Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery

Taehyun Kim
KAIST

Deondre Martin Ng
KAIST

Junzhi Gong
Harvard University

Youngjin Kwon
KAIST

Minlan Yu
Harvard University

KyoungSoo Park
KAIST

Abstract

The recent advancement of high-bandwidth I/O devices enables scalable delivery of online content. Unfortunately, the traditional programming model for content servers has a tight dependency on the CPU, which severely limits the overall performance. Our experiments reveal that over 70% of CPU cycles are spent on simple tasks such as disk and network I/O operations in online content delivery.

In this work, we present IO-TCP, a split TCP stack design that drastically reduces the burden on CPU for online content delivery. IO-TCP offloads disk I/O and TCP packet transfer to SmartNIC while the rest of the operations are executed on the CPU side. This division of labor realizes the separation of control and data planes of a TCP stack where the CPU side assumes the full control of the stack operation while only the data plane operations are offloaded to SmartNIC for high performance. Our evaluation shows that IO-TCP-ported lighttpd with a single CPU core outperforms the Atlas server and lighttpd on Linux TCP for TLS file transfer by 1.8x and 2.1x, respectively, even if they use all 10 CPU cores.

1 Introduction

The demand for online content delivery is booming in recent years [1, 5]. Especially, the popularity of high-quality video streaming is growing rapidly [9, 56].For cost-effective streaming service, it is highly important for online video service providers [3, 4, 11, 19, 26, 44] to optimize their content delivery systems.

However, improving the content delivery performance is increasingly challenging as the growth of CPU capacity stagnates [50]. While modern innovation in I/O devices such as high-bandwidth NICs and NVMe disks has alleviated the I/O bottleneck, the lack of CPU cycles often fail to translate the high I/O performance into content delivery throughput. The root cause lies in the inefficiency of the modern OS abstraction which requires all disk data to be brought to main memory before being delivered to remote clients. For this reason, CPU (or more precisely, the memory subsystem) easily becomes the performance bottleneck for I/O-intensive applications like video content delivery as over 70% of its entire cycles are spent on simple I/O operations. To effectively harness the recent advancement in the I/O devices, the OS abstraction must reduce the dependency on CPU and its memory system for I/O operations.

Our approach to breaking the CPU dependency is to employ peripheral processors to handle the I/O operations. We observe that the recent programmable I/O devices such as SmartNICs [7, 24, 27, 33] or Computational SSDs [28, 40] may make up for the insufficient compute cycles in CPU. As the PCIe standard allows peer-to-peer DMA (P2PDMA) without the intervention of CPU [35], one can conceive a server system whose NIC offloads disk I/O operations completely from the CPU. In fact, recent works like DCS [46] and DCS-Ctrl [63] have demonstrated that an FPGA-based coordinator can perform all disk I/O operations via P2PDMA for a content delivery server. The main drawback of these systems is that they do not support TCP-based delivery commonly adopted by today's video streaming [3, 11, 26].

However, supporting TCP for an I/O-offloaded server raises an interesting question of function placement – if disk I/Os are offloaded to SmartNIC, where do we run the TCP stack? Running the TCP stack on CPU is impossible as the data for packet payload is unavailable. So, the obvious alternative is to run it on the NIC side. While it is non-trivial to implement a full TCP stack on an FPGA [1], it is possible to run it on SmartNIC. Actually, recent SmartNIC platforms support Arm-based embedded processors that run Linux with a full TCP stack [7, 33]. However, running the full TCP stack on NIC typically requires its application to co-execute on the same platform, with limited resources. In fact, we observe that the throughput of `nginx` on SmartNIC with 8 Arm cores is smaller than that with even a single CPU core.

We tackle this question with I/O Offloading TCP (IO-TCP), a split TCP stack design for I/O-intensive applications. The

[1] There are a few TCP/IP stacks on the FPGA [87, 88], but they simplify the key features with assumptions on the data center environment.

key idea of IO-TCP is to run only the "control plane" operations on CPU while delegating all "data plane" operations to SmartNIC that can access disks via P2PDMA. Figure 1 shows the overview of our design. The control plane includes all core functionalities of the TCP protocol – connection management, reliable data transfer, and congestion/flow control. On the other hand, the data plane operations refer to all aspects of data packet creation and transmission including content fetching from disks. This design ensures that the CPU side assumes full responsibility of controlling all operations while actual disk and network I/O operations are offloaded to SmartNIC under the hood. This design enables dedicating CPU cycles to complex control operations while exempting them from simple but repetitive I/O operations. The rationale for the design is that the split stack avoids CPU cache pollution from intensive disk IO [96] that slows down the control path operations, stretches the RTT, and lowers the throughput. In addition, the control path would benefit from advanced hardware features of modern CPU as it is compute-intensive with frequent random accesses and branches. In contrast, the data path depends more on memory bandwidth than computation, and it is easily parallelizable and can be even built into hardware.

While IO-TCP presents a great potential for saving CPU cycles, it brings a few new challenges. First, IO-TCP must handle TCP packet retransmission on SmartNIC without the timeout or packet loss information, which may issue redundant disk I/Os. To avoid the inefficiency, IO-TCP employs an internal ACK protocol to notify the SmartNIC of the data delivery so that it can safely throw it out from memory. Second, the RTT measurement in the host TCP stack could be inaccurate due to disk-induced delay on SmartNIC before transmission. In IO-TCP, actual data packet transfer is delayed until the packet content is fetched from the disk. However, disk I/O could add significant delay to packet transfer even without any congestion in the network path. IO-TCP addresses this challenge by carefully removing the disk-induced delay from RTT measurement. It employs an echo packet that allows the host stack to keep track of the packet departure time accurately. Third, IO-TCP must provide a well-defined API for an application to flexibly construct file or non-file content for data transfer. For this, IO-TCP extends the Berkeley socket API with a few "offload" functions that open a file and send the file content from the NIC. The "offload" functions are implemented as a form of API remoting, and the results are seamlessly delivered to the application on the CPU side.

We implement IO-TCP with the Mellanox BlueField-2 SmartNIC [31] that can directly access NVMe disks with P2PDMA. For the host stack, we extend an existing user-level TCP stack [58] to support I/O offloading while we implement the NIC stack with the DPDK library [12]. It requires 1,793 lines of code modification for the host stack and 1,853 lines of C code for the NIC stack. To evaluate the effectiveness with real-world applications, we also port lighttpd [23] to

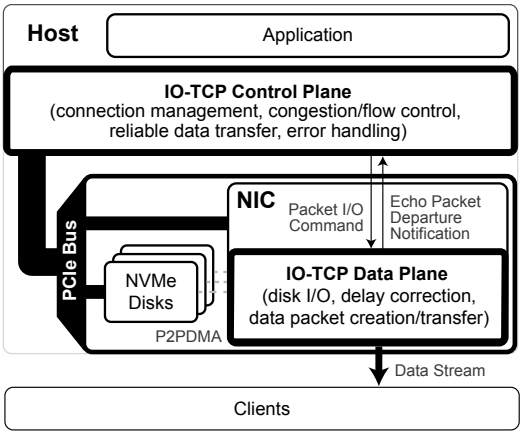


Figure 1: Overview of IO-TCP stacks

using IO-TCP with only about 10 lines of code modification.

Our evaluation demonstrates that IO-TCP-ported lighttpd achieves 77.4 Gbps of TLS video content delivery with a single CPU core, nearly saturating the full bandwidth of four NVMe disks. In contrast, the Atlas server [74] on FreeBSD and lighttpd on Linux reach only 44.2 and 37.4 Gbps, respectively, even with all 10 CPU cores. We observe that the current bottleneck of IO-TCP lies in the low memory bandwidth of the BlueField-2 NIC, but we believe the future version will achieve better performance. The main contributions of this work are summarized as follows. (1) We analyze the impact of CPU usage and cache interference by disk I/O on the performance of modern content delivery systems. (2) We present the design and implementation of IO-TCP, a split TCP stack design that fully leverages recent I/O advances in SmartNICs by separating TCP control and data planes. (3) We demonstrate how IO-TCP can surpass the limitations of the CPU bottleneck to achieve I/O bandwidths far greater than what the CPU could have normally performed.

2 Background & Motivation

We provide a brief background on content delivery systems in terms of recent trends in computing hardware.

2.1 Inefficiencies in Content Delivery System Stacks

Modern content delivery systems [2, 14, 26] consist of a large number of geographically distributed content delivery Web or reverse proxy servers. These systems serve as the basis for many applications such as video streaming and Web page accesses. Among them, the video traffic takes up about 60% of the entire Internet traffic and the overall volume has increased due to the recent pandemic [39, 56]. Average Web object sizes range from 0.01 to 1 MB while average video chunk sizes are between 0.2 to 1.5 MB [89].

For high performance, the server design has traditionally focused on optimizing disk access and CPU utilization because hard disk I/O is many orders of magnitude slower

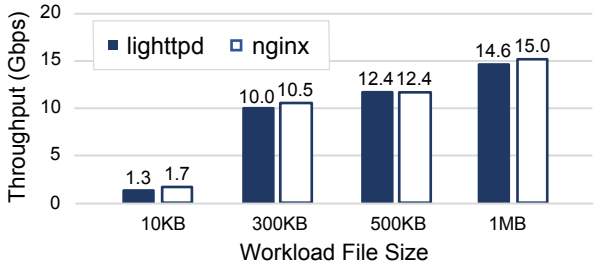


Figure 2: Throughputs of lighttpd and nginx on a single CPU core.

Function	% CPU
Read data from disk to a kernel buffer	33.53%
Memory management	21.93%
Move data to TCP send buffer (no copy)	10.30%
Open files and get stat	6.00%
Control Plane	*28.24%*
Total	**100%**

Table 1: CPU usage per data plane function in nginx when serving disk-bound workload with `sendfile()`. The breakdown for lighttpd is also similar.

than modern storage. For fetching small objects such as Web page content, the server is optimized to minimize the disk seeks while maintaining a small memory footprint for indexing [47]. For large-object access like video download, the server exploits sequential disk reads to maximize the disk throughput. Also, it typically employs `sendfile()` to avoid redundant memory copy and context switching between user and kernel spaces. To improve CPU utilization, the server typically takes the event-driven architecture [48, 67, 77, 81].

Traditional disk-based optimizations have become largely obsolete due to the advent of inexpensive large RAM and flash-based disks (e.g., NVMe SSDs) that removed the seek-induced limitations. Since the major disk bottleneck is lifted, the memory subsystem becomes the next bottleneck in today's server [74]. The problem is exacerbated by multiple memory copies due to disk and network I/O as well as content scanning for encryption and decryption. While a recent work [74] optimizes the disk access layer and exploits Intel Data Direct I/O (DDIO) [21] to arrange all such operations to perform with the data in CPU cache, it does not dissipate the workload from CPU. Also, it may be hard to expect a similar benefit if the workload exceeds the CPU cache size.

To better understand the performance of Web-based content delivery, we run experiments with two popular Web servers, lighttpd (v1.4.32) [23] and nginx (v1.16.1) [29] for disk-bound workload, which simulates a typical setting for HTTP-adaptive video streaming. The server setup is the same as in §5.1, and we use various file sizes that represent Web objects and video chunks of different quality. We configure the servers to use `sendfile()` for good performance.

Figure 2 shows the results with a single CPU core (refer to Figure 7 for performance trend over multiple CPU cores). The performances of both servers are similar, and they generally improve with larger file sizes. As our NVMe disk achieves

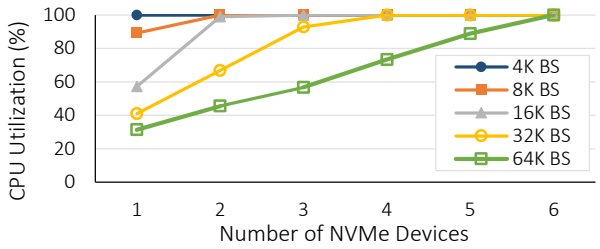


Figure 3: CPU utilization of `fio` with for varying number of NVMe disks. BS refers to block size.

around 2.5 GB/s (or 20 Gbps) per disk for random file reading, the single CPU core utilizes around half the bandwidth (10 Gbps) of a single NVMe disk for 300KB files. Considering that a server-class machine can carry 8 to 10 NVMe disks per CPU, CPU is a major source of resource bottleneck.

We analyze the CPU overhead in popular Web servers for content delivery. Table 1 shows the CPU cycle breakdown of nginx reported by `perf`[36]. `sendfile()` and `open()` take up the majority of the CPU cycles, which amounts to 71.76% of the consumed cycles. `sendfile()` reads the data on disk to kernel buffers, and serve it to clients without memory copy (33.35%). This clearly shows where the most of CPU cycles are spent in a content delivery server – disk and network I/O. Offloading these operations from the CPU would have a great potential for improving the performance.

2.2 Mismatch between I/O Device Advances and CPU Capacity

The capacity growth with recent I/O devices is impressive. Two decades ago, the fastest hard disk could achieve only about 200 random I/O operations per second (IOPS), but the recent NVMe disk can perform over 1 million IOPS [41, 97], a speedup of almost four orders of magnitude. For the same period, the bandwidth of an Ethernet NIC has improved by 400 times (from 1 Gbps in 1997 to 400 Gbps in 2021) while 800 Gbps / 1.6 Tbps Ethernet is expected to be standardized in a few years [55]. In contrast, CPU capacity improvement has been largely hampered by the end of Moore's law and breakdown of Dennard scaling [2] [54]. The first general-purpose multicore CPU appeared in 2005 [6], but the number of cores of Intel CPU has increased by only 28 times for 16 years [22].

Figure 3 shows the utilization of a single CPU core when saturating the NVMe disks with `fio` [15]. We use Intel Xeon Silver 4210 (2.20GHz) for CPU and Intel Optane 900P for NVMe devices. The figure indicates that it is relatively easy to handle large block sizes but a single core cannot saturate even 2 NVMe disks with a block size of 4KB. For 16KB blocks, it can handle up to 3 NVMe disks in parallel. Even when serving large files, disk I/O could still spend a significant portion of the CPU cycles as metadata access in filesystem would require frequent random accesses for small blocks.

[2]Dennard scaling dictates that the power density stays constant as the transistors become smaller. It is said to stop in 2006 and CPU capacity could only scale out since then.

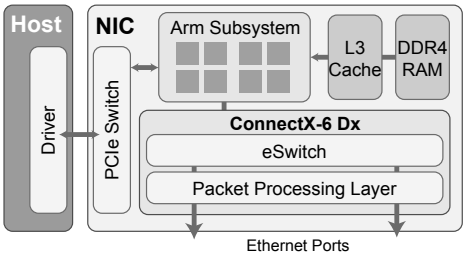


Figure 4: Architecture of the BlueField SmartNIC

In addition to NVMe, the use of persistent memory (PM) sees similar CPU bottlenecks. A recent PM performance study [94] on Intel Optane DC memory shows that 16 cores are required to fully utilize PM read bandwidth even with large I/O sizes like 64KB or 256KB. Due to the CPU bottleneck, many PM-based storage systems fall back to a lightweight storage stack design that misses features [51, 52, 53, 59, 64, 93, 98].

The performance disparity between CPU and I/O devices calls for revisiting the current OS abstraction for I/O operations, especially for serving large files with the growing trend of high-throughput content delivery. Existing OS requires CPU intervention for performing I/O operations such as reading disk content and transferring it via NIC. This is because the programming model on the current OS requires the content of the I/O device to be brought to main memory before performing any operation on the content. This memory-centric execution model wastes CPU cycles for frequent memory access stalls due to memory operations.

2.3 Opportunities with SmartNIC

The key idea of our work is to offload data I/O from CPU to a programmable I/O device while supporting TCP-based content delivery. Any programmable device that can perform direct disk I/O and network packet I/O can meet our goal, but we use SmartNIC as it serves as a convenient place to interact with remote clients. For example, recent SoC-based SmartNICs [7, 33] offer an Arm-based embedded system on top of a NIC data processing unit. These systems support direct access to NVMe disks on the same domain without intervention of CPU or main memory. More specifically, the Mellanox BlueField NIC supports P2PDMA via NVMe over Fabrics (NVMe-oF) target offload [18] through which the Arm processors can read directly from local NVMe disks. These disks are directly mounted on the Linux environment running on the Arm processors, and they run on the same file system as seen by the host OS.

Figure 4 shows the architecture of the Mellanox BlueField-2 NIC (BF-2) [31] that we use for our platform. It is equipped with 8 Armv8 cores and 16 GB of DDR4 memory that runs on Linux [3]. The Arm subsystem allows running DPDK applications to perform fast packet I/O either with remote machines or with the local host. In addition, applications can offload TCP/IP checksum calculation as well as TCP segmentation (i.e., TSO) to its ConnectX-6 Dx NIC hardware. The BlueField-2 also supports hardware acceleration for cryptographic operations that we use for supporting TLS.

[3]We run CentOS 7.6, but one can run embedded Linux like Poky [37].

lighttpd setup	Throughput (Gbps)
Linux TCP on BF-2 only	11.98
Linux TCP on BF-2 and 1 CPU core	22.02
IO-TCP-on BF-2 and 1 CPU core	44.13

Table 2: Performance of lighttpd with Linux TCP vs. IO-TCP for serving 300KB files over 1600 connections on BlueField-2 (BF-2) and a single CPU core. We use four Intel Optane 900P in all experiments.

With the Linux-operated SmartNIC, one might be tempted to use it as an extra server system [90]. However, running a server directly on SmartNIC does not efficiently use the resources. Table 2 compares the performances of lighttpd on only BF-2 (w/ all 8 cores), lighttpd on BF-2 and the host's single CPU core combined (by evenly dividing the request load), and IO-TCP-ported lighttpd on the same setup. Naïve scaling of processing power with SmartNIC ends up with only half the throughput of our solution (§4).

The experiments clearly show the current limitation with the SmartNIC – the processors and their memory are not so powerful as the host system. In fact, the Arm processors on BF-2 have 2.2x and 4.2x smaller L3 cache and memory bandwidth than those of our host CPU, which limits the overall performance. While this is not an inherent limitation as the next version [32] is reported to have 3.5x larger memory bandwidth, one should carefully design the offload functionality to effectively exploit the architectural difference.

3 Design

In this section, we present the design of IO-TCP that enables content delivery systems to leverage recent SmartNIC I/O advances. The key design choice of IO-TCP is to *separate the control and data planes* of the TCP stack such that the CPU stack takes the full control of every operation (control path) while individual I/O operations (data path) are offloaded to the SmartNIC stack. The core rationale for this is to save the majority of CPU cycles for performing I/O operations while keeping the SmartNIC stack simple to implement. Simplicity is the key to achieve the performance scalability.

There are three design goals for IO-TCP: (1) IO-TCP must conform to the TCP protocol and should be able to support various congestion control implementations. For example, handling disk I/O in the NIC stack should not compromise the congestion control logic in the host stack due to imprecise RTT measurements induced by disk access latency (§3.5). (2) The modification of existing applications should be minimal for migrating to IO-TCP – it should use the same socket API except for offloading file I/O (§3.2). (3) The IO-TCP host stack needs to communicate with the NIC stack for I/O offloading, and its overhead should be made small. In addition, the host stack should be notified of any failures in the NIC stack to

`int offload_open(const char *filename, int mode)` – opens a file in the NIC and returns a unique file ID (fid).
`int offload_close(int fid)` – closes the file for fid in the NIC.
`int offload_fstat(int fid, struct stat* buf)` – retrieve the metadata for an opened file, fid.
`size_t offload_write(int socket, int fid, off_t offset, size_t length)` – sends the data of the given length starting at the offset value read from the file, fid, and returns the number of bytes virtually copied to the send buffer.

Table 3: IO-TCP offload API functions

handle them in time (§3.3 and §3.6).

3.1 Separating TCP control and data planes

To save host CPU cycles, we need to determine which operations would benefit the most from offloading based on the capabilities of SmartNICs and CPU. The embedded processors on either SoC or ASIC-based SmartNIC are better fit for simpler data plane operations while x86 CPUs with advanced features [4] can handle complex control plane operations faster. To better reflect the architectural difference into the design, we divide the TCP stack into control and data plane operations.

The control plane functions refer to the key TCP protocol features such as connection management, reliable data transfer, congestion/flow control, and error control. These typically require complex state management as the behavior depends on the response from the other end. For example, reliable data delivery on the receiver side requires tracking all received data ranges that are disjoint for proper in-order delivery and ACK generation. It is also tightly coupled with congestion control as loss detection and packet retransmission for reliable delivery in turn re-adjust the send window size. Similarly, flow control needs to run with congestion control as they collectively determine the window size. Error control cannot run alone, either, as it requires tracking detailed flow states to infer any erroneous behavior. Theoretically, each individual operation can be offloaded, but it would be more efficient to offload them together. However, offloading them all could overload the SmartNIC as seen in experiments in §2.3.

The data plane operations refer to all operations that involve data packet preparation and transfer, which supports the implementation of control plane functions. These include managing data buffers, segmenting data into packets, calculating TCP/IP checksums, etc. IO-TCP offloads only the operations in the send path because they are simple, stateless, and easily parallelizable. In addition, IO-TCP offloads the file/disk I/O and combines it into TCP data plane operations. The rationale for offloading is that these operations would interfere with control plane operations on CPU as recent innovation like Intel DDIO would pollute the CPU cache by huge disk data [96]. Offloading them to SmartNIC would allow the control path to execute on CPU much faster, which in turn improves the data path performance. Also, SmartNICs tend to have hardware-based crypto accelerators [24, 27, 31], which enables TLS data encryption at line rate. Section 5.5 analyzes the source of performance improvement in depth.

[4] Such as larger CPU cache and vectorized instructions like AVX/AVX2.

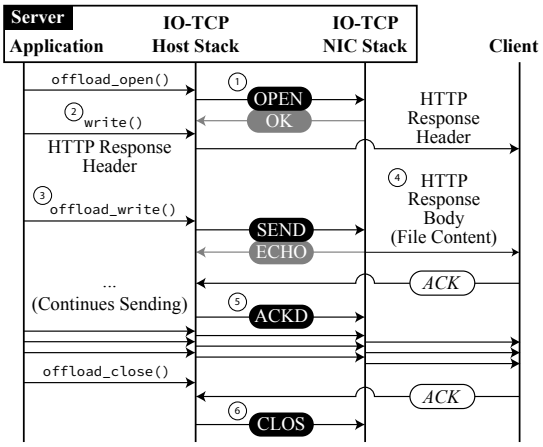


Figure 5: Content delivery from a Web server on IO-TCP

3.2 IO-TCP Offload API Functions

Ideally, porting an application to IO-TCP should require little modification of its core logic, yet it should flexibly express the application needs. For example, an IO-TCP application should be able to compose any data to transfer regardless of whether it is file content or not. Towards this goal, we extend the existing socket API by adding only four functions (see Table 3) for offloaded file and network I/O.

`offload_open()` asks the NIC stack to open a file and to report the result (either success or any error). It returns a file ID (instead of a file descriptor) that identifies the opened file in the NIC stack for later operations. `offload_open()` is an asynchronous function whose result should be checked with `epoll()` or subsequent function calls as file opening can fail for various reasons. After all file operations, the application can call `offload_close()` to close the file on the NIC stack. In addition, IO-TCP supports `offload_fstat()` that retrieves the metadata for a file (e.g., file size and permission).

With the opened file ID, the application can call `offload_write()` to send the file content on a TCP connection. Essentially, `offload_write()` carries out the same operation as `sendfile()` in Linux with the file opened at the NIC embedded system. The application can still call an existing socket API like `write()` to send out any custom data (e.g., HTTP response headers), or it can send the content from multiple files opened by the NIC stack. Figure 5 illustrates a subset of these operations with the API functions in the context of an HTTP server.

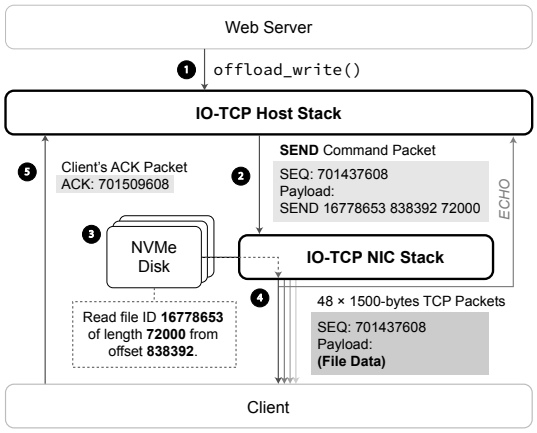


Figure 6: Generation of data packets with `offload_write()`

3.3 IO-TCP Host Stack

The role of IO-TCP host stack is to provide the full TCP functionality to applications while it interacts with the NIC stack to offload the data plane operations. The key challenge in the host stack design is how to create data packets with "missing" file data. Similarly, it should handle TCP packet retransmission without actual file data in the host side.

IO-TCP addresses the challenge by *virtually* performing data plane operations on the host stack. The host stack keeps track of which data in the sequence number space is "virtual" and performs only the bookkeeping operations while it delegates the real I/O operations to the NIC stack. For example, an application can call a mix of `write()` and `offload_write()`, and the host stack writes the immediate content directly into the send buffer while it virtually fills out the buffer range for `offload_write()` by metadata update. `offload_write()` returns immediately with the number of "virtual" bytes that can be written to the send buffer.

Then, the host stack determines the send window size with its congestion and flow control parameters, and posts a "SEND" command to the NIC stack to transfer the virtual data (Refer to ③ and ④ in Figure 5). Note that any data packets with real content (written by `write()`) in the host stack are sent out directly bypassing the NIC stack. [5] The "SEND" command is carried on a TCP packet destined to the NIC stack (with an internal MAC address of the NIC). The TCP/IP headers of the command packet contain the full connection information (i.e., four connection tuples, sequence and ACK numbers for the next data packet, etc.) while its payload contains the "SEND" command that is eventually replaced by the real content before it is sent out to the client. The "SEND" command specifies a file ID, the start offset to read, and the length of the data. With this information, the NIC stack reads the file content and creates and sends real data packets with the header information. Depending on the file content size, one "SEND" command can be translated into multiple MTU-sized data packets. Figure 6 illustrates how a "SEND" command packet is processed.

The host stack handles packet retransmission in the same manner – sending a "SEND" command with the file content information for retransmission. The rationale for this design is to make the NIC stack as simple as possible. An obvious alternative is to have the NIC stack handle retransmission so that it ensures reliable delivery of whatever data is transmitted due to the "SEND" command. Then, the NIC stack must keep track of all ACKs from the client and run the congestion control logic to determine when to retransmit packets. This would make the NIC stack stateful and more complex, which would be challenging to implement efficiently on some other SmartNIC platforms (e.g., FPGA-based ones).

For all other operations, IO-TCP behaves similarly to the normal TCP stack. All complex operations such as per-connection state and buffer management on the receive path, timer management, reliable data transfer, congestion/flow control, and error control are executed on the host stack. In addition, for the control packets or packets whose data is available on the host stack, the host stack creates and sends them directly to the client bypassing the NIC stack. All incoming packets from the client get delivered directly to the host stack as well. (See ② and client-sent ACKs in Figure 5) This is not only because going through the embedded system on the NIC incurs extra latency, but it also places an unnecessary burden on the NIC stack. This packet steering can be easily enforced in the separated mode of the Mellanox BlueField-2 NIC where an embedded system on NIC has different IP and MAC addresses.

3.4 IO-TCP NIC Stack

The IO-TCP NIC stack is responsible for performing all real data plane operations for the host stack – it handles offloaded file I/O and network I/O for data packet transfer. It operates by handling custom commands from the host stack where each command is carried on a special packet destined to the NIC stack. Currently, four commands are defined: "OPEN", "CLOS", "SEND", and "ACKD". "OPEN" and "CLOS" are for fille opening or closing. "SEND" is the main command for sending the file content to the client. "ACKD" is used to efficiently handle retransmission without redundant disk access.

The "SEND" command is the key driver for I/O operations. Conceptually, it extends TCP segmentation offload (TSO) with the metadata that describes how to fill in the packet payload. Given the "SEND" command, the NIC stack checks if the target file is opened, and reads the file content into a fixed-sized memory buffer. The file read offset and its length are aligned to the NVMe disk page boundary (e.g., 4KB), and the actual file I/O is executed asynchronously to prevent blocking of the main thread. When the file content becomes available on the memory buffer, the NIC stack creates a TSO packet with the TCP/IP headers in the "SEND" command packet, and sends it out to the NIC hardware data plane. The NIC

[5] If real data has to be sent after virtual data, the host stack delays transmission until the arrival of an echo packet (§3.5) to keep the order.

hardware data plane takes care of TCP packet segmentation and TCP/IP checksum calculation.

3.5 Challenges with Integrated I/O

Combining file I/O into the network I/O in the NIC stack brings a few unique challenges in the correctness of the TCP stack operation.

Retransmission timer and RTT measurement. TCP relies on delay measurement for setting up retransmission timers. However, the delays induced by disk I/O could confuse the RTT measurement. Even with fast NVMe disks, the disk access delay for reading a few KBs of data is in the order of microseconds, and it can be up to milliseconds if the I/O requests for the same disk are backlogged. We observe that our early implementation of IO-TCP often retransmits the packets even if the original packets have not been sent out to the client.

To address this problem, we have the NIC stack send back an *echo* packet to the host stack just before transferring data packets for the corresponding "SEND" command. The host stack starts the retransmission timer only when it receives an echo packet for the SEND command. For accuracy, the host stack adds a one-way delay of the echo packet (~3 microseconds on our platform) from the NIC stack to itself to the timeout value. The CPU overhead for the echo packet is small as it is sent per "SEND" command and a typical "SEND" command is translated to tens to even hundreds of MTU-sized packets for large-file delivery.

Also, for precise RTT measurement, the NIC stack reflects the real "packet processing" delay into the TCP timestamp option value, i.e., the delay between the arrival of a "SEND" command to the NIC stack and the departure of the corresponding data packets from the NIC stack. That is, the "SEND" command packet carries the TCP timestamp option filled by the host stack, and the NIC stack updates the value before sending out the packets. As the timestamp option value is in the millisecond granularity [38] and the time feed in the host stack is on the order of microseconds, the host stack sends the extra time information in the microsecond granularity to the NIC stack. Then, the NIC stack can round up the timestamp value if necessary.

Handling retransmission. Since retransmission of I/O-offloaded packets is also implemented with the "SEND" command, a naïve implementation that re-reads the file content would waste the disk and memory bandwidth. To avoid the inefficiency, the NIC stack keeps the original data content in memory until the host stack confirms the delivery to the client. When the host stack sees the ACKs for the I/O-offloaded sequence space range, it periodically informs the NIC stack of the delivered portion with the "ACKD" command packet. Then, the NIC stack can recycle the memory buffers holding the delivered data. To minimize the overhead, the host stack informs the NIC stack whenever it sees a threshold amount of data (e.g., we use 32KB for now) acknowledged by the client from the last time. Note that this buffer memory essentially serves as the socket send buffer in the normal TCP stack, and the required memory in practice roughly corresponds to the bandwidth-delay product. A 100 Gbps NIC with 30ms of average RTT for the connections would require 375MB of the buffer memory in aggregate.

3.6 Handling Errors

In IO-TCP, the host stack is responsible for handling all TCP-level errors such as handling packet losses, malformed packets, or abrupt connection failures. Since the NIC stack only sends packets on behalf of the host stack and all incoming packets bypass the NIC stack, the host stack can reason about any TCP-level errors as other TCP stacks do.

In contrast, the NIC stack must report errors in file I/O to the host stack. For an "OPEN" command, the NIC stack responds to the host stack whether opening a file was successful or not. Then, the host stack raises an event to the corresponding file ID so that the application learns the result. Since the host stack caches the metadata for an offloaded file (see §4), it can return an error if `offload_write()` is passed wrong parameter values. In case a file read operation itself fails, it is reported to the host stack with an "Error" command packet with the file ID and the error code. Then, `offload_write()` would return −1 with the error code at `errno` next time the application calls it.

3.7 Support for TLS and QUIC

TLS is widely used in the modern Internet as QUIC [65] and HTTP/2 [20] adopt it by default. IO-TCP can support TLS similarly to kTLS [92] except that it offloads the encryption to the SmartNIC. This is feasible as many SmartNICs (including Bluefield-2) [24, 27, 31] already support AES and SHA in hardware. So, the CPU side runs the TLS handshake and sets up the encryption and hashing keys with the SmartNIC. All data in the receive path should be decrypted by the CPU stack similarly to other receive-path processing in IO-TCP. One complication lies in how to encrypt the non-offloaded data, but one can forward such packets to SmartNIC for encryption or encrypt them with CPU's AES-NI instructions. Support for TLS is still in progress as we have implemented content encryption with AES-GCM in NIC hardware and plan to support TLS handshake and TLS record structures.

The key idea of IO-TCP can be easily applied to other transport layer protocols like QUIC – Appendix A briefly explains the architecture of IO-QUIC. We plan to elaborate on the detailed design in the follow-up work.

4 Implementation

IO-TCP host stack. We implement the IO-TCP host stack by modifying mTCP [58], a high-performance user-level TCP stack. We choose mTCP as its socket API is similar to the Berkeley socket API and it supports event-driven programming with `epoll`. The host stack extends the mTCP API

functions with four offload functions (as shown in Table 3). Each offload function is implemented by exchanging special command packets with the NIC stack. The NIC stack detects a command packet by checking the special value in the ToS field in the IP packet. The "SEND" command packet has valid TCP/IP headers with the full connection information so that only the payload (as well as checksums) needs to be replaced with the real file content before being sent to the client.

For `offload_open()`, the host stack generates a unique file ID for the file path and returns it to the user. Under the hood, it attaches the file ID to the "OPEN" command to refer to the opened file on the NIC stack. As part of response, the NIC stack provides the metadata of the opened file (e.g., output of `fstat()`) so that the host stack can handle `offload_stat()` locally on its own. This should cut back the round trip to the NIC stack. For file operations, both the host and NIC stacks share the same file system – the host OS mounts the file systems on the NVMe disks as read/writable while the NIC stack mounts them as read-only. One problem is that any update on the host-side file system does not automatically propagate to the NIC stack as they run on a separate operating system. While we currently assume that the files do not change during the content delivery service, one should add support for dynamic synchronization of the two file systems in the future.

IO-TCP NIC stack. The NIC stack is implemented as a DPDK application. It operates by handling command packets from the host stack. Each Arm core runs one main thread and a few disk reading threads that are pinned to the core. The command packets are distributed to the main threads by receive-side scaling (RSS) on the NIC hardware, which ensures in-order packet delivery in the same connection.

For efficient memory buffer management, the NIC stack pre-allocates all buffers for file content at startup. Each main thread owns $1/n$ of them to avoid any lock contention, and a simple user-level memory manager allocates and frees the buffers at low cost. We implement zero-copy DMA of file data and packet header with DPDK (i.e., scatter-gather DMA), which improves the large-file delivery throughput (in the experiments for Figure 8) by 63%.

File reading, even with faster NVMe disks, is slower than memory operations, so each main thread employs a few disk reading threads to prevent blocking of the main thread. Disk reading threads use direct I/O to bypass the inefficiency in the file system cache [74], and communicate with the main thread through shared memory. An alternative is to use a user-level disk I/O library like Intel SPDK [42]. In fact, we observe that SPDK reaches the peak disk read performance with half the Arm processor cycles used by direct I/O, but we stick to a regular file system here (i.e., ext4 on Linux) as SPDK's support for file system is not mature yet.

IO-TCP TLS implementation. We modify the DPDK NIC driver to offload TLS symmetric key encryption with the

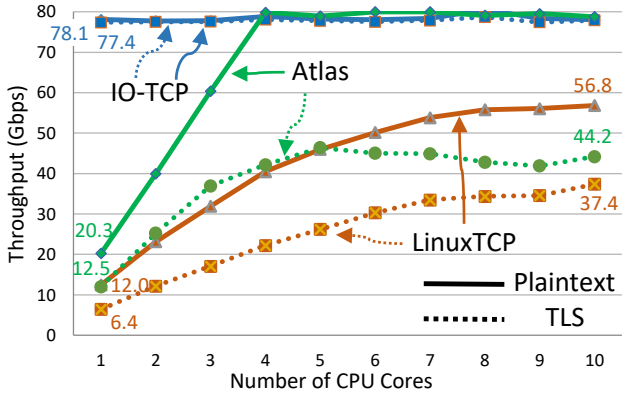


Figure 7: Comparison of throughputs of lighttpd on Linux TCP and IO-TCP, and those of the Atlas server [74] over varying number of CPU cores serving 500KB files. Dotted lines are for TLS traffic.

BlueField-2 NIC. Our TLS module initializes the NIC with TLS offloading feature enabled, and it registers a chosen ciphersuite with the NIC and returns an ID for it. When the TLS module marks the packets with the ID, the packets are encrypted with the corresponding ciphersuite. We implement AES-GCM with 256bit keys with ConnectX-6 Dx, which supports the encryption almost at line rate.

Porting lighttpd to IO-TCP. To evaluate the effectiveness of IO-TCP in the real-world applications, we port lighttpd v1.4.32 to IO-TCP. We obtain the mTCP-ported lighttpd code in Github [16], and have modified it to support offloaded I/O operations. Porting it to IO-TCP was straightforward as we needed to modify only about 10 out of 41,871 lines of the lighttpd code.

5 Evaluation

We evaluate IO-TCP with the following questions in mind: (1) how much performance improvement does IO-TCP bring over Linux or custom TCP stacks for content delivery systems? (2) does it result in significant CPU cycle saving? and (3) do our design choices (Retransmission timer and RTT measurement correction) serve their purposes well? Before running the experiments, we first verified the correctness (integrity of transferred files) of the IO-TCP stack with the IO-TCP-ported lighttpd server even in the case of many packet losses and multiple concurrent connections.

5.1 Experiment Setup

Our experiment setup consists of one server and two client nodes. The server machine has two Intel Xeon Silver 4210 CPU @ 2.20 GHz (20 cores) [6], with two 100G Mellanox BlueField-2 SmartNICs and four Intel Optane 900P NVMe SSDs. We attach one SmartNIC with two NVMe disks using NVMe-oF target offload so that the NIC can access the two NVMe disks directly. The host CPU runs on Linux 4.14 while the SmartNIC runs on Linux 4.20. The client machines are each equipped with an Intel E5-2683v4 CPU @ 2.10GHz (16

[6] We use only one CPU (i.e., 10 cores) for experiments.

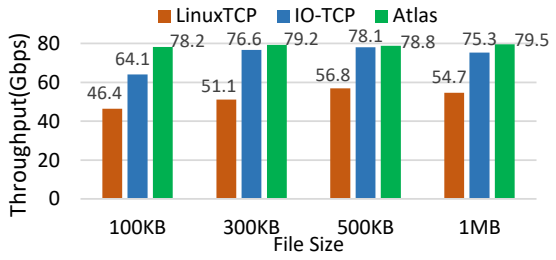
cores) and a 100G Mellanox ConnectX-5 NIC. All clients run on Linux 4.20, and we confirm that the clients are not the bottleneck for the experiments. All NICs are connected to a 100 Gbps Dell EMC Networking Z9100-ON switch.

We populate the NVMe disks with 100KB, 300KB, 500KB and 1MB files, which represent the video chunks of different quality [74, 89]. We make sure that the workload is disk-bound so that the working set size exceeds the main memory size. Each disk has an advertised read throughput of 2500 MB/sec, which would imply that we have a theoretical limit of 80 Gbps when reading from our four disks.

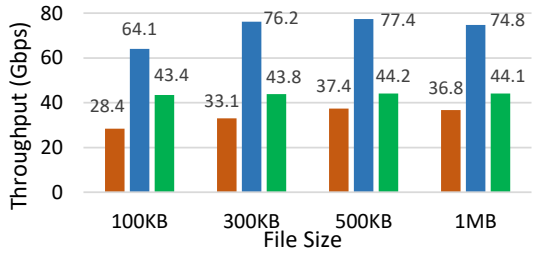
5.2 IO-TCP Throughput

We evaluate the effectiveness of IO-TCP in the large-file content delivery. We compare the throughput of IO-TCP-ported lighttpd and that of the stock version with `sendfile()` over varying numbers of CPU cores. We also compare against the Atlas server of Disk|Crypt|Net [74] that runs on FreeBSD 1.10. The Atlas server integrates raw disk reading into large-file transfer over a user-level TCP stack. For Atlas, we use a dual-port Chelsio 100Gbps NIC (T-62100) as FreeBSD 1.10 does not support the netmap [86] driver for BF-2. We have added support for TSO to the NIC driver. We note that it is not an apples-to-apples comparison as the current implementation of the Atlas server deviates from the correct operation of a typical Web server – the current version does not support regular file systems, so it simply returns a random content whose HTTP response headers are also hard-coded into NVMe disks. While implementing a proper custom file system should fix the problem, the current version benefits from avoiding the overhead. Nevertheless, comparing with Atlas would give us the rough idea of how well an IO-TCP-ported server fares over the state-of-the-art CPU-based server. Clients run `wrk` [43] to concurrently request on 1600 persistent connections. For testing Atlas, we reduce the number of concurrent connections to 800 for plaintext transfer as its custom TCP stack becomes unstable at high concurrency.

Comparison with Linux TCP and Disk|Crypt|Net. Figure 7 shows the results for serving 500KB files. lighttpd on IO-TCP achieves 78.1 Gbps with a single CPU core on the host side for plaintext transfer, which demonstrates that a single CPU core is sufficient to handle the control plane operations for all 1600 clients. IO-TCP saturates the full bandwidth of the four NVMe disks, and each NIC reaches 39 Gbps, indicating that the performance scales to the number of NICs. In contrast, Linux TCP does not go beyond 57 Gbps even with 10 CPU cores. Even when we use both CPUs (i.e., 20 cores), we do not see performance improvement (56.2 Gbps). This shows that the memory bandwidth is inefficiently utilized [74] despite the usage of a zero-copy API like `sendfile()`. When `lighttpd` on each CPU runs with a distinct port and serves a disjoint set of files, the performance goes up slightly (59 Gbps) as it benefits from local memory bandwidth. However, the improvement is limited because the content often has to



(a) Plaintext



(b) TLS

Figure 8: Comparison of maximum performance of lighttpd on LinuxTCP, and IO-TCP and Atlas for varying file sizes.

cross the NUMA domain to a NIC and the placement of kernel objects are not NUMA-aware. On the other hand, Atlas performs much better, reaching the same performance of IO-TCP at four CPU cores. When we add more NVMe disks (up to 8) and use two CPUs with both NIC ports, the performance of Atlas peaks at 107 Gbps where the memory bandwidth becomes a bottleneck. The performance of IO-TCP goes up to 95.2 Gbps if IO-TCP serves a random content with raw disk access like in Atlas, but the bottleneck lies in the memory bandwidth of the BF-2 NIC. This shows that efficient use of the host memory bandwidth is highly effective in achieving a very good throughput with only CPU. However, the performance advantage disappears when serving TLS traffic where the memory bandwidth becomes a bottleneck much earlier (discussed in the next paragraph). Figure 8a compares the performances with different file sizes. All performances of Atlas are similar as it avoids calling filesystem APIs. The performance of IO-TCP is comparable to those of Atlas from 300 KB files. IO-TCP outperforms Linux TCP by 38% to 51% and it uses 2x to 10x smaller number of CPU cores to reach the peak performance.

TLS performance. IO-TCP excels at serving TLS traffic. We enable packet encryption with AES-GCM with 256bit keys on the NIC crypto hardware for IO-TCP. For stock lighttpd, we use OpenSSL 1.0.2k [34] with TLSv1.2, and use the same algorithm for symmetric key encryption. Both Atlas and the IO-TCP-ported lighttpd do not implement the TLS handshake, but the overhead for the handshake with stock lighttpd is negligible as we use persistent connections. Figure 7 and Figure 8b show that IO-TCP experiences little performance degradation with TLS due to the dedicated crypto hardware on NIC. In contrast, Atlas achieves only 44.2 Gbps even with

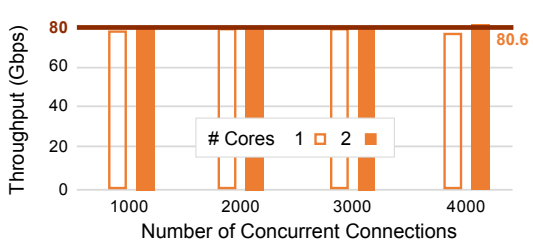
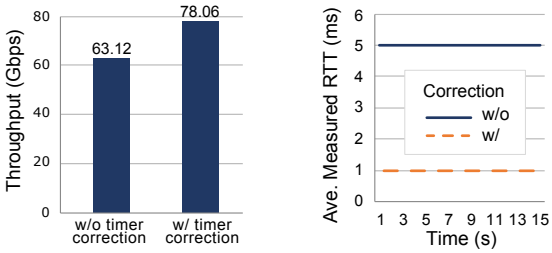


Figure 9: Comparison of TLS performance of IO-TCP over different number of connections using 1 and 2 CPU cores.



(a) Correcting retransmission timers.

(b) Correcting timestamps for more accurate RTT estimates.

Figure 10: Time measurement correction in IO-TCP

10 CPU cores as the main memory bandwidth becomes the bottleneck. The performance goes up to 54.6 Gbps with two CPUs, but overall, the TLS performance is 51% to 56% of the plaintext throughput. The similar trend is seen with Linux TCP - the TLS performance drops by 48% to 63%. We confirm that both Atlas and Linux TCP benefit from the AES-NI instructions of the CPU, but their TLS performances are poor due to content scanning for encryption. We note that the working set size of Atlas exceeds the CPU cache, so their TLS performance is bottlenecked by the memory bandwidth much faster. So, the claim that Disk|Crypt|Net manages the workload within the CPU cache depends on the hardware.

Connection scaling. Figure 9 shows the performance of IO-TCP serving 1000 to 4000 concurrent TLS connections. We observe that the performance is more or less stable over different number of connections. For 4000 connections, IO-TCP loses about 5% of performance with a single CPU core, but it reaches 80 Gbps again with two CPU cores. We check that the plaintext performance exhibits comparable trends. At 4000 connections, each connection would get around 20 Mbps, a comfortable bandwidth to stream 4K videos.

Comparison with user-level TCP stacks. One might be tempted to compare the performance with recent user-level TCP stacks like mTCP [58], IX [49], TAS [62], and F-Stack [13] as they use the CPU cycles efficiently. However, we find that these stacks are not optimized for large-file content delivery as most of them do not implement `sendfile()` nor benefit from TSO. In fact, we measure the performance of TAS, mTCP, and F-Stack on the same platform with all 10 cores, but they showe 8, 21.4, and 36 Gbps, respectively, for 500KB file delivery. Even if they implement a zero-copy API, we doubt that it would substantially outperform Linux TCP because the primary goal of the kernel-bypass networking stacks is to avoid the overhead of frequent system calls and kernel data structures for small-message transactions. However, transferring large messages would rarely impose the system call overhead nor suffer from the overhead of kernel structures. Instead, insufficient memory bandwidth (or CPU cycles) is the main cause for poor performance in large-file content delivery, which kernel-bypass TCP stacks do not help.

TCP fairness. We also evaluate if IO-TCP provides bandwidth fairness among the competing connections. Jain's Fairness Index of IO-TCP ranges from 0.91 to 0.97 for different numbers of concurrent connections. We see a similar range (0.90 to 0.97) with Linux TCP for the same experiments.

5.3 Evaluation of IO-TCP Design Choices

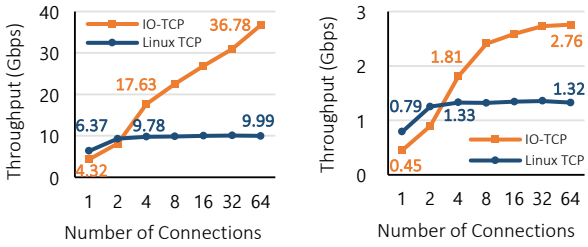
Retransmission timer correction. We evaluate the impact of echo packets that adjust the retransmission timer – when the real data packets are sent out. Figure 10a compares the throughput of lighttpd on IO-TCP with and without timer correction. Without timer correction, the IO-TCP host stack stays at 63.12 Gbps. With timer correction, IO-TCP improves the throughput by 22.6%. This is because IO-TCP without timer correction experiences highly variable RTTs and produces 600x more timeouts than that with timer correction. Such performance drop due to premature timeouts can be more severe in the wide-area-networks (WANs) where the end-to-end RTTs are larger.

RTT measurement correction. We compare the impact of fixing the TCP timestamps on the NIC stack. We measure the average RTT values recorded by the TCP stack every second with 200 concurrent connections. Figure 10b shows that the average RTT is 1 ms with timestamp correction. When we disable the TCP timestamp correction, the average RTT reaches 5 ms, a blowup by a factor of 5. This is because the RTT includes disk access delay that adds a few milliseconds. More accurate latency measurement is critical to trigger the timeout in time when there is a packet loss.

5.4 Overhead Evaluation

The split architecture of IO-TCP may suffer from the communication overhead between host and NIC stacks as well as lower computing capacity of the Arm-based subsystem in the NIC. For this reason, the CPU-only approach on Linux TCP would perform better than IO-TCP for a small number of concurrent connections as CPU can comfortably handle the connections without the overhead. However, this trend will change as the number of connections increases.

Figure 11a shows the throughputs over different numbers of concurrent connections requesting 300KB files. With a single persistent connection, Linux TCP outperforms IO-TCP by over 1.5 times. However, it reaches the peak performance with as few as four connections and the performance stays the same beyond that. In contrast, the throughput of IO-TCP



(a) Throughputs over varying # of connections w/ 300KB files **(b)** Throughputs over varying # of connections w/ 10KB files

Figure 11: Overhead Evaluation.

slowly increases due to the overhead, but it outperforms the Linux TCP at four connections.

The performance trend continues to hold with smaller file sizes. Figure 11b shows the throughput for serving 10KB files over different numbers of connections. Like in the previous case, Linux TCP and IO-TCP reach the peak performance at 4 and 64 connections, respectively, but their performance is much lower than in Figure 11a due to the increased overhead of file operations. Nevertheless, IO-TCP outperforms Linux TCP at four or more concurrent connections.

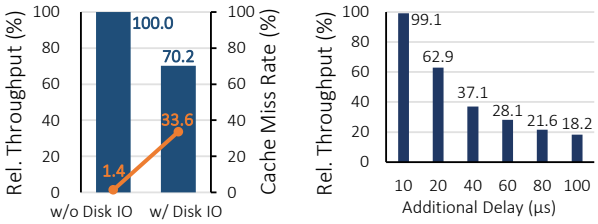
To evaluate the latency overhead, we compare the latency of single-file downloading with IO-TCP vs. Linux TCP. IO-TCP shows 50 to 80us of an extra delay for 10KB files, but the extra overhead goes up to 150 to 200us for 300KB files. This latency overhead at low congestion is inevitable as the host CPU is much faster than the Arm processor in the NIC, but it is negligible for content delivery in wide-area networks.

Memory bandwidth limitation. The performance bottleneck of our current prototype lies in the low memory bandwidth of the BlueField-2 NIC when it accesses more than two NVMe disks. We confirm this by running the same test as in Figure 8a without disk I/O – we observe that the performance reaches 80 Gbps per NIC. Note that disk I/O is the only memory copy for packet payload in our NIC stack as we employ scatter-gather DMA. Nevertheless, we still think our design is promising in the future. First, Arm-based SoC can be designed with much higher memory bandwidth and future SmartNICs would benefit from it. For example, Bluefield-3 [32] is reported to have 3.5x better memory bandwidth (~90 GB/s) than Bluefield-2, and we expect over 100 Gbps per NIC [7] for the same workload as in Figure 8a. Cavium ThunderX2 [8], an Armv8-based SoC server, has 166 GB/s of peak memory bandwidth, even larger than that of our CPU. Second, improving the memory bandwidth of the SmartNIC is more cost-effective as Arm SoCs are less expensive than server-class x86 CPUs [70] and one can easily scale the overall performance by employing multiple NICs. We note that the current SmartNIC price is very high, but it will go down with wider adoption as evidenced in the GPU prices.

[7] 40 Gbps (per NIC) × 90 GB/s (BF-3 BW) / 25.6 GB/s (BF-2 BW) = 141 Gbps

Functions	Instr. per cycle	
	Before	After
APP: Parsing HTTP request	0.82	1.57
APP: Writing response header	0.92	1.89
TCP: Check RTO expire	2.65	3.26
TCP: Process ACK	0.13	1.24
Overall IPC	0.93	1.47

Table 4: Comparison of the IPC of lighttpd and control plane functions before and after data plane offloading with IO-TCP.



(a) Relative throughputs and cache **(b)** Relative throughputs over extra miss rates with and without disk IO. delays by the control plane.

Figure 12: Analysis on the source of performance improvement.

5.5 Source of Performance Improvement

We analyze the source of performance improvement with IO-TCP. First, we observe that the control plane functions in the IO-TCP stack run faster after the separation of the data plane. Table 4 indicates that the instructions per cycle (IPC) of the control path in the IO-TCP stack improves by 58% with the division of labor. Especially, ACK processing benefits the most from the split – note that it is the key function that initiates complex operations frequently such as looking up the TCB in the connection table, determining packet loss/duplicate ACKs, computing the new send window size, etc. After the split, the IPC of this function improves by 9.53x. The performance gain mainly comes from reduced cache/memory contention as we find that the cache miss rate of last-level cache (LLC) improves by 27% with the separation. Then, how come the cache miss rate is reduced? This is because DDIO of NVMe disk IO evicts the data in the CPU cache if both planes run together [96]. To confirm this, we measure the TCB lookup performance with and without NVMe disk reading (`fio`). Figure 12a shows that the cache miss rate of the TCB lookup goes up by a factor of 24 if we co-run disk IO, which in turn reduces the lookup performance by 30%. Finally, we observe that the faster execution of the control plane actually improves the content delivery throughput. To show this, we add redundant code into the ACK processing function so that we can delay its execution by as much as we want. Figure 12b shows that the throughput degrades significantly as the extra delay increases. This implies that the faster control plane reduces the end-to-end RTT and increases the send window size quickly, which ends up improving the overall performance.

6 Related Work

PCIe P2P communications. Enabling PCIe P2P communication between external devices could reduce CPU overhead significantly when transmitting data among them. NVIDIA GPUDirect RDMA [17], GPUDirect Async [45], and AMD DirectGMA [10] techniques, provide a way for other devices to directly access data from GPU by exposing GPU memory directly to PCIe memory space. EXTOLL [79] proposes enabling direct communication between Intel Xeon Phi coprocessors (accelerators) and the NICs, so accelerators can communicate with each other over the network without CPU involvement. Morpheus [91] enables communications between NVMe devices and other PCIe devices. DCS [46] and DCS-ctrl [63] propose a hardware-based framework to enable P2P communication among various types of external PCIe devices. However, all these P2P solutions only consider data communication on hardware, without considering the kernel stacks. As a result, those solutions still suffer from kernel stack overhead when running content delivery applications.

Accelerated networking stacks. There are several existing works that attempt to improve the performance of networking stacks. Some works try to improve the performance of existing kernel stacks. Fastsocket [71] improves the TCP stack performance by achieving table-level connection partition, increasing the connection locality, and eliminating the lock contention. StackMap [95] dedicates network interfaces to applications and offer a zero-copy, low-overhead network interface for applications. Megapipe [57] leverages partitioned, lightweighted sockets, and batches system calls to improve the performance. Another approach is bypassing the heavyweight kernel stack and running the whole stack in the user level. mTCP [58], IX [49], Sandstorm [73], F-Stack [13], and PonyExpress/Snap [75] leverage user-level packet I/O libraries, and leverage multiple CPU cores to process incoming flows simultaneously, in order to increase the processing throughput and reduce latency from kernel calls. ZygOS [85], Shinjuku [60], and Shenango [80] further improve the tail latency of packet processing by improving the load balancing of the tasks among CPU cores. Arrakis [82] and IO-TCP share the same vision of separation of data and control planes, but Arrakis is focused on bypassing the kernel involvement on the data path while IO-TCP harnesses extra processors for work division of the TCP stack operations. TAS [62] builds a TCP fast path as a separated OS service, which targets to improve the performance of RPC calls in the data center. Disk|Crypt|Net [74] builds a scalable video streaming stack, containing a novel kernel-bypass storage stack and an existing kernel-bypass network stack, which achieves lower latency and higher throughput for video streaming applications. However, all these solutions still require huge CPU involvements in packet processing, which still consumes a lot of CPU power on transmitting data among external devices. A recent work called AccelTCP [78] offloads TCP connection management as well as connection relaying into SmartNIC, which relieves a part of packet processing computation from host CPU cores. However, it focuses only on improving the throughput for short-lived connections and L7 proxies.

NIC offload. Traditionally, there have been a spectrum of NIC offload schemes. Stateless schemes like TCP/IP checksum offload, TCP segmentation offload (TSO) and large receive offload (LRO) have become ubiquitous in modern NICs while stateful schemes like TCP Engine Offload (TOE) and Microsoft Chimney Offload [25] have largely been deprecated due to security and maintenance concerns coming from its complexity. IO-TCP is essentially TSO with file reading, and we believe it can be easily implemented into commodity NIC hardware due to its simplicity.

More recently, several works have focused on offloading various tasks to SmartNICs to improve the performance for specific applications. KV-Direct [68] leverages FPGA-based SmartNIC to improve the performance of in-memory key-value stores. Floem [83], ClickNP [69], and UNO [66] leverage SmartNICs to accelerate general packet processing for network applications. Metron [61] offloads packet tagging into the NICs to reduce the latency of packet processing for network functions. iPipe [72] builds a general framework for offloading distributed applications into SmartNICs. Lynx [90] uses the SmartNIC as part of an accelerator-centric architecture where the SmartNIC allows direct networking with the accelerators. Gimbal [76] uses SmartNIC as the traffic orchestrator for disk IO, and realizes efficient multi-tenancy using congestion control algorithms and fair scheduling. LeapIO [70] offloads disk IOs to SmartNIC and provides the seamless address space for cloud tenants while [84] handles NVMe-oF on NIC for remote storage access. However, neither supports TCP operations to clients from NIC. To the best of our knowledge, our IO-TCP is the first work that leverages SmartNICs to accelerate disk and packet I/O for content delivery systems.

7 Conclusion

In this paper, we have presented IO-TCP, a split TCP stack design that offloads I/O operations from CPU for scalable content delivery. IO-TCP provides a new abstraction that leverages SmartNIC processors to perform I/O operations, which significantly relieves the pressure on CPU and its main memory system. Also, our proposal maintains the simplicity in the NIC stack design so that it can be easily implemented with low-powered processors on I/O devices.

Our evaluation shows IO-TCP significantly saves CPU cycles while it delivers the benefit even for small-file transfer when it serves enough connections. Along with the benefit, we also discuss the limitations of the current prototype, and we hope that SmartNIC vendors will consider higher memory bandwidth for the embedded system when designing the next version of their SmartNIC. The source code of IO-TCP is available at https://iotcp.kaist.edu/

Acknowledgements

We appreciate the insightful feedback and suggestions from USENIX NSDI 2022 reviewers on revising our original submission. We thank Ilias Marinos for sharing the source of Disk|Crypt|Net and for helping us with setting up the Atlas server. This work is in part supported by the ICT Research and Development Program of MSIP/IITP, Korea, under [2018-0-00693, Development of an ultra low-latency user-level transfer protocol]. Junzhi Gong and Minlan Yu are supported in part by the NSF CNS-1955422 and CNS-1955487.

References

[1] Akamai braces for huge streaming audiences in 2021. https://www.fiercevideo.com/tech/akamai-braces-for-huge-streaming-audiences-2021. Last Accessed: 2021-09-15.

[2] Akamai Technologies, Inc. https://www.akamai.com/. Last Accessed: 2021-09-15.

[3] Amazon Prime Video. https://www.primevideo.com/. Last Accessed: 2021-09-15.

[4] Apple TV+. https://tv.apple.com/. Last Accessed: 2022-08-23.

[5] As Covid pushes more people online, companies that help the web stay speedy are having a moment. https://www.cnbc.com/2020/12/13/cdn-providers-cloudflare-fastly-benefit-from-covid-web-traffic-boost.html. Last Accessed: 2021-09-15.

[6] Athlon 64 X2. https://en.wikipedia.org/wiki/Athlon_64_X2. Last Accessed: 2021-09-15.

[7] Broadcom Stringray SmartNIC. https://www.broadcom.com/products/ethernet-connectivity/smartnic. Last Accessed: 2021-09-15.

[8] Cavium ThunderX2 Arm-based Processors. https://www.marvell.com/products/server-processors/thunderx2-arm-processors.html. Last Accessed: 2021-09-15.

[9] Cisco Visual Networking Index 2021. https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2021_Forecast_Highlights.pdf. Last Accessed: 2021-09-15.

[10] DirectGMA on AMD's FirePro GPUs. https://www.amd.com/Documents/SDI-techbrief.pdf.

[11] Disney+. https://www.disneyplus.com/. Last Accessed: 2021-09-15.

[12] DPDK. https://www.dpdk.org. Last Accessed: 2021-09-15.

[13] F-Stack | High Performance Network Framework Based on DPDK. https://github.com/F-Stack/f-stack. Last Accessed: 2021-09-15.

[14] Fastly, Inc. https://www.fastly.com/. Last Accessed: 2021-09-15.

[15] fio - Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html. Last Accessed: 2021-09-15.

[16] GitHub - mtcp-stack/mtcp. https://github.com/mtcp-stack/mtcp. Last Accessed: 2021-09-15.

[17] GPUDirect. https://developer.nvidia.com/gpudirect. Last Accessed: 2021-09-15.

[18] HowTo Configure NVMe over Fabrics (NVMe-oF) Target Offload. https://community.mellanox.com/s/article/howto-configure-nvme-over-fabrics--nvme-of--target-offload. Last Accessed: 2021-09-15.

[19] Hulu: Stream TV and Movies Live and Online. https://www.hulu.com/. Last Accessed: 2021-09-15.

[20] IETF RFC 7540. https://tools.ietf.org/html/rfc7540. Last Accessed: 2021-09-15.

[21] Intel Direct Data I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html. Last Accessed: 2021-09-15.

[22] Intel® Xeon® Platinum 9282 Processor. https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html. Last Accessed: 2021-09-15.

[23] Lighttpd - fly light. https://www.lighttpd.net/. Last Accessed: 2021-09-15.

[24] Marvell LiquidIOII Smart NICs. https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics.html. Last Accessed: 2021-09-15.

[25] Microsoft Windows Scalable Networking Initiative. http://download.microsoft.com/download/5/b/5/5b5bec17-ea71-4653-9539-204a672f11cf/scale.doc. Last Accessed: 2021-09-15.

[26] Netflix - Unlimited movies, TV shows, and more. https://www.netflix.com/. Last Accessed: 2021-09-15.

[27] Netronome Agilio LX SmartNICs. https://www.netronome.com/products/agilio-lx/. Last Accessed: 2021-09-15.

[28] NGD Newport NVMe Computational Storage Drive. https://www.ngdsystems.com. Last Accessed: 2021-09-15.

[29] nginx. http://nginx.org/. Last Accessed: 2021-09-15.

[30] NGINX and Netflix Contribute New sendfile(2) to FreeBSD. https://www.nginx.com/blog/nginx-and-netflix-contribute-new-sendfile2-to-freebsd/. Last Accessed: 2021-09-15.

[31] NVIDIA BlueField-2 Programmable SmartNIC. https://www.mellanox.com/files/doc-2020/pb-bluefield-2-smart-nic-eth.pdf. Last Accessed: 2021-09-15.

[32] NVIDIA BlueField-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf. Last Accessed: 2021-09-15.

[33] NVIDIA BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf. Last Accessed: 2021-09-15.

[34] OpenSSL. https://www.openssl.org/. Last Accessed: 2021-09-15.

[35] PCI Express Base Specification. https://pcisig.com/specifications. Last Accessed: 2021-09-15.

[36] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Last Accessed: 2021-09-15.

[37] Poky – Yocto Project. https://www.yoctoproject.org/software-item/poky/. Last Accessed: 2021-09-15.

[38] RFC 7323. https://tools.ietf.org/html/rfc7323. Last Accessed: 2021-09-15.

[39] Sandvine Global Internet Phenomena Report COVID-19 Spotlight. https://www.sandvine.com/phenomena. Last Accessed: 2021-09-15.

[40] SmartSSD Computational Storage Drive. https://samsungsemiconductor-us.com/smartssd/index.html. Last Accessed: 2021-09-15.

[41] StoPool Distributed Storage. https://storpool.com/blog/7-million-iops-and-0-15-ms-latency-for-an-nvme-powered-vdi-cloud. Last Accessed: 2021-09-15.

[42] Storage Performance Developement Kit. https://spdk.io/. Last Accessed: 2021-09-15.

[43] wg/wrk - Modern HTTP benchmarking tool. https://github.com/wg/wrk. Last Accessed: 2021-09-15.

[44] YouTube TV - Watch and DVR Live Sports, Shows & News. https://tv.youtube.com/. Last Accessed: 2021-09-15.

[45] Elena Agostini, Davide Rossetti, and Sreeram Potluri. Offloading communication control logic in GPU accelerated applications. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[46] J. Ahn, D. Kwon, Y. Kim, M. Ajdari, J. Lee, and J. Kim. DCS: A Fast and Scalable Device-centric Server Architecture. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[47] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[48] G. Banga and J.C. Mogul. Scalable Kernel Performance for Internet Servers under Realistic Loads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1998.

[49] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[50] Steve Blank. What the GlobalFoundries' Retreat Really Means. https://spectrum.ieee.org/nanoclast/semiconductors/devices/what-globalfoundries-retreat-really-means, 2018. Last Accessed: 2021-09-15.

[51] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.

[52] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS

User-Space NVM File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[53] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

[54] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[55] Ethernet Alliance. The 2020 Ethernet Roadmap. https://ethernetalliance.org/technology/2020-roadmap/, 2020. Last Accessed: 2021-09-15.

[56] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. A Year in Lockdown: How the Waves of COVID-19 Impact Internet Traffic. *Communications of the ACM (CACM)*, 64(7):101–108, 2021.

[57] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[58] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[59] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[60] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[61] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[62] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019.

[63] D. Kwon, J. Ahn, D. Chae, M. Ajdari, J. Lee, S. Bae, Y. Kim, and J. Kim. DCS-ctrl: A Fast and Flexible Device-Control Mechanism for Device-Centric Server Architecture. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[64] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[65] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, and J. Iyengar. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[66] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. UNO: Unifying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2017.

[67] Jonathan Lemon. KQueue–A Generic and Scalable Event Notification Facility. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2001.

[68] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[69] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[70] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan Ports, Irene Zhang,

Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[71] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. *ACM SIGARCH Computer Architecture News*, 44(2):339–352, 2016.

[72] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019.

[73] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.

[74] Ilias Marinos, Robert NM Watson, Mark Handley, and Randall R Stewart. Disk|Crypt|Net: rethinking the stack for high-performance video streaming. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[75] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[76] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 106–122, 2021.

[77] G. Banga J.C. Mogul and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1999.

[78] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[79] Sarah Neuwirth, Dirk Frey, Mondrian Nuessle, and Ulrich Bruening. Scalable communication architecture for network-attached accelerators. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[80] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[81] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1999.

[82] Simon Peter, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2013.

[83] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: a programming system for NIC-accelerated network applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

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

[85] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[86] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[87] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015.

[88] David Sidler, Zsolt Istvan, and Gustavo Alonso. Low-Latency TCP/IP Stack for Data Center Applications. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2016.

[89] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K. Sitaraman. Footprint Descriptors: Theory and Practice of Cache Provisioning in a Global CDN. In *Proceedings of the International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2017.

[90] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[91] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. *ACM SIGARCH Computer Architecture News*, 44(3):53–65, 2016.

[92] Dave Watson. KTLS: Linux Kernel Transport Layer Security. *Proposal by Facebook Engineer*, 2016.

[93] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*, 2016.

[94] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[95] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[96] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't Forget the I/O When Allocating Your LLC. In *Proceedings of the 48th IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2021.

[97] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable Parallel Flash Firmware for Many-core Architectures. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[98] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019.

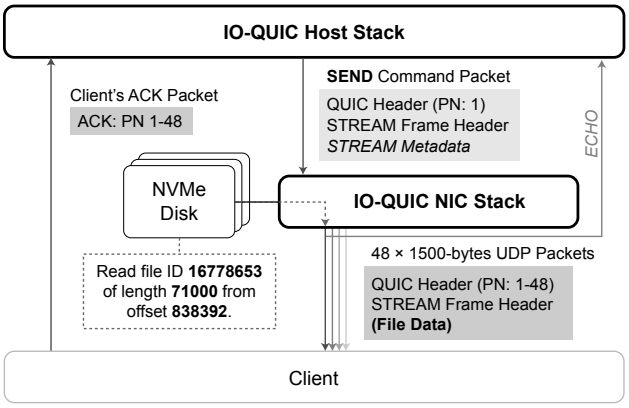


Figure 13: An adaptation of IO-TCP to QUIC.

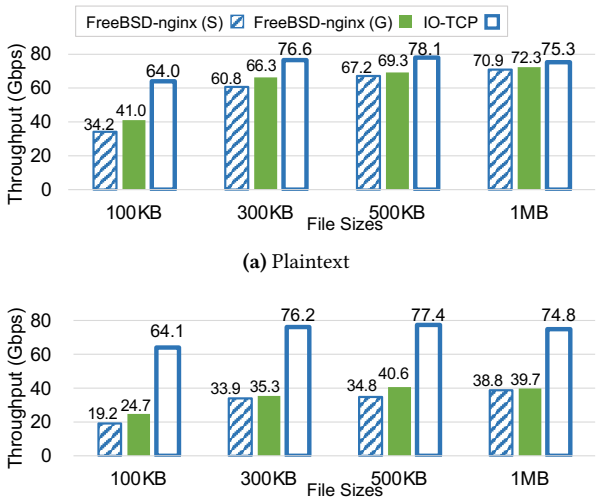


Figure 14: Comparison of maximum performance of nginx that uses asynchronous `sendfile()` on FreeBSD vs. IO-TCP. IO-TCP uses the same number as in Figure 8a and Figure 8b.

Appendix

A Support for the QUIC protocol

Unlike TCP/IP headers, the QUIC header is variable-sized, so the host stack should carefully estimate the header size before offloading. Since the data length can exceed an MTU size, the SmartNIC should perform QUIC packet segmentation with generation of QUIC headers as well as UDP/IP headers. In addition to file IO offloading, UDP packet segmentation with large content on SmartNIC could improve the performance further as QUIC on the Linux UDP stack suffers from frequent context switchings for invoking a system call for each UDP packet. For reliable transfer, the host stack should keep track of STREAM frame packet numbers and data offsets that are sent out. Retransmissions can be handled similarly to IO-TCP as the IO-TCP NIC stack manages the file content buffers independently of the particular transport layer protocol. Likewise, the "ACKD" command can free the file content buffers that are confirmed to be delivered to the QUIC client.

We have finished implementing "IO-QUIC" in the plaintext version, and we will add support for TLS in the future. Unlike the IO-TCP implementation, the host stack of our IO-QUIC implementation uses unmodified Linux kernel as it communicates with the NIC stack with a special UDP packet.

B Performance Comparison with Asynchronous `sendfile()` on FreeBSD

Recent FreeBSD supports asynchronous `sendfile()` that does not block on disk reading [30], so we compare the performance of nginx (v1.20.1) on FreeBSD (v13.0) that utilizes this feature. Since FreeBSD does not allow disabling individual CPU cores, we use all 20 CPU cores for FreeBSD experiments (FreeBSD-nginx(S)). Also, to gauge the impact of higher-capacity CPU, we employ a different server with two Intel Xeon Gold 6142 CPUs @ 2.60 GHz, a 100G Mellanox ConnectX-5 NIC and 32GB memory of DRAM (FreeBSD-nginx(G)). Again, we use all 32 cores in the two CPUs for the experiments.

Figure 14a shows the results over different file sizes. Overall, FreeBSD achieves better performance than Linux for plaintext transfer, but it does not reach the performance reported in [74] (~70 Gbps with 8 cores). This is because the stock FreeBSD version does not support other features in [74] except asynchronous `sendfile()`. In contrast, IO-TCP outperforms all other setups despite that FreeBSD uses 19 to 31 more CPU cores. In terms of the TLS performance, Figure 14b shows that FreeBSD suffers from the same issue as Linux.