Remote TCP Connection Offload with XO

Shuo Li*
University of Edinburgh
Edinburgh, United Kingdom
s.li@ed.ac.uk

Steven W.D. Chien*
University of Edinburgh
Edinburgh, United Kingdom
steven.chien@ed.ac.uk

Tianyi Gao
University of Edinburgh
Edinburgh, United Kingdom
tianyi.gao@ed.ac.uk

Michio Honda
University of Edinburgh
Edinburgh, United Kingdom
michio.honda@ed.ac.uk

Abstract

Efficient resource utilization in server clusters is essential for maximizing service capacity and minimizing latency while reducing infrastructure costs, whether in edge clouds or hyperscale deployments. Current approaches face significant limitations: layer 4 load balancing (L4LB) alone causes load imbalance over time with long-lived connections, while layer 7 load balancing (L7LB) introduces substantial CPU, memory, and network overhead despite enabling fine-grained server selection based on application-level requests.

This paper presents XO, a set of concept and techniques to enable a TCP server to offload entire TCP connection and application-request processing to another machine at request granularity. Together with L4LB, XO achieves L7LB-level load distribution without the associated overheads.

CCS Concepts

• **Networks** → **Transport protocols**; • **Software and its engineering** → *Distributed systems organizing principles.*

ACM Reference Format:
Shuo Li, Steven W.D. Chien, Tianyi Gao, and Michio Honda. 2025. Remote TCP Connection Offload with XO. In *9th Asia-Pacific Workshop on Networking (APNET 2025), August 07–08, 2025, Shang Hai, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3735358.3735377

1 Introduction

TCP has been widely used in web applications, disaggregated storage systems, and distributed data processing frameworks running in a private or public cloud. OS kernels have added a number of enhancements in their network stack, such as zero copy [8] and I/O batching [16], and NIC vendors have implemented many offloading capabilities, such as segmentation offload and TLS offload [22]. The research community has further advanced the space of TCP acceleration towards terabit Ethernet, including radical NIC redesign with FPGA [23] and the use of multiple CPU cores within a single stream [6].

Individual TCP servers are usually part of a *scale-out* cluster that expands the service capacity beyond that of a single server. Many systems distribute network requests to the servers that run replicated service instances to enhance the throughput, whereas some systems expand the storage capacity by partitioning the whole dataset into multiple storage servers (i.e., sharding). Those strategies are often employed together (e.g., selected replication [17]). Scale-out services are usually transparent to the clients; they see a single service endpoint (e.g., remote IP address and port) and are thus unaware of the exact server that is serving their requests.

To distribute the network requests to multiple servers, operators employ load balancers (LBs) [2, 10, 11]. Layer 4 load balancers (L4LBs) are lightweight because they only need packet-level operations without reassembling the TCP bytestream. Further, traffic from the server to the client can bypass the LB device, often called Direct Server Return (DSR). However, since a connection needs to be permanently handled by the chosen server and it can last long, a server cluster that is load-balanced solely by L4LBs could cause load imbalance over time. Further, L4LBs cannot be used to route the requests to sharded servers, because they cannot see the request payload that contains the information to determine the server (e.g., object id) and the next request sent over the same connection cannot be routed to another server.

Layer 7 load balancers (L7LBs) [4, 12, 14, 20, 24] *proxy* the client TCP connections (and usually TLS sessions) to the backend servers. They run in the application layer and relay data between the client- and server-side communication sections. L7LBs can apply more complex server selection policy than L4LBs, because they reassemble the TCP bytestream and thus can read the application-level message. Further, since the client connections are terminated at the L7LB, they can select another server without the client to notice, for example, when the chosen server does not have a requested storage data or becomes overwhelmed. However, relaying the data between the client and server stresses the L7LB's CPU, memory and network bandwidth resources.

This paper proposes Remote TCP Connection Offload (XO), a set of concept and techniques that enables a host to offload a TCP connection and application-level processing to another host to utilize the resource of cluster servers in a fine-grained, dynamic manner. XO achieves the best aspects of L4LB and L7LB. Like L4LB, ingress traffic goes to the offload target machine via the LB device but with a lightweight packet-level processing and egress traffic can bypass the load balancer or the host that has accepted the

*Joint first authors.

APNET 2025, Shang Hai, China

ACM ISBN 979-8-4007-1401-6/25/08
https://doi.org/10.1145/3735358.3735377

Method	System	Server select	HW	Overheads
Legacy proxy	Nginx, RGW, HAProxy	Per-request	–	App-level data relay
Connection splicing	Linux	Per-connection	–	In-kernel data copy
	AccelTCP [19]		SmartNIC	In-NIC processing
Connection migration	Prism [15]	Per-request	Programmable switch	Connection serialization, transfer and restoration
	Capybara [7]			
	XO		–	Above and ingress packet redirection

Table 1: Summary of L7LB Architectures and Features.

connection. Like L7LB, XO enables request-granularity, dynamic server selection transparently to the clients.

XO is applicable to various scale-out systems, because it supports both replicated and sharded backend architectures. We validate this through integration in real applications: `nginx` as an replicated backend example, and Ceph, a widely-used object storage system that scales the storage capacity, as a sharded backend example. XO supports various deployment scenarios, preserving those of L7LBs. Cluster servers do not have to connect to the same switch, because XO does not require switch support, which existing L7LB enhancements based on TCP connection migration, including Prism [15] and Capybara [7], require. XO thus can be used by cloud instances that could be instantiated in an unspecific physical machine or rack in the datacenter.

2 Motivation

Since the performance issues of L7LBs are known, there have been a series of enhancements that we summarize in Table 1.

Connection splicing reduces the data movement overheads by relaying data between two connections inside the kernel [5, 18]. This avoids two data copies, one from the kernel to the application and the other from the application to the kernel. AccelTCP [19] takes this step further, also removing DMA overheads by relaying data inside the SmartNIC.

However, those approaches, including AccelTCP, cannot perform payload-touching operations that legacy proxies can. Therefore, they are useful only when L7LB's job is merely to decide the (permanent) server (e.g., the least loaded one) at the connection setup time and simply copy the application-layer data between the client- and backend-side connections. Many L7LB systems require data-touching operations. For example, L7LB in object storage systems, including RGW in Ceph, require pprotocol translation between HTTP and `msgr2` (§ 5), which needs to read application-level headers that constantly appear in the TCP bytestream.

TCP connection migration takes a more radical step, moving a TCP endpoint itself to another server with the aid of a programmable switch. It has been achieved by Prism [15] and Capybara [7]. This class of approaches enables the L7LB to *handoff* the TCP endpoint that has connected with the client to a server (e.g., the one that holds the requested object). Connection migration eliminates data relaying overheads of L7LBs, because the upstream switch redirects the ingress (client to server) packets to the server and egress packets are directly sent to the client while the switch rewrites their source IP address to that of L7LB. Unlike connection splicing, those approaches can select a server in a request granularity as with legacy L7LBs, because every request is read by the application, which might be an instance that has been migrated and restored with the connection, and the application can migrate the connection and its state again based on the next request.

TCP connection migration has deployment problems. The servers between which a connection endpoint moves must be connected to the same switch. This is a serious drawback, because cloud computing instances are often VMs and they could be instantiated in unpredictable server or rack [13]. Further, the switch needs to be programmable one and the tenants must be able to configure it. Further, the major programmable switch ASIC released in the market, Intel/Barefoot Tofino has been discontinued for production since early 2023. Finally, it is unclear whether Prism and Capybara are general, because they have not been applied to real applications.

Therefore, we explore a solution that can mitigate the legacy L7LB overheads while enabling request-granularity server selection and flexible instance allocation by designing XO.

3 XO Architecture

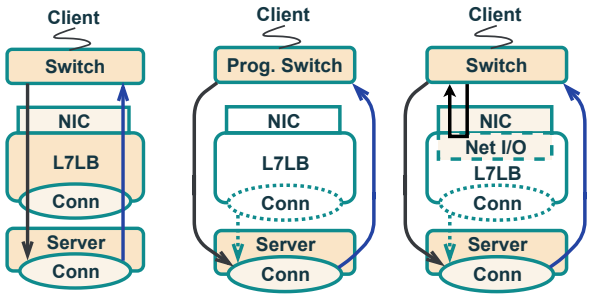
The XO architecture is based on the service model that repeats:
(1) Receiving a request from the client over a TCP connection;
(2) Executing a task (e.g., reading a data from the storage and preparing the response data) and then;
(3) Sending back a response to the client over the TCP connection.

The server, which we call ***host***, offloads the last two steps to another server, which we call ***target***; once the offload begins, the first step is also offloaded. After one or more offloaded processing, XO completes by the host reclaiming the service execution state from the target. The host is equivalent to L7LB in terms of establishing a TCP connection with the client and selecting a backend server.

XO is transparent to the clients. They do not notice that the offload is ongoing (except for through the side channels such as performance characteristics), because the client-side TCP (and TLS) endpoint is not disrupted. This significantly contributes to XO being general, because client and server applications are often implemented by different parties. For example, various S3 storage client implementations access the S3 cloud storage.

When and how offload begins? The application on the host makes an offload decision when it reads a request in the TCP connection (and decrypts the request if TLS is used). Since this enables request-granularity server selection, XO can use the same server selection policy as legacy proxies. For example, the application would choose a server in a round-robin fashion or based on the current load among the replicated servers. Note that communication between the LB control plane and servers to signal the current load is trivial [3, 17]. In the sharded server cluster, it would choose the server whose local storage has the data requested by the client in the sharded server cluster.

When and how offload completes? offload completes with the service execution returns to the host from the target. Completion decision can be made by the host, for example, when it finds a better offload target based on monitoring, or by the target, for example, finding itself overwhelmed or identifying that it cannot serve the requested storage data. After the completion of offload, the host



(a) Proxy (b) Prism (c) XO

Figure 1: Data paths of regular L7LB, Prism, and XO. Traffic flows are represented by directional arrows: black arrows indicate client requests, while blue arrows show server responses.

continues to serve the requests without offload (i.e., like a legacy L7LB) or makes another offload decision.

4 XO Design

XO has two key components: TCP and application state transfer and HW/SW hybrid traffic steering.

4.1 Connection State Transfer

Offloading the server task execution needs transferring TCP and other application-level (e.g., TLS) connection state from the host to the target (outbound) when the offload begins and the other way around (inbound) when the offload completes. Figure 2 illustrates the outbound (left) and inbound (right) state transfer sequence, which we describe next.

Connection state transfer needs endpoint operations and flow steering operations. The former serializes and restores the endpoint state; the latter redirects the ingress packets, which would otherwise go to the host, to the target. The source address of the packets sent by the target is modified that of the host, enabling DSR widely used by L4LBs [10][1].

Correctly and efficiently transferring connection state requires a strict order of endpoint operations and flow steering operation. XO's connection state transfer protocol is based on two objectives. First, to ensure correctness, it prevents undesirable packet arrivals at the transport layer during state transfer, since packets reaching a closed TCP socket trigger client connection resets [9, 15]. Second, to enable efficient transfer, it minimizes RPCs between the host and target.

4.1.1 Outbound Transfer. To prevent the ingress packets from reaching the TCP state being serialized, the host installs a packet filter rule (we discuss the method in § 4.2) that drops those packets (① in Figure 2). Note that this rule just drops *unimportant* packets, such as keep-alive or spurious retransmission. It does not block the connection progress, because the outbound state transfer process begins upon the offloading decision made by the application that has already received the request (§ 3); the next expected connection data is the response to the client.

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

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 target via a BEGIN_OFFLOAD RPC. The target then restores the connection, handling potential port conflicts by remapping ports if needed when connections transferred by different hosts share the same port (③). The target then installs a packet filter rule that rewrite the source IP address of egress packets to match the host's address (④).

Upon state restoration, the target sends a TARGET_READY RPC that requests the host install a packet filter rule that redirects the ingress packets in the connection to the target (⑤) and remove the rule that blocks the ingress traffic (⑥).

After that, the host sends a HOST_READY RPC to the target, which then begins the offloaded task.

4.1.2 Inbound Transfer. This operation is used when offload completes. The process is similar to the outbound transfer, but differs in flow steering operations. The target blocks ingress packets (⑦), serializes current connection state (⑧), and transmits state back to the host with a RPC TARGET_DONE. Upon receiving the state, the host restores the connection (⑨), removes redirection rules (⑩), and sends a final RPC END_OFFLOAD asking the target to remove source IP rewriting (⑪) and unblock ingress traffic (⑫). Upon completion, the system returns to its original configuration prior to the offload.

4.2 Flow Steering

Among various flow steering operations (redirection, blocking, and source address modification), flow redirection at the host (⑤ in Figure 2) is the most 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 offload target changes frequently.
- **Efficient traffic redirection**: The host must redirect the traffic, including ingress ACKs and subsequent requests, at a high rate to maintain the connection performance. Further, it must be done with low CPU usage to allocate as many CPU cycles as possible to the applications on the host.

We analyze the available options (§ 4.2.1), and based on our observation, 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-granularity traffic redirection:

One option is `tc-flower` [1], which is a legacy yet efficient approach that matches ingress/egress packets against flow rules based on keys such as IP addresses and ports. Its match and action procedures can be offloaded to NIC ASIC, as demonstrated by Open vSwitch, which uses `tc-flower` to offload the datapath operations on the cached flows[2]. This hardware offloading capability is available in various commodity (not particularly "smart") NICs, including Intel E810 (2020), NVIDIA/Mellanox ConnectX-5 (2016) and their successors, and Netronome Agilio CX (2018).

[2]This is also called ASAP in NVIDIA/Mellanox NICs.

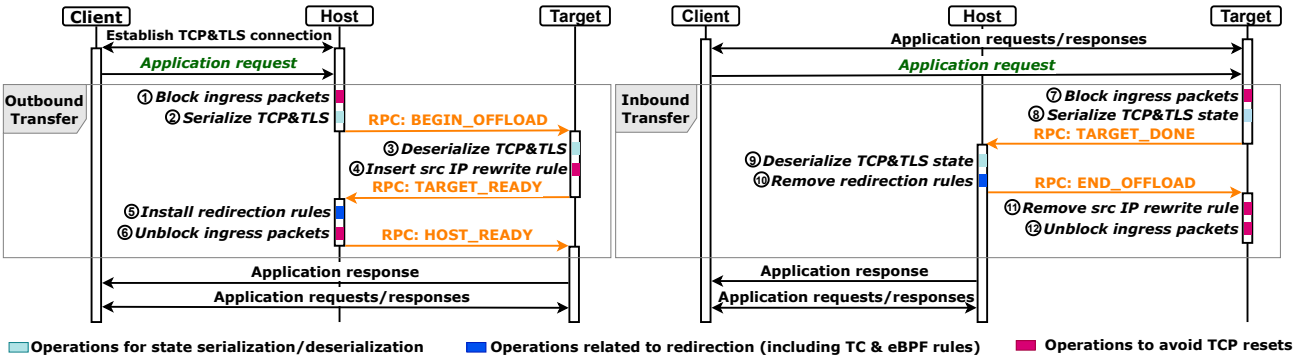


Figure 2: XO connection state transfer protocol (§ 4.1).

The other option is eBPF programs attached to `tc` classifier or `XDP`. They run below the TCP implementation in the network 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 packet processing decision.

We measured their flow installation/withdrawal time and packet forwarding performance. Our results in Table 2 highlight their key characteristics.

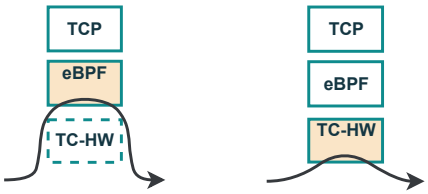
	Command (µs)		Rate (Mpps)		Latency (µs)	
	Insert	Remove	64B	1500B	64B	1500B
eBPF-tc	4.01	3.77	0.79	0.78	21.06	22.42
eBPF-XDP	38.31	7.41	6.65	2.07	16.52	18.45
tc (CX5)	476	404	33.01	2.07	8.26	9.89
tc (CX7)	2143	1134	33.08	2.07	8.41	9.97
tc (Agilio)	68	65	22.12	2.07	19.77	20.58

Table 2: eBPF and `tc-flower` flow command execution time and packet forwarding rate and latency. The bottom three rows indicate `tc-flower` hardware offload in NVIDIA ConnectX-5, ConnectX-7 and Netronome Agilio NICs, respectively.

eBPF achieves rapid flow installation. That in `tc` (eBPF-tc) is faster than in XDP (eBPF-XDP) due to its higher position in the stack. However, XDP provides superior packet forwarding rates by operating at the device driver level. `tc-flower` (with hardware offload) exhibits longer flow installation times due to kernel locks and device configuration, with significant variations across NICs: ConnectX-5/7 (CX5/7) requires 297–597 µs, while Agilio needs only 65–69 µs. However, all the NICs achieve high packet forwarding rates through hardware offload. Hardware-based forwarding is crucial also for CPU efficiency. For comparison, an eBPF program with XDP consumes 77 % of a CPU core when forwarding 1500 byte packets between 25 Gb/s links.

4.2.2 Hybrid Packet Redirection Design. Based on our observations, we design a hardware-software 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 host runs an eBPF program that processes ingress packets based on the flow table stored in the map. When



(a) Before HW rule activation. **(b) After HW rule activation.**

Figure 3: HW-SW hybrid packet redirection. Dotted frames indicate pending hardware rules, red circular arrows show redirection points, and black arrow lines represent data flow.

traffic redirection is needed (⑤ in Figure 2), the application on the host inserts the flow rule in the eBPF map in a blocking manner (synchronously) while initiating `tc-flower` hardware offload in a non-blocking manner (asynchronously). As a result, packets are initially redirected by the eBPF rule while the hardware rule is still being configured (Figure 3a). Once hardware-based rule has been activated, packet redirection is done in the hardware (Figure 3b), enabling more efficient processing.

While this hybrid mechanism enables efficient outbound state transfer, flow redirection withdrawal (⑩) must be synchronous for both software and hardware rules. Consider when a connection returns to the host after request completion (⑨). If hardware-based redirection at the host remains active due to asynchronous withdrawal, the remote would receive packets after removing its rules (⑫), triggering unwanted connection reset packets. Similarly, when the connection is offloaded to a new target device, lingering hardware rules on the host would prevent new eBPF rules from taking effect, causing packets to flow to the previous target. Also, if the host needs to send data (e.g., cached content) directly to the client, active hardware redirection in the host would prevent receiving ACKs, stalling the TCP connection.

4.2.3 Queue-based Rule Management. Synchronous flow rule withdrawal is challenging when the rules are installed asynchronously in parallel. As shown in Figure 4 left (Without user queue), when application threads issue syscalls to request the kernel to insert or remove flow redirection rules, each syscall execution acquires

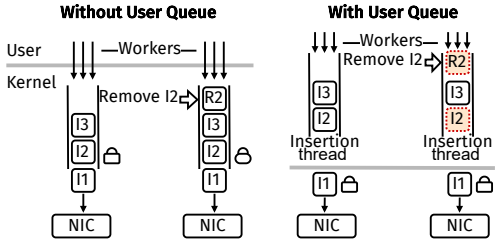


Figure 4: In-kernel command backlog (left) and user-space queue (right) discussed in § 4.2.3. In this example three worker threads issue rule insertions (I) and removals (R). With the user-space queue, offsetting removal (R2) cancels the unexecuted insertion (R1.)

kernel locks used by the `tc` subsystem or NIC driver to track hardware rules. This creates a backlog when multiple asynchronous rule insertion commands are issued, and those commands are serialized over the locks, blocking each syscall execution for a duration of the hardware rule operation time multiplied by the number of preceding commands in the backlog.

While delayed rule insertion is tolerable due to our hybrid packet redirection design, delayed rule withdrawal is problematic. Once a command (e.g., I2 in Figure 4 left) enters the kernel backlog, it cannot be *cancelled* even when an offsetting deletion command (R2) arrives. This is particularly problematic for XO where flow rules sometimes have short lifetimes (e.g., single request duration). A deletion command must wait for its corresponding insertion to complete before executing.

To address this issue, we implement a *user-space queue* (Figure 4 right) to allow the offsetting 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 the 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 Filtering and Egress Source 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` due to its quick deployment. This is sufficient because after connection handoff begins (① in Figure 2), incoming packets consist only of spurious retransmissions or keep-alive packets (§ 4.1.1). Similarly, source address modification (④) is implemented using `eBPF` or software `tc`, because this operation incurs negligible overheads.

5 Evaluation

XO runs on the Linux network stack, which is crucial for practicality [6, 23]. XO does not require kernel modification, instead leveraging existing stack features including `tc` and `eBPF` subsystems.

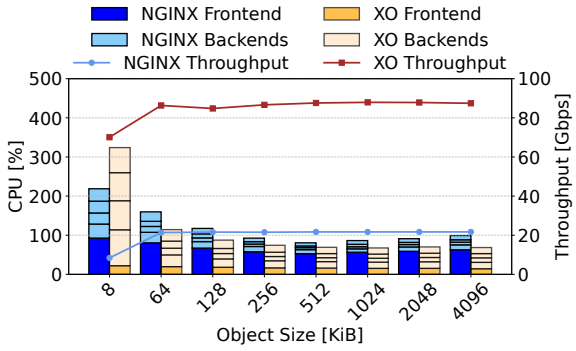


Figure 5: `nginx` aggregate throughput and CPU utilization of each machine. Frontend refers to the server with L7LB role.

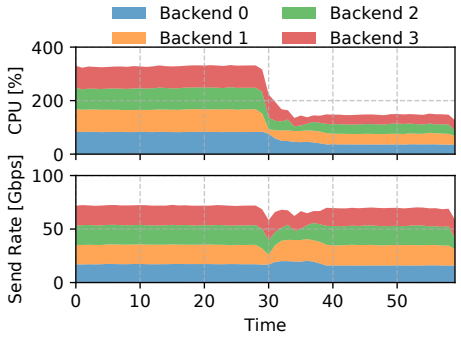


Figure 6: `nginx` load redistribution with XO. At time 30s, connections at backend 0 leave, then some connections at other backends are moved to backend 0 by XO.

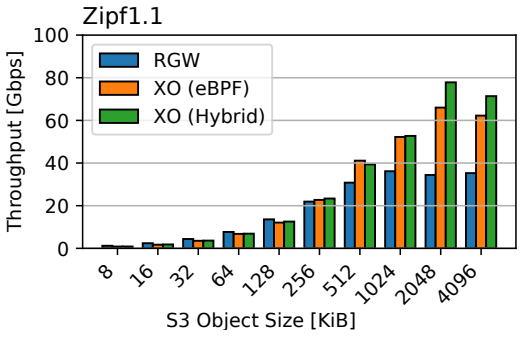


Figure 7: Ceph throughput with the default storage gateway (RGW), eBPF-only flow redirection (eBPF) and hybrid one (Hybrid). Request distribution is based on Zipfian 1.1.

Applications. To demonstrate the applicability of XO to different types of backend servers, we integrated XO in two systems. The first is `nginx`, a web server that also acts as L7LB (legacy proxy) for replicated servers. The other is Ceph, widely-used object storage system. It partitions the data into multiple servers and its Rados Gateway (RGW) subsystem acts as L7LB that computes the location

(server) of the requested object and issues storage I/O requests to that server over the `msgr2` protocol.

Experiment setup. Our testbed consists of six machines connected to the same switch with following role assignment: one client with a 100 Gb/s link, one frontend (host in XO) and four backends (target serverss) with 25 Gb/s link. The client machine is equipped with two AMD EPYC 9334 CPUs, 256 GB of RAM. The other machines are equipped with two Intel Xeon E5-2630v4 CPUs and NVIDIA CX5.

Figure 5 shows throughput and CPU usage of the `nginx` servers. XO efficiently utilizes the aggregate bandwidth of all the backend servers, as shown in the resulting throughput, while utilizing their CPU resources efficiently or without overwhelming the frontend or L7LB.

Next, we implement a simple offload policy that selects the server based on CPU usage, where each cluster server periodically reports the current load to the frontend. Figure 6 shows that XO achieves flexible load balancing that the live connections are dynamically shifted to more idle servers.

Finally, in Figure 7, we show that XO improves throughput in the Ceph cluster, a shareded backend deployment with efficient data path. This experiment also shows that our hybrid approach (XO (Hybrid)) exhibits the advantage of hardware-based packet redirection for large objects in comparison to software-only approach (XO (`eBPF`)).

5.1 Connection State Transfer Latency

Operation	Latency [µs]
(H) Block flow	5
(H) Serialize TLS	3
(H) Serialize TCP	15
(H) State transferring to the target	404
(H) Sending RPC: BEGIN_OFFLOAD	
(T) Block flow	4
(T) Restore TCP	42
(T) Restore TLS	20
(T) Install source IP rewrite rule	95
(T) Unblock flow	4
(H) Received RPC: TARGET_READY	
(H) Install redirection rule and unblock	4
(H) Send RPC: HOST_READY	110
Total	**541**

Table 3: Outbound state transfer latency breakdown. H and T indicate operations at the host or target remote sides respectively. All indented operations are included in the state transfer duration.

Table 3 shows latency breakdown of outbound state transfer latency of XO (§ 4.1.1). Since RPC latency is measured at the host, the last HOST_READY RPC time is a conservative estimate, because the target can start data transfer as soon as receiving that RPC request. Inbound state transfer takes similar time, plus, hardware rule removal time that must occur in a synchronous manner (§ 4.2.2), which is shown in the `Remove` column in Table 2.

The vast majority of latency comes from RPCs. Fortunately, there exist many low-latency RPC techniques even without relying on kernel-bypass networking. For example, Homa [21] reports 15 µs of RPC RTT with low tail latency.

TCP connection serialization or restoration involves 13 syscalls, each requires socket locking. If we modify the stack, it is trivial to merge those syscalls and locks.

6 Discussion

The interplay between throughput, latency, and CPU usage demonstrates XO's effectiveness across different scenarios. Performance differences between XO-CX5 and XO-Agilio highlight how hardware capabilities influence system behavior. As NIC hardware continues to evolve with faster rule processing and improved offloading capabilities, XO's benefits should become even more pronounced, promising greater performance advantages in throughput, latency, and CPU efficiency.

On the other hand, XO calls for the need for faster, more parallel NIC reconfiguration. Further, we also observed that, even if the hardware returns completion of the rule insertion, the NIC rule is still inactive for a short time.

As more and more offloaded network processing is imposed on the NICs, we believe it's time to think about better hardware-software interfaces, at least taking into account the reconfiguration time, internal parallelism, consistency model, and even transactional interfaces to atomically execute multiple commands. Such NIC abstractions would greatly simplify building efficient XO-like systems, for example, enabling multiple flow rule insertions in parallel.

Some modern NICs, such as Pensando Elba, are equipped with programmable P4 ASIC. 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.

7 Conclusion

This paper presented XO, which combines efficiency of L4LBs and flexibility of L7LBs. We addressed challenges in enabling a new concept of TCP offload to improve the resource utilization and service capacity of the scale-out systems, which makes up our contributions. First, we designed efficient TCP state transfer protocol that does not require switch support and thus enables location-independence property of the server instances. Second, we enabled a hardware-software hybrid method that steers network traffic at the end system based on the features available in commodity NICs. Finally, unlike the state-of-the-art L7LB architecture, such as Prism and Capybara, we integrated XO with real-world applications that use replicated or sharded backends, `nginx` and Ceph, respectively, demonstrating XO's generality and applicability.

Acknowledgments

We are grateful to the anonymous APNet reviewers for valuable comments. This work was in part supported by gift from Meta and EPSRC grant EP/V053418/1.

References

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

[2] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the edge: Transport affinity without network state. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 111–124.

[3] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. 2020. A {High-Speed}{Load-Balancer} Design with Guaranteed {Per-Connection-Consistency}. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 667–683.

[4] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. 2010. Finding a Needle in Haystack: Facebook's Photo Storage.. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vol. 10. 1–8. http://static.usenix.org/legacy/events/osdi10/tech/full_papers/Beaver.pdf

[5] Daniel Borkmann and John Fastabend. 2018. Combining kTLS and BPF for Introspection and Policy Enforcement. In *Linux Plumbers Conference*, Vol. 18.

[6] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μ s tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 767–779.

[7] Inho Choi, Nimish Wadekar, Raj Joshi, Joshua Fried, Dan RK Ports, Irene Zhang, and Jialin Li. 2023. Capybara: μSecond-Scale Live TCP Migration. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. 30–36.

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

[9] Wesley Eddy. 2022. Transmission Control Protocol (TCP). RFC 9293. (Aug. 2022). https://doi.org/10.17487/RFC9293

[10] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A fast and reliable software network load balancer.. In *Nsdi*, Vol. 16. 523–535.

[11] Rohan Gandhi, Y. Charlie Hu, Cheng-kok Koh, Hongqiang Liu, and Ming Zhang. 2015. Rubik: Unlocking the Power of Locality and End-point Flexibility in Cloud Scale Load Balancing. In *2015 USENIX Annual Technical Conference (USENIX ATC 15) (ATC'15)*. USENIX Association, Santa Clara, CA, USA, 473–485. https://www.usenix.org/conference/atc15/technical-sessions/presentation/gandhi

[12] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. 2016. Yoda: A Highly Available Layer-7 Load Balancer. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*. ACM, London, UK, Article 21, 16 pages. https://doi.org/10.1145/2901318.2901352

[13] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. 51–62.

[14] HAProxy. [n. d.]. HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/. ([n. d.]). http://www.haproxy.org/

[15] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. 2021. Prism: Proxies without the Pain.. In *NSDI*. 535–549.

[16] Kanchan Joshi, Anuj Gupta, Javier Gonzalez, Ankit Kumar, Krishna Kanth Reddy, Arun George, Simon Lund, and Jens Axboe. 2024. I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 107–121. https://www.usenix.org/conference/fast24/presentation/joshi

[17] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating skewed workloads in distributed storage with {In-Network} coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 387–406. https://www.usenix.org/system/files/osdi20-li_jialin.pdf.

[18] David A Maltz and Pravin Bhagwat. 1999. TCP Splice for application layer proxy performance. *Journal of High Speed Networks* 8, 3 (1999), 225–240.

[19] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and Kyoung-Soo Park. 2020. {AccelTCP}: Accelerating network applications with stateful {TCP} offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 77–92.

[20] Nginx. [n. d.]. NGINX | High Performance Load Balancer, Web Server, Reverse Proxy. https://www.nginx.com/. ([n. d.]). https://www.nginx.com/

[21] John Ousterhout. 2021. A Linux Kernel Implementation of the Homa Transport Protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association. https://www.usenix.org/conference/atc21/presentation/ousterhout

[22] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. 2021. Autonomous NIC offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 18–35.

[23] 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. 2024. High-throughput and Flexible Host Networking for Accelerated Computing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 405–423. https://www.usenix.org/conference/osdi24/presentation/skiadopoulos

[24] Squid. [n. d.]. Squid: Optimising Web Delivery. http://www.squid-cache.org/. ([n. d.]). http://www.squid-cache.org/