Remote TCP Connection Offload and Applications

Shuo Li*, Steven W. D. Chien*, Tianyi Gao, and Michio Honda
University of Edinburgh

Abstract

Layer 7 load balancers (L7LBs) play an important role in per-request server selection within long-lived connections and transport- or application-layer protocol translation. However, L7LBs introduce substantial CPU and network overhead.

We present XO, which enables an L7LB to *offload* transport-layer and application request processing to backend servers at request granularity. XO outperforms conventional L7LBs by 27–365 % in throughput through efficient utilization of server CPU and network resources. We apply XO to two real-world applications, Ceph and `nginx`, improving throughput by up to 135 % and 300 %, respectively.

1 Introduction

TCP has been widely used in web applications, disaggregated storage systems, and distributed data processing frameworks running on a private or public cloud. The network stack in commodity OSes has thus evolved over time with enhancements such as zero copy [6], I/O batching [24, 12] and hardware-assisted encryption [41]. The research community has further advanced the space of host TCP stack acceleration towards terabit Ethernet, including radical NIC redesign with FPGA [43] and flexible multi-core parallelism [7].

To expand the service capacity beyond what a single host can offer, operators typically organize individual TCP servers into a *scale-out* cluster and spread the incoming network requests across the cluster servers. Those servers may be service replicas that handle any request or service *shards* that maintain partitions of a larger dataset or process specific request types, for example, with an accelerator. In either case, it is crucial for the operator to efficiently utilize the compute and network resources in the cluster to minimize hardware cost and physical footprint in large datacenters [33, 45, 54] or edge clouds [1, 38, 29].

Network requests are distributed at multiple levels to the servers. Layer 4 load balancers (L4LBs) [1, 37, 15, 3] distribute ingress traffic based on the flow 5-tuple. Requests are then processed by layer 7 load balancers (L7LBs) [39, 21, 44, 36, 4, 19]. L7LBs are responsible for application-level tasks, including per-request server selection based on application-layer headers, protocol translation (which may change the size of requests or responses), and ingress traffic decryption with corresponding egress traffic encryption. In doing so, L7LBs *proxy* client TCP connections at the application layer and relay request and response data to and from servers.

These L7LB tasks are CPU-intensive [39], requiring the operators to allocate a considerable amount of resources. This not only reduces the resources available to backend servers, but also makes efficient cluster-wide resource utilization challenging, as it could leave either L7LB or servers underutilized. L7LBs can also be network-intensive, because they aggregate traffic for multiple servers. As a result, operators need to allocate more network bandwidth for L7LBs than for backend servers (e.g., with higher bandwidth or multiple links).

This paper proposes Remote TCP Connection Offload (XO). It allows L7LBs to offload most of their request processing cycles to backend servers and avoid sending egress traffic—which is typically larger than ingress traffic—through the L7LB, thereby reducing network bandwidth and CPU usage at the L7LB. Once the L7LB starts offloading to a backend server, it still receives the client ingress traffic, but forwards it to the backend using lightweight packet-level processing—possibly offloaded to the NIC hardware. The backend performs the application-level tasks, including those that would otherwise be handled by the L7LB, and sends the response to the client—in a direct server return manner, bypassing the L7LB. The backend server then receives subsequent requests on the connection. It can continue handling the requests or return the connection and application state to the L7LB to finish the offload or select another server.

The contributions of this paper are twofold:

- We design XO, a new approach to L7LB enhancement. Unlike existing systems that allow direct server return on L7LBs like Prism [22] and Miresga [42], it does not rely on programmable switches (see § 2 for other L7LB enhancements). This is a significant advantage, because it eliminates the need for operators to replace their switches and enables servers to reside across the top-of-the-rack (ToR) switch boundaries. This property is useful, as cloud operators often virtualize a per-tenant flat network across servers or racks [27, 38]. XO addresses the challenges of achieving its architecture via software-hardware hybrid flow redirection and resilient flow rule management (§ 4).
- We demonstrate the applicability of XO by integrating it with two real-world applications: `nginx`, a general-purpose HTTP reverse proxy for replicated backends, and Ceph, a distributed object storage system that consists of sharded storage backends and its own application gateway (§ 5). Since state-of-the-art L7LB-based systems with backend application coordination, such as Prism [22] and HA/TCP [20], have not been applied to real applications, we believe our experience helps practitioners understand the porting effort of XO in their own systems.

*Co-first authors. Steven W. D. Chien is at the University of St Andrews as of January 2026.

2 Scope and Motivation

L7LBs typically have the following qualitative properties, and thus we focus on enhancing such systems.

- An L7LB can parse the TCP stream, reconstruct an application-level request (which may span multiple packets), and select the backend server to which it forwards the request. It can do so for every subsequent request in the same stream.
- When forwarding the traffic to and from the backend, it can apply application-level processing that involves buffering or changing of the data size or content, such as protocol translation and TLS encryption/decryption.
- It is transparent to the client. The client does not notice when the backend server that generates a response changes during the connection, except for through side-channels such as performance characteristics.

Real-world L7LBs, including `nginx` [36], HAProxy [21], Envoy [17] (general-purpose HTTP reverse proxies), Rados Gateway (RGW) [8] (storage gateway for Ceph) and Hermes [39] (proprietary cloud L7LB), meet those properties. Other systems that do not meet those properties but are relevant to XO are discussed in § 8.

2.1 L7LB Performance Case Study

Relaying data at an L7LB is expensive. First, it consumes network bandwidth of the L7LB's link shared by all the backends. Second, it performs CPU-intensive operations, involving multiple data copies—moving data from the kernel to user space and then moving it back to the kernel—along with applying other data-touching operations, such as protocol translation and encryption.

We characterize the impact of these L7LB tasks using the Ceph object store (detailed in § 5.2). Clients access RGW, which acts as an L7LB, over the S3 protocol with TLS encryption; it relays data to and from the storage backends over the `msgr2` protocol.

We use four storage backend machines, with RGW co-located on one of them, and a separate client machine. All machines connect to the same switch. The backend machines use links configured at either 25 or 100 Gb/s, while the client link is always 100 Gb/s to resemble the switch uplink.

Figure 1 shows the throughput of reading the objects uniformly across the storage backends. With 25 Gb/s backend links, throughput is constrained by the RGW link, because all the traffic from the backends shares it. Using 100 Gb/s links shifts the bottleneck to RGW software. Significant factors include packet I/O, syscalls and request parsing for small data, and data copying and encryption for large data. Those overheads at RGW cause a cluster-wide imbalance in CPU and network resource utilization, leaving those resources in the backends underutilized.

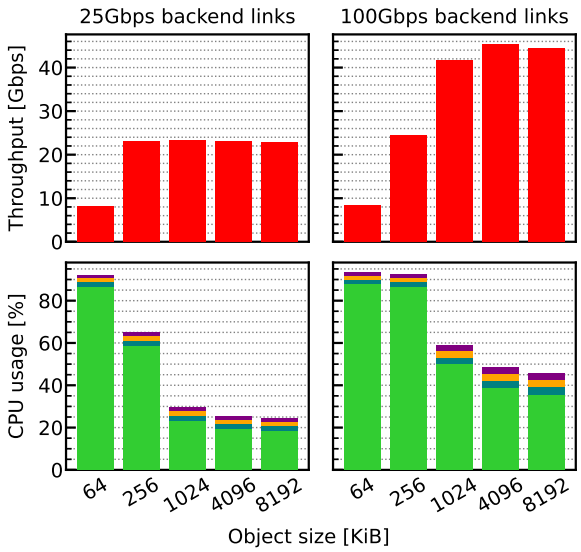


Figure 1: Ceph throughput with the RGW L7LB in a 4-server cluster (top) and per-node CPU usage in different colors (bottom).

2.2 L7LB Enhancements

In search of L7LB acceleration, we found performance-practicality trade-offs in existing L7LB architectures. Prism [22] enables L7LBs to *handoff* the TCP connection established between the client and the L7LB to a server. It has the best performance benefit, because it enables line-rate ingress traffic redirection to the server without L7LB involvement and direct server return (DSR) on the egress datapath. However, it is impractical due to its reliance on programmable switches[1] and the requirement that all servers connect to the same switch. Although switch programmability has evolved, its main users are operators (e.g., for measurement tasks [23, 35]), not host applications; cloud operators usually do not allow their tenants to configure their infrastructure switches.

Finally, it is unclear whether Prism is general, because it has not been applied to real applications. Furthermore, since the TCP and application endpoint state must be restored at the server, server application modification is required. An interesting contrast is Miresga [42]. It avoids server application modification and enables DSR with the aid of P4 programmable switch, but it does not support protocol translation, including TLS termination.

Those observations raise a natural question: *how can we achieve the performance advantage of Prism with DSR while not using programmable switches? Furthermore, given that server application modification seems unavoidable, how difficult is it?* We answer these questions by enabling host-based ingress packet redirection and DSR in XO (§ 4) and applying it to two real-world applications (§ 5).

[1]Further, a major programmable switch ASIC released in the market, Intel/Barefoot Tofino, has been discontinued for production since early 2023.

3 XO System Model

XO supports all the L7LB properties listed in § 2. The XO architecture is based on the following service model offered by the cluster:

1. Receiving a request from the client over a TCP connection;
2. Executing a task (e.g., reading data from the storage server over the network);
3. Sending a response to the client over the TCP connection.

XO offloads steps 2 and 3 from the L7LB to a cluster server; once the offload begins, step 1 is also offloaded.

When and how does offload begin? The application makes an offloading decision when it receives a request over the TCP connection. This enables the highest flexibility in defining an application-specific offloading policy, as flexible as L7LBs. The decision can be based on the application-level content after decryption if the original request has been encrypted (e.g., TLS). For example, the application would choose the least loaded server among the replicated servers, the server whose local storage has the data requested by the client, or the server with specialized capabilities like GPU acceleration.

When and how does offload complete? Offload completes when the request processing returns to the L7LB from the server. The completion decision can be made by the L7LB, for example, when it finds a better server based on continuous load monitoring. It can also be made by the server, for example, when it finds itself overwhelmed or when it identifies that it cannot serve the requested storage data. After the completion of the offload, the L7LB may make another offload decision.

4 XO Design

XO implements the aforementioned architecture using two key components: *TCP and application state transfer* (§ 4.1) and *HW/SW hybrid traffic steering* (§ 4.2). We discuss how XO integrates with real-world applications in § 5.

4.1 TCP and Application State Transfer

Offloading execution of L7LB tasks to a server requires transferring the TCP connection and other application-level state (e.g., request data and metadata, and TLS session key and state) from the L7LB to the server (outbound) when the offload begins and in the other way around (inbound) when it completes. Connection state transfer requires *endpoint operations* that serialize and restore the TCP and application state. It also needs *flow steering operations* at L7LB and the server. L7LB configures its network stack to redirect the ingress packets, which would otherwise go to L7LB's TCP implementation, to the server; the server configures its stack to modify the source address of the egress packets to that of L7LB, which enables DSR [15][2].

XO's state transfer protocol has two design objectives. The first is correctness. Since endpoint and flow steering operations require many non-atomic commands, we must enforce a strict execution order to avoid connection failure. Second, for fast state transfer, we must minimize RPCs between L7LB and the server. Figure 2 illustrates the outbound (left) and inbound (right) state transfer sequence, described next. We use the Linux `TCP_REPAIR` feature [31] for TCP state serialization and restoration to avoid kernel modification; we discuss an optimized method with kernel modification in § 7.1.

Outbound transfer. To prevent ingress packets from reaching the TCP state being serialized, which leads to connection resets [14, 22], the L7LB installs a packet filter rule (we discuss the method in § 4.2) that drops such packets (① in Figure 2). Note that this rule drops only *unimportant* packets, such as keep-alives and spurious retransmissions. It does not block connection progress, because the state transfer process begins upon the offloading decision made by the application that has already received the request (§ 3); the next expected data packet is the response to the client.

After this step, the TLS state (initialization vector, session key, and record sequence number) along with the TCP state (sequence numbers, negotiated options, window sizes, and buffer data) can be serialized safely (②). The state is transferred to the server via a BEGIN_OFFLOAD RPC. The server then restores the connection, handling potential port conflicts by remapping ports if needed when connections transferred by different L7LBs share the same port (③). The server then installs a packet filter rule that rewrites the source IP address of egress packets to match the L7LB's address (④).

Upon state restoration, the server sends a SERVER_READY RPC that requests the L7LB to install a rule that redirects the ingress packets in the connection to the server (⑤) and to remove the rule that blocks the ingress traffic (⑥). The L7LB then sends an L7LB_READY RPC to signal the server to begin the offloaded task.

Inbound transfer. This operation is used when offloading completes. It is similar to the outbound transfer, but differs in flow steering operations. The server blocks ingress packets (⑦), serializes current connection state (⑧), and transmits state back to the L7LB with a SERVER_DONE RPC. Upon receiving the state, the L7LB restores the connection (⑨), removes redirection rules (⑩), and sends a final END_OFFLOAD RPC asking the server to remove source IP rewriting (⑪) and unblock ingress traffic (⑫). After this, all offload-related state is cleared, and thus the L7LB is ready to start another offload.

[2]Since networks are typically virtualized in datacenters, cluster nodes across racks usually use the same subnet. Therefore, this address modification, whether by an L4LB or XO, does not cause a spoofing problem.

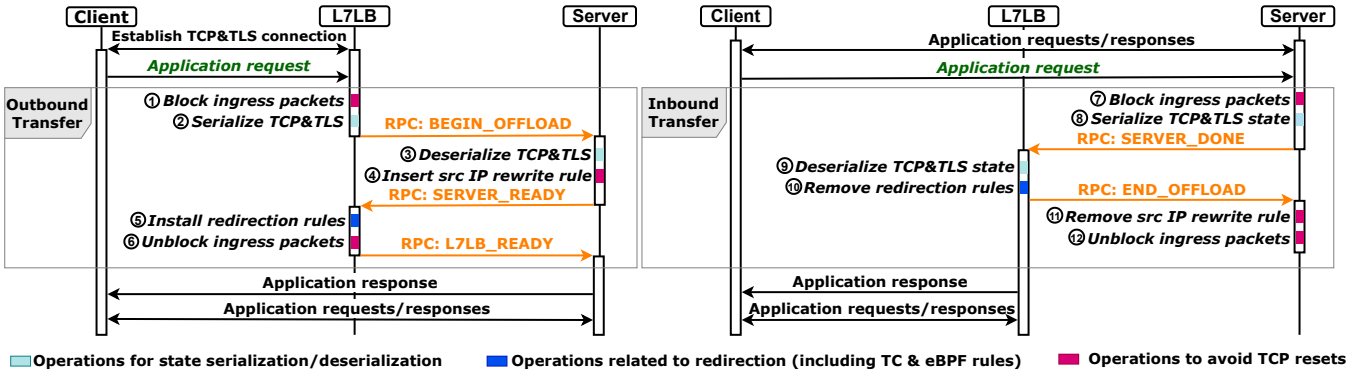


Figure 2: XO connection state transfer protocol (§ 4.1).

4.2 Flow Steering

Among flow steering operations (redirection, blocking, and source address modification), flow redirection at the L7LB (⑤ in Figure 2) is challenging to achieve two properties:

- **Fast rule installation**: We must install flow rules rapidly to minimize connection state transfer latency. This is particularly critical when the server changes frequently.
- **Efficient traffic redirection**: We must redirect the traffic, including ingress ACKs and subsequent requests, at a high rate and with low CPU usage.

Based on our analysis (§ 4.2.1), we design a HW/SW-hybrid traffic redirection mechanism (§ 4.2.2 and § 4.2.3).

4.2.1 Quickness and Efficiency Trade-Off

Linux provides two options for flow-level traffic redirection. One option is `tc-flower` [46], which matches ingress/egress packets against flow rules based on keys such as IP address and port. Its match-and-action processing can be offloaded to NIC ASICs, which are used by Open vSwitch[3]. This hardware offloading capability is available in various commodity (not particularly "smart") NICs, including Intel E810 (2020), NVIDIA ConnectX-5 (2016) and their successors, Broadcom Thor, and Netronome Agilio CX (2018).

The other option is `eBPF` programs attached to a `tc` classifier or `XDP`, both run below the TCP implementation in the stack to apply custom packet-level operations to the ingress/egress traffic. An `eBPF` program can refer to a *map*, a shared memory between the kernel, user-space, and `eBPF` programs, to make a packet processing decision.

We measured their flow installation/withdrawal time and packet forwarding performance. Our results in Table 1 highlight their key characteristics.

`eBPF` achieves rapid flow installation. Although `eBPF-XDP` provides superior packet forwarding rates by operating at the device driver level, `eBPF-tc` exhibits faster flow updates. This is because updating eBPF maps (`bpf_map_update_elem()`) is not just storing variables in shared memory but involves synchronization between the potential readers in the kernel,

[3]This is also called ASAP in NVIDIA NICs.

	Operation (µs)		Rate (Mpps)		Latency (µs)	
	Insert	Remove	64B	1500B	64B	1500B
`eBPF (tc)`	4.01	3.77	0.79	0.78	21.06	22.42
`eBPF (XDP)`	38.31	7.41	6.65	2.07	16.52	18.45
`tc (CX5)`	476	404	33.01	2.07	8.26	9.89
`tc (CX7)`	2143	1134	33.08	2.07	8.41	9.97
`tc (Agilio)`	68	65	22.12	2.07	19.77	20.58

Table 1: `eBPF` and `tc-flower` flow processing performance.

which is simpler for `tc`-attached programs as they operate at a higher position in the stack.

`tc-flower` (with hardware offload) exhibits longer flow installation times than `eBPF` due to kernel locks and device configuration, with significant variations across NICs: ConnectX-5/7 (CX5/7) requires 404–2143 µs, while Agilio needs only 65–68 µs. However, all NICs achieve high packet forwarding rates through hardware offload. Hardware-based forwarding exhibits lower CPU usage. For comparison, an eBPF program with XDP consumes 77 % of a CPU core when forwarding 1460 byte packets between 25 Gb/s links.

4.2.2 Software-Hardware Hybrid Packet Redirection

Based on our observations, we design a software-hardware hybrid approach for packet redirection in XO, which combines the advantages of both methods: fast rule insertion from `eBPF` and hardware-based packet redirection from `tc-flower`.

In our design, the L7LB runs an `eBPF` program that processes ingress packets based on a flow table stored in an `eBPF` map. When traffic redirection is needed (⑤ in Figure 2), the application on the L7LB inserts the flow rule in the `eBPF` map in a blocking manner (i.e., synchronously) while initiating `tc-flower` hardware offload in a non-blocking manner (i.e., asynchronously). As a result, packets are initially redirected by the `eBPF` rule while the hardware rule is still being configured. Once a hardware-based rule has been activated, packet redirection is done by the hardware.

Flow redirection withdrawal (⑩) must be synchronous for both software and hardware rules. This is because the remaining rule prevents the endpoint state, which has been returned to the L7LB (⑨) or offloaded to another server, from receiving connection packets (by redirecting them to the server previously used for offload).

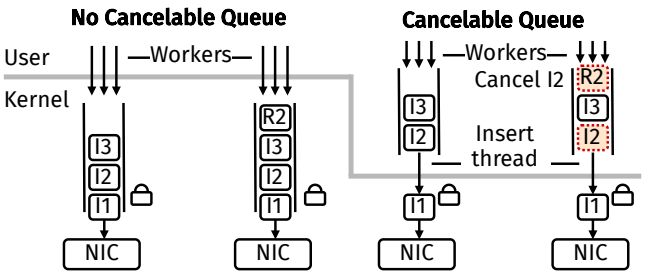


Figure 3: Command backlog with and without cancelable queue (discussed in § 4.2.3). Rule insertion and withdrawal commands are represented with I and R, respectively, followed by the flow identity.

4.2.3 Command Cancellation

Synchronous flow rule withdrawal is challenging when the rules are installed asynchronously in parallel. As shown in Figure 3 (left, no cancelable queue), when the application threads issue flow rule insertion or withdrawal commands via syscalls, those commands are serialized over global kernel locks in the `tc` subsystem and NIC driver to track hardware rules. Those commands are processed at the rate of hardware rule manipulation (Table 1).

While a delay in rule insertion is tolerable due to the co-located `eBPF` rule, a delay in rule withdrawal is problematic. Once a command (e.g., I2 in Figure 3 left) enters the kernel backlog, it cannot be *cancelled* even when the withdrawal command that offsets the insertion (R2) arrives. A deletion command must wait for its corresponding insertion to complete, and it then must be executed. This is problematic, particularly for XO where a flow rule sometimes has a short lifetime (e.g., for a single request processing).

To address this issue, we implement a *cancelable queue* (Figure 3 right) to enable the deletion command to cancel the execution of the preceding insertion command. It is a multiple-producer single-consumer queue that allows application worker threads to enqueue commands while a dedicated insertion thread pops a command and pushes it to the kernel synchronously. This design also enables bounded rule insertion latency through configuration of the queue size, allowing operators to adapt to different NICs' rule insertion speeds.

4.2.4 Ingress Blocking and Egress Address Modification

The implementation of the other flow steering operations is straightforward. For ingress traffic blocking (① and ⑦ in Figure 2), we use `eBPF-tc` due to its quick installation. This is sufficient because after connection handoff begins (① in Figure 2), incoming packets consist only of spurious retransmissions or keep-alives. Similarly, source address modification (④) can be implemented using `eBPF-tc`, because this operation incurs negligible overhead.

4.3 Limitations

Hardware redirection availability. In the practical deployment, the server or L7LB hosts could be VMs with virtual network interface (e.g., `virtio`) and thus installing flow redirection to hardware may not be possible. Therefore, in § 6, we evaluate both eBPF-only method and hardware-software hybrid method, and demonstrate superior performance in both.

Hardware reconfiguration time. Slow hardware flow rule removal may limit opportunities to use hardware-based packet redirection to when the application knows that the transmission is large or the connection stays at the same server over many subsequent requests. In § 6.3, we show that it is possible for the application to know the send size beforehand and choose whether it has the L7LB redirect ingress packets with hardware or not. To generalize this mechanism, we would define another RPC command that requests the L7LB to switch the redirection method in the existing packet redirection rules.

Failure handling. In the regular L7LB architecture, client connections could survive server failures, because they are terminated at the L7LB, which could switch the server to retrieve the response transparently to the client. With XO, client connections that have offloaded to the failed server will terminate. However, the L7LB can clean up the offload state cleanly, because it anyways tracks the offloaded connections for flow redirection rule management. Applications are expected to handle server failure with their own logic, such as activating another replica, as in Ceph (§ 5.2).

Open-loop application support. Although many L7LB systems are closed-loop (§ 2), open-loop ones where the client sends the next request without waiting for the response to the previous one in the same connection also exist. When the next request arrives at the L7LB or server while state transfer triggered by the previous request is in progress, that request packet could be dropped due to the flow blocking rule. However, since state transfer completes a few 100s of µs (§ 6.1 and § 7.1), which is before the retransmission timeout, the lost packets could be recovered by fast retransmit. This would not be a problem if the backend servers are replicas, because state transfers are infrequent. When frequent state transfers are needed and loss is critical, we could adopt packet-level buffering as with HA/TCP [20] and Capybara [10].

TCP parameters. Our current implementation relies on the Linux kernel's TCP state serialization mechanism, which does not restore some state variables from previous TCP connections. Those parameters include the congestion window, measured RTT and other congestion control parameters like RACK state [9]. Restoring those parameters would improve the server transmission performance.

4.4 Implementation

XO runs on the Linux network stack, which is crucial for practicality [43, 7]. XO does not require kernel modification, instead leveraging the existing stack features including `tc` and `eBPF` subsystems. We implement `libforward`, a library that implements the cancelable queue and interfaces for hardware and `eBPF` flow redirection and other rule management.

We also implement a custom RPC protocol for state transfer between L7LB and servers (§ 4.1). Since it directly operates on file descriptors, applications that monitor regular communication sockets using standard event interfaces, such as `epoll` and `io_uring`, can easily add monitoring of those RPCs.

5 Real-World Application Integration

Approaches that enable direct server return on L7LBs require application modification (§ 2.2), because the server must perform the L7LB's tasks and maintain the accepted sockets associated with the clients. Those systems have proposed creating a new programming abstraction (e.g., `libuv`-based event loop for Prism), but we found such approaches to be impractical when applied to real-world applications, because they already have considerable complexity. In fact, Prism and other connection-migration approaches (§ 8) have not been applied to real-world applications, but instead build custom applications from scratch.

In this section, we report our experience of adding support for XO in two real applications—`nginx` and Ceph—to create case studies that help other L7LB-based systems support XO. We have chosen those applications for the diversity of their features. `nginx` is a general-purpose HTTP reverse proxy that assumes replicated backend servers. RGW is an application-specific (i.e., Ceph) L7LB for sharded storage backend servers. We discuss our lessons learned from our application integration effort in § 7.3.

5.1 Nginx

`nginx` is an event-driven web server that also acts as an L7LB or reverse proxy. It functionalities are organized into modules, such as the event module, which runs an epoll event loop, the HTTP module, which implements the HTTP parsers, and the mail module, which implements POP3 proxying. Modules can be chained and depend on other modules, which is described in the `nginx.conf` configuration file.

We introduce the XO module that implements connection serialization and restoration operations, as well as flow steering operations using `libforward`. It also implements the policies to make the offload decision.

The XO module depends on the HTTP module. An incoming request is first processed by the HTTP module, which passes the result to the XO module using `nginx`'s standard interface between modules. Therefore, once the `nginx` is compiled with the XO module, the operator can activate the use of XO for a specific service through the configuration file (found in the artifact mentioned in § 9).

The XO module implements two commands invoked by the HTTP module. The first one is `xo_out` used by the L7LB that makes a connection offload decision and initiates state transfer. The other is `xo_in` used by the server. It receives the connection state and subsequent requests in that connection from the L7LB.

Although the XO module mostly follows the intended way to extend `nginx`, we needed a small modification in the `nginx` core. Since XO requires offload termination procedure when the connection is closed, The XO module registers the connection closure callback. However, the current `nginx` does not allow external modules to store additional metadata used by that callback. Therefore, we needed to add one data pointer in the core connection structure.

5.2 Ceph

Ceph [51] is an open-source object storage system designed for scalability, reliability, and availability. Its storage backends are organized as object storage daemons (OSDs) running on different hosts. Objects are mapped into OSDs based on the CRUSH algorithm, where any party can *compute* the OSD that holds a given object without RPCs so as to maximize scalability and minimize latency. Objects are replicated over multiple OSDs, but each object is served only by its primary OSD to ensure data consistency. OSD failures are detected by heartbeats exchanged among OSDs. The Ceph monitor then updates the OSD map to mark the failed OSD as down, allowing CRUSH to return new OSDs for objects previously mapped to the failed OSD.

Ceph provides multiple abstractions, including CephFS (file system), RBD (block), and RGW (S3-compatible object storage). We focus on RGW. As shown in Figure 4, after receiving a request over the S3 protocol, RGW computes the OSD and issues a request to the OSD host using the `msgr2` protocol. The OSD retrieves the data from the storage and returns it to the gateway. The gateway encapsulates the data into an HTTP reply, encrypts it with TLS, and sends it to the client. Optionally, the gateway caches the object to avoid future communication with the OSD.

Since objects are served by specific OSD hosts, bandwidth and locality could be improved if they directly return data to a client. We implement XO-Ceph, which replaces RGW and handles offloaded connections at OSDs. Once receiving a GET request, the XO-Ceph gateway computes the OSD of the object and offloads the connection to that OSD host. The OSD host restores the connection, reads and encapsulates data locally, and sends the response to the client directly. If the next request asks for an object on another OSD, the OSD returns the connection to the gateway. The gateway then offloads the connection to that OSD host. Therefore, unlike XO-`nginx`,

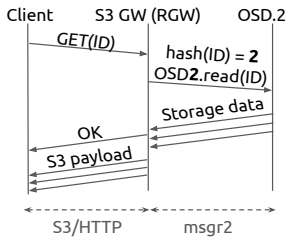


Figure 4: The workflow of an ordinary Ceph S3 object gateway.

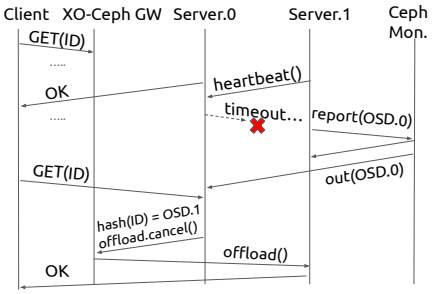


Figure 5: A XO-Ceph gateway offloading to a secondary OSD host due to failover.

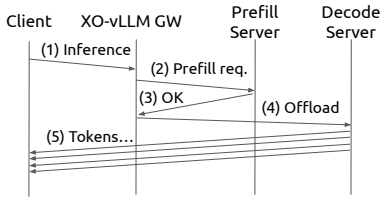


Figure 6: vLLM with XO which offloads the decoding stage (discussed in § 7.4).

offload decisions are dictated by object location, rather than load balancing policies. Furthermore, XO eliminates the need for caching at the gateway, because data is directly sent to the client by the OSD.

XO-Ceph does not undermine the OSD failure handling in Ceph. In Figure 5, a connection currently offloaded to Server.0 receives a request for an object residing there. However, the OSD process is down, and since CRUSH now points to a secondary OSD, the server returns the connection to the gateway. The gateway then offloads the connection to Server.1, the secondary OSD.

XO-Ceph is implemented using `librados`, the Ceph storage abstraction accessed by all the Ceph abstractions (i.e., CephFS, RBD and RGW), which means no core modification to Ceph core is necessary. We slightly extended `librados` to return object location from its internal memory mapping with a small access function, which we intend to issue a pull request to Ceph. Similar to `nginx`, XO-Ceph manages state transfer and flow steering using `libforward`.

6 Evaluation

We evaluate XO using a custom microbenchmark application (§ 6.1) and two real-world applications discussed in § 5: `nginx` (§ 6.2) and Ceph (§ 6.3). All the systems use TLS/TCP for client-facing communication unless otherwise stated. Our main results are:

1. XO achieves up to 365.6 % higher throughput than regular L7LB by efficiently utilizing network and CPU resources of the backend servers (§ 6.1).
2. XO improves throughput of `nginx`, which employs replicated backend servers, by up to 300%. We also confirm CPU usage convergence over dynamic load (§ 6.2).
3. XO improves throughput of Ceph, which employs sharded storage backend servers, by up to 135% (§ 6.3).

6.1 Basic Performance Characteristics

We begin by measuring XO's basic performance using nophttpd, a custom HTTP server with minimal application-level overhead. It also acts as an L7LB. The nophttpd cluster operates as a regular L7LB-based system (denoted as Proxy in the plots), where the L7LB relays data between the client and servers, or with XO enabled, where the L7LB offloads client connections to the servers. nophttpd is multi-threaded, and each thread processes state transfer RPCs (§ 4.1) and client- or L7LB-facing connections on an `epoll` event loop. For TLS, it uses kTLS [25]. The client runs `wrk` that issues HTTP/1.1 GET requests over 600 persistent connections. Unless otherwise stated, requests are evenly distributed to backend servers.

We use six machines connected to the same switch. One client, equipped with two EPYC 9334 CPUs and 256 GB of RAM, connects to the switch with a 100 Gb/s link, emulating the mass of the external clients over the high-bandwidth switch uplink. The other five machines are equipped with two Xeon E5-2630v4 CPUs, 32–64 GB of RAM, and NVIDIA ConnectX-5 NIC, and connect to the switch with a 25 Gb/s link, acting as the cluster nodes at the switch downlinks. One of them is dedicated as an L7LB, and the rest are used as servers. The L7LB machine has an additional Netronome Agilio NIC to test the effect of different reconfiguration times.

Figure 7 plots throughput, latency, and L7LB and server CPU usage. We vary the object size that the client downloads and the number of requests processed in a single offload cycle (i.e., between one outbound and inbound state transfer). Frequent state transfer cases represent systems with sharded backends, because the next request is more likely needed to be handled by another server. Infrequent state transfer cases represent load balancing between replicated servers, where the server unlikely has to return the connection to the L7LB for every request. For XO variants, XO-eBPF only uses eBPF (i.e., software) for packet redirection (thus no hardware rule synchronization cost, see § 4.2), whereas XO-CX5 performs hybrid packet redirection with a ConnectX-5 NIC, and XO-Agi. uses an Agilio NIC.

Throughput and link utilization. In a regular L7LB (Proxy), throughput is limited by L7LB's link speed for 64 K B or larger objects (we experiment cases where Proxy is unconstrained by link bandwidth in § 6.3). When objects are 64 K B or smaller and state transfer happens for every request (leftmost column), Proxy outperforms XO variants by at most

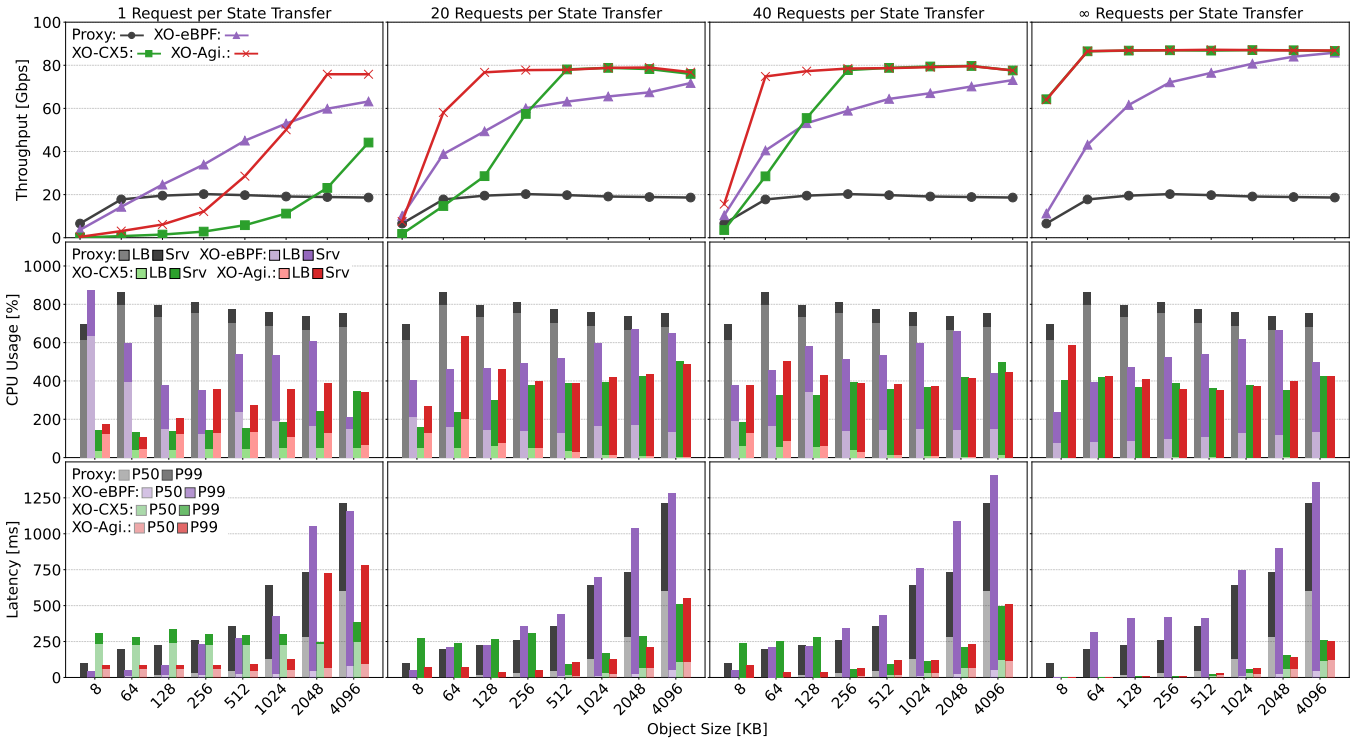


Figure 7: Throughput, CPU usage and request latency of regular L7 proxy and XO. Each column indicates the number of requests processed by the same server before it returns the connection to the L7LB, which then re-migrates it to the server. This parameter is irrelevant for `Proxy` so the same values are plot across the columns. Server CPU usage represents that of one server out of four (all show similar usage due to uniform connection distribution). Each machine has 8 cores, so the maximum CPU usage is 800%.

25.4 % due to the state transfer costs. In other scenarios XO variants outperform Proxy by up to 365.6 %. XO's throughput scales towards the aggregate link capacity of all the servers (4× 25 Gb/s) as the state transfer cost is amortized over larger responses (left to right in each plot) or more requests processed in a single state transfer (left to right columns).

XO-eBPF exhibits higher throughput than XO-CX5 and XO-Agi. when the object size is small and state transfers are frequent (left plot), because the efficiency of hardware-based packet redirection does not outweigh the hardware rule withdrawal cost (§ 4.2.2). XO-Agi. performs better than XO-CX5 due to lower hardware reconfiguration delay (Table 1).

L7LB and server CPU usage. Figure 7 middle row plots CPU usage of the L7LB and one of the servers (equally loaded due to uniform request and connection distribution). When comparing Proxy and XO-eBPF, except for when the relative (i.e., per-request) state transfer costs are very high (see aforementioned discussion in throughput), XO exhibits lower CPU usage at the L7LB, even though achieving higher throughput (see corresponding throughput data points), because of packet-level ingress traffic redirection and direct server return. XO shows a higher server-to-LB CPU usage ratio than Proxy, because the server shoulders L7LB tasks like encryption.

Proxy exhibits almost full (800% for 8 cores) CPU usage at the L7LB. This implies that it would not achieve much higher throughput even if L7LB's link had higher bandwidth.

With hybrid redirection (XO-Agi./CX5), we still see some CPU usage at the L7LB caused by the state transfer operations, but it decreases when the state transfer costs are amortised over larger responses or more requests between state transfers.

Request latency. Figure 7 bottom shows P50 and P99 request latency. When state transfer latency is dominant, XO exhibits constant latency (XO-CX5 in the leftmost plot), whereas when queuing latency is dominant, it increases latency with higher throughput (other plots of XO variants). In many cases, XO variants achieve lower request latency than Proxy although achieving higher throughput, because the request backlog that otherwise forms at the L7LB is spread across the servers, and its advantage outweighs state transfer latency. As a result, XO achieves lower latency than Proxy by up to 95.3 % at P50 (second left plot, 64 K B with XO-CX5) and by up to 85.7 % (second left plot, 128 K B with XO-Agi.) at P99 .

Connection state transfer latency. Table 2 shows the latency breakdown of outbound state transfer of XO; inbound state transfer takes a similar time, plus, hardware rule removal

Operation	Latency [µs]
(L) Block flow	4
(L) Serialize TLS	2
(L) Serialize TCP	11
(L->S)State transferring to the server	148
(L) Sending RPC: BEGIN_OFFLOAD	
(S) Restore TCP	39
(S) Restore TLS	15
(S) Install source IP rewrite rule	5
(L) Received RPC: SERVER_READY	
(L) Install redirection rule and unblock	4
(L) Send RPC: L7LB_READY	56
Total	**225**

Table 2: Outbound state transfer latency breakdown. L and S indicate operations at L7LB or the server, respectively. All indented operations are included in the state transfer RPC.

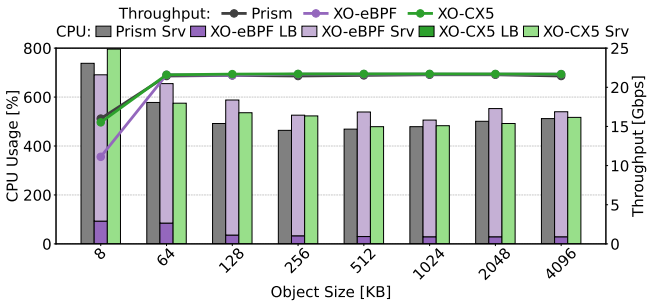


Figure 8: Throughput and CPU usage in comparison to Prism.

time (Remove column in Table 1) that must occur in a synchronous manner (§ 4.2.2). We discuss and measure optimizations that can be used if we modify the kernel in § 7.1.

6.1.1 Comparison to Prism

Prism is an L7LB enhancement based on programmable switches (§ 2.2). We cannot compare Prism and XO with state transfer operations because we do not have a programmable switch. However, we can compare the stable-state performance of these systems (i.e., the endpoint has been transferred to a server and stays there), because in Prism the switch redirects ingress traffic to the migrated endpoint at a line rate and performs direct server return like XO. We can thus just use an ordinary server application behind a regular switch to emulate a Prism endpoint.

Figure 8 plots throughput and CPU usage of XO variants and emulated Prism. With hardware-based packet redirection (XO-CX5), XO achieves comparable throughput with Prism even for small objects, indicating negligible overhead of extra packet hops (i.e., switch-to-L7LB and L7LB-to-switch). This implies that, if hardware rule installation (required by XO) is done as quickly as remote switch reconfiguration (required by Prism), XO could achieve a comparable performance with Prism in the presence of frequent state transfers.

On the other hand, without hardware-based packet redirection (XO-eBPF), small object (8 K B) throughput is penalized by at most 30.5 %. However, since local eBPF map update (takes 4 µs, see Table 1) could be faster than remote switch

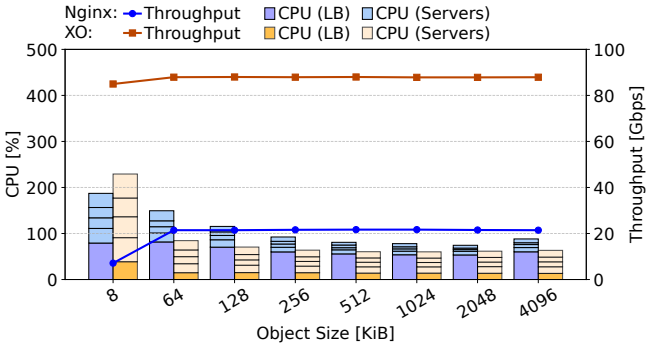


Figure 9: Throughput and CPU utilization of `nginx` cluster.

ASIC reconfiguration and thus connection migration is faster, when state transfer happens at a certain frequency, packet redirection inefficiency would be offset in some degrees; our result shows the case where it is not offset at all.

6.2 Nginx Experiment

We now evaluate XO integrated with real-world applications that have higher software complexity and specific usage patterns. We first test `nginx` (§ 5.1) using the setup in § 6.1, where one machine acts as a reverse proxy. Since `nginx` statically maps client connections to the servers, which are replicas that can process any client request, using consistent hashing[4], XO-`nginx` applies the same policy by migrating a connection to the server only at the first request. We use XO-eBPF; since we do not move connections frequently, we expect larger benefit of XO with hardware-based redirection. We do not use TLS encryption in `nginx` experiment.

Figure 9 shows throughput and CPU utilization. Although the L7LB and servers are connected to the switch over a 25 Gb/s link, XO-`nginx` achieves 87 Gb/s, utilizing the aggregated link bandwidth of all the servers, whereas `nginx` achieves 22 Gb/s, constrained by the L7LB link bandwidth. XO-`nginx` utilizes cluster CPU resources efficiently by shifting the load from the L7LB to the servers; higher total CPU usage in XO-`nginx` is due to higher overall throughput.

Next, as an advanced use case, we demonstrate dynamic load balancing with XO-`nginx`. We implement a simple mechanism and policy, where load monitoring daemons on each backend exchange their current CPU load every half second via a key-value store in the L7LB and write everyone's load in their local shared memory; every time it receives a request, the `nginx` XO module in each backend reads that shared memory and decides whether it returns the connection to the L7LB or not based on the load of its own and others.

In Figure 10, requests to fetch an 8 KiB file are initially served by all the servers, each handling the same number of connections and thus a similar request load. After 30 seconds, all the connections handled by server 1 (dark blue area) leave,

[4] https://nginx.org/en/docs/http/load_balancing.html

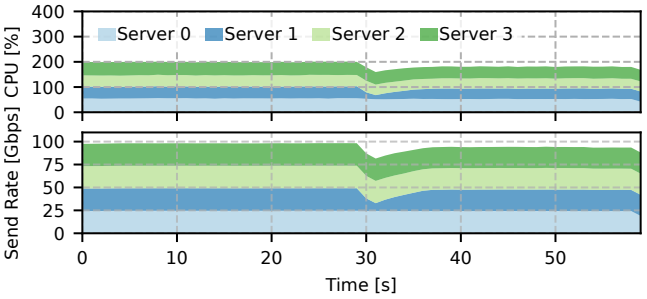


Figure 10: XO rebalancing load when all the client connections at Server 1 leave at $T = 30$.

making this server idle. Once the load discrepancy is detected by the other servers, they return some connections to the L7LB, which subsequently offloads those connections to the least loaded server, which is server 1. After a brief fluctuation, it results in a balanced CPU and network load again; overall CPU utilization reduces due to fewer connections.

6.3 Ceph Experiment

Finally, we evaluate XO-Ceph (§ 5.2). Since we need NVMe SSDs, we use another cluster consisting of five machines. Four of those are equipped with two Xeon Silver 4314 CPUs, 256GB of RAM, NVIDIA ConnectX-6 NIC and Samsung PM9A3 SSD. We use three of those to run servers (OSDs, storage backends), and the other to run the L7LB (XO-Ceph gateway or RGW, the baseline). The remaining machine is equipped with two EPYC 9334 CPUs and used as the client. All the machines install Linux kernel 6.6 and connect to the same switch with 100 Gb/s links.

We populate the object store beforehand, and the client issues requests following three object popularity distributions: Zipf-1.1 (most skewed), Zipf-0.9, and uniform. Figure 11a and Figure 11b plot the throughput and CPU utilization of L7LB and storage backends, respectively. We apply one optimization. Since the OSD knows the object size it is sending upon receiving the request (i.e., before making a decision whether it returns the connection to the L7LB or not), it requests the software-hardware hybrid packet redirection method to the L7LB only for objects larger than 2MiB (in SERVER_READY RPCs in § 4.1), because software redirection is likely more efficient when the object is small (§ 6.1).

XO outperforms RGW in throughput with 256KiB or larger objects in all the workloads, reaching up to 135 % improvement. This indicates that, at this object size, state transfer costs are amortized by direct server return and improved utilization of the storage server CPU resources. This is reflected in CPU utilization; servers behind RGW remain largely underutilized. Note that 256KiB objects are not particularly large in Ceph; a recent industry-led analysis [40] uses 64KiB, 4MiB, 32MiB, 64MiB and 256MiB objects as representative sizes.

XO-CX6 shows the advantage of hardware packet redirection for 2MiB or larger objects, reducing L7LB CPU usage compared to software packet redirection in XO-eBPF. XO-CX6 achieves similar performance to XO-eBPF for smaller objects, because it actually avoids using hardware-based redirection for those objects based on the optimization described earlier in this subsection. When XO is used and the workload is highly skewed (Zipf-1.1), CPU utilization on one backend is significantly higher than that on others, because Ceph serves each object only from the primary OSD (replicas are used for fault tolerance; see § 5.2) and thus OSDs hosting popular objects handle more connections or requests.

7 Discussion

7.1 State Transfer Optimization

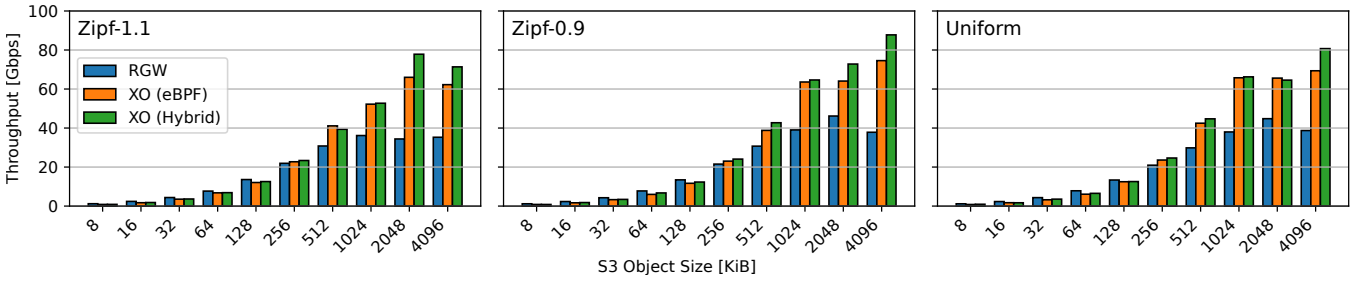
Although we built XO on Linux without network stack or kernel modifications for easy deployment, such modifications could accelerate XO. The current Linux APIs to serialize or restore TCP connection state need 13 syscalls with each acquiring and releasing the socket lock. Since the actual operation in each call is cheap (e.g., just setting a flag), we could accelerate connection serialization and restoration by consolidating these operations into a single new call and socket lock. We thus implemented these consolidated calls—`TCP_REPAIRALL` and `TCP_RESTORE`—as new commands for `get/setsockopt`. These calls serialize a connection in 8 µs (11 µs with existing APIs, see Table 2) and restore it in 28 µs (39 µs otherwise).

Those consolidated calls improve end-to-end performance. Figure 12 plots the results with XO-eBPF when connection offload occurs on every request (setup in Figure 7 left). We observed 38.4–83.1 % improvement in throughput, with latency reduction of 1.7–54.2 and 9.3–87.7 % at P50 and P99, respectively.

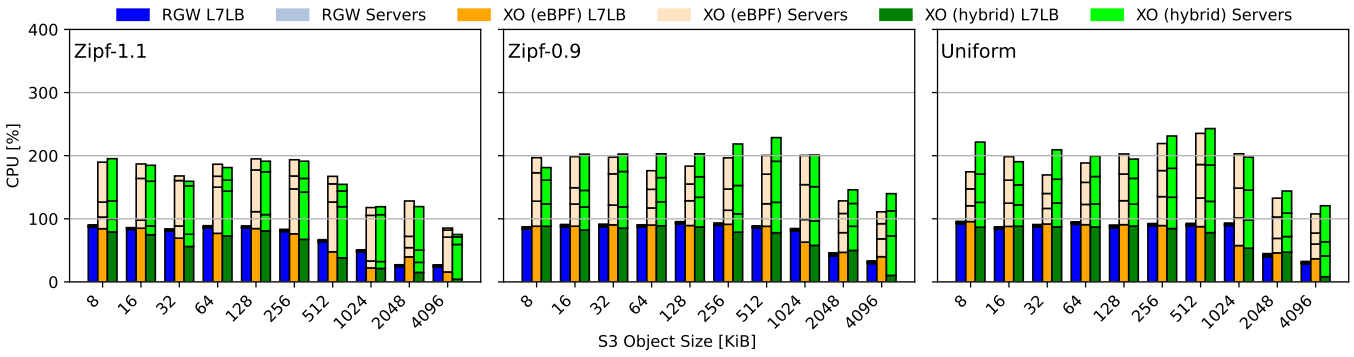
7.2 NIC Design Implications

The performance gap between XO-CX5 and XO-Agilio highlights the impact of hardware capabilities on XO and calls for new NIC designs.

We believe future hardware-software interfaces should take into account at least reconfiguration time, concurrency, and the consistency model. Such NIC abstractions would accelerate XO-like systems, for example, enabling multiple flow rule insertions in parallel. Some modern NICs, such as Pensando Elba, are equipped with programmable P4 ASICs. Since those ASICs are designed with frequent match-action rule updates in mind, they will enable faster hardware rule updates than those we tested in this paper.



(a) Throughput



(b) CPU Utilization

Figure 11: XO-Ceph performance. RGW refers to the default storage gateway for S3 access in Ceph. hybrid uses a CX6 NIC.

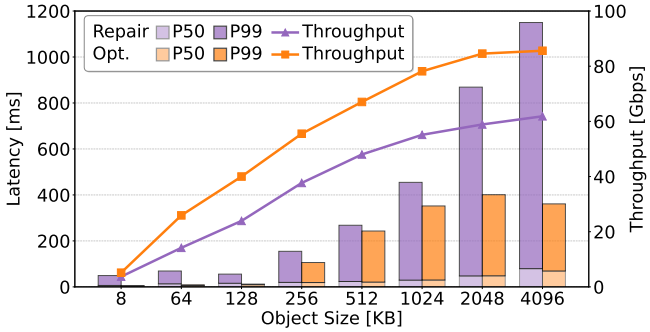


Figure 12: Throughput and latency improvement with optimized connection serialization/restoration (§ 7.1).

7.3 Application Support

Connection migration-based approaches inherently require server application modification because the application must handle both migrated and accepted connections. We can reuse the server selection policy of the base L7LB. For example, in Ceph, we can use the object location, whereas we can use a simpler strategy in `nginx` because it assumes replicated servers. The software architecture also dictates porting effort. `nginx` has a modular design, which exposes sockets and its internal `epoll` loop across the modules, enabling trivial integration. Ceph requires a slight modification of its `librados` to expose locality information beyond its intended boundary. In total, it took approximately two weeks to integrate XO as an `nginx` module; whereas it took one month to develop XO-GW in Ceph where we found complexity in supporting S3-compatible services.

Our choice of using the Linux kernel network stack eased real application integration, because those applications heavily rely on the underlying semantics of file descriptors, asynchronous I/O, and threading. It would have been difficult if we opted for kernel-bypass stacks like Demikernel used by Capybara [10] (§ 8), because they often require an unusual programming model for applications [2]. The feature-rich Linux stack is also suitable for Internet-facing L7LBs. If we assume in-datacenter clients and that the server does not need modern TCP features, using kernel-bypass stacks that offer multicore capability, such as Junction [18], possibly with other techniques like Crab [26, 52] (§ 8), would be an option. We leave this direction to future work.

7.4 Further Use Case

XO could support other L7LB-based applications, such as vLLM [28], a popular large-language model serving and inference system. LLM inference has two stages: prefill (tokenizing prompts and generating the first token) and decode (producing the remaining output). vLLM improves resource utilization by running these stages on different hosts, where each host provides an OpenAI-compatible HTTP API service, and a gateway [49] similar to an `nginx` reverse proxy relays requests between the two stages.

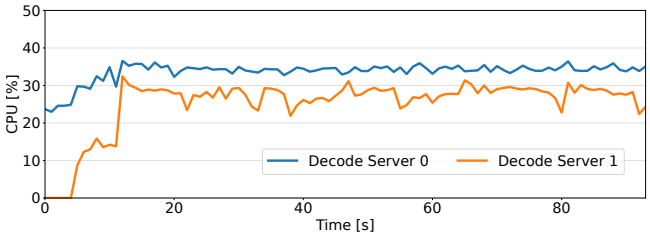


Figure 13: Load balancing between vLLM decode servers.

We prototype XO-vLLM, which provides similar functionality to the vLLM gateway. It supports disaggregated prefilling while allowing the decode server to directly return the response to the client. Figure 6 shows the workflow of XO-vLLM. It offloads the connection *after* processing the prefill request, so that continuation decoding and token streaming can be directly served by the decode server. In the figure, after the gateway accepts an inference request (1), it performs a request for prefill (2 and 3), then executes connection offload to the decode server (4). The decode server generates response tokens and sends them directly to the client (5). On the next client request, it returns the connection to the gateway, repeating the same process (1).

Although this architecture requires frequent connection migrations, XO's state transfer costs could be negligible, because inference takes 10s of milliseconds per token [48], while XO adds only several 100s of microseconds (Table 2).

To demonstrate load-balancing, we deploy an XO-vLLM gateway with one prefill and two decode servers, using a portion of the cluster used in the Ceph experiment (§ 6.3). Since we do not have access to GPUs, we emulate vLLM instances using a lightweight Python server. This emulation is reasonable, because it follows the OpenAI-compatible API and returns valid responses. In Figure 13, the client gradually increases the number of connections until $t = 12$. All the arriving connections are offloaded to server 0 until its CPU load reaches to 30%, which we define as the target capacity of each server, at around $t = 4$. The gateway then starts offloading subsequent connections to server 1 and thus load increase at server 0 stops. The load at server 1 increases with further connection arrivals, which stop around $t = 12$.

8 Related Work

This paper extends our previous short paper [30] with detailed design, evaluation, and discussion.

Connection migration-based systems. HA/TCP [20] uses connection migration for failover of L7 middleboxes between active-standby replicas, where the primary node mirrors ingress traffic to the secondary node. Its connection migration is simpler than XO, because the nodes share the same IP address and only one of them is active at a time, obviating address translation and packet redirection. It is unclear how to use HA/TCP in existing applications, because its current applications are all built from scratch. Capybara [10] migrates TCP connections between servers using a programmable switch, sharing the deployment problem with Prism.

Connection splicing. Connection splicing is often discussed in the context of L7LB systems, because it terminates client TCP connections and relays data to and from the server. It enables efficient data movement in the kernel [47, 32, 5] or hardware [34]. However, it is incompatible with general L7LBs (see § 2), because it can only perform the server selection at the connection setup or first request and cannot perform application-level processing, such as per-request server selection and protocol translation. Performance Enhancing Proxies (PEPs) also split TCP connections, but they are used to apply different congestion control algorithms in the Internet [53], rather than application-level processing.

Other LB enhancements. Yoda [19] is an L7LB design that stores the connection information in shared storage for failure recovery. The concept is orthogonal to XO and would help the XO L7LB handle server failures more proactively. LB-NIC [11] runs L7LBs with kernel-bypass TCP stack in the host SmartNIC. It could apply XO in its L7LB to further reduce the L7LB load.

QDSR [50] proposes that multiple backends serve different streams aggregated into a single QUIC connection at the client. It is unclear if the solution is general, because not only is it specific to QUIC, but the object needs to be served simultaneously by the servers. For example, Ceph designates one backend to serve each particular object to ease consistency guarantee, although objects are replicated for fault tolerance. DISC [16] applies a similar concept to QDSR but to TCP.

9 Conclusion

This paper presented XO, which accelerates L7LBs by streamlining the client-to-L7LB-to-server datapath while supporting essential L7LB requirements, such as per-request, application-driven server selection and protocol translation. XO's design has a strong emphasis on practicality, requiring no access to the network switches and being built on the feature-rich Linux kernel network stack. This enabled us to apply XO to real-world L7LB systems, including `nginx`, which is a general HTTP reverse proxy, and Ceph, which contains the RGW L7LB specifically designed for it. All the code used in this paper and the documentation are available in https://github.com/uoenoplab/xo.

Acknowledgments

We thank Jamal Hadi Salim, Yutaro Hayakawa, anonymous reviewers, and our shepherd, Gianni Antichi, for their valuable comments or discussion to improve this paper. We also thank the anonymous artifact evaluation committee member who assessed our code and documentation using xl170 nodes in CloudLab [13]. This work was in part supported by EPSRC grant EP/V053418/1, Meta Research Award and gift from Google and NetApp.

References

[1] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. "Balancing on the edge: Transport affinity without network state". *Proc. USENIX NSDI*. 2018.

[2] Shinichi Awamoto and Michio Honda. "Opening Up Kernel-Bypass TCP Stacks". *Proc. USENIX ATC*. 2025.

[3] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. "A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency". *Proc. USENIX NSDI*. 2020.

[4] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. "Finding a Needle in Haystack: Facebook's Photo Storage." *Proc. USENIX OSDI*. 2010. 2010.

[5] Daniel Borkmann and John Fastabend. "Combining kTLS and BPF for Introspection and Policy Enforcement". *Linux Plumbers Conference*. 2018.

[6] Willem de Bruijn and Eric Dumazet. *sendmsg copy avoidance with MSG_ZEROCOPY*. https://netdevconf.info/2.1/papers/debruijn-msgzerocopy-talk.pdf. 2017.

[7] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. "Towards μ s tail latency and terabit ethernet: disaggregating the host network stack". *Proc. ACM SIGCOMM*. 2022.

[8] Ceph authors and contributors. *Ceph Object Gateway*. https://docs.ceph.com/en/reef/radosgw/.

[9] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. *The RACK-TLP Loss Detection Algorithm for TCP*. RFC 8985. 2021. URL: https://www.rfc-editor.org/info/rfc8985.

[10] Inho Choi, Nimish Wadekar, Raj Joshi, Joshua Fried, Dan RK Ports, Irene Zhang, and Jialin Li. "Capybara: μSecond-Scale Live TCP Migration". *Proc. ACM APSys*. 2023.

[11] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. "Offloading load balancers onto smartnics". *Proc. ACM APSys*. 2021.

[12] Eric Dumazet and Coco Li. *Going big with TCP packets*. https://lwn.net/Articles/884104/.

[13] 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". *Proc. USENIX ATC*. 2019.

[14] Wesley Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. 2022. URL: https://www.rfc-editor.org/info/rfc9293.

[15] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. "Maglev: A fast and reliable software network load balancer." *Proc. USENIX NSDI*. 2016.

[16] Brice Ekane, Djob Mvondo, Renaud Lachaize, Yérom-David Bromberg, Alain Tchana, and Daniel Hagimont. "DISC: Backpressure Mitigation In Multi-tier Applications With Distributed Shared Connection". *Proc. USENIX NSDI*. 2025.

[17] Envoy Project Authors. *Envoy*. https://www.envoyproxy.io/.

[18] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. "Making kernel bypass practical for the cloud with junction". *Proc. USENIX NSDI*. 2024.

[19] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. "Yoda: A Highly Available Layer-7 Load Balancer". *Proc. ACM EuroSys*. 2016.

[20] Haoyu Gu, Ali José Mashtizadeh, and Bernard Wong. "HA/TCP: A Reliable and Scalable Framework for TCP Network Functions". *Proc. USENIX NSDI*. 2025.

[21] HAProxy. *HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer*. http://www.haproxy.org/. URL: http://www.haproxy.org/.

[22] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. "Prism: Proxies without the Pain." *Proc. USENIX NSDI*. 2021.

[23] Andrew Johnson, Ryan Beckett, Xiaoqi Chen, Ratul Mahajan, and David Walker. "Sequence Abstractions for Flexible, Line-Rate Network Monitoring". *Proc. USENIX NSDI*. 2024.

[24] Kanchan Joshi, Anuj Gupta, Javier Gonzalez, Ankit Kumar, Krishna Kanth Reddy, Arun George, Simon Lund, and Jens Axboe. "I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux". *Proc. USENIX FAST*. 2024.

[25] *Kernel TLS offload*. https://www.kernel.org/doc/html/latest/networking/tls-offload.html.

[26] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. "Bypassing the load balancer without regrets". *Proc. ACM SoCC*. 2020.

[27] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. "Network virtualization in multi-tenant datacenters". *Proc. USENIX NSDI*. 2014.

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention". *Proc. ACM SOSP*. 2023.

[29] Raul Landa, Lorenzo Saino, Lennert Buytenhek, and João Taveira Araújo. "Staying alive: Connection path reselection at the edge". *Proc. USENIX NSDI*. 2021.

[30] Shuo Li, Steven W.D. Chien, Tianyi Gao, and Michio Honda. "Remote TCP Connection Offload with XO". *Proc. ACM APNet*. 2025.

[31] LWN.net. *TCP connection repair*. https://lwn.net/Articles/495304/.

[32] David A Maltz and Pravin Bhagwat. "TCP Splice for application layer proxy performance". *Journal of High Speed Networks* (1999).

[33] Jeffrey C Mogul and John Wilkes. "Physical Deployability Matters". *Proc. ACM HotNets*. 2023.

[34] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. "AccelTCP: Accelerating network applications with stateful TCP offloading". *Proc. USENIX NSDI*. 2020.

[35] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. "Sketchovsky: Enabling Ensembles of Sketches on Programmable Switches". *Proc. USENIX NSDI*. 2023.

[36] Nginx. *NGINX | High Performance Load Balancer, Web Server, Reverse Proxy*. https://www.nginx.com/. URL: https://www.nginx.com/.

[37] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. "Stateless datacenter load-balancing with beamer". *Proc. USENIX NSDI*. 2018.

[38] Tian Pan, Kun Liu, Xionglie Wei, Yisong Qiao, Jun Hu, Zhiguo Li, Jun Liang, Tiesheng Cheng, Wenqiang Su, Jie Lu, Yuke Hong, Zhengzhong Wang, Zhi Xu, Chongjing Dai, Peiqiao Wang, Xuetao Jia, Jianyuan Lu, Enge Song, Jun Zeng, Biao Lyu, Ennan Zhai, Jiao Zhang, Tao Huang, Dennis Cai, and Shunmin Zhu. "LuoShen: A Hyper-Converged Programmable Gateway for Multi-Tenant Multi-Service Edge Clouds". *Proc. USENIX NSDI*. 2024.

[39] Tian Pan, Enge Song, Yueshang Zuo, Shaokai Zhang, Yang Song, Jiangu Zhao, Wengang Hou, Jianyuan Lu, Xiaoqing Sun, Shize Zhang, et al. "Hermes: Enhancing Layer-7 Cloud Load Balancers with Userspace-Directed I/O Event Notification". *Proc. ACM SIGCOMM*. 2025.

[40] Daniel Alexander Parkes and Anthony D'Atri. *Benchmarking the Ceph Object Gateway: A Deep Dive into Secure, Scalable Object Storage Performance. Part 2*. https://ceph.io/en/news/blog/2025/benchmarking-object-part2/. 2025.

[41] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. "Autonomous NIC offloads". *Proc. ACM ASPLOS*. 2021.

[42] Xiaoyi Shi, Lin He, Jiasheng Zhou, Yifan Yang, and Ying Liu. "Miresga: Accelerating Layer-7 Load Balancing with Programmable Switches". *Proceedings of the ACM on Web Conference*. 2025.

[43] 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". *Proc. USENIX OSDI*. 2024.

[44] Squid. *Squid: Optimising Web Delivery*. http://www.squid-cache.org/. URL: http://www.squid-cache.org/.

[45] Yevgeniy Sverdlik. *How is a Mega Data Center Different from a Massive One?* https://www.datacenterknowledge.com/archives/2014/10/15/how-is-a-mega-data-center-different-from-a-massive-one.

[46] *tc-flower(8) — Linux manual page*. https://man7.org/linux/man-pages/man8/tc-flower.8.html.

[47] *Transparent proxy support*. https://docs.kernel.org/networking/tproxy.html.

[48] vLLM. *vLLM v0.6.0: 2.7x Throughput Improvement and 5x Latency Reduction*. https://blog.vllm.ai/2024/09/05/perf-update.html.

[49] vLLM Router project. *vLLM Router*. https://blog.vllm.ai/2025/12/13/vllm-router-release.html.

[50] Ziqi Wei, Zhiqiang Wang, Qing Li, Yuan Yang, Cheng Luo, Fuyu Wang, Yong Jiang, Sokoe Yang, and Zhenhui Yuan. "QDSR: Accelerating Layer-7 Load Balancing by Direct Server Return with QUIC". *Proc. USENIX ATC*. 2024.

[51] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. "Ceph: A scalable, high-performance distributed file system". *Proc. USENIX OSDI*. 2006.

[52] Rui Yang and Marios Kogias. "Heels: A host-enabled ebpf-based load balancing scheme". *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*. 2023.

[53] Gina Yuan, Thea Rossman, and Keith Winstein. "Internet Connection Splitting:What's Old is New Again". *Proc. USENIX ATC*. 2025.

[54] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C Mogul, and Amin Vahdat. "Minimal rewiring: Efficient live expansion for clos data center networks". *Proc. USENIX NSDI*. 2019.