

zcIO: Enabling Transparent Zero-copy for NVMe/TCP

Junghan Yoon*, SeungYeon Shin*, Jonghyeon Park, Jaehyun Hwang†, and KyoungSoo Park

Seoul National University *Sungkyunkwan University*†

Abstract

NVMe-over-Fabrics via TCP (NVMe/TCP) has become a datacenter standard for disaggregated storage, yet it remains severely CPU-bound in high-speed environments. While zero-copy I/Os could alleviate the load, traditional zero-copy TCP stacks often fail to meet the strict page-alignment requirements necessary for efficient hardware-to-software handoffs.

We present zcIO, a novel Linux NVMe/TCP stack that achieves seamless zero-copy I/Os through record-aware networking. By aligning TCP payloads with the NVMe PDU formats and leveraging large MTUs with header-data split capabilities, zcIO ensures that incoming DMA transfers are perfectly aligned with memory page boundaries. This precise alignment allows zcIO to replace expensive memory copies with low-overhead virtual memory page remapping. zcIO provides a transparent zero-copy data path requiring no application modifications and ensures safe page reclamation. Our evaluation shows that zcIO improves remote storage performance by up to 2.3×, reaching throughput of up to 67.6 GB/s (540.8 Gbps) with 15 CPU cores.

1 Introduction

Disaggregated storage has become increasingly prevalent [14, 15, 26, 58] in modern datacenters (DCs [1, 2]), where compute nodes access remote storage resources over high-speed network fabrics. By decoupling compute from storage, this architecture enables independent resource scaling, higher utilization, and enhanced fault tolerance—if a compute node fails, its associated storage can be seamlessly remounted to a healthy node, ensuring continuous data availability.

NVMe-over-Fabrics (NVMe-oF [38]) serves as the foundational protocol for disaggregated storage [50, 57], utilizing either TCP (NVMe/TCP) or RDMA (NVMe/RDMA) as the underlying transport. While NVMe/TCP [40] is often preferred for its operational simplicity and lower maintenance overhead, NVMe/RDMA [39] typically delivers superior performance. By leveraging hardware-based offloading and zero-copy data transfers, NVMe/RDMA minimizes CPU intervention, bringing a performance advantage that becomes increasingly pronounced as payload sizes grow.

One approach to narrowing this performance gap is the employment of a zero-copy TCP stack [12, 21, 49, 55]. Implementing zero-copy in the transmit (TX) path is relatively straightforward [49,55] as it is similar to direct I/O—it directly sends user pages to NIC by pinning. However, zero-copy in the receive (RX) path presents a significant challenge [12, 21] as incoming packets are DMAed across different kernel pages that must be remapped to user's virtual memory space. Unfortunately, this remapping requires strict page-level alignment. When packet headers or metadata cause payloads to deviate from page boundaries, the system must revert to a conventional copy-based path. Consequently, existing zero-copy TCP stacks often fail to provide meaningful performance gains in production environments.

This difficulty stems from a fundamental mismatch between *page*-based virtual memory management and *packet*-based data transfer network protocols. Virtual memory requires page-aligned data sharing between the kernel and applications, whereas network protocols handle packet-level content reassembly, where standard packet sizes do not align with page boundaries. Thus, packet RX events would rarely result in payloads perfectly aligned for page remapping. RDMA bypasses this issue because the RoCEv2 NIC (RNIC) hardware controller manages DMA with in-order payload placement directly into application memory. In contrast, standard TCP lacks this specialized hardware steering; because payloads DMAed from NIC often cannot be placed directly into the final destination, the system must perform expensive, CPU-intensive data copies to achieve alignment.

In this paper, we address these challenges with zcIO, a novel zero-copy NVMe/TCP stack architecture. The key idea of zcIO is to leverage *record-aware networking* that aligns each packet payload with the application-level data format.

*They contributed equally to this work.

Although record-aware networking is often undesirable for high performance due to its perceived inflexibility, we find it remarkably well-suited for intra-DC remote storage access. Specifically, zcIO aligns the NVMe block content size in the Protocol Data Unit (PDU) payload of individual packets to the memory page size. Each NVMe data packet carries block data of exactly 4 KB or 8 KB as payload, so each packet RX would result in one or two pages that are remappable into application virtual memory. This design is made possible by three key observations. First, the NVMe disk block size aligns with the common 4KB memory page size [1]. Thus, the I/O granularity eliminates the need for buffering network-transferred data. Second, intra-DC environments can employ Maximum Transmission Units (MTUs [46]) significantly larger than the standard 1500 B limit. For example, 9000 B jumbo frames can easily accommodate an NVMe PDU command header alongside up to 8 KB of disk blocks as payload. Third, modern NICs support split DMA of packet headers and payloads [7, 24, 35]. By leveraging this capability, one can cleanly separate NVMe PDU data from TCP/IP headers in distinct memory regions.

However, introducing zcIO presents several significant challenges. First, standard header-data split is insufficient for isolating block data, as NICs typically treat the entire NVMe PDU (including its header) as the packet payload. To achieve true zero-copy, the system must isolate only the NVMe block content into memory pages destined for remapping. We address this by modifying the sender-side stack to encapsulate the NVMe PDU header within TCP header options (§4). By shifting the PDU header into protocol metadata, we ensure the NIC recognizes only the block content as the payload. Complementing this, we configure receive-side buffering to ensure DMA arrives in page-sized chunks (§5). Second, the receiver must transition these RX pages into the I/O destination without violating protocol semantics. Instead of copying data, zcIO reuses the received pages by either swapping the request-buffer pages with the RX pages or remapping page tables to map the RX pages into the target virtual address space (§6). To minimize overhead, zcIO employs deferred TLB batching, consolidating flushes until I/O completion rather than flushing per page. For partially-aligned requests, zcIO zero-copies the aligned prefix and falls back to conventional copies only for the remaining tail bytes. Third, adopting explicit reference counting for page reclamation would require invasive application modifications. To maintain transparency, we implement implicit page reclamation: pages are automatically recycled when the application either overwrites them with subsequent I/Os or releases the memory mapping.

We implement zcIO in the Linux kernel (v6.11) using 3566 lines of C source code. In remote disk access benchmarks, zcIO outperforms stock Linux and SPDK [52] by up to 2.30× and 1.70×, respectively. Furthermore, zcIO improves Nginx throughput by up to 1.33×, while delivering performance gains of 1.45× for MLPerf Storage and 1.58× for RocksDB.

[1] 512 bytes of legacy size is possible but not popular in high-speed DCs.

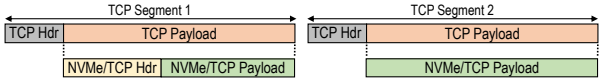


Figure 1: NVMe/TCP PDU layout on TCP segments

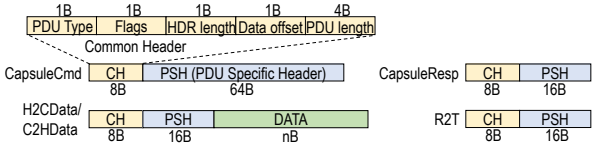


Figure 2: PDU formats

2 Background & Motivation

2.1 NVMe-oF and NVMe/TCP

NVMe-oF [38] extends the NVMe interface beyond the PCIe bus, allowing a host initiator to access a remote target over a network while preserving the standard block-device abstraction. NVMe-oF supports multiple transport bindings, including NVMe/TCP [40] and NVMe/RDMA [39]. In this work, we focus on NVMe/TCP, whose kernel-based data path offers broad deployability but incurs additional CPU overhead, especially when data copies are involved.

TCP delivers application data as a continuous byte stream, where each segment comprises a TCP header followed by a TCP payload. NVMe/TCP operates atop this layer by encoding protocol messages into Protocol Data Units (PDUs). Each PDU starts with an **NVMe/TCP PDU Header**; for data-transfer operations, this is followed by an **NVMe/TCP PDU Payload** containing disk block data. As illustrated in Figure 1, an NVMe/TCP PDU is carried within the TCP payload.

Because TCP is byte-stream oriented, PDU boundaries are not necessarily aligned with TCP segment boundaries. A single PDU may span multiple segments, or conversely, multiple PDUs may be coalesced into a single segment. Accordingly, the receiver reconstructs individual PDUs by parsing the stream and utilizing the length fields within the PDU headers to delineate each NVMe/TCP PDU Payload region.

In the common read/write path, NVMe/TCP uses five PDU types: `CapsuleCmd` (command submission), `CapsuleResp` (completion), `C2HData` (read data from target to host), `H2CData` (write data from host to target), and `R2T` (ready-to-transfer; target request for subsequent `H2CData`). `CapsuleCmd`, `CapsuleResp`, and `R2T` are control PDUs that are typically small in size, whereas `C2HData` and `H2CData` are data PDUs whose `DATA` field carries the actual disk block content and therefore dominates network traffic for large I/Os.

As shown in Figure 2, every PDU starts with a mandatory header (`CH+PSH`), optional integrity fields (`HDGST/DDGST`), and a `DATA` field for NVMe/TCP PDU payload. Data PDUs (`C2HData/H2CData`) use a fixed 24 B header (8 B common,

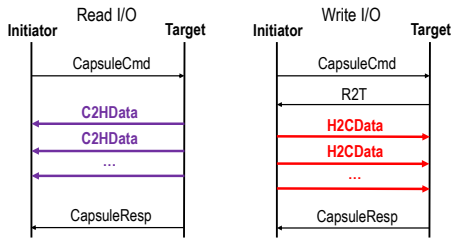


Figure 3: PDU exchange sequence for read/write operations

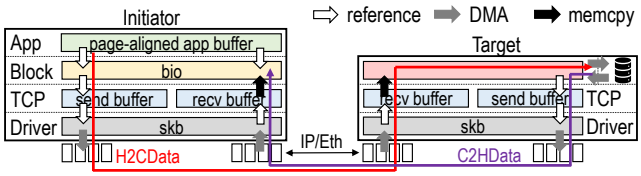


Figure 4: Data movement in `C2HData`/`H2CData` PDU

16 B specific). Figure 3 illustrates the PDU exchange during an I/O operation. A read involves a `CapsuleCmd`, multiple `C2HData` PDUs, and a final `CapsuleResp`. A write uses `H2CData` PDUs; if it exceeds the in-capsule limit, the target sends `R2T` PDUs to request specific offset ranges. The Linux NVMe/TCP receive path is not a simple socket buffer copy. The driver must parse headers, track request offsets, and place payload bytes directly into destination pages, such as the page cache for buffered I/O or pinned user pages for direct I/O.

2.2 Data Copies in Protocol Processing

We identify data copies in the NVMe/TCP stack implementation on Linux and quantify their overhead.

Linux receive socket buffers. In Linux, incoming packets are represented as socket buffer (`sk_buff` or `skb`) objects. These structures store protocol metadata and reference the received payload—often as page-backed fragments populated via NIC DMA. As the generic internal representation for all networking data within the kernel, `skb` facilitates data movement across the stack. We provide a detailed analysis of the `skb` architecture and the layout of NVMe/TCP PDU bytes across `skb` regions in §5.

Write operation. The red line in Figure 4 illustrates the data movement during `H2CData` PDU processing. On the initiator side, the TX path for `H2CData` can leverage zero-copy transmission. For `O_DIRECT` writes, these pages backing the request may correspond to pinned user pages. Using mechanisms such as `MSG_ZEROCOPY`, the TCP stack can transmit these pages without additional copying, allowing the NIC to DMA directly from user memory. As a result, the initiator software stack avoids extra copies of the write payload.

In contrast, the NVMe/TCP target incurs a memory copy on the receive path. `H2CData` packets are first DMAed into pre-allocated socket buffers (`skbs`). In Linux, `nvmet-tcp` allocates request pages for incoming write data, receives `H2CData` through the generic socket receive path, and submits

Bucket	Read(%)	Write(%)
Payload copy	**41.99**	**40.52**
Network stack (TCP/IP, driver)	19.18	19.63
Kernel overhead (sched, lock, syscall, etc.)	17.07	17.99
NVMe/TCP & block I/O	10.97	4.96
Buffer & page management	9.36	16.88
User space	1.43	0.02

Table 1: CPU breakdown of the NVMe/TCP initiator/target under random read/write workloads (9000 B MTU)

the resulting `bios` to the backend block device. As a result, the conventional target path still copies data from `skb`-backed receive buffers into request pages before backend write submission. This copy is necessary because the received `skb` data cannot be reused as-is as backend write payload: it may include protocol headers, arbitrary offsets, or partial-page regions, and must therefore be reorganized into the layout expected by the backend write path.

Read operation. The violet line in Figure 4 illustrates the data movement during `C2HData` PDU processing. While the target can transmit data without intermediate copies (e.g., via `kernel_sendpage()`), the initiator in standard Linux implementations necessarily incurs a memory copy. As with the write path, incoming packets are first DMAed into socket buffers. Because the final destination pages can be identified only after protocol processing, the `C2HData` payload must be copied from the TCP receive buffers into the request's pinned user pages. Thus, the standard initiator read path includes at least one unavoidable memory copy.

Data copy overhead. Our profiling confirms that these data copies dominate the receive (RX) path. Table 1 quantifies the CPU time distribution for 128 KB random reads/writes at a saturated initiator/target. The dominant cost is payload copy, which alone accounts for **41.99%/40.52%** of total CPU cycles. This overhead exceeds the combined cost of network protocol processing and the NVMe/TCP plus block-I/O control path. In other words, memory copying is the single largest CPU consumer in the NVMe/TCP stack, significantly outweighing protocol logic. This bottleneck motivates our focus on eliminating the redundant data copy.

2.3 Limitations of Existing Approaches

DPDK and AF_XDP. Modern kernel-bypass techniques such as `DPDK` [9] and `AF_XDP` [16] eliminate memory copies by maintaining packet buffers (e.g., `mbufs` or `umem` chunks) entirely in user space. By bypassing the kernel TCP/IP stack, packets received via DMA are processed directly, allowing applications to reuse original buffers without intermediate copies. While this enables a theoretically optimal zero-copy NVMe/TCP data path, it requires re-implementing a significant portion of the transport and storage layers—including TCP state machines and block I/O semantics—within user space. This re-engineering substantially increases architectural complexity and breaks compatibility with standard ker-

nel abstractions. Thus, most production NVMe-oF initiators continue to rely on Linux kernel modules for their seamless integration with established OS ecosystems.

TCP zero-copy RX. The Linux kernel provides `mmap`-based TCP zero-copy RX mechanisms [12, 21], such as `TCP_ZEROCOPY_RECEIVE`, which can avoid data copying by remapping kernel RX data pages into user space. However, blindly applying this mechanism to NVMe/TCP is insufficient, as the zero-copy TCP stack is oblivious to the application-level protocol formats. Since the NVMe/TCP protocol prepends headers to block data within the TCP payload, the block data typically starts at a mid-page offset. Because the block layer requires block data to be page-aligned for direct `bio` attachment, this misalignment necessitates a copy, defeating the purpose of the zero-copy mechanism.

`io_uring` zero-copy RX. Recent Linux kernels also support zero-copy RX through `io_uring` [29,43]. Unlike remapping-based TCP zero-copy, this design adopts a partial kernel-bypass model: a dedicated hardware RX queue is reserved for the application, and hardware flow steering (e.g., Flow Director) directs matching traffic to that queue. Then, the kernel places TCP payloads directly into user-provided buffers, eliminating both `skb`-to-user copies and page remapping overhead. However, this suffers from the same limitation as the TCP zero-copy RX—the NVMe/TCP protocol handling will be shifted to the application.

SPDK. Importantly, user-space storage frameworks such as SPDK [41, 52] do not fundamentally circumvent this data movement problem. Because `SPDK` typically relies on the kernel TCP stack for NVMe/TCP, it inherits the same `skb`-based data path. Even if `io_uring` zero-copy RX were utilized beneath SPDK, the NVMe/TCP header placement and block-layer alignment constraints would continue to require memory copies for data realignment.

3 Overview of zcIO

zcIO eliminates memory copies when handling NVMe/TCP disk I/O operations. Our key mechanism is to align the NVMe data PDU payload within each TCP segment to page boundaries. Since modern NVMe I/Os are typically issued in multiples of the standard memory page size (4 KB), zcIO achieves zero-copy delivery by ensuring that (a) disk blocks are prepared and transmitted at page granularity, and (b) the received NVMe/TCP disk blocks are DMAed by the NIC directly into the correct page-aligned memory region. The resulting skb fragment pages carrying the I/O payload are then remapped into a user-level buffer for read I/Os. For write I/Os, the corresponding fragment pages are handed off to the block write path without a memory copy. All changes are made transparently within the kernel's NVMe and TCP modules, requiring no modifications to applications or standard I/O interfaces.

3.1 Challenges and Ideas

There are multiple sources of misalignment between the sender's NVMe/TCP data stream and the receiver's page-based I/O buffers [2].

C1: Sender-side misalignment. TCP is a byte-stream protocol. A sender may segment data without awareness of disk block boundaries. Simply tuning the MTU to 4 KB [12] does not ensure that TCP segment boundaries coincide with page-aligned block boundaries, as the TCP payload carries an NVMe PDU header, with the actual block data following it. Furthermore, the presence of control NVMe PDUs can shift the offset of subsequent data PDU content, further complicating alignment.

I1: Protocol-aware packetization. To address this issue, we propose *protocol-aware TCP segmentation* for TX that guarantees each data packet carries a multiple of pages filled with pure block data. The sender performs segmentation with an awareness of NVMe/TCP block boundaries, ensuring that block data within each segment is aligned to disk block granularity. The detailed design is presented in §4.

C2: Receiver-side misalignment. Even if the sender transmits packets containing page-aligned block data, alignment can still be broken at the receiver. In a conventional NIC receive path, the entire Ethernet frame — including Ethernet, IP, TCP, and NVMe/TCP headers — is DMAed into a page starting at the page boundary. As a result, the actual block data begins at a non-zero offset within the page. This misalignment necessitates subsequent byte-reshaping or memory copies when delivering data to block-layer buffers. The header-data split in modern NICs [7, 24, 35] does not help because it leaves the NVMe/TCP header in the payload.

I2: Page-aligned RX DMA. To eliminate this source of misalignment, we propose a *protocol-aware header-data split* mechanism. This design separates L2-to-L4 and L5 (NVMe/TCP) headers into a distinct buffer while steering the pure block data to the beginning of a page-aligned DMA region. With this design, each `skb` fragment encapsulates a perfectly page-aligned block of data, eliminating the need for intermediate byte-reshaping. The details are in §5.

With **I1** and **I2**, zcIO eliminates write path memory copies by repurposing received `skb` fragment pages to serve as data pages for backend write I/O. The target achieves this through page handoff, replacing the request's original page references with the received network pages (§6.1).

C3: Transparent management of disk block pages. The read path presents a different challenge. The initiator must deliver the received block data pages into application buffers referenced by `bio` pages. In this case, the destination pages are already allocated and actively managed—either as page cache entries for buffered reads or pinned user pages for direct

[2] In read operations, the target acts as the sender and the host as the receiver; for write operations, these roles are reversed.

I/O. The receiver therefore cannot simply swap in references to `skb` fragment pages, as the existing memory mappings must be maintained.

I3: Lightweight page remapping. To overcome this limitation, we revitalize a classical technique: *virtual page remapping* [11, 56]. Instead of copying data between buffers, we remap the `bio` pages to the received `skb` fragment pages, enabling zero-copy delivery without physical data movement. The specific mechanism, safety considerations, block-layer integration, and overhead amortization are detailed in §6.2.

3.2 Assumptions

zcIO targets controlled datacenter storage networks with large TCP transfers, including environments that favor TCP over lossless RoCE to avoid operational issues such as PFC storms, head-of-line blocking, and deadlocks. It remains transparent to applications: standard block and file I/O interfaces are unchanged, as all optimizations are confined to the kernel NVMe/TCP data path.

Full-stack deployment. Enabling TX/RX page alignment requires coordinated kernel and driver updates at both the initiator and target, as well as NIC support for header-data split DMA. This requirement is practical in typical NVMe/TCP deployments, where both endpoints are often operated within a single administrative domain, such as a cloud service provider managing both compute and storage nodes. Although our prototype uses Linux and NVIDIA ConnectX-7 NICs [37], the design is not vendor-specific.

Large MTU support. zcIO requires that the end-to-end path between the initiator and the target supports an MTU larger than the logical block size (4 KB), ensuring that at least one full block can be carried within a single TCP segment. In practice, modern cloud infrastructures commonly utilize 9000 B jumbo frames for intra-datacenter traffic, a configuration widely supported by commodity NICs and switches. We assume a 9000 B MTU as the default configuration. Under this setup, a single TCP segment can carry up to 8 KB of aligned block data after accounting for protocol headers.

4 Protocol-aware TCP Segmentation

4.1 Strawman Approaches

Aligning NVMe PDUs with packet payloads at the TCP sender cannot be easily achieved through simple configuration. One might attempt a fixed MTU (e.g., 4,160 B) to match a 4 KB page of block data plus headers. However, this fails because control PDUs without block data can shift alignment, and subsequent data PDUs would lose header-to-packet synchronization. Even for data PDUs, the packet will stop carrying the PDU header from the second block.

Alternatively, using `TCP_NODELAY` with a large MTU to force immediate transmission of aligned blocks is also insufficient. While it disables Nagle's algorithm [20] to prevent small-write buffering, it does not enforce fine control over segmentation. If the NVMe/TCP layer invokes send functions faster than the TX path drains the socket, data accumulates. Once this backlog exceeds the Maximum Segment Size (MSS [6] = MTU - TCP/IP header size), the TCP stack performs MSS-based segmentation, inevitably producing segments misaligned with PDU and page boundaries.

4.2 Protocol-aware TCP Segmentation

To provide a strict alignment guarantee, the TCP stack must be modified to perform NVMe/TCP-aware segmentation. While embedding upper-layer protocol logic into the transport layer may seem to violate traditional layering principles, the Linux kernel already employs similar Layer 5 (L5) awareness for features like kTLS offloading [44, 53].

We identify two primary mechanisms that cause PDU misalignment and develop strategies to mitigate them.

Window-based constraints. The first source of misalignment is window-based transmit control, constrained by the sender's available congestion window (*cwnd*), the peer's advertised receive window (*rwnd*), and the socket's send buffer limits. These variables collectively determine the maximum payload size for any given transmission opportunity. Our design conservatively caps the transmit length to the nearest lower NVMe/TCP PDU or block data boundary. Instead of filling the window to its exact byte limit, we truncate the payload to ensure that no NVMe block data is split across segments. This approach preserves TCP semantics, as it simply transmits fewer bytes than are allowed. If the send buffer contains only a partial PDU or a partial disk block, transmission is deferred until sufficient buffer space becomes available. However, ACKs would continuously drain the send buffer, which allows the remaining data to be enqueued and transmitted in an aligned manner [3]. Therefore, conservatively capping window-limited transmissions at PDU or page boundaries does not violate TCP semantics.

Socket buffer (`skb`) boundaries. The second source of misalignment arises from the internal organization of the TCP send buffer, which consists of a linked list of socket buffers (`skb`s). Each `skb` typically holds up to 64 KB of payload and TCP segmentation operates within the context of a single `skb`. Because `skb` allocations are independent of NVMe PDU sizes, a single PDU or page-aligned block often straddles the boundary between two `skb`s, resulting in an inevitable split during transmission. zcIO repairs these misalignments by "pulling" the trailing portion of a PDU or page-aligned block from the subsequent `skb` into the current one. This is

[3] If some ACKs get lost, retransmission of data packets would eventually trigger the ACK retransmission.

achieved via a shallow copy—reusing data references rather than duplicating the actual payload—to append the missing bytes to the tail of the current `skb`.

A potential risk of this mechanism is delayed transmission if the upper layer submits non-aligned data. However, since NVMe/TCP modules generate block data strictly in PDU and page-aligned units, partial submissions do not occur in practice. Thus, our repair mechanism ensures alignment without introducing additional latency or blocking beyond standard TCP behavior.

Benefiting from TSO. To minimize the overhead of protocol-aware TCP segmentation, zcIO actively exploits hardware TSO. The kernel TCP stack typically constructs a large `skb` so that the NIC performs fixed-MSS segmentation with negligible CPU cost. While protocol-aware segmentation must respect NVMe/TCP PDU and page-aligned block boundaries, only the initial segment of a large data PDU requires software segmentation. For example, given a 128 KB `C2HData`/`H2CData` PDU and 9000 B MTU, the first segment contains a 24 B NVMe/TCP header followed by an 8 KB page-aligned block (8,216 B total payload). All subsequent segments consist strictly of 8 KB page-aligned block data without headers and can therefore be segmented using standard hardware TSO [4]. Since the vast majority of the bytes are segmented using unmodified hardware offload, the extra overhead of protocol-aware segmentation is negligible.

5 Protocol-aware Packet Placement

In a conventional NIC RX path, the entire Ethernet frame—comprising L2/L3/L4 headers and the NVMe/TCP PDU (i.e., L5) header—is DMAed into RX buffers without protocol-aware separation of the NVMe/TCP header from the block data payload. As a result, block data begins at a non-zero offset within the receive buffer, breaking the page alignment required for zero-copy delivery to the block layer. Page-aligned RX requires two conditions: (i) header bytes must be steered away from block data pages, and (ii) each DMA fragment containing block data must occupy its own 4 KB page, without sub-page packing or tail-room spillover.

5.1 L5 Protocol-aware Header-Data Split

Network packets in Linux are represented by the socket buffer (`sk_buff` or `skb` structure), which partitions packet storage into a linear buffer and a non-linear buffer to balance processing efficiency with memory utilization. The linear buffer provides a contiguous memory region for cache-friendly access to frequently-used protocol headers. On the receive path, headers—and occasionally an initial payload prefix—are stored here to enable fast, in-place parsing and metadata updates. In contrast, the non-linear buffer stores the remaining bulk payload as page-backed fragments described by `skb_shared_info`. Each fragment references a specific portion of a physical page, typically a DMA-filled page provided by the NIC driver, using a `(page, offset, length)` descriptor. This hybrid structure avoids the overhead of large contiguous allocations and facilitates zero-copy delivery, as the majority of the payload resides in reusable, page-backed buffers.

However, this split storage alone is insufficient for zero-copy NVMe/TCP RX processing. Without strict header-data separation, block data may be scattered across both the `skb` linear and non-linear buffers as shown in Figure 5a. Modern NICs [7, 24, 30, 35] partially address this through hardware-based header-data split, which places L2–L4 headers in the linear buffer while directing the TCP payload to page-backed fragments. This separates data at the transport boundary, but it does not solve the NVMe/TCP case—the NIC treats the NVMe/TCP PDU header as ordinary TCP payload and co-locates it with the block data within the fragment pages as shown in Figure 5b.

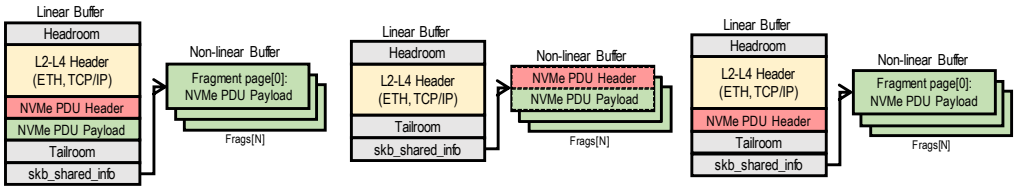
L5 header-data split. To overcome this limitation, zcIO performs *protocol-aware* header-data split. The key idea is to have the NIC treat the NVMe/TCP PDU header as part of the TCP header. To do so, the sender encodes the NVMe/TCP PDU header in the TCP option field by increasing the TCP data offset (`doff`). As a result, the receiving NIC places the L2–L4 headers together with the encapsulated NVMe/TCP PDU header in the `skb` linear buffer, while directing only the NVMe/TCP PDU payload to the page-backed fragments (Figure 5c). This confines protocol metadata to the linear buffer and leaves the fragment pages to contain only block data, which is required for efficient zero-copy delivery.

TCP option space availability. NVMe/TCP data PDUs (such as `H2CData` and `C2HData`) carry a 24 B header, which fits within the 40 B of TCP option space when encapsulated via `doff` adjustment. Even with the TCP timestamp (12 B including padding) and the NVMe/TCP header digest (4 B) enabled, the total remains within the available space. However, Selective Acknowledgment (SACK [45]) introduces a constraint. Since a single SACK block spans 8 B (plus 2 B for the option kind and length), it cannot co-exist with a fully encapsulated NVMe/TCP header in the same segment. In such cases, zcIO temporarily disables the encapsulation for those specific segments, falling back to the conventional copy-based delivery path. This fallback occurs only when packet loss during bidirectional block-data transfers triggers SACK generation at the receiver, and it affects only those data segments. Since NVMe/TCP typically operates in low-loss data centers or cloud environments [1, 2], such fallback is rare in practice.

5.2 Page-aligned Receive Stride

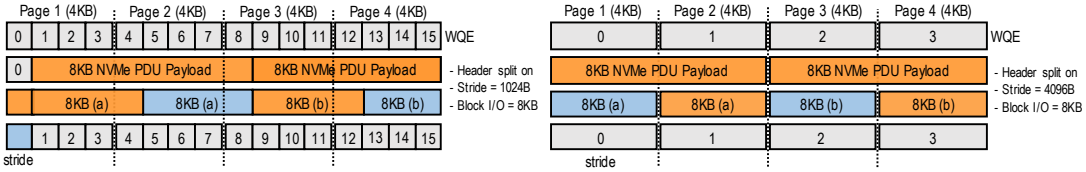
Protocol-aware header-data splitting alone does not guarantee page alignment due to *strided* RX data placement.

[4] Modern NICs execute GSO/TSO based on the specified `gso_size`, generating uniformly sized segments that adhere to TCP semantics.



(a) No header-data split **(b)** L4 header-data split **(c)** Protocol-aware header-data split

Figure 5: Comparison of header-data split mechanisms in the NVMe/TCP receive path.



(a) Sub-page stride (1 KB): payload crosses page boundaries **(b)** Page-sized stride: payload lands as clean 4 KB pages

Figure 6: Impact of RX stride size on payload page alignment.

Strided RX. Many high-performance NICs employ strided RX placement (e.g., NVIDIA Multi-Packet WQE [42]), where a large receive buffer is posted and DMA is performed in fixed-size strides across multiple incoming packets [10, 42]. While this amortizes descriptor management and DMA setup costs, it inevitably causes multiple packets to share a single physical page. If a small packet (e.g., a control PDU or a packet from an unrelated flow) occupies the initial part of a page, subsequent block data DMA begins at a non-zero offset, instantly breaking page alignment. With sub-page strides (e.g., 1 KB), an 8 KB block payload spans several strides and frequently crosses 4 KB page boundaries (Figure 6a). Because multiple strides are packed into a single physical page, data from different packets can coexist within that page. This "contaminates" the block data pages with unrelated metadata or interleaved traffic, making it impossible to map those pages directly into user-space buffers without expensive re-alignment copies.

Page-sized stride. To prevent such contamination, we configure the RX stride to the OS page size (4 KB) and align the stride base so that each DMA fragment occupies a full page (Figure 6b). This approach is consistent with existing practice; for example, the ConnectX NIC driver uses page-sized strides in its XDP path [16] to avoid multi-packet packing within a page. Combined with protocol-aware header-data splitting, this configuration ensures that block data for large `H2CData`/`C2HData` PDUs arrives as clean, page-aligned 4 KB pages. Control PDUs and non-aligned tails are handled through the conventional path. Together, these mechanisms produce a stream of page-aligned block data pages suitable for zero-copy delivery without modifying NIC hardware.

5.3 Interaction with GRO and Hardware LRO

Our RX scheme transparently benefits from GRO [8, 54] and LRO [13] while maintaining zero-copy data placement.

TCP reassembly and GRO. TCP reassembly reconstructs the byte stream according to sequence numbers without modifying fragment-page boundaries. Since alignment is enforced at a 4 KB DMA granularity, reassembly simply concatenates page-aligned fragments, preserving their internal offsets even under out-of-order arrival or retransmission. Essentially, TCP reassembly reorders `skb`s rather than modifying the internal layout within each `skb`, thereby preserving alignment. Similarly, Generic Receive Offload (GRO [8, 54]) merges adjacent segments in software by coalescing `skb` fragments. Because these fragment pages are already clean and page-aligned, GRO aggregation maintains alignment by design.

LRO. Large Receive Offload (LRO [13]) aggregates packets in hardware prior to DMA. If consecutive NVMe/TCP PDUs are pipelined, merging segments could, in principle, insert a subsequent PDU header into the middle of a block data page, breaking alignment. However, modern LRO (or hardware GRO [31]) implementations typically avoid merging segments that carry distinct TCP header metadata or options. Since our design encodes NVMe/TCP headers as TCP options, each PDU-bearing segment contains unique header information and is bypassed by hardware LRO merging. This approach preserves both transport semantics and page alignment.

Partial payloads. If a payload violates the alignment constraints—for instance, due to middlebox-induced fragmentation [5]—zcIO transparently falls back to the standard copy-based path. This fallback ensures functional correctness while maintaining a performance floor equivalent to the baseline Linux NVMe/TCP stack, even under sub-optimal network conditions.

6 Transparent Disk Block Delivery

We now describe how zcIO exposes page-aligned block data to the requester. Unlike prior solutions that depend on application-managed buffers or shared-memory interfaces, our design aims to seamlessly integrate with the existing block I/O subsystem and the standard NVMe/TCP stack. zcIO op-

[5] Packet loss, out-of-order delivery, and retransmissions do not by themselves violate the alignment.

portunistically enables zero-copy only when both the RX payload and the user buffer layout satisfy the alignment requirements for whole-page substitution, a common case in block storage. For unaligned or non-page-sized requests, zcIO falls back to the copy-based path, ensuring data integrity and correctness. The next two subsections detail the target-side write path and the initiator-side read path, respectively.

6.1 Zero-copy Write via Page Handoff

For write I/Os, the received payload serves as input to the subsequent backend disk writes. To avoid memory copies, zcIO hands off the RX fragment pages to the block I/O layer so that the backend write consumes those pages directly. In the standard Linux implementation, incoming payload is copied into the request buffer within the generic socket receive path as packets arrive. Because this copy is performed within the generic socket receive path, the NVMe/TCP write path cannot make the copy-or-handoff decision at that point. As a result, zero-copy handoff cannot be implemented solely in the NVMe/TCP write path.

zcIO addresses this problem by decoupling fragment discovery from page handoff. As `H2CData` PDUs arrive, the RX path identifies disk write payloads that land in page-aligned regions and marks the corresponding fragment pages as deferred handoff candidates. After the NVMe/TCP write request is fully received, zcIO replaces the corresponding request-buffer pages with the candidate fragment pages so that the backend write can consume them directly while preserving the expected buffer layout.

Reclaiming fragment pages for zero-copy writes is more deterministic than for reads as their lifecycle is tied directly to the backend I/O. These pages remain occupied until the backend write operation completes. So, we only need to ensure that handed-off RX fragment pages are not recycled back into the NIC driver's per-core page pool, which supplies packet-DMA buffers, prior to the completion of backend DMA. To guarantee this, zcIO increments the page reference count for each fragment while the write is in-flight; this extra reference is released only upon I/O completion, ensuring the pages remain valid throughout the entire data path.

6.2 Zero-copy Read via Page Remapping

For read I/Os, the received data must become visible in the requester-specified destination pages without memory copies. zcIO achieves this by remapping the RX pages directly to the application's virtual address space. Our current implementation targets direct I/O [6], a popular choice in high-performance storage systems such as RocksDB and MySQL, where the block I/O layer exposes pinned, page-backed user buffers. The pinned pages allow zcIO to replace them with RX pages without requiring specialized user-space buffer management or API changes. Supporting buffered I/O can be done similarly by installing DMA-filled pages as page-cache folios rather than remapping user pages. This would eliminate the network-to-page-cache copy, although conventional `read()` would still require a final page-cache-to-user copy. We leave that extension to our future work.

Page remapping [11, 56] introduces a new challenge that is absent from the write path: once destination pages are replaced, the application must observe the new mappings rather than stale translations cached in the TLB. So, after updating page-table entries, one must invalidate TLB entries to prevent CPU cores from using stale translations. These invalidations can trigger cross-core TLB shootdowns [3, 4, 28], often imposing a non-trivial overhead. This overhead is particularly harmful for small requests, which are difficult to amortize and can erode the benefits of enabling zero-copy.

Inline per-I/O TLB flush. Our initial design invalidates TLB entries immediately after each remapping during packet RX. Although simple, this approach places expensive MMU synchronization on the critical path of packet processing, interfering with protocol handling and making it difficult to amortize the fixed shootdown cost. We quantify this overhead in §7.4.

Deferred, batched TLB flush. To reduce shootdown frequency and packet-path interference, zcIO defers both MMU synchronization and I/O completion, batching TLB invalidations across multiple remapped pages. Each NVMe/TCP queue stages remapped pages and enqueues completed commands into a per-queue pending ring. Once the staged pages exceed a threshold, a zcIO-managed workqueue handler (`zc_flush_work`) executes a coalesced invalidation before releasing the deferred completions. This mechanism ensures that an I/O is reported complete only after translations are globally consistent, allowing the packet RX path to remain focused on parsing and data placement.

Batching introduces a new challenge under sparse traffic. When the request rate is low, the batch threshold would delay the I/O completion. To avoid this, zcIO uses a timer-based tail flush that commits staged mappings even if the batch is incomplete. However, under extremely sparse workloads, this timer can become the effective upper bound on completion latency. To avoid the problem, zcIO implements an *end-of-burst flush*: when the queue becomes quiescent (i.e., no outstanding I/Os remain), it immediately schedules the flush worker to commit any pending mappings and release deferred completions rather than waiting for the idle timer to expire.

6.3 Safe Recycling of Shared Data Pages

One fundamental hurdle in zero-copy networking is the asynchronous buffer ownership between the network stack and the application. Note that the NIC driver's page pool provides

[6] zcIO primarily targets large-transfer workloads whose locality is weak or whose working sets exceed cache capacity.

the RX fragment pages that are remapped to the application virtual address space (VA). Once remapped, pages must remain under application ownership while reachable. However, if they are not reclaimed in time, they can accumulate in user space and eventually deplete the page pool. zcIO resolves this lifecycle conflict transparently by leveraging the native page reference counts for both I/O types. To bridge the ownership gap, zcIO employs two distinct reclamation mechanisms tailored to the read path.

Overwrite-triggered page reclamation. In many applications, successive read operations overwrite the fixed destination buffers. zcIO leverages this pattern to implement *overwrite-triggered page reclamation*. When a new I/O triggers a remap to a destination VA, zcIO checks if the page belongs to the page pool installed by zcIO, and if the new I/O fully overwrites it. If so, zcIO replaces the mapping and returns the stale page to the NIC recycling domain. In case the new I/O updates only a sub-page region, zcIO keeps the existing mapping and takes the copy path for the region. This prevents premature page reclamation, preserving both the destination VA mapping and application-modified data. As a result, standard I/O flow recycles pages into the NIC pool without explicit cleanup overhead.

MMU notifier for final cleanup. To handle final deallocations such as `munmap`, process exit, or VMA changes, zcIO leverages the `mmu_notifier` subsystem. By subscribing to `invalidate_range` events, zcIO intercepts virtual address teardowns. Upon notification, zcIO scans its metadata to identify any remapped NIC pages within the affected range and reclaims them into the pool, ensuring memory consistency.

7 Evaluation

We evaluate zcIO using `fio` [5]-based NVMe/TCP microbenchmarks and three representative real-world workloads: MLPerf [33], RocksDB [47], and Nginx [34]. Since zcIO optimizes the NVMe/TCP receive path, its benefits apply to the receiver in each I/O type (initiator for reads, target for writes). Our evaluation addresses three questions: (i) How much performance improvement does zcIO provide over the standard Linux NVMe/TCP implementation and alternative receive-side designs? (ii) How many CPU cores are required to reach a target throughput, and how effectively does performance scale with additional cores? (iii) Do these gains translate into consistent end-to-end improvements under practical, real-world configurations? We validated correctness by verifying data integrity and ensuring that application-visible behavior remains identical to the baseline.

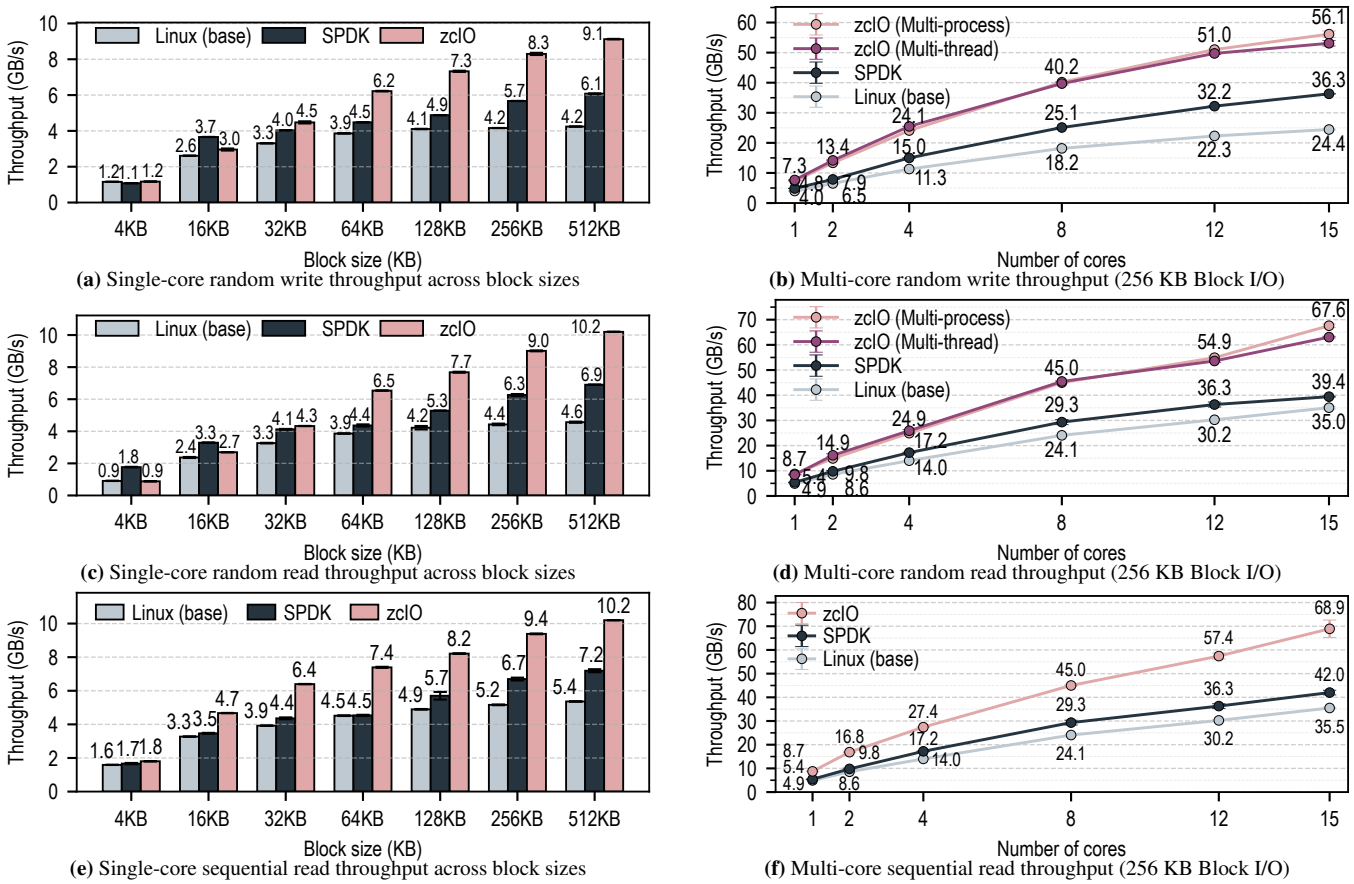
7.1 Experiment Setup

Experimental testbed. Our testbed consists of an NVMe/TCP target and an NVMe/TCP initiator connected by two 400 GbE links. The target server is equipped with a Xeon Gold 6418H CPU (24 cores), 8 modules of DDR5 DRAM, and two NVIDIA ConnectX-7 400 GbE NICs [37] (800 Gbps aggregate). For storage, it utilizes eight Samsung 9100 PRO NVMe SSDs [48], each capable of 14,700 MB/s sequential reads and 13,300 MB/s sequential writes. The initiator is equipped with a Xeon Gold 5512U CPU (28 cores), 8 modules of DDR5 DRAM, and two ConnectX-7 400 GbE NICs. Both machines run Ubuntu 22.04.5 LTS with Linux kernel 6.11 unless otherwise noted. All network interfaces are configured with an MTU of 9000 B.

NIC offloads. We enable GRO and TSO to reflect production settings. For receive-side coalescing, we enable `rx-gro-hw`, which not only merges the contiguous TCP packets with compatible headers but also performs split-DMA of protocol headers and payloads into the `skb` linear buffer and non-linear fragments, respectively. zcIO exploits this separation to implement protocol-aware packet placement (§5).

zcIO implementation. We implement the zcIO prototype by adding 3566 lines of code to the Linux kernel (v6.11). For protocol-aware TX, we extend `tcp_output.c` to maintain per-flow NVMe/TCP contexts, requiring 409 lines of C code. For page-aligned RX, we modify the `mlx5_core` driver for our ConnectX-7 NICs. Specifically, we add 10 lines to roll back the TCP data offset and 9 lines to enable strided RX. While our prototype uses `mlx5_core`, our design can be easily applied to other NICs with similarly minimal driver modifications. On the target RX path, we add 1021 lines to reuse `skb` fragments directly as the source buffer for disk I/O. On the initiator RX path, we add 2117 lines to remap the physical pages referenced by `skb` fragments to the application buffer, including TLB-flush batching.

Baselines. We compare zcIO against two baselines: (1) **Linux:** the standard Linux 6.11 NVMe/TCP implementation, which copies received payloads from `skb` fragments into the I/O buffer. (2) **SPDK:** the SPDK v25.09 NVMe-oF/TCP implementation running in user space. While SPDK reduces storage-side overhead with its user-space, polled-mode design, its TCP transport still relies on SPDK's socket layer over the Linux kernel TCP stack. As a result, received NVMe/TCP payload is still copied from kernel networking buffers into SPDK user-space buffers. We also measured the performance of i10 [17], but we excluded its results from the comparison because i10 was implemented on an older kernel version [19] that lacks recent improvements to the network stack. Unfortunately, we cannot downgrade the kernel version used for our zcIO implementation, as only recent kernels support header-data split functionality. Other approaches based on DPDK, AF_XDP, Linux TCP RX zero-copy, or `io_uring` RX zero-copy could, in principle, eliminate copies in the TCP receive path, but none of them currently support NVMe/TCP. Supporting NVMe/TCP would require extensive modifications across multiple layers, including the application interface and



(a) Single-core random write throughput across block sizes

(b) Multi-core random write throughput (256 KB Block I/O)

(c) Single-core random read throughput across block sizes

(d) Multi-core random read throughput (256 KB Block I/O)

(e) Single-core sequential read throughput across block sizes

(f) Multi-core sequential read throughput (256 KB Block I/O)

Figure 7: NVMe/TCP write and read throughput comparison

the NVMe/TCP host and target modules.

7.2 NVMe/TCP Write Performance

We run a random write workload on `fio` over NVMe/TCP to evaluate the throughput and CPU efficiency of the target. We saturate the target CPU (≈100% utilization) to make RX overhead directly visible.

Single core. Figure 7a compares the single-core target performance. zcIO outperforms the Linux baseline and SPDK by up to 2.17× and 1.49×, respectively, for large block sizes, achieving 9.1 GB/s. While SPDK reduces SSD interrupt overhead, it paradoxically achieves lower throughput for 4 KB transfers. This sub-optimal performance stems from the frequent context switching between the kernel-level TCP stack and the user-space SPDK target, whereas Linux and zcIO benefit from unified, kernel-resident data paths.

Scalability. To evaluate whether zcIO maintains its performance gain at high line rates, we fix the block size at 256 KB and scale the target from 1 to 15 cores, which matches the maximum number of hardware queue pairs supported by our storage device. To avoid a network bandwidth bottleneck, we employ eight SSDs and two 400 GbE NICs, providing an aggregate link capacity of 100 GB/s (or 800 Gbps). As shown in Figure 7b, zcIO consistently outperforms both Linux and SPDK, achieving up to 2.30× and 1.70× higher throughput, respectively. With only 8 cores, zcIO reaches 40.2 GB/s, already surpassing the peak throughput of both baselines, and ultimately scales to 56.1 GB/s (449 Gbps) with 15 cores.

7.3 NVMe/TCP Read Performance

Single core. We next measure the random read throughput sustained by a single-core initiator with `fio` in the same setup. As Figure 7c shows, SPDK leads at 4 KB. Unlike the target side, the initiator path crosses the user–kernel boundary on every I/O submission; SPDK runs entirely in user space and avoids this cost, and its interrupt-free, polled design suits small transfers. From 32 KB onward, however, zcIO overtakes both baselines. At 512 KB, it delivers 2.22× the throughput of Linux and 1.48× that of SPDK, while cutting median and p99.99 latency by 55.4% and 60.5% relative to Linux. Sequential reads (Figure 7e) follow the same trend: zcIO's advantage is negligible at 4 KB and widens with block size, reaching 10.2 GB/s at 512 KB—1.89× Linux and 1.42× SPDK. Eliminating data copies therefore pays off under both access patterns, and the benefit grows with transfer size.

Scalability. To evaluate scalability, we repeat the same experiments on a 100 GB/s testbed while scaling the number

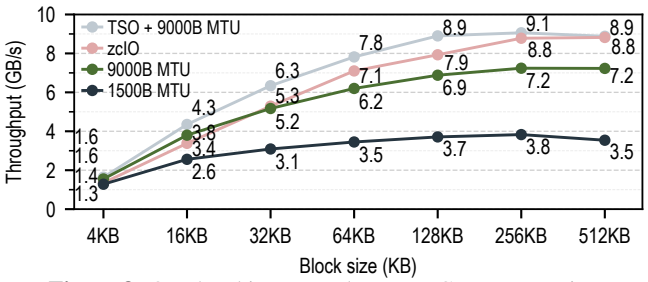


Figure 8: Overhead in protocol-aware TCP segmentation

of initiator cores for 256 KB reads. Similar to the target-side results, Figure 7d demonstrates that zcIO scales substantially better as the core count grows. zcIO delivers up to 1.93× and 1.71× higher throughput than Linux and SPDK, respectively. zcIO achieves comparable throughput across both multi-process and multi-threaded `fio` configurations, as the TLB flush overhead is effectively amortized over large I/O operations. Overall, the gain is slightly more modest than in the target-side (random write) case, owing to the remapping overhead and TLB flushes. Nevertheless, zcIO still achieves substantial performance improvements. With only 8 cores, zcIO reaches 45.0 GB/s, surpassing the peak throughput of the baselines, and scales up to 67.6 GB/s (540.8 Gbps) with 15 cores. Sequential reads (Figure 7f) behave the same way, with speedups of up to 1.96× over Linux and 1.72× over SPDK. Together, these results confirm that zcIO's advantage widens with both transfer size and concurrency.

7.4 Overhead Analysis

Segmentation overhead. To enable zero-copy data delivery at the receiver, zcIO's sender TCP stack performs protocol-aware segmentation. However, this may incur CPU overhead and reduce TX performance, as it often fails to leverage the full MTU for the payload. We evaluate the sender-side overhead while providing sufficient CPU cores to the receiver. The NVMe/TCP target is restricted to a single CPU core. Figure 8 compares four segmentation strategies: Software TCP segmentation (1500 B and 9000 B MTU), Hardware TSO with a 9000 B MTU (TSO + 9000 B MTU), and our protocol-aware TCP segmentation (zcIO). When the requested block size is small (4 KB), zcIO does not benefit from large MTUs nor TSO because each block contains only a single PDU. In this scenario, its TX performance mirrors the 1500 B-MTU baseline, reaching 1.4 GB/s—roughly 14% lower than the TSO throughput of 1.6 GB/s. However, as the requested block size increases, multiple 4 KB disk pages fit within a single block, allowing TSO to become effective. For large blocks, the performance of zcIO reaches 8.8 GB/s, only 3% lower than TSO. Overall, the extra overhead introduced by PDU alignment becomes negligible as the I/O size increases. Given that this approach yields a performance improvement of up to 2.30× on the RX side, this marginal TX overhead represents a highly favorable trade-off.

TLB flush overhead. We quantify the TLB flush overhead at the initiator by profiling the CPU-time share of `flush_tlb_mm_range` using `perf record`. On the 3-core, 24-thread setup, inline flushing accounts for 15.3%, 14.4%, and 14.5% of the total CPU cycles for 4, 8, and 16 KB I/Os, respectively. In contrast, deferred and batched flushing substantially reduces these shares to 3.2%, 2.1%, and 2.8%, a 4.8–6.9× reduction. This optimization translates to throughput improvements of 27.1%, 29.6%, and 22.1%. While zcIO is primarily motivated by large-transfer efficiency, deferred batching keeps remapping costs at a level that preserves or even slightly exceeds the baseline Linux performance for small I/Os.

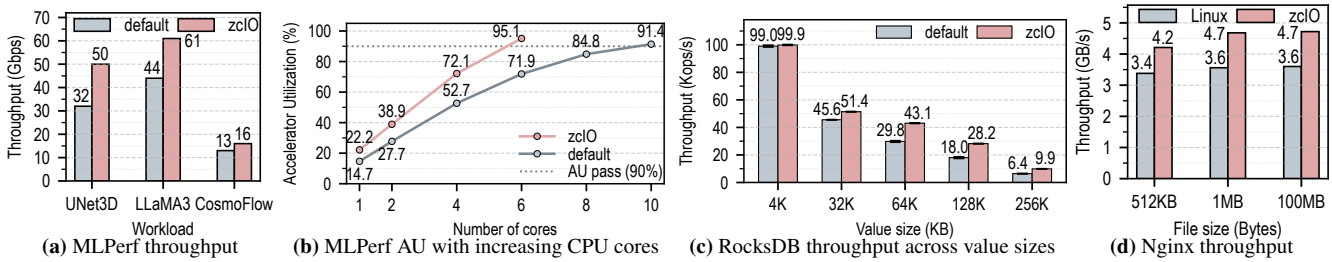
7.5 Real-world Application Performance

MLPerf Storage. We evaluate zcIO with three representative MLPerf Storage [33] workloads: UNet3D (medical image segmentation training), CosmoFlow (cosmology simulation training), and LLaMA3-8B checkpoint loading. For training workloads, the primary metric is Accelerator Utilization (AU), which quantifies the fraction of time an emulated accelerator spends computing versus stalling on I/O. A threshold of 90% AU is typically used to verify that the storage subsystem can sufficiently saturate the accelerator pipeline. For LLaMA3-8B checkpoint loading, where no accelerator is emulated, we report performance directly in terms of I/O throughput.

Setup. We apply workload-specific optimizations to both the baseline Linux kernel and zcIO. We enable the `O_DIRECT` option and optimize the benchmark library to reuse aligned I/O buffers, thereby eliminating memory allocation overhead. Our testbed consists of four NVMe drives exported via NVMe/TCP over a ConnectX-7 400 GbE NIC. We present two primary evaluations: (1) a single-core throughput comparison where we emulate 2 H100 accelerators for UNet3D and 5 H100s for CosmoFlow; and (2) a scalability study on UNet3D emulating 8 H100 accelerators to identify the minimum CPU resources required to meet the 90% AU threshold.

Throughput. Figure 9a reports single-core read throughput for all workloads. zcIO consistently outperforms the Linux baseline, delivering 1.56×, 1.39×, and 1.23× higher throughput for UNet3D, LLaMA3-8B checkpoint loading and CosmoFlow, respectively. The gains are most pronounced for UNet3D and checkpoint loading, both of which involve large sequential reads where RX-side copy overhead is the primary bottleneck. In contrast, CosmoFlow's reliance on smaller files increases the significance of fixed per-request overheads, which partially offsets the benefits of zero-copy delivery.

AU scalability. Figure 9b shows accelerator utilization (AU) as we scale the number of cores allocated to the UNet3D training pipeline. With the Linux baseline, AU starts at 14.7% with a single core and requires 10 cores to finally exceed the 90% passing threshold. In contrast, zcIO reaches this threshold with only 6 cores, achieving 95.1% AU. Overall,



(a) MLPerf throughput **(b)** MLPerf AU with increasing CPU cores **(c)** RocksDB throughput across value sizes **(d)** Nginx throughput

Figure 9: Performance comparison across real-world application workloads

zcIO achieves the same level of accelerator utilization with 40% fewer CPU cores.

RocksDB. We further evaluated zcIO using a RocksDB-based application [47] deployed on disaggregated NVMe storage. The setup utilizes four NVMe/TCP SSDs, each mounted with `ext4` and accessed via `O_DIRECT`. We perform random point-lookups using the `Seek(k)` API with value sizes ranging from 4 KB to 256 KB, pinning the application to a single core to isolate per-core efficiency. As shown in Figure 9c, at 4 KB values, Linux and zcIO deliver nearly identical performance, indicating that RocksDB's internal per-request overhead masks any potential gains from zero-copy. However, as the value size increases from 32 KB to 256 KB, the relative impact of TCP data movement becomes more pronounced. zcIO's performance advantage scales accordingly, achieving throughput improvements of 1.13×, 1.45×, 1.58×, and 1.55× for 32, 64, 128, and 256 KB values, respectively.

Nginx. To evaluate end-to-end performance, we deploy Nginx [34] in a three-node topology comprising an NVMe/TCP target, an Nginx server (initiator), and a dedicated `wrk` client. The Nginx process is pinned to a single core, with NIC IRQ processing offloaded to separate cores to isolate application-level CPU costs. We disabled `sendfile` and employed `O_DIRECT` to bypass the page cache, thereby stressing the storage-to-network data path. Large output buffers (512 KB to 1 MB) are configured to maximize direct-I/O read sizes, limited only by the underlying block-layer constraints. As shown in Figure 9d, zcIO improves single-core Nginx goodput from 3.38–3.60 GB/s (Linux) to 4.21–4.72 GB/s, representing a 1.25–1.33× speedup. For objects 1 MB and larger, throughput remains stable, indicating that the server reaches a steady-state streaming pattern with consistent read-send iterations. These results demonstrate that zcIO effectively accelerates the end-to-end data path by eliminating redundant copies between the storage and network stacks.

8 Discussion

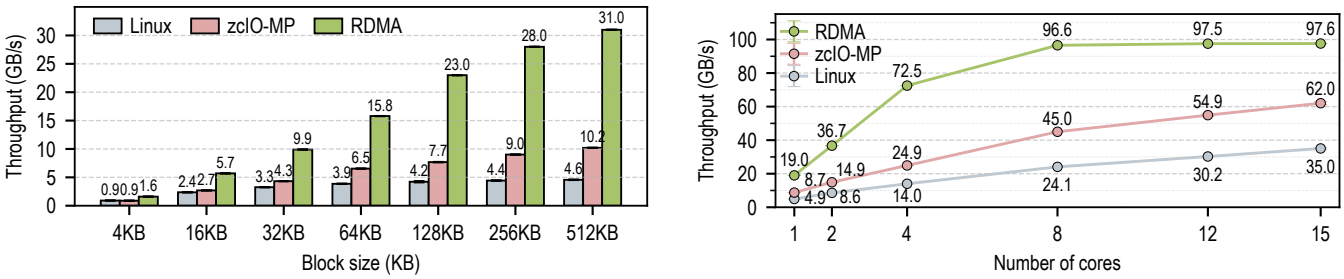
Comparison with NVMe/RDMA. While zcIO substantially outperforms Linux NVMe/TCP, it does not fully match the performance of NVMe/RDMA. Figure 10 reports single- and multi-core random read performance of NVMe/RDMA along with zcIO and Linux NVMe/TCP. On a single core, NVMe/RDMA's advantage shrinks from 4.1–6.7× over Linux NVMe/TCP to 2.4–3.1× over zcIO, confirming that payload copies dominate the cost of large I/Os. The same reduction holds in the non-saturated 1–4 core range, where the advantage falls from 3.9–5.2× to 2.2–2.9×. Beyond eight cores, NVMe/RDMA plateaus at 96.6 GB/s, bounded by the 100 GB/s (800 Gbps) network capacity rather than by software. The residual gap comes from hardware offload: NVMe/RDMA hands transport processing to the RNIC, which writes data directly into destination buffers. zcIO deliberately forgoes this in order to preserve the standard TCP/IP stack and transparent kernel interfaces, so the host still performs TCP/IP and NVMe/TCP processing, skb management, and read-side page remapping with TLB flushes.

NVMe/TLS. We examine zcIO's extensibility to NVMe/TCP with TLS protection [40]. Since NVMe/TCP is kernel-resident, decryption is managed via kTLS [53]. In software kTLS, zcIO yields no benefit because decryption is already fused with the mandatory data copy from socket buffers to the destination. However, zcIO can effectively complement hardware kTLS, where packets are decrypted before reaching host memory. The primary challenge is TLS framing: the 5 B header and 16 B authentication tag can disrupt zcIO's page-alignment assumptions. While current NIC header-data split mechanisms can isolate transport headers, they cannot yet isolate the trailing authentication tag. Enabling zcIO for NVMe/TLS may thus require modest NIC extensions, such as a header-data-trailer split DMA.

HW kTLS robustness. Current hardware TLS offloads (e.g., L5O [44]) are highly sensitive to packet reordering, often due to misalignments between TLS records and TCP segments. Aligning TLS records with packet boundaries would improve reordering robustness and simplify decryption by removing the dependency on TCP flow reassembly. Such record-packet alignment aligns with zcIO's design philosophy and represents a promising direction for future research.

9 Related Work

Network storage optimizations. Several recent works have targeted the overheads of NVMe/TCP. i10 [17] optimizes the in-kernel data path through request batching and delayed doorbells, primarily addressing fixed per-request overheads for small I/O. In contrast, ReFlex [27] bypasses the kernel entirely with a polling-based user-space dataplane, which



(a) Single-core NVMe/RDMA random read throughput across block sizes

(b) Multi-core NVMe/RDMA random read throughput with 256 KB I/Os

Figure 10: NVMe/RDMA random read throughput comparison.

requires a custom stack and invasive application refactoring. Intel Application Device Queues (ADQ) [18] improves tail latency by dedicating hardware queues to specific applications, thereby mitigating interference. Further, NVMe-oAF [23] optimizes for locality by dynamically switching between shared-memory and TCP transports. zcIO is complementary to these approaches: while prior works focus on per-request or scheduling overheads, zcIO eliminates the byte-proportional payload copy that becomes the dominant bottleneck at medium-to-large transfer sizes.

Hardware-assisted I/O. Several specialized hardware solutions have been proposed to offload NVMe/TCP processing. L5O [44] utilizes a NIC that parses PDU headers to DMA data directly into user buffers. However, this requires tight coupling between application-managed buffer layouts and NIC firmware. Similarly, ZeroNIC [51] offloads transport processing to an FPGA-based NIC to enable direct DMA to PCIe devices or host buffers. While effective for NVMe/TCP targets via P2P DMA, ZeroNIC demands integration with a custom provider library and invasive application-level modifications for host-side consumption. IO-TCP [25] employs a split-stack design on SoC SmartNICs (e.g., NVIDIA BlueField [36]) to offload disk I/O, primarily targeting the target's TX path. While these approaches offer significant offloading, they rely on proprietary hardware and often necessitate rigid application-hardware co-design. In contrast, zcIO is a software-only kernel enhancement that achieves zero-copy performance using standard NIC features available in commodity hardware.

Aligned transport. SOLAR [32] redesigns the transport layer to facilitate data-path execution on FPGA-based accelerators. It adopts a one-block-one-packet design that strictly aligns network packets with storage blocks, thereby eliminating packet buffering and the need for packet-to-block conversion. This approach shares a similar intuition with zcIO's record-aware networking, as both aim to align transport units with storage semantics. However, SOLAR introduces a new UDP-based transport protocol, requiring significant changes to the network stack. In contrast, zcIO preserves the standard NVMe/TCP protocol, achieving record-aware delivery through a transparent TCP overlay that maintains compatibility with existing internet infrastructure.

Performance analysis tools. Understanding performance bottlenecks in NVMe/TCP is inherently complex, as requests must traverse disparate subsystems, including the block I/O layer, the TCP/IP stack, and the NIC driver. To address this, ntprof [22] offers a cross-layer profiling framework designed to trace NVMe/TCP requests and isolate bottlenecks along the end-to-end data path.

10 Conclusion

This work identifies that the layered structure of NVMe/TCP inherently prevents transparent zero-copy due to the misalignment between TCP segments and NVMe block boundaries. To address this, we introduce zcIO, which enables transparent zero-copy through *record-aware networking*. Our evaluation demonstrates that zcIO improves RX performance by up to 2.2× on the initiator and 2.3× on the target compared to the Linux baseline. By preserving standard application interfaces, zcIO seamlessly accelerates existing workloads without requiring code modifications. Beyond NVMe/TCP, we believe the principles of record-aware networking offer a generalizable framework for optimizing other page-aligned protocols like iSCSI and NFS, opening new avenues for high-performance networked systems.

Acknowledgments

We appreciate the insightful feedback and suggestions from the FAST 2027 reviewers, as well as our shepherd, Vasily Tarasov. This work was supported in part by the ICT Research and Development Program of MSIT/IITP, Korea, under [RS-2024-00349594, Development of networked systems technologies leveraging SmartNIC], [2022-0-00531, Development of in-network computing techniques for efficient execution of AI applications], [RS-2024-00418784, Next-generation Cloud-native Cellular Network Center], [RS-2025-02217106, Ethernet-based high-performance Smart Switch for Data Center], [RS-2026-25507250, Development of a Fabric Platform for Ultra-low Latency, Lossless, and High-bandwidth AI Data Centers], and the New Faculty Startup Fund from Seoul National University. KyoungSoo Park is the corresponding author of this paper.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: minimal near-optimal datacenter transport. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[3] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2017.

[4] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down TLB shootdowns! In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2020.

[5] Jens Axboe. fio - flexible i/o tester. https://fio.readthedocs.io.

[6] David Borman. TCP Options and Maximum Segment Size (MSS) (RFC-6691). https://www.rfc-editor.org/rfc/rfc6691, 2012. Last Accessed: 2026-03-14.

[7] Broadcom Inc. EtherNIC Controller User Guide (UG2XX). https://techdocs.broadcom.com/content/dam/broadcom/techdocs/us/en/pdf/data-center-solutions/netxtreme/EtherNIC-Ctrl-UG2XX.pdf, 2026. Last Accessed: 2026-03-14.

[8] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding Host Network Stack Overheads. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2021.

[9] DPDK. DPDK: Data Plane Development Kit. https://www.dpdk.org/. Last Accessed: 2026-03-14.

[10] DPDK Project. NVIDIA MLX5 Ethernet Driver. https://doc.dpdk.org/guides-25.11/nics/mlx5.html, 2025. Last Accessed: 2026-03-14.

[11] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[12] Eric Dumazet. The Path To TCP 4K MTU and RX ZeroCopy. https://netdevconf.info/0x14/session.html?talk-the-path-to-tcp-4k-mtu-and-rx-zerocopy, 2020. Last Accessed: 2026-03-14.

[13] Leonid Petrovich Grossman. Large Receive Offload implementation in Neterion 10GbE Ethernet driver. https://www.kernel.org/doc/ols/2005/ols2005v1-pages-203-208.pdf, 2010. Last Accessed: 2026-03-14.

[14] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the ACM International Systems and Storage Conference (SYSTOR)*, 2017.

[15] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2013.

[16] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[17] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP $\approx$ RDMA: CPU-efficient remote storage access with i10. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2020.

[18] Intel and Lightbits Labs. Scalable, Low-Latency Storage Using NVMe over TCP. https://thelinuxcluster.com/wp-content/uploads/2021/04/intel-lightbits-scalable-low-latency-storage-using-nvme-tcp-wp-092220-1.pdf, 2020. Last Accessed: 2026-03-14.

[19] Jaehyun Hwang. i10-implementation. https://github.com/i10-kernel/i10-implementation, 2020. Last Accessed: 2026-03-14.

[20] John Nagle. Congestion Control in IP/TCP Internetworks (RFC-896). https://www.rfc-editor.org/rfc/rfc896, 1984. Last Accessed: 2026-03-14.

[21] Jonathan Corbet. A reworked TCP zero-copy receive API. https://lwn.net/Articles/754681/, 2018. Last Accessed: 2026-03-14.

[22] Yuyuan Kang and Ming Liu. Understanding and profiling NVMe-over-TCP using ntprof. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2025.

[23] Arjun Kashyap and Xiaoyi Lu. Nvme-oaf: Towards adaptive nvme-of for io-intensive workloads on hpc cloud. In *Proceedings of the ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2022.

[24] Khalid Manaa. [pull request][net-next 00/14] mlx5 HW GRO. https://lists.openwall.net/netdev/2021/10/27/9, 2021. Last Accessed: 2026-03-14.

[25] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. Rearchitecting the TCP stack for I/O-Offloaded content delivery. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2023.

[26] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[27] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[28] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[29] Linux Kernel Organization. io_uring zero copy Rx. https://docs.kernel.org/networking/iou-zcrx.html. Last Accessed: 2026-03-14.

[30] Alexander Lobakin. Re: [PATCH iwl-next] ice: implement configurable header split for regular Rx. https://www.mail-archive.com/intel-wired-lan@osuosl.org/msg13879.html, 2025. Last Accessed: 2026-03-14.

[31] LWN.net. net/mlx5e: SHAMPO, Enable HW GRO once more. https://lwn.net/Articles/975774/, 2024. Last Accessed: 2026-03-14.

[32] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: the evolutions of the compute-to-storage networks in Alibaba cloud. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2022.

[33] MLCommons. Mlperf storage benchmark. https://mlcommons.org/benchmarks/storage/. Accessed: Mar. 2026.

[34] NGINX. NGINX. https://nginx.org/. Last Accessed: 2026-03-14.

[35] NVIDIA Corporation. Changes and New Features History. https://docs.nvidia.com/networking/display/Kernelupstreamv517/Changes%2Band%2BNew%2BFeatures%2BHistory, 2023. Last Accessed: 2026-03-14.

[36] NVIDIA, Inc. NVIDIA BlueField Platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. Last Accessed: 2026-03-14.

[37] NVIDIA, Inc. NVIDIA ConnectX-7. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf. Last Accessed: 2026-03-14.

[38] NVM Express, Inc. NVM Express™ over Fabrics. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1a-2021.07.12-Ratified.pdf, 2021. Last Accessed: 2026-03-14.

[39] NVM Express, Inc. NVM Express NVMe over RDMA Transport Specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-NVMe-over-RDMA-Transport-Specification-Revision-1.2-2025.08.01-Ratified.pdf, 2025. Last Accessed: 2026-03-14.

[40] NVM Express, Inc. NVM Express NVMe over TCP Transport Specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-NVMe-over-TCP-Transport-Specification-Revision-1.2-2025.08.01-Ratified.pdf, 2025. Last Accessed: 2026-03-14.

[41] Tatsuki Oyama and Kohei Shiomoto. Implementation of NVMe Over TCP Using SPDK and Its Performance Measurement. In *Proceedings of the IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR)*, 2024.

[42] Patchwork (ozlabs.org). Support RX multi-packet WQE (Striding RQ). https://patchwork.ozlabs.org/patch/611526/, 2016. Last Accessed: 2026-03-14.

[43] Pavel Begunkov and David Wei. Fast ZC Rx Data Plane using io_uring. https://netdevconf.info/0x17/sessions/talk/fast-zc-rx-data-plane-using-io-uring.html, 2023. Last Accessed: 2026-03-14.

[44] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC offloads. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[45] Matthew Podolsky, Sally Floyd, Jamshid Mahdavi, and Matt Mathis. An Extension to the Selective Acknowledgement (SACK) Option for TCP (RFC-2883). https://www.rfc-editor.org/rfc/rfc2883, 2000. Last Accessed: 2026-03-14.

[46] Jon Postel. Internet Protocol (RFC-791). https://www.rfc-editor.org/rfc/rfc791, 1981. Last Accessed: 2026-03-14.

[47] RocksDB. RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/. Last Accessed: 2026-03-14.

[48] Samsung Electronics Co., Ltd. Samsung V-NAND SSD 9100 PRO: Data sheet (revision 1.0). https://download.semiconductor.samsung.com/resources/data-sheet/Samsung_NVMe_SSD_9100_PRO_Datasheet_Rev.1.0.pdf, 2025. Last Accessed: 2026-03-14.

[49] Marcos Schwarz, Brian Tierney, Kiran Vasu, Eli Dart, Christian Esteve Rothenberg, Jeronimo Bezerra, and Italo Valcy. Recent Linux Improvements that Impact TCP Throughput: Insights from R&E Networks. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC-W)*, 2024.

[50] Nikita Sharma, Ruihao Li, Qinzhe Wu, and Lizy K. John. Performance Impact of NVMe-Over-TCP on HDFS Workloads. In *Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2022.

[51] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, Raghu Raja, Daniel Walton, Rachit Agarwal, Shrijeet Mukherjee, and Christos Kozyrakis. High-throughput and Flexible Host Networking for Accelerated Computing. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[52] SPDK. SPDK: Storage Performance Development Kit. https://spdk.io/. Last Accessed: 2026-03-14.

[53] The Linux Kernel. Kernel TLS. https://www.kernel.org/doc/html/latest/networking/tls.html. Last Accessed: 2026-03-14.

[54] The Linux Kernel. Segmentation Offloads. https://docs.kernel.org/networking/segmentation-offloads.html#generic-receive-offload. Last Accessed: 2026-03-14.

[55] The Linux Kernel. MSG_ZEROCOPY. https://www.kernel.org/doc/html/latest/networking/msg_zerocopy.html, 2020. Last Accessed: 2026-03-14.

[56] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[57] Jiexiong Xu, Yiquan Chen, Yijing Wang, Wenhui Shi, Guoju Fang, Yi Chen, Huasheng Liao, Yang Wang, Hai Lin, Zhen Jin, Qiang Liu, and Wenzhi Chen. LightPool: A NVMe-oF-based High-performance and Lightweight Storage Pool Architecture for Cloud-Native Distributed Database. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.

[58] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. Efficient user-level storage disaggregation for deep learning. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, 2019.

A Artifact Appendix

Abstract

This artifact provides the zcIO prototype and an automated evaluation framework for reproducing the principal experimental results presented in the paper. zcIO is implemented using three Linux kernel variants for initiator-side zero-copy receive, target-side zero-copy receive, and sender-side protocol-aware packetization. The artifact includes scripts for configuring the testbed, executing NVMe/TCP microbenchmarks and application workloads, and collecting performance results.

Scope

This artifact allows evaluators to validate the main claims made in the paper. Specifically, it supports evaluation of whether eliminating receive-side payload copies improves NVMe/TCP read and write throughput for medium-to-large I/Os and whether these gains persist as additional CPU cores are allocated. It also evaluates the sender-side overhead of protocol-aware TCP segmentation and the application-level benefits for MLPerf Storage, RocksDB, and Nginx.

Contents

The artifact consists of the following components:

- The `initiator-rx-zc` kernel branch for initiator-side zero-copy receive.
- The `target-rx-zc` kernel branch for target-side zero-copy receive.
- The `pdu_align_only` kernel branch for sender-side protocol-aware packetization.
- The `zcio-ae/` directory containing experiment scripts and a dedicated README for each reproduced result.
- Per-experiment `all_in_one.sh` scripts for configuring the participating machines, running the workload, and collecting summarized results.

Hosting

The source code and artifact evaluation framework are publicly available at https://github.com/TNET-SNU/zcIO. The evaluation framework is located in the `zcio-ae/` directory. The artifact uses the following source revisions:

- `initiator-rx-zc`: `1b1fb3f`
- `target-rx-zc`: `99d3563`
- `pdu_align_only`: `311dc4f`
- `main` (`zcio-ae/`): `8a52454`

Requirements

Hardware dependencies. Most experiments use two physical machines: an NVMe/TCP initiator and an NVMe/TCP target. The target contains multiple Samsung 9100 PRO NVMe SSDs, and the two machines are connected through two 400 GbE links using NVIDIA ConnectX-7 NICs. The Nginx experiment additionally uses a third machine as an HTTP workload generator.

The network path is configured with a 9000 B MTU. The current prototype and evaluation environment use ConnectX-7 header-data split functionality and coordinated kernel support at both NVMe/TCP endpoints.

Software dependencies. The machines run Ubuntu 22.04 with the required baseline and zcIO kernel images preinstalled. The testbed also provides `fio`, SPDK, MLPerf Storage, RocksDB, Nginx, `wrk`, and the dependencies required by the individual experiments.

Kernel configurations. The read and write evaluations use different kernel combinations:

Experiment	Initiator	Target
Read-side zcIO	`initiator-rx-zc`	`pdu_align_only`
Write-side zcIO	`pdu_align_only`	`target-rx-zc`
Linux baseline	baseline Linux	baseline Linux

Evaluation Setup

Detailed instructions are provided in `zcio-ae/README.md` and in each experiment directory. The evaluator first runs the one-time deployment script on the initiator machine:

```
cd ~/zcIO/zcio-ae
./deploy.sh
```

Each experiment is then executed from its corresponding directory:

```
cd ~/zcIO/zcio-ae/<figure-directory>
./all_in_one.sh
```

The scripts verify the required kernel configuration, configure the participating machines, run the workload, and collect the results.

Directories `7a`–`7d` reproduce the single-core and multi-core random-write and random-read NVMe/TCP results in Figure 7. Directory `8` reproduces the sender-side protocol-aware segmentation overhead in Figure 8. Directories `9a`–`9d` reproduce the MLPerf Storage, RocksDB, and Nginx application results in Figure 9.

Each per-experiment README specifies the workload, participating machines, required kernel combination, execution procedure, expected results, and recovery instructions.