X-IO: A High-performance Unified I/O Interface using Lock-free Shared Memory Processing

Shixiong Qi*, Han-Sing Tsai†, Yu-Sheng Liu†, K. K. Ramakrishnan*, Jyh-Cheng Chen†
*University of California, Riverside, †National Yang Ming Chiao Tung University

***Abstract*—Cloud-native microservice applications use different communication paradigms to network microservices, including both synchronous and asynchronous I/O for exchanging data. Existing solutions depend on kernel-based networking, incurring significant overheads. The interdependence between microservices for these applications involves considerable communication, including contention between multiple concurrent flows or user sessions. In this paper, we design X-IO, a high-performance unified I/O interface that is built on top of shared memory processing with lock-free producer/consumer rings, eliminating kernel networking overheads and contention. X-IO offers a feature-rich interface. X-IO's zero-copy interface supports building provides truly zero-copy data transfers between microservices, achieving high performance. X-IO also provides a POSIX-like socket interface using HTTP/REST API to achieve seamless porting of microservices to X-IO, without any change to the application code. X-IO supports concurrent connections for microservices that require distinct user sessions operating in parallel. Our preliminary experimental results show that X-IO's zero-copy interfaces achieve 2.8x-4.1x performance improvement compared to kernel-based interfaces. Its socket interfaces outperform kernel TCP sockets and achieve performance close to UNIX-domain sockets. The HTTP/REST APIs in X-IO perform 1.4x-2.3x better than kernel-based alternatives with concurrent connections.**

***Index Terms*—DPDK, communication in service function chains, shared memory, unified I/O interface**

I. Introduction

Cloud-native applications are shifting from monolithic to microservices-based architectures. A typical example is the evolution of the cellular core network (going from purpose-built hardware appliances, to monolithic software-based implementations, to a disaggregated microservices-based 5G core). A microservice paradigm provides flexibility in development and deployment, modularity, and scalability at the individual function level. However, the loose coupling between microservices imposes *communication* overheads for the microservice-based components to together complete a task. For example, in the 5G core (5GC) control plane, the various components for access and mobility management, authentication, and data management, work together to complete the task of registering a User Equipment's (UE) session.

Typical communication paradigms adopted by microservices can be broadly classified into (1) *synchronous* data exchange (I/O in short) between a pair of microservices, and (2) *asynchronous* I/O. Synchronous I/O is *bidirectional*, where the source microservice (caller) sends a message to the destination microservice (callee) and waits until a response is returned. A typical example of synchronous I/O in the dataplane between microservices is a Remote Procedure Call (RPC). On the other hand, asynchronous I/O is *unidirectional* where the caller is not blocked after sending the message to the callee. Asynchronous I/O has been widely used to organize communication among a set of interdependent microservices as a Directed Acyclic Graph (DAG), with only directed asynchronous communication pattern considered between microservices. Asynchronous I/O is not able to support a synchronous 'request-response' communication paradigm. The I/O operation (initiated by the caller) in 'request-response' type communication is always blocked for the synchronization to be performed with the callee, *i.e.,* the callee returns a response back to the caller, indicating to the caller that the I/O operation is complete and it can proceed. This leads to a mismatch in how asynchronous I/O and synchronous I/O are implemented.

Despite distinct operating modes, existing solutions adopted by cloud-native microservices for synchronous I/O (*e.g.,* gRPC, 3GPP SBI) and asynchronous I/O, typically interact through the kernel-based networking stack. However, communicating through a full-fledged kernel-based networking stack proves to be heavyweight and imposes significant overheads on the dataplane, including data copy, protocol processing, serialization/de-serialization, heavyweight kernel/userspace boundary crossing, and interrupts. This adds additional latency and limits the performance of backend microservices [1], [2].

Moreover, within a group of coupled microservices, an individual microservice may consume messages from multiple upstream microservices (producers) or produce messages to multiple downstream microservices (consumers). This results in a multiple-producer, multiple-consumer communication pattern, leading to contention when there is concurrent processing. Using a lock can ensure correct operation when there is contention, but in turn, exacerbates the communication-related performance degradation for microservice chains. This is especially concerning for use cases that have stringent latency requirements, such as 5GC, Network Function Virtualization (NFV), and Middleboxes, which demand sub-millisecond or even microsecond scale networking between microservices.

Shared memory can be a very effective means for communicating information between microservices within a node, as it eliminates the overhead of a full-fledged kernel-based networking stack. Further, having a lock-free producer/consumer ring framework can help reduce contention between microservices accessing this shared memory. With the help of such a lock-free, shared memory communication, the performance of both synchronous and asynchronous I/O can be considerably improved, benefiting applications that use microservices, with low-latency, high-performance networking.

However, harmonizing cloud-native microservices that need either synchronous, asynchronous I/O or both, with a single high-performance communication framework is not straightforward, due to the distinct needs for the communication paradigms (synchronous vs. asynchronous). This motivates us to create a unified I/O interface that can flexibly work either synchronously **OR** asynchronously with shared memory processing in a lock-free manner - one that would allow us to avoid the dataplane overheads that exist in current kernel-based networking. In addition, a careful design of the Application Programming Interface (API) that is exposed to a microservice is required to avoid extensive rewriting of the application's implementation. Furthermore, cloud-native microservices, such as the 5GC, often need to handle messages of different user sessions or flows in parallel. This requires maintaining the underlying transport state (*e.g.,* connection or user session state) and distinguishing clients that a particular microservice communicates with.

In addition to the mismatch between synchronous and asynchronous I/O, another challenge comes from programming language incompatibility. A considerable number of cloud-native microservices, which place more emphasis on functionality and development velocity, often choose to use a high-level programming language (*e.g.,* Golang). On the other hand, approaches designed for high-performance networking use cases (*e.g.,* DPDK), utilize programming languages with low-level primitives, such as C.

This paper proposes X-IO, a high-performance **unified** I/O interface designed for cloud-native microservice applications. We implement X-IO using shared memory processing with lock-free producer/consumer rings between microservices on the *same* node as the "X-IO stack" (§III-B), which is a far simpler alternative to the kernel networking stack. This achieves zero-copy packet delivery, bringing substantial improvement to data plane performance. X-IO stack exposes a set of primitives for the zero-copy interface (§III-C) to be leveraged by upper-layer applications to construct a zero-copy data plane between microservices. It also supports a set of POSIX-like primitives for the socket interface (*e.g.,* Read(), Write()), built with the shared memory processing in X-IO stack (§III-D). An abstraction layer, X-IO library, is designed to harmonize with the underlying X-IO stack, and provides the POSIX-like I/O primitives to upper-layer applications. This eases the porting of existing microservices that depend on a POSIX socket to benefit from X-IO's lock-free, shared memory processing. Both synchronous and asynchronous communication paradigms can be constructed by exploiting X-IO's zero-copy or socket interfaces (§III-E). Specifically, we demonstrate X-IO's capability in supporting synchronous I/O by using the 3GPP SBI in 5GC as a case study. We use X-IO to construct the equivalent HTTP/REST APIs (*e.g.,* GET(), POST()) to replace the kernel-based HTTP/REST APIs for existing 5GC functions, while keeping the function implementation *unchanged*. X-IO is designed to support concurrent connections by by having each microservice maintain a local connection table in its X-IO stack to track connection state. This is helpful when the microservice needs to support multiple user sessions, each with its own distinct connection. Requests are de-multiplexed to the correct connection endpoint after looking up the local connection table in the X-IO stack (§III-D). X-IO offers cross-language support by leveraging the CGo interface [3] to mitigate the programming language incompatibility between the lower-layer X-IO stack (implemented in C) and upper-layer socket interfaces and HTTP/REST APIs (implemented in Golang), thus reducing the porting effort for microservices developed with Golang.

X-IO is available at https://github.com/nycu-ucr/xio.git

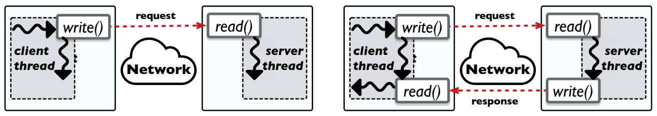

Fig. 1: (Left) Asynchronous I/O; (Right) Synchronous I/O.

II. Related work & Background

I/O Primitives in Linux: Linux offers a variety of I/O primitives to support I/O tasks, such as the socket interfaces (*e.g.,* listen(), accept(), connect(), read(), and write()). *Asynchronous* I/O between a pair of client/server can be easily constructed by leveraging the basic I/O primitives, *e.g.,* read() and write(). As shown in Fig. 1 (left), the client sends a message via write(), which is received by the server via read(). Since *asynchronous* I/O is used, the client continues with other tasks without being blocked and does not wait for a response from the server. The *synchronous* I/O mode requires at least two pairs of read()/write() calls between the client and server, as depicted in Fig. 1 (right). After the client invokes the write() call to send the message to the server, the client will be blocked on its read() call until a response is returned by the server. The synchronous/asynchronous I/O developed on socket interfaces relies on kernel-based networking, which introduces a number of overheads (*e.g.,* data copies, context switches, etc) in the datapath, impacting performance.

Linux also provides other advanced I/O primitives, *e.g.,* aio [4], io_uring [5]. However, aio is only available for storage-related I/O operations and cannot be used for network I/O. io_uring on the other hand does support network-related I/O operations. It uses a lock-free producer/consumer ring design that reduces interrupts and context switches for I/O operations, unlike the read()/write() socket interface. However, io_uring still depends on kernel-based networking, incurring the overheads of data copies and protocol processing. X-IO improves I/O performance using the combination of shared memory communication and lock-free producer/consumer rings, thus delivering much better performance compared to kernel-based io_uring (evaluated in §IV-A).

Network I/O optimizations: Multiple proposals exist for optimizing network I/O [6]–[9]. However, these designs are targeted for specific use cases. *e.g.,* [6] improves the NFV dataplane by using shared memory processing and lock-free producer/consumer rings, but [6] does not support synchronous I/O, unlike X-IO. [7] uses DMA to avoid CPU-based copies involved in data transfers, but DMA introduces additional PCIe delays, resulting in lower performance compared to the shared memory communication used in X-IO. [8], [9] seek to design a zero-copy interface in supporting synchronous I/O. But they both lack optimization for specific use cases (*e.g.,* NFV) that requires high-performance asynchronous I/O. X-IO addresses these concerns by offering a unified I/O, while still using shared memory to achieve a high-performance dataplane.

Data Plane Development Kit (DPDK): DPDK [10] supports both inter-host kernel-bypass (between userspace and NIC) and intra-host kernel-bypass (between processes on the same host). It has been widely used in NFV to build high-performance data planes, *e.g.,* in OpenNetVM [6]. DPDK offers a Poll Mode Driver (PMD) to deliver packets between userspace and the NIC, bypassing the kernel for inter-host communication. DPDK also features a set of useful libraries (RTE RING lib [11], multi-process support [12], Mempool lib [13]) to help construct an intra-host kernel-bypass data plane based on shared memory. DPDK makes targeted optimizations on packet processing within its libraries, *e.g.,* the use of Linux HugePages, and efficient synchronization for memory access (lightweight locking). This feature-rich toolkit enables a customized, high-performance packet processing pipeline.

DPDK's RTE RING is a shared memory ring buffer that can be used as a low-latency Inter-Process Communication (IPC) channel between individual applications. Working in conjunction with DPDK's multi-process support [12] and Mempool lib, to construct a shared-memory-based dataplane, the DPDK RTE RING producer/consumer ring framework can enable zero-copy packet delivery between applications. It passes packet descriptors (containing pointers to data in shared memory) between applications, thus operating at memory speed. This guarantees high dataplane throughput and low latency, especially when a networked microservice is organized as a chain to follow a certain execution order. Due to these desirable characteristics, we choose DPDK as the basic building block of X-IO's communication stack (*i.e.,* X-IO stack). However, our ideas are generally applicable and can work with other alternatives, *e.g.,* the event-driven shared memory processing utilizing the extended Berkeley Packet Filter (eBPF) in the Linux kernel (as in SPRIGHT [1]).

III. Design of X-IO

We first provide an overview of X-IO and introduce the key building blocks for constructing a high-performance unified communication paradigm that avoids the typical overheads of synchronous/asynchronous communication using the kernel protocol stack. We then discuss each part in detail, including shared memory processing, lock-free producer/consumer rings, different I/O interfaces in X-IO, and connection management. Specifically, X-IO offers two alternative interfaces: a zero-copy interface, and separately, a POSIX-like socket interface. We also describe how to construct a synchronous/asynchronous communication channel between microservices using different interfaces offered by X-IO, and discuss a case study of using X-IO to support 3GPP SBI.

A. *Overview of X-IO*

Fig. 2 depicts the architecture of X-IO. A shared memory pool is created on each node to support shared memory communication between X-IO-based microservices. We introduce the X-IO manager, a per-node userspace component responsible for initializing the shared memory pool. We also use the X-IO manager to perform lock-free communication (*i.e.,* packet descriptor delivery) between multiple microservices.

We introduce an X-IO stack with each microservice to support shared memory communication with other microservices co-located on the *same* node. This avoids a number of dataplane overheads in kernel-based networking (*e.g.,* protocol processing, copies, serialization/deserialization). Within the X-IO stack, there is a packet handler that handles all incoming/outgoing requests. Requests related to connection handling (*e.g.,* establishment/teardown) are also processed by the packet handler to complete connection management tasks. Each X-IO stack maintains a local connection table to support concurrent connections. For a certain request, its connection is identified through the connection table lookup using IP 4-tuples (source IP and port number, destination IP and port number). The request is then forwarded to the correct connection endpoint (*i.e.,* a certain user thread) by the packet handler in the X-IO stack (details in §III-D).

Since X-IO exploits shared memory for communication between microservices, the request payload always resides in shared memory without being moved. Requests are exchanged in the form of packet descriptors, which contain critical metadata such as the request type, and the pointer to the payload in shared memory. To exchange packet descriptors between different X-IO stacks, the packet handler in the X-IO stack is assigned a pair of producer/consumer rings. The X-IO stack uses the producer/consumer rings to facilitate a lock-free packet descriptor exchange with the X-IO manager (§III-B). We use the X-IO manager to route the packet descriptor (based on IP 4-tuples) between the X-IO stacks belonging to different microservices. X-IO exposes a set of zero-copy interfaces (§III-C) from the X-IO stack to facilitate shared memory communication between microservices.

The higher-layer application code interacts with the X-IO stack through the X-IO library (X-IO lib in short). The X-IO lib also offers a separate socket interface and HTTP/REST APIs for higher-layer applications to use. Our current implementation of X-IO lib is based on Golang because of its popularity in developing cloud-native microservices. Since X-IO lib keeps the *same* semantics as Golang's socket and HTTP/REST APIs, porting applications to use X-IO is seamless.

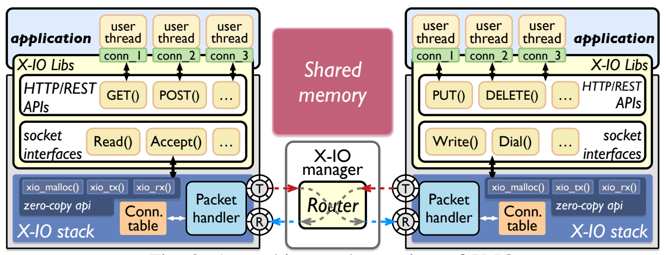

Fig. 2: An architectural overview of X-IO

We base the implementation of X-IO on top of OpenNetVM [6]. Specifically, we build the X-IO manager on top of OpenNetVM's NF manager for shared memory pool creation and descriptor routing. We implement the packet handler in X-IO stack using the NFLib in OpenNetVM, which offers basic primitives to operate producer/consumer rings to support the descriptor exchange. The packet handler is implemented using `C` language. The packet handler (in C) interacts with the higher-layer X-IO lib (in Golang) through the *CGo* interface [3] provided by Golang. The C-to-Go boundary crossing incurs negligible additional latency (70ns in our testbed) and thus has very little impact on the dataplane performance. The cross-language support for other high-level programming languages (*e.g.,* Python) is part of our ongoing work.

B. Lock-free shared memory communication

Two key elements are required to support shared memory communication: (1) a pool of shared memory buffers; (2) packet descriptor delivery mechanism. The shared memory pool provides a shareable backend to store the payload accessed by microservices. The packet descriptor delivery passes the pointer to the payload in the shared memory between different microservices instead of moving the payload (*i.e.,* memory-memory copy). Microservices use the packet descriptor to access the payload in shared memory, achieving zero-copy packet delivery.

Shared memory pool: X-IO uses the X-IO manager (Fig. 2) to manage the initialization of the shared memory pool that contains a certain number of shared memory buffers. The X-IO manager runs as the DPDK primary process, which gives it privileged permission to create the memory pool in the Linux file system. It utilizes DPDK's Mempool Library [13] to create pools of memory buffers (using *rte_mempool_create()* API). It is also necessary to specify a unique 'shared data file prefix' that the X-IO manager uses when creating memory buffers in the file system. The DPDK's Environment Abstraction Layer (EAL) takes the 'shared data file prefix' to distinguish between different shared memory pools. Since all shared memory buffers are pre-allocated during the initialization of the shared memory pool, this avoids adding unnecessary latency to the creation of shared memory buffers during message transfer. Note: our current implementation only supports fixed-size shared memory buffers.

An X-IO-based microservice attaches to the shared memory pool before accessing the shared memory buffers. To enable sharing of memory buffers created by X-IO manager, X-IO makes use of DPDK's multi-process support. DPDK's multi-process leverages the 'shared data file prefix' to attach DPDK secondary processes[1] (X-IO-based microservices in our context) to the shared memory pool. An X-IO-based microservice is permitted to access a shared memory pool, only if it is specified to have the same 'shared data file prefix' as the X-IO manager during its startup. This 'shared data file prefix' feature

[1]DPDK secondary processes are those that have read/write access to the shared memory buffers but do not have create/destroy permission.

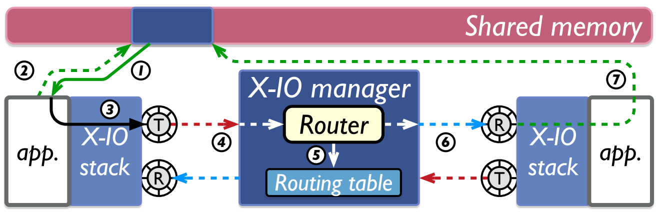

Fig. 3: Shared memory communication in X-IO using raw I/O primitives.

can be further extended for isolation purposes when X-IO-based microservices are divided into different security domains co-located on the same node [1]. Each security domain has a private shared memory pool.

Lock-free packet descriptor delivery: X-IO utilizes DPDK's RTE RING to pass packet descriptors between the X-IO stacks of different microservices. DPDK's RTE RING, in essence, is a shared memory circular buffer that is shared between a producer and consumer. Though it offers a high-speed IPC channel to deliver packet descriptors between producer/consumer at memory speed, locks would be needed when multiple producers write to the same buffer simultaneously. Acquiring and releasing locks would increase the latency and overheads for descriptor delivery in X-IO, and offset the benefit of this high-speed IPC model.

To achieve a lock-free producer/consumer ring design, we assign each X-IO stack with a pair of RTE RINGs, one for receive (RX) and the other for transmit (TX). Further, we only allow the X-IO stack to share its RTE RING pair with the X-IO manager. This restricts access to a single producer and consumer to write/read to the ring, thereby avoiding having to acquire a lock [6]. We leverage the X-IO manager to pass descriptors between different X-IO stacks.

Lock-free access to shared memory: The lock-free access to the shared memory buffer is controlled by the ownership of the descriptor. Only the microservice that currently owns the descriptor has write access to the shared memory buffer.

C. Raw I/O primitives in X-IO: zero-copy interfaces

X-IO exposes raw I/O primitives from the X-IO stack to construct a zero-copy communication channel between microservices. Zero-copy I/O primitives provided by X-IO include xio_malloc(), xio_tx(), and xio_rx(), which are built on top of DPDK's RTE RING and Mempool APIs. xio_malloc() uses DPDK's *rte_mempool_get()* API to retrieve a packet descriptor pointing to an empty memory buffer from the shared memory pool. xio_tx() enqueues the descriptor to the TX ring using the *rte_ring_enqueue()* API and xio_rx() dequeues the RX ring to retrieve the descriptor (using *rte_ring_dequeue()*). Note that our current implementation only supports polling mode of DPDK's RTE RING. The consumer of the ring (either X-IO stack or X-IO manager) uses up one CPU core to constantly poll the ring to receive packet descriptors. This leads to unnecessary CPU wastage at low traffic intensities. X-IO can adopt the NFVnice [14] scheduling principles to mitigate the CPU resource consumption, and multiplex a CPU core across multiple X-IO stacks.

Fig. 3 depicts the zero-copy, shared memory communication in X-IO using raw I/O primitives. The source microservice (caller) calls xio_malloc() to retrieve an empty memory buffer used to write the payload. The caller then invokes xio_tx() to enqueue the descriptor of that memory buffer to its TX ring. On the other side, the X-IO manager polls the TX ring of the caller to retrieve the descriptor. The X-IO manager parses the routing information in the descriptor, looks up the routing table, and enqueues the descriptor into the RX ring of the destination microservice (callee). The callee uses xio_rx() to receive the descriptor from its RX ring, which is then used to access the payload in shared memory. **No copying** of the message is involved with the transmission and we eliminate a number of other overheads compared to kernel-based networking. Although the exchange does involve descriptor copies, it is a very small amount of data with negligible transmission overhead compared to the actual payload.

Limitations: X-IO's zero-copy interface enables true zero-copy data transfer between microservices. But, its I/O primitives are not compatible with existing cloud-native microservices that rely on a POSIX-like socket interface and/or HTTP/REST APIs, thus requiring application modifications.

D. POSIX-like I/O primitives in X-IO: socket interfaces

To support seamless porting of applications that depend on the POSIX socket API to leverage X-IO's shared memory processing of the X-IO stack, we further introduce a separate socket interface in X-IO. X-IO's socket interfaces are exposed via an abstraction layer, namely X-IO lib, which coordinates with the X-IO stack and exposes a set of POSIX-like I/O primitives as listed in Table-I, including Read(), Write(), Listen(), Accept(), Dial(). X-IO lib currently supports equivalent Golang-style socket interfaces.

Design of X-IO's Read() interface: Read() is the basic read socket interface in X-IO. Similar to POSIX read(), X-IO's Read() supports both "blocking" and "non-blocking" modes. In blocking mode, the caller of Read() is blocked until it receives a request from the X-IO stack (via the packet handler) and moves the payload of the request into the *receive buffer* provided by the caller. In non-blocking mode, the caller of Read() is not blocked waiting for the request. This requires additional programming to ensure the caller can receive the request, *e.g.,* polling the Read() interface in a while loop until a valid request is returned.

Two building blocks are required in the X-IO lib to support a blocking Read(): (1) First is a blocking primitive used to hold the Read() call unless the condition variable of the blocking primitive is changed by the X-IO stack. The condition variable determines whether the blocking primitive is in effect (based on it being set to TRUE). In our Golang-style socket interfaces, we use Golang's built-in "Condition Variable" (*Cond* in Golang's `sync` package []) as the implementation

TABLE I: A list of POSIX-like socket interfaces in X-IO.

APIs	Input parameters	Description
Listen()	server address	Server listens for connections via X-IO stack
Dial()	server address	Client dials server to establish connection
Accept()	null	Server accepts connection with client
Read()	receive buffer	Server/client receives message via X-IO
Write()	send buffer	Server/client sends message via X-IO

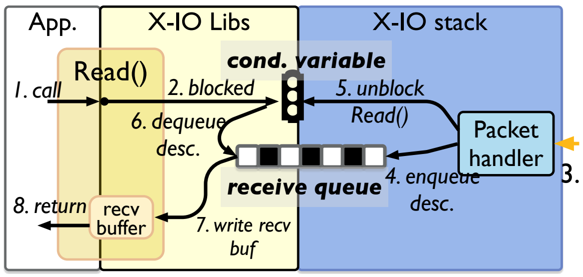

Fig. 4: Read() interface in X-IO

of the blocking primitive in Read(). *Cond* offers two methods, `Wait()` and `Signal()`, for us to implement the blocking primitive: `Wait()` blocks the caller, which owns the *Cond*, until `Signal()` is invoked. (2) Second, a receive queue to buffer the requests (in the form of packet descriptors) is required. The X-IO stack, as the producer, enqueues the received packet descriptor to the queue. On the other hand, the X-IO lib, as the consumer, dequeues the packet descriptor from the queue and use the descriptor to copy the payload in the shared memory to the receive buffer of the Read() call. Since there is only a single producer and a single consumer, the queue is lock-free.

Fig. 4 depicts the processing flow of a Read() call in X-IO. ① After the application calls the Read(), ② the Read() call is blocked on the condition variable (via `Cond`'s `Wait()` method). ③ The X-IO stack receives a request (in the form of a packet descriptor from a writer application on the other side). ④ The X-IO stack enqueues the descriptor into the receive queue. ⑤ The X-IO stack then calls the `Signal()` method to unblock the Read() call. ⑥ The Read() call is then unblocked and dequeues the descriptor from the receive queue. ⑦ The X-IO lib *copies* the payload from the shared memory to the caller's receive buffer, and ⑧ returns back to the caller. The caller is then unblocked and continues its execution.

Configuring Read() to operate in non-blocking mode can be simply done by bypassing the blocking primitive and directly returning to the caller. This avoids the context switching penalty compared to the blocking-mode Read(), which depends on system calls (invoked by `Signal()` method) with the kernel involved in order to unblock the Read() call, resulting in extra context switches and interrupts, which adds additional delays. However, non-blocking Read() requires the caller to busy-poll the Read() to inspect the arrival of the request, leading to high CPU usage. To strike a balance between the CPU efficiency of the blocking mode and the high performance of the non-blocking mode, we design a batch wake-up mechanism that unblocks a set of Read() calls in a batch to reduce the system call overheads. Thus, we avoid the use of busy polling and save CPU resources.

Design of X-IO's Write(): Write() is the basic write socket interface in X-IO. We only support blocking Write() in X-IO, which is to ensure all of the request payload is written into the shared memory buffer before the Write() returns. The Write() is blocked until the X-IO stack moves the payload from the *send buffer* to the shared memory and enqueues the packet descriptor into the TX ring. There are some cases where a

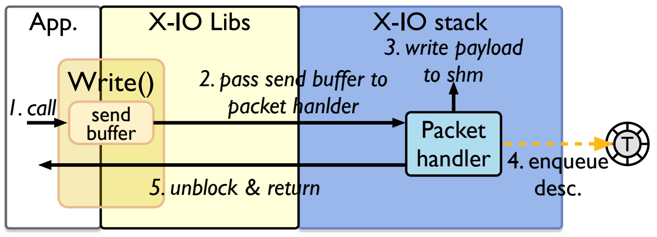

Fig. 5: Write() interface in X-IO

Write() may be blocked for a longer period, when there are no free shared memory buffers or the TX ring buffer is full. The Write() will be blocked until shared memory buffers or TX ring buffers become available. Fig. 5 shows the processing flow of a Write() call in X-IO. ① After the application calls the Write() with a send buffer input, ② the X-IO lib passes the pointer of the send buffer to the packet handler in X-IO stack. ③ The packet handler *copies* the payload from the send buffer to the shared memory. After that, ④ the packet handler enqueues the descriptor to the TX ring and ⑤ returns back to the Write() call. The Write() is then unblocked and control returns back to the caller.

Concurrent connection support: Both Read() and Write() interfaces in X-IO require an a priori established connection for data transmission, which is set up by using Listen(), Accept(), and Dial(). This retains alignment with the kernel-based HTTP/TCP communication model using socket interfaces. Further, it is desirable to leverage thread-based concurrency (*i.e.,* multi-threading) to allow the server application to service multiple clients in parallel or allow a single client to connect with multiple servers. Each server-client connection is assigned a dedicated thread to independently handle the corresponding data transmission.

The above design, however, increases the complexity of managing multiple client-server connections simultaneously. Especially for the receiver (server) side, since the request (in our case the packet descriptor) has to be transferred to the Read() call of the correct server thread. Managing simultaneous connections on the sender (client) side is straightforward, as the client application is aware of the server it wants to make a request to, and can find the appropriate connection before issuing a Write() call.

To support concurrent data exchanges between client(s) and server(s), we enable concurrent connections to be setup. We maintain a local connection table in the X-IO stack. Each connection table entry contains the connection-specific information, including the IP and port number of the server and client application (IP 4-tuple), connection state, and the receive queue of the Read() interface of that connection.

As shown in Fig. 6, ① after X-IO stack receives a packet descriptor, ② it performs a connection table lookup, using the

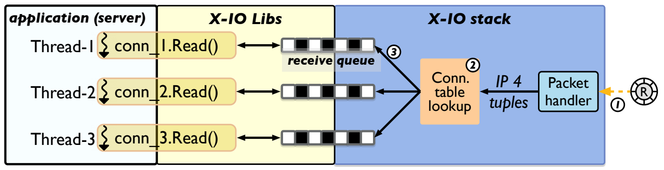

Fig. 6: Concurrent connections support in X-IO: concurrent Read()

IP 4-tuple contained in the packet descriptor as the key, to find the corresponding connection table entry. ③ X-IO stack then enqueues the packet descriptor into the receive queue of the corresponding Read() interface.

Connection Establishment & Teardown: We expose several connection-related socket interfaces in the X-IO lib for higher-layer applications to utilize: Listen(), Accept(), Dial(), and Close(). Listen() and Accept() are used by the server application for opening the socket, creating a receive queue for the new connection and waiting for connection establishment requests; Dial() is used by the client application to initiate connection establishment with the server. The connection establishment in X-IO is completed through a two-way handshake between the client and the server. The server-side X-IO stack allocates a new connection table entry, sets the connection state to active, and then responds with an acknowledgment message back to the client for confirmation of the connection establishment. The client's X-IO stack, after receiving the acknowledgment, allocates a new entry in the local connection table and sets the connection state to 'active' to complete the connection establishment. When Close() is called on a connection, the X-IO stack releases both its own and the peer's connection by deleting the corresponding connection table entry in the connection table. We reuse the packet handler in the X-IO stack to process messages related to connection establishment and teardown.

```go
/* X-IO-based server */
listener, _ := xio.Listen(server_address)
conn, _ := listener.Accept()

receive_buffer := make([]byte, RECV_MSG_SIZE)
n, err := conn.Read(receive_buffer)

conn.Close()

/* X-IO-based client */
conn, err := xio.Dial(server_address)

send_buffer := make([]byte, SEND_MSG_SIZE)
n, err := conn.Write(send_buffer)

conn.Close()
```

Listing 1: An example X-IO server/client pair using POSIX-like socket APIs

Porting existing applications to use X-IO's socket interfaces: Listing 1 shows an example of the server and client application using X-IO's socket interfaces. The server application calls Listen() to set up the server socket and then calls Accept() to wait for new connection establishment requests from the client. After the client calls Dial(), providing the correct input of the server's address and port, a connection is established between the client and the server. Once the connection establishment is complete, the client and server can use Read() and Write() for data transmission. Our implementation of X-IO's socket interfaces is *strictly* aligned with Golang's native socket interfaces, thus achieving seamless porting of Golang-based microservices to use X-IO's socket interfaces.

Limitation of X-IO's socket interfaces: Although X-IO's socket interfaces are built on top of the same shared memory communication framework as X-IO's zero-copy interfaces, an additional copy is introduced when moving the payload between shared memory and the application's send/receive buffer. While it would be ideal to eliminate this additional copy, the need to be aligned with POSIX-like socket interfaces to achieve transparent porting currently required this approach to move the payload between the send/receive buffer and the shared memory buffer. The design and implementation of a true zero-copy, POSIX-like socket interface supporting transparent portability is part of our ongoing work.

E. Synchronous & Asynchronous data exchange with X-IO

Asynchronous and synchronous data exchange between microservices can be built using either X-IO's socket interfaces or X-IO's zero-copy interfaces.

Asynchronous data exchange with zero-copy interfaces: For this data exchange, the source microservice (caller) uses xio_malloc() to retrieve a shared memory buffer and write the request payload. It then uses xio_tx() to send the request descriptor to the destination microservice (callee), which uses xio_rx() to receive the incoming descriptor.

Synchronous data exchange with zero-copy interfaces: For this setup, the source microservice (caller) and the destination microservice (callee) both need a pair of xio_tx()/xio_rx() to enable the 'request-response' communication paradigm. xio_ malloc() is called on demand when writing a request or response into the shared memory.

Asynchronous data exchange with socket interfaces: This setup requires a pair of Read()/Write() interfaces to establish a unidirectional channel between the source microservice (caller) and destination microservice (callee). The caller uses Write() to send a request to the callee (using Read() to receive). The caller proceeds with other tasks without waiting for a response from callee.

Synchronous data exchange with socket interfaces: This exchange requires two pairs of Read()/Write() interfaces to establish a bidirectional channel between the source microservice (caller) and destination microservice (callee). The caller uses Write() to send a request to the callee (using its Read() to receive). The caller waits on its Read() interface for a response from the callee (which uses its Write() to send the response).

Case study – using X-IO to support 3GPP SBI: The 5GC is implemented using virtualized 5GC functions as independent cloud-native microservices for flexibility and scalability. Because of this disaggregated design, 5GC functions, especially the control plane functions, interact over the Service Based Interface (SBI) recommended in the 3GPP specifications. A popular implementation of 5GC control plane is free5GC [15], which implements the 3GPP SBI utilizing the kernel-based networking stack. However, the kernel-based networking stack limits the performance of the cellular core control plane, especially in achieving low latency [16], [17]. Compared to the existing kernel-based SBI in free5GC, X-IO offers a much better and performant service for interfacing 5GC functions with the underlying lock-free shared memory processing.

In addition, free5GC uses Golang as the programming language for its SBI implementation, returned in REST-based interfaces over HTTP/2 protocol. X-IO also takes into account the need to maintain I/O compatibility by offering HTTP/REST APIs in Golang. This facilitates seamless porting of 5GC function to use X-IO and helps improve the cellular core to have much lower control-plane latency.

We build the HTTP/REST APIs using the socket interfaces in X-IO. X-IO's HTTP implementation has several optimizations to reduce the HTTP data transmission overhead compared to Golang's native HTTP implementation (referred to as Go-HTTP in short) in the "net/http" package [18], which was extensively used to construct the 3GPP SBI.

X-IO's HTTP interface sends the HTTP request header and request's payload together to reduce the delay for HTTP transactions. Go-HTTP lacks the optimization for receive buffer management on a Read() call[2]. For a large HTTP data frame whose size exceeds the receive buffer size, Go-HTTP continues reading the receive buffer in a loop until it detects "End-of-file", indicating the end of the payload. To assemble multiple receive buffer reads into the payload buffer, each time Go-HTTP completes a receive buffer read, it copies the data from the receive buffer to the payload buffer. The size of the payload buffer is determined by the "Content-length" contained in the HTTP header. X-IO instead chooses to allocate a large enough shared memory buffer to store the complete payload, avoiding the assembly and disassembly on both the sender and receiver side, thus reducing copies during the HTTP transaction. The shared memory buffers are allocated during the initialization of the X-IO manager, resulting in a fixed memory footprint. The resource wastage of shared memory buffers can be alleviated since the shared memory buffers are recycled after previous HTTP transactions are completed, and reused for subsequent HTTP transactions. We leave this last optimization of buffer management as part of our future work.

We build X-IO's HTTP implementation as a Golang package named "xio/http". X-IO's HTTP package keeps the same semantics as Go-HTTP. Porting an HTTP application (*e.g.,* a 5GC function) built on Golang's net/http package involves simply replacing net/http package with our xio/http. This achieves seamless porting to the X-IO framework.

F. Deployment Strategy

To leverage X-IO's shared memory processing, microservices need to be placed on the same node, especially when they have large data dependencies among others and are often organized in a chain. This can be achieved by specifying the node affinity of microservices in the same chain, when working with orchestration engine such as Kubernetes [19]. For inter-node communication (*e.g.,* when microservices that have data dependencies are placed on different nodes), users would switch to kernel-based networking, *e.g.,* using Linux TCP sockets. However, the lower bound for the performance

[2]based on examining the source code [18] of Golang's net/http package.

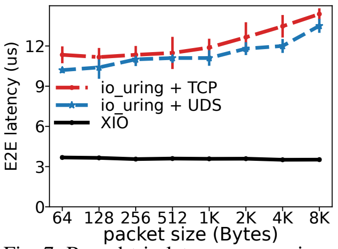

Fig. 7: Round-trip latency comparison between X-IO's zero-copy interfaces and io_uring alternatives

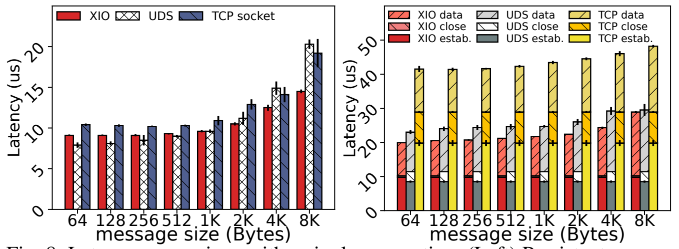

Fig. 8: Latency comparison with a single connection: (Left) Persistent connection; (Right) Short connection. 'data' - data transmit delay; 'close' - connection close delay; 'estab.' - connection establishment delay.

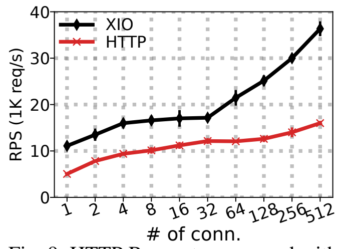

Fig. 9: HTTP Request per second with increasing # of connections: X-IO VS. kernel-based HTTP APIs

of X-IO would be no worse than a complete kernel-based networking solution, while the upper bound performance of X-IO would far outperform kernel-based networking by considering node affinity when placing microservices so as to fully exploit shared memory processing.

IV. Evaluation

Experiment Setup: We compare the performance of X-IO's zero-copy interfaces and socket interfaces with different I/O alternatives based on Linux kernel networking, including basic socket interfaces (*e.g.,* read()/write()) and io_uring [5]. We consider both TCP sockets and Unix-domain sockets (UDS in short) as the backend for kernel-based interfaces. We use the "net" package in Golang that offers the implementation of socket interfaces. We use liburing [20] (in C) to implement io_uring. We further examine the performance of the HTTP APIs in X-IO compared with the "net/http" package in Golang. We set up our experiments on a single node, with an 8-core CPU, 192GB memory, and Ubuntu 20.04 (kernel version 5.15). All results show the standard deviation.

A. Zero-copy interface performance: compared with io_uring

We implement an L2 forwarder function and a packet generator function (both placed on the same node) using io_uring and X-IO's zero-copy interfaces separately. The packet generator sends raw packets to the L2 forwarder, which then returns packets back to the packet generator. We measure the round-trip latency. We enable both submission polling and completion polling for io_uring [5]. With submission polling, the kernel polls the submission ring that contains the I/O request posted by the application. With completion polling, the application polls the completion ring that contains notification of completed I/O requests. Using submission/completion polling avoids the use of interrupts with the io_uring for I/O operations between userspace application and kernel. This achieves the best possible performance for io_uring.

Fig. 7 compares the round-trip latency between different alternatives. X-IO achieves 2.8×∼4.1× lower round-trip latency than io_uring (either with TCP socket or UDS). Specifically, X-IO has constant latency (3.6us) across various message sizes, demonstrating the benefit of *zero-copy* shared memory communication with X-IO. On the other hand, when using io_uring, 4 packet copies are incurred for every packet round-trip. These packet copies in io_uring inevitably impact packet forwarding performance. The superior performance of X-IO's zero-copy interfaces makes it much more attractive for latency-sensitive applications.

B. POSIX-like socket interface performance

To compare the performance for POSIX-like socket interfaces, we implement a benchmark application that contains a pair of client and server, using various socket interfaces: X-IO, TCP socket, and UDS. We configure multiple concurrent connections between the client and server to emulate the behavior with multiple user sessions. Each user session has a dedicated connection using a strictly 'request-response' pattern, with the client sending the next request only after it receives a response from the server.

Fig. 8 shows the performance across the different alternatives with a single connection. When using a persistent connection (Fig. 8 (left)), the performance of X-IO is better than using a TCP socket and has performance close to that achieved with a UDS. This is not surprising, since X-IO's socket interfaces have the same overhead (4 copies and 2 interrupts for a single round-trip) as with the TCP socket and UDS. It is just that the TCP socket has in-kernel protocol processing overhead in addition, which makes its performance slightly worse. Switching to short-lived connections, X-IO shows 2× and 1.1× latency improvement compared to the TCP socket and UDS (Fig. 8 (right)). This benefit comes from the fast connection establishment/close in X-IO, which can significantly reduce the overall packet transfer latency when using short-lived connections.

We further extend to having multiple concurrent persistent connections, over which the client and server exchange packets. We additionally disable X-IO's batch wake-up mechanism (refer to as X-IO-no-batch) to compare against default X-IO (refer to as X-IO-with-batch) to understand the effect of batched wake-up in handling frequent wake-up operations. The "net" package in Golang, by default, leverages Linux epoll to achieve batched wake-up. The Linux epoll [21] is an I/O event notification facility that adaptively batches wake-up operations to unblock higher-layer applications when multiple I/O events appear, which we consider as a representative implementation

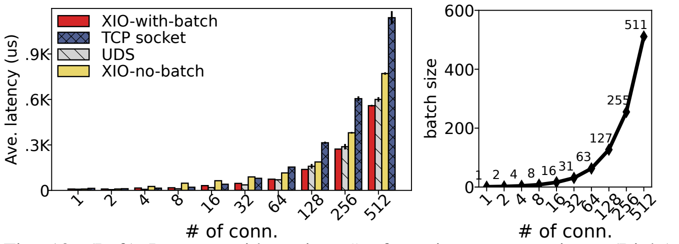

Fig. 10: (Left) Latency with variant # of persistent connections; (Right) Batched wake-up (using epoll) in TCP socket and UDS.

in servers with concurrent connection support. Fig. 10 (right) shows that the batch size of Linux epoll correlates with the number of connections. We only show the latency results when the message size is set to 64 Bytes (Fig. 10). The observations are consistent across a range of message sizes (512B, 1KB, 2KB) that we tested.

As shown in Fig. 10 (left), X-IO-with-batch consistently has lower latency than TCP socket and is close to that with the UDS. In contrast, the latency of X-IO-no-batch is always higher than using UDS. This clearly shows the benefit of performing wake-up (unblocking) of multiple connections in a batch that can amortize the overheads of interrupts and context switches, thus reducing overall latency. However, compared to UDS, X-IO's socket interface does not achieve significant performance improvement. This is mainly constrained by the copy overhead when moving payload between shared memory and the application's send/receive buffer (as the input to the Read()/Write() call). This motivates our ongoing work to design a truly zero-copy socket interface for X-IO to further improve its performance.

C. HTTP/REST API Performance

We study the HTTP performance using a simple HTTP echo server/client pair built with X-IO's HTTP APIs and Golang's "net/http" package. We vary the number of concurrent HTTP connections. Each connection is allowed to have one in-flight request (64Bytes) at a time, *i.e.,* the client can only send out a new request after receiving the response from the server. We consider persistent connections in this experiment.

Fig. 9 shows the results of HTTP request per second (RPS) comparison. X-IO achieves $1.4\times\sim2.3\times$ improvement in RPS and latency (not shown) because of the optimizations in the HTTP support (§III-E). Golang's "net/http" package introduces extra data copies when passing the request between the socket interfaces (*i.e.,* Read()/Write()) and HTTP APIs (*e.g.,* Get()/Post()), which limits its performance. The performance gap further increases because of the increasing CPU overhead from copies with more connections performing communication in parallel. This clearly shows the benefits of X-IO's optimization for HTTP support, which eliminates data copies for HTTP calls. X-IO scales when handling a large number of concurrent HTTP exchanges.

V. Conclusion

This paper describes X-IO, a high-performance, unified I/O interface designed for cloud-native microservices. X-IO offers a set of versatile interfaces that can be used to construct synchronous/asynchronous communication methods under different microservice scenarios, *e.g.,* NFV, Middlebox, and 5GC. X-IO builds the communication stack using shared memory, with lock-free producer/consumer rings. Using X-IO's zero-copy interface, X-IO delivers $2.8\times\sim4.1\times$ better performance compared to kernel-based io_uring interface. X-IO also abstracts the Golang-style socket interfaces to support seamless porting from a kernel-based socket interface to X-IO. X-IO's socket interfaces achieve competitive performance compared to kernel-based TCP socket and Unix-domain socket. X-IO further builds HTTP support using the socket interfaces in X-IO, which helps to ease the development of cloud-native microservices that often adopt HTTP-style, "request-response" communication paradigm, while in the meantime, X-IO's HTTP interfaces achieve $1.4\times\sim2.3\times$ better performance compared to the native HTTP interfaces available in Golang.

Acknowledgment

We thank the US NSF for their generous support through grant CRI-1823270. This work was also supported in part by the National Science and Technology Council of Taiwan under grant numbers 111-2221-E-A49-093-MY3, 111-2218-E-A49-023, and 111-3114-E-A49-001. We thank Poyi Lin of National Yang Ming Chiao Tung University, Taiwan, for his help in the implementation.

References

[1] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. K. Ramakrishnan, "Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 780–794.

[2] S. Qi, Z. Zeng, L. Monis, and K. K. Ramakrishnan, "Middlenet: A unified, high-performance nfv and middlebox framework with ebpf and dpdk," *IEEE Transactions on Network and Service Management*, 2023.

[3] "Cgo," https://pkg.go.dev/cmd/cgo, 2023, [ONLINE].

[4] https://man7.org/linux/man-pages/man7/aio.7.html, 2023, [ONLINE].

[5] https://kernel.dk/io_uring.pdf, 2023, [ONLINE].

[6] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. K. Ramakrishnan, and T. Wood, "Opennetvm: A platform for high performance network service chains," in *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*, 2016, pp. 26–31.

[7] Q. Su, C. Wang, Z. Niu, R. Shu, P. Cheng, Y. Xiong, D. Han, C. J. Xue, and H. Xu, "Pipedevice: a hardware-software co-design approach to intra-host container communication," in *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, 2022, pp. 126–139.

[8] A. Kalia, M. Kaminsky, and D. G. Andersen, "Datacenter rpcs can be general and fast," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. USA: USENIX Association, 2019, p. 1–16.

[9] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang, "Socksdirect: Datacenter sockets can be fast and compatible," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 90–103.

[10] "Data Plane Development Kit," https://www.dpdk.org/, 2023, [ONLINE].

[11] "DPDK Ring Library," https://doc.dpdk.org/guides/prog_guide/ring_lib.html, 2023, [ONLINE].

[12] "DPDK Multi-process Support," https://doc.dpdk.org/guides/prog_guide/multi_proc_support.html, 2023, [ONLINE].

[13] "DPDK Mempool Library," https://doc.dpdk.org/guides/prog_guide/mempool_lib.html, 2023, [ONLINE].

[14] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017.

[15] "free5GC," https://github.com/free5gc/free5gc, 2023, [ONLINE].

[16] V. Jain, H.-T. Chu, S. Qi, C.-A. Lee, H.-C. Chang, C.-Y. Hsieh, K. K. Ramakrishnan, and J.-C. Chen, "L25gc: a low latency 5g core network based on high-performance nfv platforms," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 143–157.

[17] V. Jain, S. Panda, S. Qi, and K. K. Ramakrishnan, "Evolving to 6g: Improving the cellular core to lower control and data plane latency," in *2022 1st International Conference on 6G Networking (6GNet)*. IEEE, 2022, pp. 1–8.

[18] "net/http," https://pkg.go.dev/net/http@go1.19.4, 2023, [ONLINE].

[19] https://kubernetes.io/, 2023, [ONLINE].

[20] https://github.com/axboe/liburing, 2023, [ONLINE].

[21] https://man7.org/linux/man-pages/man7/epoll.7.html, 2023, [ONLINE].