequentially. It blocks the profiling agent temporarily, collects all profiling records associated with the NVMe/TCP session, and manipulates their profiling records based on the statistic type field. For a load primitive, we copy the statistics, build the response capsule, and then directly push it to the completion path. A store/reset opcode causes updating/clearing up the profiling record and sending an acknowledgment back. Since there might be multiple I/O paths for one session, when forwarding a probe request, our switch would duplicate the special command and submit them to all traversed egressed paths, resembling the switch port mirroring technique [14];
- **Responder.** Unlike TPP which relies on the destination node to construct a reply packet and piggyback queried results, `ntprof` allows each software switch to reply directly due to the large memory usage of profiling records. This indicates that one `ProbCmd` capsule incurs `ProbResp` capsules back. Further, when the probing request reaches the NVMe/PCIe layer at the target side (Figure 2), we send back a probing completion response to notify the controller.

3.6 Profiling Analyzer

During the querying phase, the host sends profiling queries and buffers the received results. `ntprof` then processes and analyzes the data. The analysis involves four steps:

First, `ntprof` provides different ways to organize profiling records, such as by NVMe/TCP session, I/O path, issuing core, target NVMe drive, or request category. We first categorize

these records based on the prescribed criteria (defined in the report specification), wherein each is further serialized based on the I/O submission timestamp (tagged when entering the block layer). Second, `ntprof` performs statistics calibration. End-to-end profiling results are obtained directly at the initiator host. The latency breakdown gathers statistics from all on-path agents which contain event processing time and queueing delay. We timestamp the enqueue and dequeue events whenever possible and use our abstracted queueing model (§3.2) with the queueing occupancy and I/O concurrency distribution for validation. Third, `ntprof` encodes grouper/aggregator functions into a multi-threaded map-reduce framework. We spawn several mappers based on the profiling data volume and group records based on the mapper keys (e.g., 4KB I/O). Intermediate results are shuffled to a couple of reducers that perform the aggregation; Fourth, we output profiling results in the JSON format, which could also be visualized.

3.7 Implementation

`ntprof` requires kernel patches, adds a new kernel module, and provides a user utility, ∼10K LOCs in total.

Kernel Modifications. Tracepoints are the key approach enabling our programmable profiling agent. Here is an example showing how we track the enqueue event at the NVMe/TCP transport layer. It defines three parameters: a pointer to a request struct, the NVMe command, and the queue id.

```
/* TRACE_EVENT Example */
TRACE_EVENT(nvme_tcp_queue_request,
    TP_PROTO(struct request *req,
             struct nvme_command* cmd,
             int qid),
    TP_ARGS(req, cmd, qid),
    TP_STRUCT__entry(...),
    TP_fast_assign(...),
    TP_printk(...));
```

We insert this tracepoint into the `nvme-tcp` module when `trace_nvme_tcp_queue_request` function is called.

```
trace_nvme_tcp_queue_request(
    blk_mq_rq_from_pdu(req),
    req->req.cmd,
    nvme_tcp_queue_id(queue));
```

Kernel Module and User-space Utility. `ntprof` introduce new kernel modules at the initiator and target side. Here is an example of the callback function for the above tracepoint. It is registered during the module initialization, and it will be executed once the tracepoint is triggered. The user-space utility provides a CLI interface, and it takes the developer's inputs, interacts with kernel modules, and reports profiling results.

```
void on_nvme_tcp_queue_request(
    void* ignore, struct request* req,
    struct nvme_command* cmd, int qid) {...}
```

Implementation Alternative. `ntprof` might also be implemented using eBPF [6] instead of a standalone kernel module. One could (1) define and insert the tracepoints to the kernel as described earlier; (2) define eBPF programs, which are functions attached to the predefined tracepoints using the `SEC` macro; (3) implement a user-space analyzer to read profiling records through the eBPF maps and generate the final profiling results. The eBPF programs can retrieve the tracepoint's parameters and store profiling records in eBPF maps [7].

Extension. One could also extend `ntprof` to support the kernel-bypass storage stack. For example, over SPDK NVMe-oF, we can use its tracing framework to define corresponding tracepoints [5, 21, 22], capture them via the eBPF mechanism, and reuse some of `ntprof`'s building blocks, like profiling agent, query protocol, and profiling analyzer.

4 Case Studies

4.1 Experimental Methodology

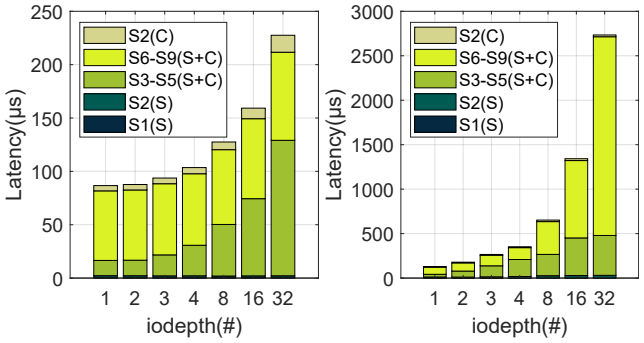
Testbed. We use two hardware setups. The first (testbed-1) consists of 2 sm110p servers from CloudLab [40]. Each node has a 32-core Intel Xeon Silver 4314 CPU, 128 GB DDR4, 1 NVIDIA/Mellanox CX6 25GbE NIC, 1 NVIDIA/Mellanox CX6 100GbE NIC, 1 Intel SATA SSD, and 4 Samsung (MZQL2960HCJR-00A07) 960GB NVMe SSDs. We configure its network MTU to 9KB. The second (testbed-2) includes 2 m510 servers from Cloudlab. Each node has a 16-core Intel Xeon D-1548 CPU, 64 GB of DDR4 memory, 1 Mellanox CX3 10 GbE NIC, 1 TOSHIBA (THNSN5256GPU7) 256 GB NVMe SSD. We disable the turbo boost and set the CPU frequency governor to "performance" in both cases. All nodes run Ubuntu 20.04 with kernel v5.15.143.

Workloads. Case studies 1–4 use synthetic workloads generated by `fio` v3.36. Case 5 runs the TSBS benchmark [20] over Apache IoTDB [4, 99] v1.2.0 in a write-heavy scenario. Case 6 runs the Filebench's [75] videoserver workload over F2FS [65] issuing read-heavy I/Os. We mainly examine I/O latency and throughput in each case. To demonstrate how `ntprof` reacts to different hardware settings, we conduct use case 1 on both testbeds. Due to space limitations, the results for the remaining use cases are presented only for testbed-1. The results for testbed-2 are presented in the Appendix A.3.

4.2 Use Case 1: Latency Breakdown

Description. `ntprof` can break down the end-to-end I/O latency into different stages (Figure 4) and provide useful system execution insights. Developers often complain about the unexpected poor performance in NVMe/TCP [15, 24], but the issue is sometimes unrelated to the protocol. More often, it's due to misconfigurations, such as having insufficient parallelism in the NVMe subsystem or mismounting a hard drive. Latency breakdown is helpful in these scenarios to pinpoint the root cause. Further, when introducing new software layers in the I/O path [55, 56, 77] or upgrading hardware, latency breakdown can help assess the impact of these changes.

Experimental Setup. On testbed-1, we break down the latency of a 4KB random read and 128KB sequential write



(a) 4K Random Read. **(b) 128K Sequential Write.**

Figure 6: Latency breakdown for the 4K random read and 128K sequential write as increasing the IO depth on testbed-1. We report the five most time-consuming parts. S1(S) refers to the time an I/O stays at the S1 stage on the submission path. S2(C) captures the time spent at the S2 stage on the completion path. S3–S5(S+C) aggregates the time spent in the S3, S4, and S5 stages across the submission and completion paths.

while increasing the iodepth from 1 to 32. We configure `fio` to run 1 job and issue requests to the Samsung drive in a direct I/O mode. The I/O engine is set to `libaio` for asynchronous operations. On testbed-2, we set the iodepth to 1, but increase the number of jobs from 1 to 32. We define the profiling task specification as follows: read(write) type, 4KB(128KB) block size, NVMe session named "nvme4n1" (the remote storage subsystem), a maximum of 10,000 profiling records, sampling frequency=1 per 1,000, grouping by NVMe session (all I/O sent to "nvme4n1" treated as one group), and `AVERAGE` as the aggregator. We launch the `fio` job with `ntprof`, which collects and analyzes profiling records and generates the report.

Result Analyses. Figure 6a illustrates the latency breakdown for 4K random read requests. As the iodepth varies from 1 to 32, "S3-S5(S+C)" increases the most, from 14.3 µs to 127.0 µs, indicating that queue is built up the networking substrate at both the initiator and target side. For example, when the iodepth equals 32, the above aggregated stages account for 92.2% of the total latency. The reason why `ntprof` can capture it is because our installed profiling agent takes timestamps at two tracepoints at the NVMe/TCP initiator (`_try_send_data` and `_handle_c2h_data`) and two tracepoints (`_done_recv_pdu` and `_try_send_reponse`) at the target, which allows the profiling agent to dissect the latency contribution from one or several stages (Figure 4). Figure 6b shows the latency breakdown for 128K sequential writes. "S6-S9(S+C)" becomes the most time-consuming part, rising from 79.2 µs to 2234.9 µs, as the iodepth increases from 1 to 32. `ntprof` is capable to further break down "S6-S9(S+C)" and reveal that the "S5-S9(S+C)" on the target side takes 95.9% of the time when iodepth is 32. Similarly, such capability comes from the fact that `ntprof` records the triggering time of the two key tracepoints (`_exec_write_req` and `_queue_response`) at the target side. Figure 11a (Appendix A.3) presents the I/O latency breakdown on testbed-2.

Takeaways. `ntprof` models the NVMe/TCP data path as a chain of software switches, where each can be equipped with a programmable profiling agent that uses tracepoints to collect statistics. This allows us to dissect the path segment and perform time accounting in a fine-grained way, and makes `ntprof` to operate across a wide range of hardware models.

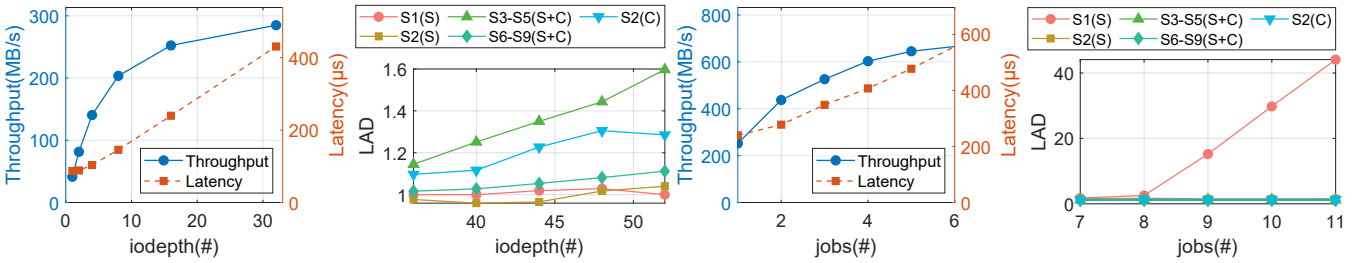
4.3 Use Case 2: Software Bottleneck Localization

Description. NVMe/TCP interacts with several vital OS kernel modules (e.g., block layer, TCP/IP networking stack, and NVMe subsystem). Any system bottleneck from multi-core execution, TCP connection, or NVMe submission/completion command delivery could throttle the achieved I/O throughput, yielding a drastic latency increase. These issues are sometimes subtle, but hard to detect and reproduce, especially given that (a) application I/O ordering is not preserved; (b) multi-tenant deployments are common. Accurate software stack bottleneck localization is a demanding feature for developers.

Experimental Setup. We configure two bottleneck scenarios. One is at the NVMe/TCP target core, where each NVMe drive is provisioned with one target core regardless of the I/O load. We launch a 4KB random read stream and gradually increase the iodepth. The other is at the TCP connection, which carries an increasing number of I/O streams. We spawn multiple `fio` jobs (iodepth=16) and make sure they are pinned to different cores on the initiator side. Both cases are common in an enterprise/cloud disaggregated setting for cost efficiency. The profiling task specification is similar to the first case (§4.2), except for the generated report format.

Result Analyses. Figure 7a depicts the performance when iodepth varies from 1 to 32. We observe a marginal throughput increase (from 252.4 MB/s to 284.9 MB/s) when the iodepth is beyond 16, while latency rises significantly, from 238.7 µs to 431.0 µs, indicating that a system bottleneck occurs. `ntprof` introduces a new metric called *Latency Amplification Degree (LAD)*–defined as T_{cong}/T_{base}, where T_{cong} and T_{base} refer to processing latency at the overloaded and unloaded scenarios–to localize the bottleneck. As shown in Figure 7b, when iodepth is 52, the "S3-S5(S+C)" sees a 1.6 LAD, dominating other parts. `ntprof` can capture this because our profiling agent records timestamps at two tracepoints on the NVMe/TCP target (`_try_send_data` and `_try_send_response`), allowing it to report the time spent in these combined stages. Further LAD analysis shows that "S6(C)" latency increases from 26.38 µs to 47.14 µs (LAD=1.8). This happens because, for the switching queue of S6, the target is not fast enough to send the I/O completion, causing queue buildup and latency increase, indicating the target core becomes the bottleneck. `ntprof` uses the `_done_recv_pdu` tracepoint on the target side to achieve this.

Figure 7c shows that when the number of `fio` jobs increases from 5 to 6, throughput only rises by 3.1%, from 645.2 MB/s to 665.5 MB/s, while latency rises from 477.3 µs to 556.7 µs. As we consolidate jobs from 6 to 11, as shown



(a) Throughput and Latency. **(b) LAD.** **(c) Throughput and Latency.** **(d) LAD.**

Figure 7: Software bottleneck localization. (a) and (b) depict the target core bottleneck case, where we run one `fio` job and increase its iodepth from 1 to 64. (c) and (d) shows the TCP connection bottleneck case, where we increase the number of consolidated `fio` jobs (with iodepth=16) from 1 to 11. LAD=Latency Amplification Degree.

in Figure 7d, the LAD of "S1" rises to 51. Such a latency increase indicates that there are inadequate TCP connections to deliver I/Os from the initiator to the target. `ntprof` locates this through a key tracepoint on the initiator (`_queue_rq`).

Takeaways. `ntprof` views each system component as a buffered software switch. A system bottleneck causes queueing buildup at one or several molded queues (§3.2). `ntprof` thus dissects the queueing delay and introduces a new metric (latency amplification degree) for bottleneck localization.

4.4 Use Case 3: Hardware Bottleneck Localization

Description. The underlying hardware infrastructure can also be a system bottleneck. For example, people find that a malfunctioned NIC [31,64,67] can reduce its bandwidth capacity dramatically. A fragmented NVMe SSD would deliver much lower throughput in a read/write or write-only workload. Further, hardware heterogeneity has become common in the data center, which can also cause hardware bottlenecks.

Experimental Setup. First, we create a "slow" SSD. On the initiator side, we run 1 `fio` job (referred to as "remote `fio`") issuing 128K sequential writes with an iodepth of 10. Simultaneously, we run 1 `fio` job on the target side (referred to as "local `fio`") also performing 128K sequential write workloads to the same drive. Second, we create a "slow" NIC. We run two `fio` instances on the initiator side, each targeting a different remote SSD. Both `fio` instances issue 128K random reads with 8 jobs and an iodepth of 5. Additionally, we run `iperf3` clients on the target side, sending data to the initiator at 100Gbps. The profiling spec is the same as before.

Result Analyses. Figure 12a (Appendix A.3) shows the performance when local `fio` iodepth increases from 2 to 8. The aggregated throughput reaches 1440.0 MB/s, maxing out the SSD bandwidth limit. Compared to no local `fio` jobs, the bandwidth shared by remote `fio` decreases to 627.2 MB/s, while the latency increases from 859.7 µs to 1984.6 µs. Figure 12b shows when local `fio` iodepth is 8, "S6-S9(S+C)" sees a 3.6 LAD, significantly higher than the other parts. `ntprof` can locate this because the profiling agent uses 2 tracepoints on the target (`_don_recv_pdu` and `_try_send_response`). A further drill-down shows that "S7-9(S+C)" is the primary contributor, with its latency increased from 303.8 µs to 1729.2 µs (LAD=5.7). This occurs because the local `fio` pushes more I/Os into the "S7-9(S+C)" stage while the SSD is already operating at its maximum speed, causing the queue to build up and latency to rise. `ntprof` is capable of capturing this because it records timestamps of `_exec_eritw_req` and `_queue_response` on the target.

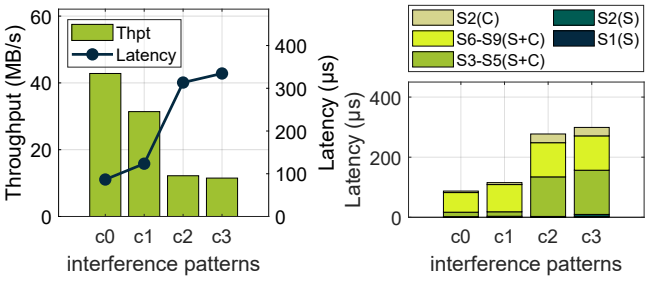
Figure 12c shows the aggregated throughput of `fio` and `iperf3` approaches 100 Gb/s, indicating the NIC is the bottleneck. The latency of `fio` I/O rises from 834.9 µs to 2803.5 µs when there are 16 `iperf3` clients, compared to when there are none. Figure 12d shows "S3-S5(S+C)" increases most significantly when increasing `iperf3` clients. It sees a 4.5 LAD when there are 16 clients. That is because on the completion path, the NVMe/TCP network flows share less bandwidth when there are more `iperf3` clients, leading to queue buildup. `ntprof` can capture it because we take timestamps at two tracepoints (`_try_send_data` and `_handle_c2h_data`) at the NVMe/TCP initiator and two tracepoints (`_done_recv_pdu` and `_try_send_reponse`) at the target.

Takeaways. `ntprof` views the SSD or NIC as one stage. Its queueing model is inherent from the upstream storage, making `ntprof` capable of localizing the hardware bottleneck.

4.5 Use Case 4: Interference Analysis

Description. Multi-tenancy is a pressing concern for storage disaggregation. When latency-sensitive I/O streams co-locate with throughput-oriented ones, one would observe significant head-of-line blocking and unfair bandwidth partition [56,77].

Experimental Setup. We run 2 `fio` instances on the target side. The latency-sensitive workload is configured to issue 4K random read requests with 1 job (iodepth=1). The throughput-sensitive one issues 128K random read requests to the same NVMe drive. We design 3 interference patterns. For case "c1", the two I/O flows are pinned to different cores on the initiator side, each using a separate NVMe/TCP session. For case "c2", they are pinned to the same core and therefore share the same NVMe/TCP session. For case "c3", they are pinned to different cores but share the same NVMe/TCP session. We define the profiling task specification as follows: read type, 4KB block size, NVMe session named "nvme4n1", a maximum of 10,000 profiling records, sampling frequency=1 per 1,000, grouping by NVMe session, and `AVERAGE` as the aggregator. We launch the `fio` job with `ntprof`, which collects



(a) Throughput and Average Latency. (b) Latency Breakdown.

Figure 8: Latency breakdown and throughput analysis of the 4K random read I/O flow when introducing a 128K random read I/O in different scenarios. There is only the 4K read I/O flow running in "c0". "c1" Two I/O flows are pinned to different cores and using different NVMe/TCP sessions in "c1". Both flows are pinned to the same core, sharing the same NVMe/TCP session in "c2". Both flows are pinned to different cores but share the same NVMe/TCP sessions in "c3".

and analyzes profiling records and generates the report.

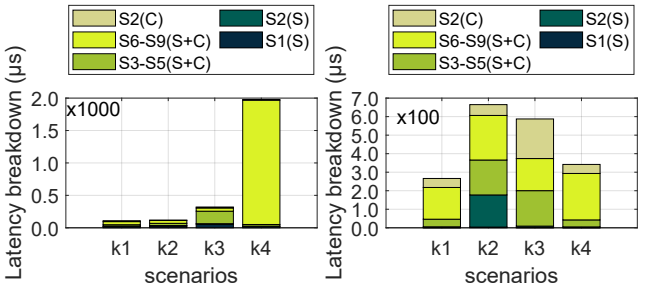
Result Analyses. Figure 8a shows the latency of 4KB reads is increased in all interference cases, with a 26.7%, 71.5%, and 73.1% throughput degradation in "c1", "c2", and "c3", respectively. Figure 8 shows the latency breakdown of the 4KB read requests. In "c1", the "S6-S9(S+C)" latency rises the most, from 65.4 µs to 90.6 µs. A further breakdown reveals that the majority of this increase, 25.2 µs, comes from the "S7-S9(S+C)". A queue is built up at the NVMe/TCP session at the target due to the increasing I/O load. `ntprof` captures this because it utilizes the timestamp when the 2 tracepoints on the target are triggered (`_exec_read_req` and `_queue_response`). In "c2", "S3-S5(S+C)" increases the most, from 14.9 to 131.6 µs, also due to "S6-S9(S+C)". The impact of latency in "c3" is similar to "c2", where "S3-S5(S+C)" is increased to 114.2 µs.

Takeaways. `ntprof` enables profiling at different I/O granularities with separated profiling paths. Under mixed I/O traffic, `ntprof` generates different profiling records to collect statistics, making interference analysis possible.

4.6 Use Case 5: Apache IoTDB

Description. Apache IoTDB [4, 59, 99] is an LSM-tree-based high-performance time-series database designed for IoT scenarios. Storage disaggregation gains significant traction in such settings given that edge computing nodes have limited storage resources. We used the Time Series Benchmark Suite (TSBS) [20, 89] to generate DevOps workloads to evaluate the performance of time-series databases. By using the default parameters, the workload is set to contain 4000 devices' information running for 3 days, with each device generating multiple time-series metrics, such as CPU utilization, memory usage, disk I/O, and network activity, respectively.

Experimental Setup. Both the Apache IoTDB server and client run on the initiator node, with IoTDB's data directory mounted to a remote Samsung SSD. Using the default con-



(a) Use case: IoTDB. (b) Use case: F2FS.

Figure 9: I/O Latency breakdowns for real-world applications under different scenarios. k1 is the baseline, where there are 32 NVMe/TCP sessions. The number is reduced to 1 in k2. 8 `iperf3` clients are running in k3, and `fio` workload is running on the target side in k4. The IoTDB and F2FS show the latency breakdown for 4KB writes and 128KB reads.

figuration of TSBS, we run 32 workers to write data into the database. We analyze the latency breakdown of all write I/Os under the following scenarios: (1) "k1" uses 32 NVMe-TCP sessions, (2) "k2" pins all I/O to a single NVMe/TCP session, (3) in "k3", there are 8 `iperf3` clients on the initiator side, each sending data to one of 8 servers on the target side, emulating a NIC bottleneck, and (4) in "k4" two `fio` jobs run on the target side, issuing 128 KB sequential writes to the same SSD (iodepth=10), creating an SSD bottleneck. We configure `ntprof` similarly to §4.2, except that we monitor ANY types of write I/Os, and results are grouped by I/O type/size.

Result Analysis. The TSBS benchmark reports the throughput of IoTDB as follows: 10.6M matrices/s in k1, 10.4M in k2, 8.7M in k3, and 9.6M in k4. `ntprof` shows different sized writes are issued, ranging from 4KB to several MB. `ntprof` reports this because it generates different profiling records of different types of I/O. Across all scenarios, 4 KB write requests constitute the majority, 80.9% in scenario k1, 80.2% in k2, 77.5% in k3, and 80.1% in k4. Figure 9a shows the latency breakdown for 4KB writes (since it dominates the write traffic). k3 sees a drastic increase in "S3-S5(S+C)". That is because NVMe/TCP sessions are competing with `iperf3` flows. The queue is built up in stage S3. In k4, "S6-S9(S+C)" increases most significantly, from 47.8 µs to 1917.3 µs. A further breakdown shows an 1859.7 µs increase in "S7-S9(S+C)" is the primary contributor, indicating that the background I/O streams consume bandwidth and make SSD a bottleneck.

Takeaways. Real-world applications exhibit diverse I/O profiles. `ntprof` exposes a rich task specification interface for developers, translates these profiling requirements into predicates, and installs them into the modeled software switch. As such, it enables concurrent heterogeneous I/O profiling.

4.7 Use Case 6: F2FS

Description. F2FS [65] is a Linux file system designed for modern flash storage devices. F2FS performs exceptionally well with NVMe SSDs, as it efficiently handles the inherent characteristics of flash memory, such as wear-leveling and

garbage collection. In this case study, we focus on analyzing how file I/Os traverse the NVMe/TCP. We take Filebench [75] and run the 'videoserver" workload, which emulates a video streaming server managing multiple concurrent streams.

Experimental Setup. We configure the "videoserver" to run 32 threads issuing 128KB read requests to the active videos, while there is another thread issuing 1MB write requests to the passive videos. The video size is set to 10 GB. Direct I/O is enabled. We also run the application under the 4 scenarios as described in §4.6, except in k3, the `iperf3` clients are on the target side, while the servers are on the initiator side. We use the same profiling task specification and focus on reads.

Result Analysis. Filebench reports 117µs per operation in k1, 319 µs in k2, 235 µs in k3 and 114 µs in k4. Figure 9b shows the latency breakdown for 128KB read I/O. Comparing to k1, "S2(S)" in k2 increases significantly, from 4.6 µs to 173.5 µs. This is because a single TCP connection in the nvme-tcp queue struggles to handle the read I/Os submitted by all the cores, indicating the need to increase the number of NVMe/TCP queues. Akin to §4.2, `ntprof` locates this via two tracepoints (`_queue_request` and `_try_send_cmd_pdu`). "S6-S9(S+C)" is increased from 171.3 µs to 240.6 µs. A further breakdown shows that "S6(C)" is increased by 55.2 µs, dominating the entire latency. That is because by reducing NVMe/TCP sessions we also reduce the dequeuing rate in S6 on the completion path. `ntprof` uses two tracepoints (`_queue_response` and `_setup_response_pdu`) on the target side for analysis. In k3, "S3-S5(S+C)" is increased from 41.0 µs to 191.1 µs. The reason is similar to the case in §4.6. In addition, "S2(C)" is increased from 48.7 µs to 213.8 µs, shown via 2 tracepoints on the initiator side (`_handle_c2h_data` and `_process_nvme_cqe`). In k4, "S6-S9(S+C)" is increased from 171.3 µs to 250.8 µs, aligning with our scenario design.

Takeaways. `ntprof` enables hierarchical I/O path analysis due to our Clos network view. Similar to the Flame Graph [8], one can gradually drill the I/O path segment using different metrics and locate the system bottleneck.

4.8 Overhead Analysis

`ntprof` operates at the data plane and interacts with critical system components. It is important to minimize overhead, including memory and CPU usage. We configure four sets of `fio` experiments (Appendix A.3) and measure the application performance, CPU usage, and memory usage when enabling and disabling (Table 5). Across all workloads, the overhead introduced by `ntprof` is marginal. The CPU usage increases by 0.6% and 2.9% for the read and write cases. We observe up to 17MB more memory usage during the profiling.

5 Related Work

Network Telemetry. `ntprof` benefits from prior network telemetry systems [32, 48, 54, 74, 81, 93, 101, 103, 104, 106]. For example, Sonata [48] develops a declarative interface to express telemetry tasks and partitions query execution to streaming processors and switches. [32, 68, 72, 74, 81, 90, 91, 101, 104, 106] introduce probabilistic data structures (like Sketch) to capture high-volume data streams with low storage on the programmable data plane, i.e., in-network telemetry.

Profiling Systems. Many software utilities are developed to identify code hotspots, analyze concurrency dependency, and break down stalled cycles [9, 13, 18, 19, 38, 45, 66, 102]. For example, Linux `perf` [38] is a widely used tool to instrument CPU performance counters, tracepoints, and probes and reports application execution statistics. Intel VTune [9] takes a top-down analysis strategy [102] and drills down the performance analysis mostly using architectural counters. Some are also integrated into the language system facilitating application development [13, 35, 46, 61, 70, 73, 82–84, 87, 100]. We focus on profiling and analyzing the NVMe/TCP protocol.

Storage Disaggregation. People have explored disaggregated storage extensively given the rising networking bandwidth and fast remote storage protocol [47, 49, 50, 52, 55, 62, 63, 71, 77–79, 105]. Ana Klimovic *et al.* characterize the performance of iSCSI-based disaggregated storage [62]. i10 [55] develops an efficient in-kernel TCP/IP remote storage based on dedicated end-to-end IO paths and delayed doorbell notifications. Researchers [49] also report the performance characteristics of server-based NVMe-oF boxes.

Intra-host Diagnosis. The host interconnect is becoming a bottleneck under high-bandwidth networks. Researchers have developed benchmarking frameworks and diagnostic tools to analyze it [29, 30, 51, 53, 67, 69, 97]. For example, Saksham Agarwal et al. [29, 30] analyze the host congestion issues and build the host congestion control protocol. NSight [51], leveraging CPU profilers, captures sources of overheads in the end-host stack at the nanosecond scale. Hostping [67] monitors and diagnoses intra-host bottlenecks in RDMA networks. `ntprof` can use them to improve its efficiency.

6 Conclusion

This paper presents a profiling and development utility for NVMe/TCP. It enables developers to understand and analyze the execution characteristics of the NVMe/TCP systematically. The key idea is to view the NVMe/TCP storage substrate as a lossless switched network and apply network monitoring techniques. `ntprof` models each on-path system module as a software switch, integrates a programmable profiling agent on the data plane, and introduces a proactive query interface for statistics collection and analysis. We build `ntprof` over Linux kernel and apply it for six case studies.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Soujanya Ponnapalli, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199, CNS-2212192, and CAREER-2339755.

References

[1] Linux Kernel 5.0 Release. https://kernelnewbies.org/Linux_5.0, 2019.

[2] x-cellent technologies Selects Lightbits for metalstack.cloud. https://www.lightbitslabs.com/press-releases/x-cellent-technologies-selects-lightbits-for-metalstack-cloud/, 2022.

[3] How We Built It: Block Storage for AI/ML Workloads, Powered by Lightbits. https://crusoe.ai/blog/how-we-built-it-block-storage-for-ai-ml-workloads-powered-by-lightbits/, 2023.

[4] Apache IoTDB: Database for Internet of Things. https://iotdb.apache.org/, 2024.

[5] BPF tracing with SPDK. https://spdk.io/news/2021/09/28/bpf_tracing_with_spdk/, 2024.

[6] eBPF. https://ebpf.io/, 2024.

[7] eBPF Maps. https://docs.ebpf.io/linux/concepts/maps/, 2024.

[8] Flame Graphs. https://www.brendangregg.com/flamegraphs.html, 2024.

[9] Intel VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.ed42rh, 2024.

[10] Linux Kernel Tracepoints. https://docs.kernel.org/trace/tracepoints.html, 2024.

[11] Lustre's Benchmarking. https://wiki.lustre.org/Category:Benchmarking, 2024.

[12] Nebul Chooses Lightbits to Deliver Powerful AI Cloud. https://www.lightbitslabs.com/press-releases/nebul-chooses-lightbits-for-ai-cloud-data-platform/, 2024.

[13] NVIDIA Nsight. https://developer.nvidia.com/tools-overview, 2024.

[14] Port mirroring. https://en.wikipedia.org/wiki/Port_mirroring, 2024.

[15] Proxmox and NVMe/TCP. https://forum.proxmox.com/threads/proxmox-and-nvme-tcp.149160/, 2024.

[16] RocksDB's Benchmarking Tools. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools/, 2024.

[17] The Block Layer I/O Tracing Utility. https://linux.die.net/man/8/blktrace, 2024.

[18] The iostat Utility. https://linux.die.net/man/1/iostat, 2024.

[19] The PAPI Performance Application Programming Interface. https://icl.utk.edu/papi/, 2024.

[20] Time Series Benchmark Suite (TSBS). https://github.com/benchANT/tsbs, 2024.

[21] Tracing Framework Guide Part 1. https://spdk.io/spdk_trace/2022/09/26/tracing-library-guide-pt1/, 2024.

[22] Tracing Framework Guide Part 2. https://spdk.io/spdk_trace/2022/11/25/tracing-library-guide-pt2/, 2024.

[23] Valgrind: an Instrumentation Framework for Building Dynamic Analysis Tools. https://valgrind.org, 2024.

[24] Very slow write speed (NVME drive) on 10G network. https://serverfault.com/questions/1065820/very-slow-write-speed-nvme-drive-on-10g-network, 2024.

[25] FI-TS Delivers Disaggregated Storage Platform. https://www.intel.com/content/www/us/en/customer-spotlight/stories/finanz-informatik-technologie-customer-story.html, 2025.

[26] Lightbits Labs: Software-Defined Storage. https://www.lightbitslabs.com, 2025.

[27] Lightbits Powers One of the Worlds Largest eCommerce Platforms. https://www.lightbitslabs.com/wp-content/uploads/2023/06/LBITS-ecommerce-case-study-digital-LBCS02-2023-03.pdf, 2025.

[28] Zenlayer Accelerates Edge Cloud Services. https://www.intel.com/content/www/us/en/customer-spotlight/stories/zenlayer-customer-story.html, 2025.

[29] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher De Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, et al. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 198–204, 2022.

[30] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, page 275–287, 2023.

[31] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association.

[32] 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 (SIGCOMM'20)*, pages 662–680, 2020.

[33] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery.

[34] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*, pages 1–12, 2024.

[36] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[37] William E Cohen. Tuning programs with oprofile. *Wide Open Magazine*, 1:53–62, 2004.

[38] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[39] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/, 2024.

[40] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[41] NVM Express. Nvm express base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.1-2024.08.05-Ratified.pdf, 8 2024.

[42] NVM Express. Nvm express overview. https://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf, 8 2024.

[43] NVM Express. Nvm express tcp transport specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-TCP-Transport-Specification-Revision-1.1-2024.08.05-Ratified.pdf, 8 2024.

[44] Python Software Foundation. The python profilers. https://docs.python.org/3/library/profile.html, 2024.

[45] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, jun 1982.

[46] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, page 916–929, 2023.

[47] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. LEED: A Low-Power,

Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, page 1012–1027, 2023.

[48] 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 (SIGCOMM'20)*, pages 357–371, 2018.

[49] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–9, 2017.

[50] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. Performance characterization of nvme-over-fabrics storage disaggregation. *ACM Trans. Storage*, 14(4), dec 2018.

[51] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 861–877, Renton, WA, April 2022. USENIX Association.

[52] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*, page 33–47, 2023.

[53] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, pages 297–312, 2024.

[54] 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 (SIGCOMM'20)*, pages 404–421, 2020.

[55] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10 . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, 2020.

[56] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, pages 113–128, 2021.

[57] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: using packets for low latency network programming and visibility. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, page 3–14, 2014.

[58] Vimalkumar Jeyakumar, Mohammad Alizadeh, Changhoon Kim, and David Mazières. Tiny packet programs for low-latency network control and monitoring. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery.

[59] Yuyuan Kang, Xiangdong Huang, Shaoxu Song, Lingzhe Zhang, Jialin Qiao, Chen Wang, Jianmin Wang, and Julian Feinauer. Separation or not: On handing out-of-order time-series data in leveled lsm-tree. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 3340–3352. IEEE, 2022.

[60] David G Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain. *The Annals of Mathematical Statistics*, pages 338–354, 1953.

[61] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 535–546. IEEE, 2010.

[62] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*, 2016.

[63] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 345–359, 2017.

[64] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 31–48, Boston, MA, April 2023. USENIX Association.

[65] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[66] David D Levine. Iometer user's guide. *Intel Server Architecture Lab*, 40, 1998.

[67] 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, Boston, MA, April 2023. USENIX Association.

[68] Ming Liu. *Building Distributed Systems Using Programmable Networks*. University of Washington, 2020.

[69] Ming Liu. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, page 118–126, 2023.

[70] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, page 318–333, 2019.

[71] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 305–323, 2020.

[72] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 795–809, 2017.

[73] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 363–378, 2019.

[74] 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.

[75] Richard McDougall and Jim Mauro. Filebench. *URL: http://www. nfsv4bat. org/Documents/nasconf/2004/filebench. pdf (Cited on page 56.)*, 2005.

[76] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM'22)*, pages 753–766, 2022.

[77] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, page 106–122, 2021.

[78] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 461–477, 2023.

[79] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage*, 20(3), June 2024.

[80] J. Nagle. On Packet Switches with Infinite Storage. *IEEE Transactions on Communications*, 35(4):435–438, 1987.

[81] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 85–98, 2017.

[82] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 663–679, 2018.

[83] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*, page 16–22, 2020.

[84] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated SmartNIC Offloading Insights for Network

Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 772–787, 2021.

[85] Linux RDMA. qperf. https://github.com/linux-rdma/qperf, 2018.

[86] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro*, 30(4):65–79, 2010.

[87] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 740–755, 2021.

[88] Steve Scott, Dennis Abts, John Kim, and William J Dally. The blackwidow high-radix clos network. *ACM SIGARCH Computer Architecture News*, 34(2):16–28, 2006.

[89] Daniel Seybold and Jörg Domaschka. Benchmarking-as-a-service for cloud-hosted dbms. In *Proceedings of the 22nd International Middleware Conference: Demos and Posters*, Middleware '21, page 12–13, New York, NY, USA, 2021. Association for Computing Machinery.

[90] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 1–16, 2018.

[91] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 685–699, 2020.

[92] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. Burstable Cloud Block Storage with Data Processing Units. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 783–799, 2024.

[93] Lizhuang Tan, Wei Su, Wei Zhang, Jianhui Lv, Zhenyi Zhang, Jingying Miao, Xiaoxi Liu, and Na Li. In-band network telemetry: A survey. *Computer Networks*, 186:107763, 2021.

[94] Ej technologies. The definitive guide to jprofiler. https://www.ej-technologies.com/resources/jprofiler/help/doc/JProfiler.pdf, 2024.

[95] David L Tennenhouse, Jonathan M Smith, W David Sincoskie, David J Wetherall, and Gary J Minden. A survey of active network research. *IEEE communications Magazine*, 35(1):80–86, 1997.

[96] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(5):81–94, oct 2007.

[97] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. Understanding the Host Network. In *Proceedings of the ACM SIGCOMM 2024 Conference (SIGCOMM'24)*, page 581–594, 2024.

[98] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 449–462, 2020.

[99] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. Apache iotdb: A time series database for iot applications. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[100] Ziyang Xu, Yebin Chon, Yian Su, Zujun Tan, Sotiris Apostolakis, Simone Campanoni, and David I August. Prompt: A fast and extensible memory profiling framework. *Proceedings of the ACM on Programming Languages*, 8:449–473, 2024.

[101] 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 (SIGCOMM'18)*, pages 561–575, 2018.

[102] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[103] Minlan Yu. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review*, 49(1):11–17, 2019.

[104] Minlan Yu, Lavanya Jose, and Rui Miao. Software {Defined}{Traffic} Measurement with {OpenSketch}. In *10th USENIX symposium on networked systems design and implementation (NSDI'13)*, pages 29–42, 2013.

[105] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.

[106] 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.

A Appendix

A.1 Tracepoint Parameters

Table 3 and Table 4 show the parameters of our added tracepoints in the `nvme-tcp` and `nvmet-tcp` module, respectively.

Tracepoint	Parameters
_queue_rq	struct request *req, void *pdu, int qid, struct llist_head *req_list, struct list_head *send_list, struct mutex *send_mutex
_queue_request	struct request *req, struct nvme_command* cmd, int qid
_try_send_cmd_pdu	struct request *rq, struct socket *sock, int qid, int len
_try_send_data_pdu	struct request *rq, void *pdu, int qid
_try_send_data	struct request *rq, void *pdu, int qid
_done_send_req	struct request *rq, int qid
_handle_c2h_data	struct request* rq, int qid, int data_remain, u64 recv_time
_recv_data	struct request* rq, int qid, int len
_process_nvme_cqe	struct request *rq, void* pdu, int qid, u64 recv_time
_handle_r2t	struct request *rq, void* pdu, int qid, u64 recv_time
_try_send	struct request *rq, int qid

Table 3: Parameters of tracepoints in the module `nvme-tcp` (initiator).

Tracepoint	Parameters
_done_recv_pdu	struct nvme_tcp_cmd_pdu *pdu, int qid, long long recv_time
_exec_read_req	struct nvme_command *cmd, int qid
_exec_write_req	struct nvme_command *cmd, int qid, int size
_queue_response	struct nvme_command *cmd, int qid
_setup_c2h_data_pdu	struct nvme_completion *cqe, int qid
_setup_r2t_pdu	struct nvme_command *cmd, int qid
_setup_response_pdu	struct nvme_completion *cqe, int qid
_try_send_data_pdu	struct nvme_completion *cqe, void *pdu, int qid, int size
_try_send_r2t	struct nvme_command *cmd, void *pdu, int qid, int size
_try_send_response	struct nvme_completion *cqe,void *pdu, int qid, int size
_try_send_data	struct nvme_completion *cqe, int qid, int size
_handle_h2c_data_pdu	struct nvme_tcp_data_pdu *pdu, struct nvme_command *cmd, int qid, int size, long long recv_time
_try_recv_data	struct nvme_command *cmd, int qid, int size, long long recv_time

Table 4: Parameters of tracepoints in the module `nvmet-tcp` (target).

A.2 Triggered Events of an NVMe/TCP I/O

Figure 10 enumerates the ordered list of events that an I/O may experience throughout its lifetime in the `nvme-tcp` layer. Some events may occur multiple times, indicated by "1+". Generally, all types of I/O begin with `QUEUE_RQ` and end with `PROCESS_NVME_CQE`.

A.3 More Evaluation

Latency Breakdown on Testbed-2. In Figure 11a, we observe similar breakdown results as in Figure 6a when the number of jobs increases in testbed-2. As shown in Figure 11b, for 128KB sequential writes, "S3-S5(S+C)" becomes the dominant component when the job count exceeds 16, indicating

read I/O

Event Name	Num
QUEUE_RQ	1
QUEUE_REQUEST	1
TRY_SEND	1
TRY_SEND_CMD_PDU	1
DONE_SEND_REQ	1
HANDLE_C2H_DATA	1
RECV_DATA	1+
PROCESS_NVME_CQE	1

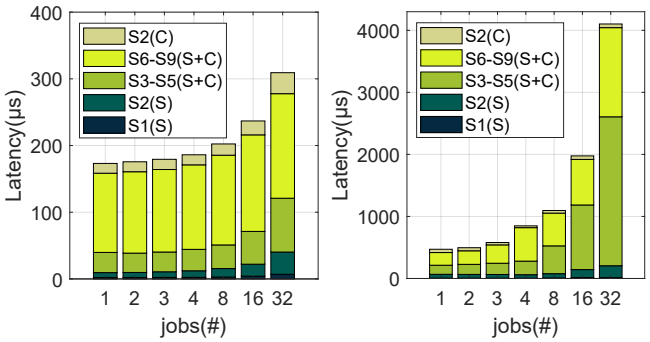
write I/O without inline data

Event Name	Num
QUEUE_RQ	1
QUEUE_REQUEST	1
TRY_SEND	1
TRY_SEND_CMD_PDU	1
DONE_SEND_REQ	1
HANDLE_R2T	1
QUEUE_REQUEST	1
TRY_SEND	1
TRY_SEND_DATA_PDU	1
TRY_SEND_DATA	1+
DONE_SEND_REQ	1
PROCESS_NVME_CQE	1

write I/O with inline data

Event Name	Num
QUEUE_RQ	1
QUEUE_REQUEST	1
TRY_SEND	1
TRY_SEND_CMD_PDU	1
TRY_SEND_DATA	1+
DONE_SEND_REQ	1
PROCESS_NVME_CQE	1

Figure 10: Event series of different types of I/O in the `nvme-tcp` layer. The event name corresponds to the tracepoint name, omitting the "nvme_tcp_" prefix.



(a) 4K Random Read. **(b) 128K Sequential Write.**

Figure 11: Latency breakdown for the 4K random read and 128K sequential write as increasing the number of jobs on testbed-2. We report the five most time-consuming parts. S1(S) refers to the time an I/O stays at the S1 stage on the submission path. S2(C) captures the time spent at the S2 stage on the completion path. S3–S5(S+C) aggregates the time spent in the S3, S4, and S5 stages across the submission and completion paths.

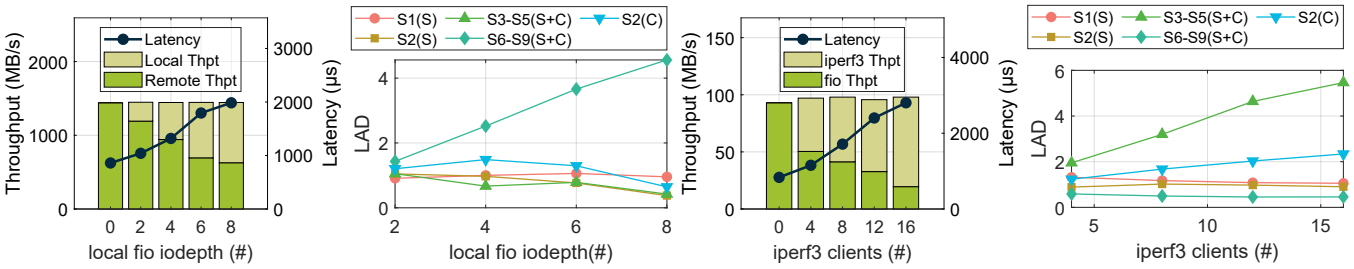
workload	without `ntprof`		with `ntprof`			
	lat(µs)	bw(MB/s)	lat(µs)	bw(MB/s)	imem(MB)	tmem(MB)
4k read	449.26	2202.91	460.57	2099.11	12	10
4k write	66.75	883.91	69.47	835.86	1	7
128k read	1136.72	3493.03	1136.24	3491.88	13	3
128k write	338.75	1439.89	337.44	1439.17	1	17

Table 5: Performance comparison with and without `ntprof`. imem represents the memory usage increase on the initiator side, while tmem indicates the increase on the target side. lat and bw denote the average latency and bandwidth, respectively.

that the queue builds up during data transmission through the network stack. When there are 32 jobs, 2400.4µs is spent out of 4102.2µs of end-to-end latency.

Use Case 3 Results. Figure 12 presents our hardware bottleneck localization results.

Experimental setup for `ntprof` overhead analysis. We configure four sets of `fio` experiments to analyze the `ntprof` overheads: (a) 4K random read, with 16 jobs and an iodepth of 16; (b) 4K sequential write, 4 jobs and an iodepth of 4; (c) 128K sequential read, with 4 jobs and an iodepth of 8; (d) 128K sequential write, with 2 jobs and an iodepth of 2. Each job is assigned to a different core. These parameters were selected based on experiments designed to prevent any



(a) Throughput and Latency. **(b) LAD.** **(c) Throughput and Latency.** **(d) LAD.**

Figure 12: Hardware bottleneck localization. (a) and (b) depict the "slow SSD" case, where we run one extra `fio` job on the target and increases the iodepth from 2 to 8. (c) and (d) show the "slow NIC" case, where we run `iperf3` server on the initiator side and clients on the target sides, and we increase the TCP flows from 4 to 16. LAD=Latency Amplification Degree.

component in the I/O path from becoming a bottleneck, while applying sufficient load to stress the system. The `fio` tests ran for 60 seconds, during which total memory consumption was recorded every second, and an average was calculated at the end of each run.