Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack

Qizhe Cai
Cornell University

Midhul Vuppalapati
Cornell University

Jaehyun Hwang
Sungkyunkwan University

Christos Kozyrakis
Stanford University

Rachit Agarwal
Cornell University

ABSTRACT

Dedicated, tightly integrated, and static packet processing pipelines in today's most widely deployed network stacks preclude them from fully exploiting capabilities of modern hardware.

We present NetChannel, a disaggregated network stack architecture for μs-scale applications running atop Terabit Ethernet. NetChannel's disaggregated architecture enables independent scaling and scheduling of resources allocated to each layer in the packet processing pipeline. Using an end-to-end NetChannel realization within the Linux network stack, we demonstrate that NetChannel enables new operating points—(1) enabling a single application thread to saturate multi-hundred gigabit access link bandwidth; (2) enabling near-linear scalability for small message processing with number of cores, independent of number of application threads; and, (3) enabling isolation of latency-sensitive applications, allowing them to maintain μs-scale tail latency even when competing with throughput-bound applications operating at near-line rate.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; • **Computer systems organization**; • **Networks → Network design principles**;

KEYWORDS

Operating system; Network stack; Terabit Ethernet

ACM Reference Format:
Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3544216.3544230

1 INTRODUCTION

In discussions about future host network stacks, there is widespread agreement that, despite its great success, today's Linux network stack is seriously deficient along one or more dimensions. Some of the most frequently cited flaws include its inefficient packet processing pipeline [7, 11, 25, 34, 51], its inability to isolate latency-sensitive and throughput-bound applications [35, 54, 61], its rigid and complex implementation [6], its inefficient transport protocols [10, 23, 55], to name a few. These critiques have led to many interesting (and exciting!) debates on various design aspects of the Linux network stack: interface (*e.g.*, streaming versus RPC [24, 36, 45, 61]), semantics (*e.g.*, synchronous versus asynchronous I/O [40, 44, 45]), and placement (*e.g.*, in-kernel versus userspace versus hardware [6, 51]).

This paper demonstrates that many deficiencies of the Linux network stack are *not* rooted in its interface, semantics and/or placement, but rather in its core architecture[1]. In particular, since the very first incarnation, the Linux network stack has offered applications the same "pipe" abstraction designed around essentially the same rigid architecture:

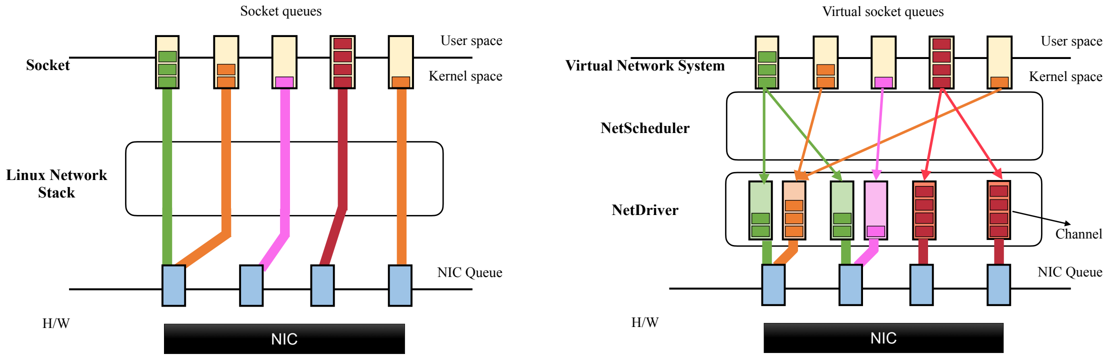
- **Dedicated pipes:** each application and/or thread submits data to one end of a dedicated pipe (sender-side socket) and the network stack attempts to deliver the data to the other end of that dedicated pipe (receiver-side socket);
- **Tightly-integrated packet processing pipeline:** each pipe is assigned its own socket, has its own independent transport layer operations (congestion control, flow control, etc.), and is operated upon by the network subsystem completely independently of other coexisting pipes;
- **Static pipes:** the entire packet processing pipeline (buffers, protocol processing, host resource provisioning, etc.) is determined at the time of pipe creation, and remains unchanged during the pipe lifetime, again, independent of other pipes and dynamic resources availability at the host.

Such dedicated, tightly-integrated and static pipelines were well-suited for the Internet and early-generation datacenter networks—since performance bottlenecks were primarily in the network core, careful allocation of host resources (compute, caches, NIC queues, etc.) among coexisting pipes was unnecessary. However, rapid increase in link bandwidths, coupled with relatively stagnant technology trends for other host resources, has now pushed bottlenecks to hosts [6, 11, 25, 30, 34, 51, 55]. For this new regime, our measurements in §2 show that dedicated, tightly-integrated and static pipelines are now limiting today's network stacks from fully exploiting capabilities of modern hardware that supports μs-scale latency and hundred(s) of gigabits of bandwidth.

[1] One exception is per-core performance, which indeed depends on its interface, semantics and placement. This paper is not about per-core performance of network stacks—our architecture is agnostic to the interface, semantics, and placement of network stacks.

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544230



(a) Linux network stack

(b) NetChannel design

Figure 1: NetChannel architecture overview. (a) The Linux network stack architecture uses dedicated, tightly-integrated, and static packet processing pipelines. (b) NetChannel disaggregates the packet processing pipeline into three loosely-coupled layers: applications interact with the network stack using a Virtual Network System (VNS) layer that enables data movement between application buffers and kernel buffers while maintaining correctness of interface semantics; NetDriver abstracts away the network and remote servers as a multi-queue device using a `channel` abstraction, and performs dynamic resource scheduling for individual `channels`; NetScheduler performs fine-grained multiplexing and demultiplexing (as well as scheduling) of data between VNS buffers and individual NetDriver `channels`. More discussion in §3.

Experimenting with new ideas has also become more challenging: performance patches have made the tightly-integrated pipelines so firmly entrenched within the stack that it is frustratingly hard, if not impossible, to incorporate new protocols and mechanisms. Unsurprisingly, existing network stacks are already at the brink of a breakdown and the emergence of Terabit Ethernet will inevitably require rearchitecting the network stack. Laying the intellectual foundation for such a rearchitecture is the goal of this paper.

The NetChannel architecture. NetChannel disaggregates the tightly-integrated packet processing pipeline in today's network stack into three loosely-coupled layers (Figure 1)[2].

Applications interact with a Virtual Network System (VNS) layer that offers standardized interfaces, *e.g.*, system calls for streaming and RPC traffic. Internally, VNS enables data transfer between application buffers and kernel buffers, while ensuring correctness for the interface semantics (*e.g.*, in-order delivery for the streaming interface). The core of NetChannel is a NetDriver layer that abstracts away the network and remote servers as a multi-queue device using a `channel` abstraction. In particular, the NetDriver layer decouples packet processing from individual application buffers and cores: data read/written by an application on one core can be mapped to one or more `channels` *without breaking application semantics*. Each `channel` implements protocol-specific functionalities (congestion and flow control, for example) independently, can be dynamically mapped to one of the underlying hardware queues, and the number of `channels` between any pair of servers can be scaled independent of number of applications running on these servers and the number of cores used by individual applications. Between the VNS and NetDriver layers is a NetScheduler layer that performs fine-grained multiplexing and demultiplexing (as well as scheduling) of data from individual cores/applications to individual `channels` using information about individual core utilization, application buffer occupancy and `channel` buffer occupancy.

NetChannel benefits. The primary benefit of NetChannel is to enable new operating points for existing network stacks without any modification in existing protocol implementations (TCP, DCTCP, BBR, etc.). These new operating points are a direct result of NetChannel's disaggregated architecture: it not only allows independent scaling of each layer (that is, resources allocated to each layer), but also flexible multiplexing and demultiplexing of data to multiple `channels`. We provide three examples. First, for short messages where throughput is bottlenecked by network layer processing overheads [25, 34], NetChannel allows unmodified (even single-threaded) applications to scale throughput near-linearly with number of cores by dynamically scaling cores dedicated to network layer processing. Second, in the extreme case of a single application thread, NetChannel can saturate multi-hundred gigabit links by transparently scaling number of cores for packet processing on an on-demand basis; in contrast, the Linux network stack forces application designers to write multi-threaded code to achieve throughput higher than tens of gigabits per second [11]. As a third new operating point, we show that fine-grained multiplexing and demultiplexing of packets between individual cores/applications and individual `channels` enabled by NetChannel, combined with a simple NetScheduler, allows isolation of latency-sensitive applications from throughput-bound applications: NetChannel enables latency-sensitive applications to achieve μs-scale tail latency (as much as 17.5× better than the Linux network stack), while allowing bandwidth-intensive applications to use the remaining bandwidth near-perfectly.

NetChannel also has several secondary benefits that relate to the extensibility of network stacks. For instance, NetChannel alleviates the painful process of applications developers manually tuning their code for networking performance (*e.g.*, number of threads, connections, sockets, etc.) in increasingly common case of multi-tenant

[2] In the hindsight, NetChannel is remarkably similar to the Linux storage stack architecture [8, 29]. This similarity is not coincidental—for storage workloads, bottlenecks have always been at the host, and the "right" architecture has evolved over years to both perform fine-grained resource allocation across applications, and to make it easy to incorporate new storage technologies.

deployments[3]. NetChannel also simplifies experimentation with new designs (protocols and/or schedulers) without breaking legacy hosts—implementation of a new transport protocol (*e.g.*, dcPIM [10], pHost [23] or Homa [53]) in NetChannel is equivalent to writing a new "device driver" that realizes only transport layer functionalities without worrying about functionalities in other layers of the stack like data copy, isolation between latency-sensitive and throughput-bound applications, CPU scheduling, load balancing, etc. Thus, similar to storage stacks (that have simplified evolution of new hardware and protocols via device drivers, and have simplified writing applications with different performance objectives via multiple coexisting block layer schedulers, etc.), NetChannel would hopefully lead to a broader and ever-evolving ecosystem of network stack designs. To that end, we have open-sourced NetChannel for our community; the implementation of NetChannel is available at `https://github.com/Terabit-Ethernet/NetChannel`.

What this paper is *not* about. Going back to our starting point, there have been a lot of interesting and exciting recent debates on various design aspects of network stacks including their interface, semantics and placement. These are important discussions, but are tangential to our goals.

First, NetChannel architecture is complementary to recent efforts in improving per-core (or, per-connection) performance of the Linux kernel network stack (*e.g.*, zero-copy mechanisms [15, 16], and the new `io_uring` interface [40])—we will demonstrate, in §5, that applications using the `io_uring` interface can also benefit from the NetChannel architecture. Given that single-core CPU speeds have long been saturated, simple calculations show that saturating multi-hundred gigabit access link bandwidths would necessitate using multiple cores. NetChannel architecture thus focuses on enabling new design points to enable applications to share and exploit all host resources (*e.g.*, multiple cores, NIC queues, multi-hundred gigabit bandwidth, etc.).

Second, NetChannel architecture is independent of where the network stack is placed—in-kernel, userspace or hardware; one could very well implement NetChannel design on top of a microkernel-style userspace stack [21, 38, 51]. We choose the Linux kernel simply because of its maturity, stability, and widespread deployment; we leave it to future work to explore integration of NetChannel with userspace and hardware network stacks.

2 MOTIVATION

In this section, we demonstrate that the dedicated, tightly-integrated, and static packet processing pipelines in today's host network stacks lead to deficiencies along multiple dimensions. We perform measurements for the Linux network stack, with various transport designs including TCP and its multi-path extension MPTCP [18], various system interfaces (standard `read/write` interface and `io_uring`), various packet processing optimization techniques (*e.g.*, packet coalescing and packet steering), and different isolation mechanisms. We start by describing our measurement setup (§2.1), and then highlight several limitations of today's Linux network stack (§2.2) using these measurements. Our key findings are:

- *Static and dedicated* packet processing pipelines preclude applications from fully utilizing host CPU resources. Even with all optimizations enabled, a single TCP long flow fails to saturate a 100Gbps link (achieving a maximum of ~60Gbps) even when ample CPU cores are available. We find that, at ~60Gbps, one of the cores at the receiver side becomes the bottleneck (specifically, the core where the application runs), and today's network stacks provide no way to dynamically scale the compute resources allocated to the packet processing pipeline during runtime (even if there are other free CPU cores available). Multipath extensions are no different—while MPTCP enables a single TCP connection to utilize multiple network paths, processing at the host is still bound to a single CPU core leaving the bottleneck unchanged.
- *Static and dedicated* packet processing pipelines also preclude Linux from dynamically scaling the number of connections when network layer processing becomes the bottleneck for short messages (*e.g.*, RPCs). MPTCP also fails to achieve scalability for network layer processing of short messages due to the same reason as above.
- *Tightly-integrated* nature of packet processing pipelines lead to poor performance isolation when Latency-sensitive (L-apps) and Throughput-bound (T-apps) applications are co-located. When L-apps and T-apps share a core, today's network stacks provide no mechanism to steer L-app packet processing and T-app packet processing to different cores—this results in high tail latency for L-apps due to head-of-line blocking; we observe as much as 37× increase in L-apps tail latency during such contention.

2.1 Measurement Setup

We set up a testbed using two servers directly connected with a 100Gbps link so as to push bottlenecks to the host network stack. Each server has 4 NUMA nodes with 8 CPU cores per-NUMA node. Direct Cache Access (Intel's Data Direct I/O (DDIO) [30]) is enabled and configured to use the maximum possible number of L3 cache ways. We use Linux kernel v5.6 for TCP and MPTCP kernel v0.95 for MPTCP[4]. To emulate multi-pathing of the MPTCP kernel over a single 100Gbps link, we increase the number of subflows manually.

To understand performance bottlenecks for long flows and short messages, we use a long-lived TCP connection for transmitting stream data and 4KB RPCs respectively, avoiding extra overheads of creating/destroying connections. When measuring the performance interference between T-apps and L-apps, for T-apps, we generate long flow traffic similar to Iperf [19], and for L-apps, we use a ping-pong style RPC workload (message size for both request/response is 4KB).

We measure total throughput of long flows and short messages; to better understand performance bottlenecks, we also perform CPU profiling and classify kernel functions into different categories as

[3] Libraries and/or schedulers outside the network stack (*e.g.*, gRPC) may be able to offer this benefit to some extent, but in many multi-tenant deployments, they may not have the global visibility of all applications, system-level metrics like CPU utilization, and in particular, network-layer metrics like congestion information to effectively offer such a benefit. Nevertheless, our point here is not that similar benefits cannot be achieved using other systems, but rather that one of the most widely used network stacks, Linux, is limited by its architecture in offering such a benefit.

[4] The full implementation of MPTCP has been maintained separately from the upstream kernel using different version numbers. As of now, the most recent version of MPTCP kernel is based on Linux kernel 4.19 [18]. Efforts to push MPTCP into the upstream kernel are still in progress [47].

Mechanism	Description
TSO (Tx)	TCP Segmentation Offload. Offloads the segmentation of "big" packets (up to 64KB) into frames to the NIC.
GRO (Rx)	Generic Receive Offload. Aggregates MTU-sized frames into a "big" packet (up to 64KB) before passing them to the TCP/IP layer.
Jumbo Frames (Tx/Rx)	Using larger MTU size (9000B)
aRFS (Rx)	accelerated Receive Flow Steering. Steers received frames to the core that the application is running on.
DCA (Rx)	Direct Cache Access. Allows NICs to DMA receiving frames directly to the processor's L3 cache.

Table 1: Packet processing optimization techniques used in many modern network stacks.

in prior work [11]. We also measure throughput-per-core of T-apps and P99.9 tail latency of L-apps for co-located T-apps and L-apps.

2.2 Limitations of Existing Kernel Stack

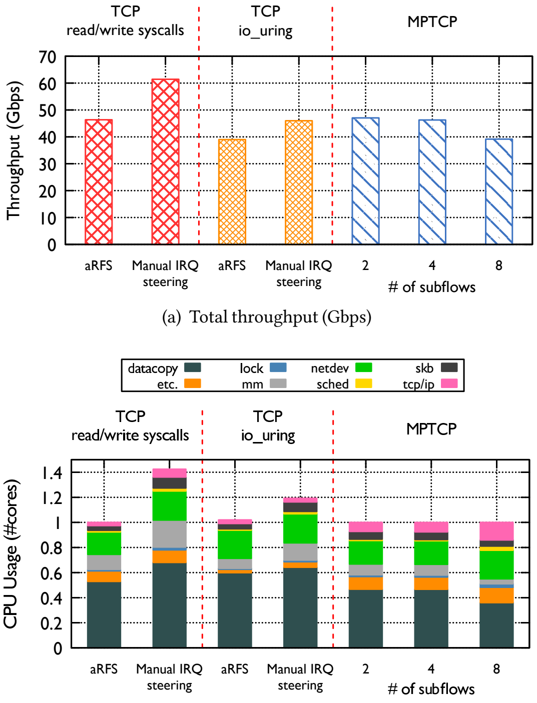
We use three measurement scenarios to showcase limitations of today's Linux network stack.

(1) Static and dedicated pipeline ⇒ lack of scalability for long flows. We run an application that transmits a stream of data through a single TCP socket using standard read/write system calls. Fig. 2(a) shows the network stack is unable to saturate the 100Gbps link for this application, even after enabling various prevalent offload and aggregation-based optimizations like Jumbo Frames, TSO, GRO, aRFS (see Table 1), new upcoming interfaces like `io_uring`, and multiple subflows for MPTCP. Jumbo Frames and TSO/GRO help reduce the per-packet processing overheads since they allow the processing pipeline to operate on larger size packet buffers (or `skbs`). aRFS allows NICs to steer received frames to the application core; along with DCA, it generally improves throughput by performing data copy directly from the L3 cache when applications are running on the cores in the same NUMA node as the NIC. For a more detailed discussion of these optimizations and their impact, refer to [11].

To dig deeper, we perform CPU profiling as shown in Fig. 2(b); our results suggest that, despite DCA being enabled, the core bottleneck is data copy from kernel to userspace *at the receiver-side* consistent with observations in recent work [11]. Data copy is performed on the application core that executes the `recv()` system calls. Additionally, with aRFS enabled, interrupts (IRQs) are steered to the application core, and hence other network layer processing such as TCP/IP, Net-device subsystem (netdev), etc., also happens on the application core.

We find that by disabling aRFS and manually steering interrupts to a different core on the same NUMA node as the application, the network stack achieves slightly higher throughput of ~60Gbps (Fig. 2(a) second bar). Part of the processing (TCP/IP, netdev, etc.) is now offloaded to a different CPU core, hence freeing up more CPU cycles for data copy on the application core.

Even with Linux's recent `io_uring` [40] feature, with aRFS and with manual NUMA-local IRQ steering (bars 3 and 4 in Fig. 2(a) respectively), we find that the total throughput does not improve—in particular, data copy remains the dominant overhead (bars 3 and 4 in Fig. 2(b))[5]. In fact, we observe a slight degradation in total throughput with `io_uring` because it dispatches some of the socket receive calls to a separate kernel thread which contends with the application thread for the common socket lock.



(a) Total throughput (Gbps)

(b) Receiver-side CPU breakdown

Figure 2: Static pipeline of the Linux network stack (using both TCP and MPTCP) fails to saturate 100Gbps access link bandwidths since it is unable to scale compute resources allocated to packet processing pipelines. More discussion in §2.2.

With MPTCP, we observe that using aRFS gives the best possible throughput. Independent of the number of subflows, all the processing happens on the core where the application runs. The total throughput is reduced when the number of subflows increases due to the increase in the amount of network layer processing. Without aRFS, interrupts for subflows get mapped to arbitrary CPU cores potentially on different NUMA nodes resulting in poor throughput[6].

Overall, independent of the configuration used, the packet processing pipeline of today's network stack is static (both data copy and network layer processing are bound to a single core). As a result, it is unable to dynamically scale resources allocated to a packet processing pipeline to utilize the full network link bandwidth, despite the availability of idle CPU cores. Even if data copy were to be eliminated (via zero-copy mechanisms [15, 16]), simple calculations show that the network stack will not be able to saturate emerging multi-hundred gigabit links using a single

[5] While `io_uring` enables "zero-copy" of the metadata associated with socket operations, payload data is copied as usual. While there is ongoing work on exploiting `io_uring` for zero-copy send [17], we are not aware of any work on zero-copy receive (which is usually the bottleneck [11]).

[6] We were not able to manually steer subflow IRQs to different cores in the same NUMA using 4-tuples, because the kernel determines the source ports of subflows at runtime.

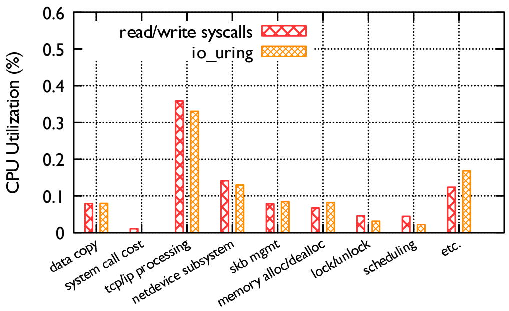


Figure 3: In today's Linux network stack, sender-side network layer processing (including protocol and netdevice subsystem processing) overheads are the bottleneck for short message processing. More discussion in §2.2.

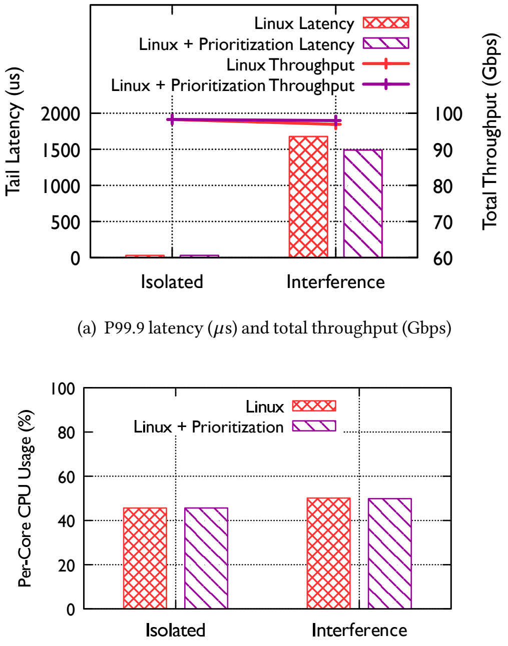
core, as the packet processing pipeline is still bound to one core. The requirement of multi-core processing is, thus, inevitable.

(2) Static and dedicated pipeline ⇒ lack of scalability for short message processing. We run a client application which sends 4KB RPC requests to a server. The sustained throughput using a single socket is roughly ~ 9.88Gbps. We find that the sender-side is the bottleneck. The sender-side CPU breakdown in Figure 3, shows that TCP/IP processing and netdevice subsystem processing are the dominant overheads, accounting for almost a half of the total CPU cycles used. Once again, all of this processing is bound to a single CPU core and is unable to dynamically scale even if additional CPU cores are available—a result of the static nature of today's network stack.

We tried this experiment using `io_uring` as well, but observed no improvement in throughput (achieving a maximum of 8.5Gbps; there is a small degradation in throughput [27, 64])). This is unsurprising given that system call cost and context-switch overheads (included in scheduling)—the main overheads that `io_uring` is supposed to minimize—account for a very tiny fraction of CPU cycles in this scenario, as seen in Figure 3. MPTCP cannot help in this case for two reasons. First, since sender-side network layer processing still happens on the application core, running multiple subflows on a single core does not help. Second, MPTCP cannot dynamically scale the number of subflows at runtime based on the CPU load.

While the application could overcome the network layer processing bottleneck by sending data over multiple sockets from different threads, it is difficult for application developers to estimate how many sockets they will need, especially in multi-tenant and virtualized deployments [11]. Moreover, the application and userspace libraries linked to the application (e.g., gRPC) have limited information about congestion in the network and utilization of CPU cores, making it hard to make informed packet scheduling decisions across sockets. An ideal networking stack should dynamically and transparently allocate new connections on idle cores and multiplex packets to different connections so that the application can achieve higher throughput without manually managing the number of connections. Further, this should happen only when the throughput is limited by CPU, not by congestion control, to maintain protocol-side properties such as TCP-friendliness.

(3) Tightly-integrated pipeline ⇒ lack of performance isolation. To understand performance interference between L-apps and T-apps when they are co-located, we run 1 L-app and 8 T-apps on the same NUMA node (number of applications > number of cores). We do not pin the applications to specific cores, thus allowing the CPU scheduler to dynamically move applications across all of the cores within the NUMA node. We enable all optimizations including aRFS.



(a) P99.9 latency (μs) and total throughput (Gbps)

(b) Per-Core CPU Usage

Figure 4: Tightly integrated pipeline of the Linux network stack fails to provide μs-scale isolation for L-apps when they are co-located with T-apps. L-apps can suffer from extremely high tail latency. More discussion in §2.2.

Fig. 4(a) (Linux) shows the tail latency (99.9th percentile) of the L-app and the overall throughput achieved, when the applications are run in isolation (Isolated) versus when they are co-located (Interference). In the latter case, the L-app experiences a 37× inflation in tail latency, relative to when it is run in isolation. This dramatic inflation in tail latency is due to the tight integration of network layer processing with the application cores. Since there are more applications than cores, it is inevitable that at certain points in time, the L-app will share a CPU core with one or more T-apps. When this happens the kernel runs the corresponding network layer processing for both applications on the same core, hence causing interference—the network layer processing for the L-app can get blocked behind network layer processing for the T-app, causing inflation in tail latency.

Using prioritization techniques to prioritize L-app traffic does not solve the problem. Prioritization can be performed at two layers today—(1) transmission of L-app packets can be prioritized at the qdisc [39] layer on the sender-side using the pfifo_fast scheduling policy [43]; and, (2) the L-app process can be prioritized at the CPU scheduler (on both sender and receiver-side) by setting the niceness value of L-app's processes to −20 (the highest CPU scheduling priority in Linux's CFS scheduler [46]). Despite applying both of these prioritization techniques, as shown in Fig. 4(a) (Linux +

Prioritization), we observe no noticeable improvement in the tail latency of the L-app. qdisc prioritization does not help because there is not much queueing at the qdisc layer to begin with. This is because of the TCP Small Queue (TSQ) [14] feature which limits the number of in-flight bytes at the qdisc layer in order to minimize bufferbloat. CPU scheduling prioritization does not help for two reasons. First, a majority of the network layer processing happens in IRQ threads (Rx packet processing at the receiver-side, and TSQ processing at the sender-side) whose scheduling is not impacted by the priority of application threads. Second, even if there were a mechanism to prioritize IRQ processing, it would not fully solve the problem, as IRQ processing is non-preemptive in nature; thus, L-apps could still get blocked by T-apps.

Since prioritization mechanisms are not effective, the only way to achieve isolation is by separating the network layer processing for L-apps and T-apps onto separate cores. However, due to tight-integration of the processing pipeline with application cores, today's network stack is unable to do so—indeed, as shown in Fig. 4(b), the average per-core CPU utilization is only 40%–50% in the above experiments. An ideal network stack should allow separation and isolation of network layer processing for L-apps and T-apps, even if the two classes of applications are sharing the same CPU core.

3 NetChannel DESIGN

As discussed earlier, the dedicated, tightly-integrated, and static packet processing pipelines in today's Linux network stack leads to several limitations. To resolve this, NetChannel disaggregates the pipeline by rearchitecting the network stack into three layers: (1) Virtual Network System (VNS) layer, (2) NetDriver layer, and (3) NetScheduler layer.

The VNS layer (discussed in §3.1) provides interfaces to applications (e.g., socket, RPC) while ensuring correctness of the interface semantics. These interfaces are "virtual", since unlike in today's Linux network stack, they only buffer data from/to applications and are disaggregated from the rest of the packet processing pipeline. At the bottom, the NetDriver layer (discussed in §3.2) abstracts the network as a multi-queue "device" through a generic `channel` abstraction exposed to the upper layer. Decoupling application interfaces (in the VNS layer) from `channels` (in the NetDriver layer), enables flexible and fine-grained multiplexing/demultiplexing and scheduling of data between the two. This multiplexing/demultiplexing is controlled by the NetScheduler layer (discussed in §3.3) which enables pluggable schedulers that can be designed to achieve various objectives including dynamic scaling of the packet processing pipeline across CPU cores, and performance-isolation of L-apps from T-apps.

3.1 Virtual Network System (VNS) Layer

The VNS layer offers application interfaces while maintaining the necessary semantics of each interface. In order to support unmodified applications, NetChannel supports the standard POSIX socket interface through *virtual sockets.* From the application's perspective, these virtual sockets have the exact same semantics as normal sockets, i.e., reliable in-order delivery between endpoints. As usual, applications can interact with these sockets using standard system calls (e.g., connect, send, recv, epoll) or even using `io_uring` [40]. While VNS also supports other interfaces such as RPCs, we focus our discussion here on the socket interface since it has the strongest requirements in terms of semantics.

Ensuring correctness of interface semantics. Each virtual socket internally maintains a pair of Tx and Rx buffers. When applications send/receive data to/from virtual sockets, data is copied from/to userspace to/from the virtual socket Tx/Rx buffers. Data in the virtual socket Tx buffer is forwarded to the NetDriver layer for transmission, while data received over the network is forwarded from the NetDriver layer to the virtual socket Rx buffer. While we can rely on the network transport (underlying the `channels` in NetDriver layer) to guarantee reliable delivery, VNS needs to do some book-keeping to ensure in-order delivery of bytes between a pair of virtual sockets. This is because, as we will discuss in §3.2, data from a virtual socket can be multiplexed/demultiplexed to/from more than one underlying `channel` in the NetDriver layer, in which case, it is possible for data to arrive in the virtual socket Rx buffer out-of-order. To that end, on the sender side, VNS embeds a sequence number (§4) in each data packet representing its order within the virtual socket stream. On the receiver-side, it can then use these sequence numbers to ensure data is delivered in-order—packets are buffered in the virtual socket Rx buffer until they are next in-sequence.

Decoupling data copy from application threads. VNS also maintains per-core worker threads for data copy between userspace and the kernel (for interfaces that require it). Data copy operations for virtual sockets can be divided into smaller parts (each with a target buffer address and length) and distributed across worker threads of multiple cores. This enables utilizing multiple cores to scale data copy processing for throughput-bound applications with long flows.

3.2 NetDriver Layer

The NetDriver layer abstracts away the network and remote servers as a multi-queue device and exposes `channels` which are analogous to queues of this device. In this subsection, we discuss the `channel` abstraction, NetDriver mechanisms for decoupling network layer processing from sockets, NetDriver mechanisms to enable ease of integration of new network transport designs, and other details relevant to managing buffer overflow and avoiding head-of-line blocking.

Channel abstraction. In NetDriver, each `channel` consists of a pair of Tx/Rx queues, and an instance of an independent network layer processing pipeline of an underlying network transport that implements functionalities such as reliable delivery, flow control, and congestion control (for example, in the case of TCP, a `channel` would map to a single underlying TCP connection). The `channel` API (Table 2) is simple and generic enabling it to encapsulate a wide range of transports. In particular, it can support both connection-oriented stream-based transports (e.g., TCP, DCTCP [5]), and connection-less message-oriented transports (pHost [23], Homa [53] or dcPIM [10]). For example, in the case of the former, upon a `channel` API create call, a connection can be created. Subsequently, arbitrary chunks of data in the stream can be transmitted through `channel` enqueue calls. In the case of the latter,

API method	Arguments	Description
create()	metadata	Create a new channel instance
destroy()	channel	Destroy a channel instance
enqueue()	channel, List<packet buffer>, metadata	Enqueue data for transmission through a given channel. Given as a list of packet buffers along with metadata.
dequeue()	channel, count	Dequeue upto count bytes of data received through a given channel. Returns a list of packet buffers along with metadata.

Table 2: Summary of core channel API calls that need to be implemented by network drivers. For a more exhaustive list, see [12]. The metadata argument in both create and enqueue calls is opaque from the perspective of the channel interface, and can be used to encode transport-specific information such as host address and port number.

no connection will be created during `channel` creation, and individual messages can directly be transmitted through the `channel` enqueue call (passing destination information in the metadata).

Decoupled network layer processing. `channels` in NetDriver are decoupled from instances of virtual interfaces (e.g., virtual sockets) in the VNS layer. This architectural choice enables *decoupling network layer processing from individual sockets and cores* that applications use—NetDriver allows creating one or more `channels` between a given pair of servers, independent of the number of applications running on these servers, and the number of sockets/cores used by these applications. Further, it allows flexible fine-grained multiplexing and demultiplexing of data from/to virtual socket to/from `channels`. This enables several interesting operating points. For instance, if one `channel` has high CPU load, the subsequent data from a virtual socket can be dynamically scheduled to different `channels`. Such dynamic multiplexing can enable utilizing multiple cores to scale network layer processing, while also enabling utilization of multiple network paths similar to MPTCP.

Integrating new transport designs. Given that NetDriver is an abstraction of a multi-queue device, integrating a network transport is now equivalent to writing a new device driver. This essentially makes it easier to integrate and experiment with new protocols. Protocol developers do not need to worry about implementing cumbersome APIs related to socket interfaces (*e.g.*, epoll) and things like data copy processing, instead focusing only on implementing their own network protocol logic plus simple APIs of the `channel` abstraction as shown in Table 2.

Piggybacking on transport-level flow control. To avoid Rx buffer overflow of virtual sockets, NetDriver naturally piggybacks on the flow control of the underlying transport protocol(s) through backpressure, without having to introduce a new flow control protocol. When a virtual socket's Rx buffer becomes full, VNS stops receiving data from `channels`, leading to accumulation of data in the `channel`'s Rx buffer, eventually triggering the flow control mechanism of the underlying transport. VNS resumes receiving data when the virtual socket's Rx buffer is available again as the application reads the data.

Alleviating HoL blocking. Since it is possible for a single `channel` to be shared by multiple virtual sockets, we need to handle corner-case scenarios where one virtual socket causes head-of-line (HoL) blocking for the others. This can happen if an application does not read data from a virtual socket for an extended period of time (e.g., because it is malfunctioning, or busy with other work). This is problematic because other virtual sockets may lose an opportunity to use that `channel` since its Rx buffer is full (potentially causing NetScheduler to unnecessarily create a new `channel`, which will be explained in §3.3).

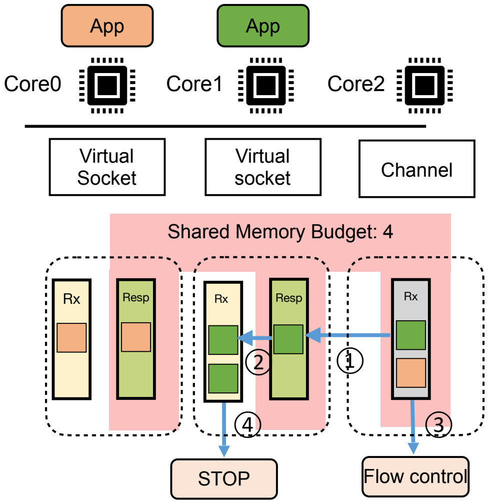


Figure 5: Alleviating HoL blocking (1) Packets received by a `channel` are pushed into their corresponding virtual socket response queue, preventing HoL blocking at the `channel` Rx queue. (2) If the Rx buffer of the virtual socket has free space, the packets will be pushed from the response queue to the Rx buffer. Otherwise, packets are buffered in the response queue. (3) Response queues of virtual sockets and `channels` share the memory budget. (4) When the virtual socket Rx buffer becomes full and the timer expires, the virtual socket sends a "stop" message to the peer virtual socket. More discussion in §3.2.

To address this issue, NetChannel maintains a response queue per virtual socket (Figure 5)—the high-level idea is to stop data transmission only for the virtual socket whose Rx buffer is full while keeping the `channel`'s Rx buffer unblocked for other virtual socket traffic. Maintaining these response queues incurs negligible memory overhead as each is simply a linked list of pointers. Whenever a new packet arrives in the `channel`'s Rx buffer, it is forwarded to the response queue of the virtual socket that the packet belongs to immediately. This resolves the HoL blocking issue at the `channel`'s Rx buffer. Next, the packet is pulled from the response queue to the virtual socket Rx buffer. We allow the response queues to use the memory budget of the `channel`'s buffer, so that the `channel`'s underlying flow control is triggered when the number of entries in the response queue increases. If the virtual socket Rx buffer becomes full, the worker thread sets a timer, and sends a `stop` control message to the peer virtual socket to avoid further data transmission when the timer expires. When the virtual socket receives a corresponding `ACK` for the `stop` message, it explicitly moves all packets in the response queue to the virtual socket Rx buffer, so the `channel`'s buffer memory is freed (the virtual socket buffer size is temporarily increased in this corner-case scenario). It sends a `resume` message and stops the timer when the virtual socket again has free space in the Rx buffer.

3.3 NetScheduler Layer

NetScheduler performs three main tasks: (1) fine-grained multiplexing and scheduling of application data to `channels` to achieve various performance objectives, (2) scaling number of `channels` between a pair of hosts dynamically, and (3) scheduling of data copy requests across per-core data copy worker threads at fine-grained timescales. Given its location in the kernel, NetScheduler has visibility into various kinds of metrics such as the occupancy level of queues, number of virtual sockets, CPU core utilization, application priority, and so forth. We note that our goal is not to design scheduling policies, but rather, to provide mechanisms to enable different policies. One can implement any scheduling policy within our NetScheduler framework. Here, we discuss some simple example policies that enable new operating points. The specific policies used in our current implementation are discussed in §4.

Dynamic scheduling of application data to channels. At any given point, there can be one or more active `channels` between a pair of hosts. Upon receiving data from an application, NetScheduler determines the target `channel` to send the data on at per-`skb` granularity using a configurable scheduling policy (*e.g.*, round-robin, shortest-queue-first, etc.) to achieve fine-grained load balancing across `channels`. As we show in §5, this enables scaling network layer processing across cores.

Dynamic scaling and placement of channels. NetScheduler scales the number of `channels` to a given remote host dynamically by monitoring scheduling metrics at coarse-grained timescales. An example policy is to increase the number of `channels` when the average CPU utilization across channel workers is persistently high. Further, NetScheduler also controls the mapping of `channels` to cores. This can be exploited to achieve performance isolation by separating `channels` for L-app and T-app processing and mapping these `channels` to different cores. As we show in §5, this enables performance isolation when L-apps and T-apps are co-located.

Dynamic scheduling of data copy requests. NetScheduler schedules data copy requests generated by virtual sockets over multiple per-core worker threads at per-request granularity. It uses the cores in the same NUMA node as the application core to avoid cross-NUMA data copy overheads. As we show in §5, this makes it possible to selectively parallelize data copy across cores for T-apps.

4 NetChannel IMPLEMENTATION

We have implemented NetChannel in Linux kernel v5.6. Throughout the implementation, our goal was to re-use existing kernel network stack infrastructure as much as possible. To that end, most of our current implementation re-uses existing code in the kernel. In this section, we discuss some interesting details of our NetChannel implementation.

Application interfaces. At VNS, our goal is to support unmodified application interfaces. We thus implement the virtual socket interface by adding a new flag— `IPPROTO_VIRTUAL_SOCK` in the standard socket interface to create virtual sockets. Applications can specify NetChannel-related attributes via `setsockopt()` — *e.g.*, the `SO_APP_TYPE` attribute determines the application class (*e.g.*, latency-sensitive, throughput-bound, etc.). The RPC interfaces are similar to those in prior work [41].

Virtual socket connections. The virtual socket interface uses the following procedure to set up connections (similar to existing socket interface): clients initiate `connect` system calls and the corresponding *listen* sockets on the remote host accept the connection requests and return a new socket per request (`accept` system call). Underneath, virtual sockets perform a handshake using `NCSYN` and `NCSYN_ACK` control packets to set up a connection. Note that since the underlying `channels` in NetDriver already provide reliability, virtual sockets only require a 2-way handshake, unlike TCP's 3-way handshake.

NetChannel headers. Since one virtual socket can use multiple `channels` and/or multiple virtual sockets can share the same `channel`, NetChannel needs to uniquely identify packets to their corresponding virtual sockets. To do so, NetChannel wraps an additional header atop of the packet payload. The NetChannel header consists of (1) a pair of virtual socket source and destination ports, to uniquely identify virtual socket-level connections; (2) virtual socket sequence number, to perform packet reordering when multiple `channels` are used; and (3) packet type, to distinguish data packets from control packets (*e.g.*, `NCSYN` and `NCSYN_ACK`). Use of NetChannel header allows virtual sockets to work in conjunction with underlying `channels` without *any* modifications in the `channel`'s header.

Reducing page allocation overheads for DMA. To reduce page allocation overheads during DMA (caused by `get_page_from_freelist()` calls), our implementation sets up a dedicated page pool for each receive queue of NIC. While a large page pool size may help reduce the page allocation overhead, it may also increase L3 cache miss rate due to DCA effects [11]. We use 256 as the default page pool size. We found that it is sufficient to achieve reasonably low page allocation overhead while still maintaining a low DCA cache miss rate. Even for a NIC with 256 receive queues, the memory overhead of maintaining these page pools is $256 \times 256 \times 4$KB = 256MB, which is negligible relative to the DRAM sizes of modern servers.

Scheduling policy. Our current NetScheduler implementation adopts a simple round-robin scheduling policy for scheduling of (1) application data to channels and (2) data copy requests to workers. For (1), we use only `channels` of the same type as the application (i.e., L-app or T-app `channels`). To avoid overloading already busy `channels`/workers, we exclude those which have queue occupancy higher than a certain threshold. Through simple sensitivity analysis, we found $2MB$ and $640KB$ to be good thresholds for (1) and (2) respectively, and use these by default for our evaluation (§5).

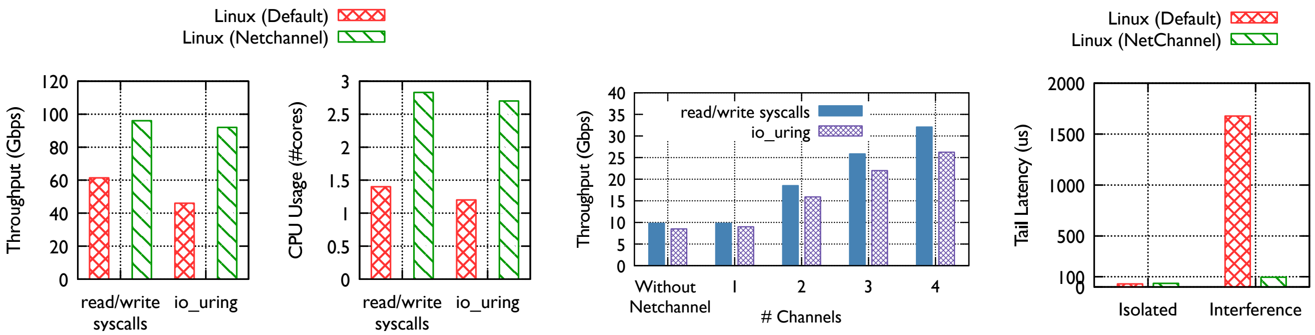


Figure 6: NetChannel enables Linux to achieve new operation points (left-right, a-d). (a, b) For a single long flow, it can saturate 100Gbps (a), by utilizing multiple cores for data copy processing (b). (c) It enables near-linear scaling of short-message throughput with an increasing number of `channels`. (d) It is able to provide performance isolation even when an L-app is co-located with 8 T-apps over 8 cores. More discussion in §5.2.

5 NetChannel EVALUATION

In this section, we demonstrate that NetChannel is able to achieve new operating points that were previously unachievable by the Linux network stack, in particular, saturating a 200Gbps link using a single socket, increasing short message throughput almost linearly with cores, and achieving μs-scale tail latency for L-apps even when they are co-located with T-apps. We describe our evaluation setup in §5.1. We use the same experimental scenarios in §2 and provide insights on how NetChannel alleviates the previously discussed limitations of today's Linux network stack (§5.2), followed by an investigation of the overheads of our current NetChannel prototype (§5.3). Next, we demonstrate NetChannel's effectiveness with real-world applications (§5.4), and finally show that it can scale to Terabit Ethernet (§5.5).

Before diving in, we make three important notes. First, NetChannel supports unmodified applications (§3), and one can run any application on top of it. In order to focus on the network stack, we use lightweight applications which perform minimal compute similar to prior works [11, 38], and additionally demonstrate NetChannel's effectiveness with two real world applications (Redis and SPDK). Second, our goal is not to show that NetChannel beats state-of-the-art network stack performance in absolute terms, but rather to demonstrate the benefits enabled by NetChannel architecture and understand its overheads. In order to do so, we naturally compare our prototype with the baseline system that it is implemented on top of (Linux). As we discuss in §6, NetChannel's ideas could very well be implemented on top of userspace stacks, hence making it complementary to these systems. Third, while parallelizing parts of the packet processing pipeline across multiple cores, NetChannel naturally introduces some (although relatively minimal) CPU overheads. Since the processing speed of a single CPU core has long since been saturated, utilizing multiple cores is essential. Hence, paying a small cost in per-core overheads to enable this is worthwhile.

5.1 Evaluation Setup

Hardware setup. Our experimental testbed consists of two servers connected directly via a 100Gbps link. Each server has a 4 NUMA nodes with 8 cores per NUMA node (Intel Xeon Gold 6234 3.3GHz CPU), 32KB/1MB/25MB L1/L2/L3 caches, 384GB DRAM and a 100Gbps NVIDIA Mellanox ConnectX-5 NIC. Both servers run Ubuntu 20.04 (Linux kernel v5.6). By default, we enable TSO, GRO, Jumbo Frames (9000B), aRFS, and Dynamically-Tuned Interrupt Moderation (DIM) [52] while disabling hyperthreading and IOMMU, since doing so maximizes performance. Direct Cache Access (Intel DDIO) is enabled and configured to use the maximum possible number of L3 cache ways for all experiments.

Evaluated workloads. Similar to §2, T-apps generate long-lived flows (i.e., stream traffic) and L-apps generate ping-pong style 4KB RPC requests/responses. Both of these applications perform minimal application-level processing, ensuring that the network stack is the bottleneck. We consider both scenarios of standard read/write system calls, and `io_uring` [40]. In all experiments, we only cores in the NUMA node where the NIC is attached. We also evaluate NetChannel with two real-world applications, Redis [59], and SPDK [33].

Performance metrics. We measure performance in terms of throughput for T-apps and P99.9 tail latency for L-apps. In order to quantify CPU efficiency and understand overheads, we use throughput-per-core which is measured as the total throughput / CPU utilization (we take the maximum of the client-side and server-side CPU utilization when computing CPU utilization).

5.2 New operating points

We now demonstrate how NetChannel enables three new operating points using the experimental scenarios from §2.2.

Scalability for long flows. In the extreme case of a T-app using a single TCP socket, Linux fails to saturate the 100Gbps link due to its static packet processing pipeline (§2.2), despite the availability of CPU cores. As shown in Figure 6(a), while using standard read/write system calls, NetChannel enables Linux to saturate the 100Gbps link by making use of multiple cores (Figure 6(b)). This is because NetChannel allows independent scaling of data copy processing (which is the bottleneck in this scenario) at VNS, so that NetScheduler can use two data copy worker threads on two different cores while maintaining one `channel` at NetDriver layer. In the `io_uring` case as well, we find that NetChannel enables Linux to nearly saturate 100Gbps by utilizing multiple cores (Figure 6(a, b)).

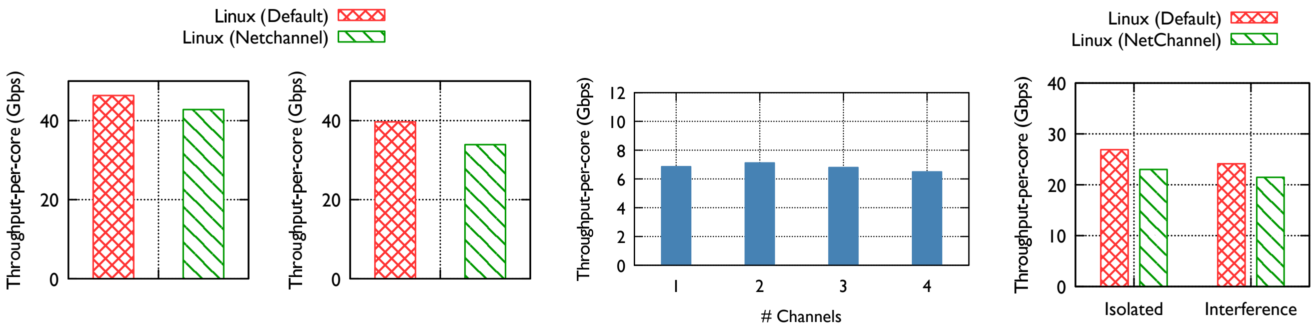


Figure 7: Understanding NetChannel overheads (left-right, a-d). NetChannel incurs minimal throughput-per-core overheads while emulating today's Linux pipeline (a), and while scaling data copy processing across cores (b). (c) throughput-per-core does not change independent of the number of `channels`. (d) it is able to isolate L-app latency with minimal throughput-per-core degradation. More discussion in §5.3.

Scalability for short messages. The second scenario in §2.2 considers a short message scenario with 4KB RPCs where network layer processing overheads are more dominant. To push these overheads to an extreme, we disable throughput optimization techniques including TSO/GRO and Jumbo Frames (both with and without NetChannel for a fair comparison). NetChannel enables Linux to dynamically scale network layer processing by allowing data from a single virtual socket to be multiplexed across multiple `channels` (§3). To demonstrate this, we measure throughput in this scenario while increasing the number of `channels` (each running on a separate core). With standard read/write system calls, we find that throughput increases near-linearly with the addition of `channels` (Figure 6(c)). We discuss overheads in §5.3. With `io_uring` as well, we see that throughput increases with the addition of `channels` (Figure 6(c)). For the same number of channels, we observe slightly lower throughput than with read/write syscalls due to `io_uring` having extra overheads on the application core.

Enabling performance isolation. In the third scenario that captures performance interference, we run 8 T-apps and one L-app over 8 cores. We focus on the case of standard read/write system calls. Since `io_uring` does not improve per-application performance, as observed in previous experiments, we omit it in the interest of brevity. As discussed in §2.2, with today's Linux network stack, the L-app suffers from high tail latency inflation due to network layer processing interference between T-apps and the L-app. NetChannel enables isolating network layer processing for L-apps from T-apps, even if they share the same core, by decoupling virtual sockets from `channels`— `channels` can be flexibly mapped to different cores. In this experiment, NetScheduler uses up to 4 `channels` for T-apps and a single `channel` for L-app as the L-app generates low load. These `channels` are assigned to different cores in order to separate network layer processing for the T-apps from that for the L-app. As shown in Figure 6(d), we see that with NetChannel, Linux is able to achieve 17.5× lower tail latency for L-app, hence demonstrating that NetChannel can indeed enable performance isolation. We also repeated this experiment with 8 L-apps instead of 1, and confirmed that benefits remain — NetChannel enables Linux to achieve 9.5× lower tail latency in this case.

5.3 Understanding NetChannel Overheads

We now investigate the overheads incurred by NetChannel in the process of enabling new operating points.

Overheads of emulating the Linux network stack. To better understand the overheads that NetChannel introduces, we emulate a single packet processing pipeline of today's Linux stack using NetChannel. To this end, we use a single application thread, and a single `channel` thread, while placing everything on the same core. For a fair comparison, Linux uses a single application thread, and we enable aRFS for both systems to ensure that the Rx packet processing is also run on the same core. In Figure 7(a), we observe that NetChannel shows a minimal ∼7% reduction in throughput-per-core (the server-side core is the bottleneck for both systems).

Overheads of scaling data copy processing. In order to understand the overheads of scaling data copy processing, we compare NetChannel and Linux using scenarios where they are both able to saturate the 100Gbps link, and compare the total CPU utilization. For NetChannel, we use 2 data copy threads and 1 `channel` thread running on separate cores (similar to Figure 6(a)). For Linux, we run 3 long-lived TCP connections over 3 cores to fully saturate the 100Gbps link (this is the minimum number needed to saturate 100Gbps). We find that NetChannel incurs a minimal 12% reduction in throughput-per-core as shown in Figure 7(b). The main reason for this is because the application buffers are not warm in the L1 cache of the cores where the data copy worker threads run, leading to higher L1 cache misses during data copy.

Overheads of scaling network layer processing. To understand overheads incurred by NetChannel while scaling network layer processing, we measure the throughput-per-core from Figure 6(c), as the number of `channels` increases from 1 to 4. As shown in Figure 7(c), throughput-per-core remains the same independent of the number of channels. Compared to Linux (default), there is a relatively small degradation in throughput-per-core. We found that the reason for this overhead is that NetScheduler needs to wake up `channel` threads more frequently as each `channel` thread goes to sleep after processing each short message (4KB). Such overheads could be reduced if we perform batching at the NetScheduler layer.

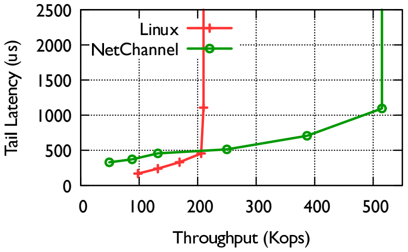


Figure 8: NetChannel enables Linux to achieve 2.4× higher throughput for Redis. More discussion in §5.4.

Overheads of achieving performance isolation. Now we consider the performance isolation scenario in Figure 6(d), where 8 T-apps and 1 L-app are running over 8 cores, to understand the overheads of achieving performance isolation. Figure 7(d) shows the throughput-per-core of T-apps in this experiment. We see that NetChannel incurs only a minimal throughput-per-core reduction (12%) in the Interference case, while achieving more than an order-of-magnitude reduction in tail latency (Figure 6(d)).

5.4 Real-world applications with NetChannel

Redis with NetChannel. We now evaluate NetChannel with Redis [59], a well-known in-memory key-value database. We use the standard YCSB workload [66], with 95%/5% read/write ratio. Each RPC request is 4KB in size. We use 8 threads on the client side to fully utilize all cores on a single NUMA node. Since the workload generates small-sized messages, we use the same configuration used in Figure 6(c).

Figure 8 shows the latency-throughput curve with and without NetChannel. We see that NetChannel enables Linux to achieve 2.4× higher throughput for Redis. This is because NetChannel enables scaling network layer processing across cores through multiple `channels`. (In this experiment, NetChannel increases the number of `channels` to 4.). In terms of tail latency, NetChannel incurs slightly higher scheduling and reordering latency as the virtual socket of the Redis server is mapped to multiple `channels`. While this latency overhead is more visible at low load, it is ~160μs.

SPDK-based remote storage stack with NetChannel. There has been significant recent work on designing remote storage stacks in the disaggregation context [22, 28, 29, 32]. To evaluate this scenario, we use SPDK [33], a widely-deployed userspace storage stack. In particular, we use SPDK's NVMe-over-TCP stack [32] that uses the Linux TCP/IP stack by default to access a remote storage device over the network. We use an experimental setup similar to prior work [29] — in our two-machine testbed, the SPDK client (we use the standard SPDK perf benchmark tool [31]) runs on one of the machines and issues I/O requests to a remote in-memory storage device (RAM block device) exposed by the SPDK server running on the other machine. Both the SPDK client and server application threads run on a single CPU core each, and use a single TCP connection for data transfer. We use a sequential 100% read workload with large I/Os (2MB) in order to maximize throughput.

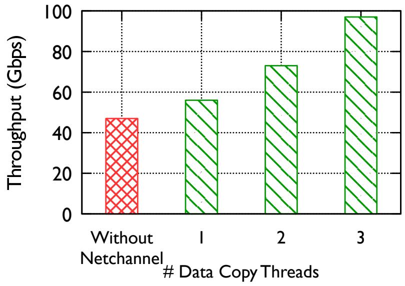


Figure 9: NetChannel enables Linux to achieve 2.06× higher throughput for SPDK. More discussion in §5.4.

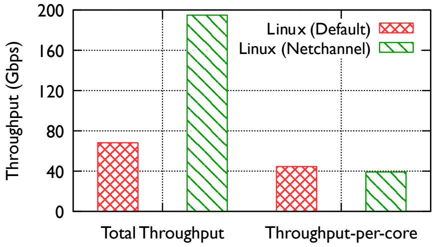


Figure 10: NetChannel enables Linux to saturate 200Gbps link bandwidth using a single socket. More discussion in §5.5.

Figure 9 shows the total throughput that SPDK achieves with and without NetChannel. With NetChannel, we use a single channel and vary the number of data copy threads. Even with a single data copy thread, using NetChannel already leads to a 1.19× increase in SPDK throughput as network layer processing is offloaded onto a separate core. Increasing the number of data copy threads leads to significant improvements in throughput as data copy is parallelized across multiple cores. With 3 data copy threads, NetChannel enables SPDK to achieve 2.06× higher throughput and saturate the 100Gbps link with a single TCP connection. Relative to the Figure 6(a) experiment, SPDK requires one extra data copy thread to saturate the link bandwidth. This is due to the SPDK client using larger buffers and additional application-level delays for processing responses before data copy. Both of these factors contribute to higher L3 cache miss rate as analyzed in prior work [11], and hence result in reduced data copy efficiency.

5.5 NetChannel with Terabit Ethernet

We now demonstrate that NetChannel scales to link speeds beyond 100Gbps, i.e., Terabit Ethernet [4]. For this we use a different testbed which has two servers directly connected by a 200Gbps link. Each server has 2 NUMA nodes (Intel Xeon Gold 6354 3.0GHz CPUs) and a Mellanox ConnectX-6 NIC. Each NUMA-node has 18 cores and 39MB of L3 cache. We re-ran the Figure 6(a) experiment of a single T-app with a single socket on this new setup. As shown in Figure 10, NetChannel enables Linux to saturate the 200Gbps link bandwidth using a single application thread. To do so, it uses 2 `channels` and 3 data copy threads. Unlike in the previous setup (Figure 6(a)), a single `channel` is no longer sufficient to saturate the link, as a single core is not able to perform all of the network layer processing at the required rate. We find that the throughput-per-core achieved by Linux both without and with NetChannel (Figure 10) increases on the new testbed (by 12% and 15% respectively) due to improved data copy efficiency as a result of larger L3 cache size.

6 DISCUSSION

We discuss the generality of NetChannel's architecture, placement of its functionality, and its scheduling policies.

Can NetChannel architecture be applied to other network stacks? In this paper, we have realized the NetChannel architecture within the Linux network stack. However, NetChannel's architecture and design ideas can be applied to host network stacks in general—even those placed in userspace and/or hardware. Microkernel-style userspace stacks [21, 51, 54] would be ideal candidates for implementing NetChannel's ideas. For example, while TAS [38] decouples the packet processing pipeline from application cores, it can benefit from additionally disaggregating different parts of the packet processing pipeline similar to NetChannel. NetChannel's ideas can also be applied to hardware network stacks. For example, in SoC based smartNICs [48], NetChannel's design could enable transport-agnostic parallelization of processing across cores, which is even more important in this context since these devices typically contain a large number of wimpy cores [48, 62]. Further, by disaggregating the host network stack, NetChannel could enable easier integration of partial hardware offloads that offload different parts of the packet processing pipeline (e.g., I/OAT [58] for data copy, and Tonic [6] for transport layer) into Linux or other software network stacks. We leave an exploration of these directions to future work.

Can NetChannel benefits be achieved with modifications outside the network stack? While it may be possible to achieve some of NetChannel's benefits using libraries and/or schedulers outside the network stack (*e.g.*, gRPC can multiplex RPCs across different underlying connections), there are two limitations of such an approach. First, decoupling and independently scaling different parts of the packet processing pipeline (*e.g.*, data copy and network layer processing) requires support from the network stack, thus necessitating modifications similar to NetChannel. Second, in multi-tenant deployments, these libraries do not have global visibility of all applications, system-level metrics like CPU utilization, and in particular, network-layer metrics like congestion information to effectively make multiplexing decisions. Thus, the network stack is the right place to realize the NetChannel architecture and its benefits.

NetScheduler Policies. The NetScheduler layer in NetChannel's architecture provides the mechanisms to implement different scheduling and placement policies. We have demonstrated in §5 that even with simple policies, NetChannel can achieve significant benefits. Similar to storage stacks where schedulers have evolved over years [1–3, 26], we expect new NetScheduler policies will be developed over time to match application requirements.

7 RELATED WORK

We discuss work that is most closely related to NetChannel's goals.

Linux network stack improvements. Linux now has support for TCP zero copy send [15] and receive [16]. However, as discussed in prior work [11], these mechanisms are far from offering a silver bullet. Zero-copy receive (which bears similarities to the older zero-copy mechanisms in Solaris [13]) in particular requires special hardware support (header-data split) in the NIC [16], and has several other limitations [20], which limits widespread adoption. Nevertheless, even the small subset of applications which use these zero-copy mechanisms can still benefit from network layer processing scalability, performance isolation, and other benefits enabled by NetChannel. Our work is complementary to many existing Linux kernel optimization efforts especially for small messages, through new interfaces [25, 67], system call optimizations [63, 65], and optimized socket implementations [42].

Recent work [11] has reported an in-depth analysis of overheads in the Linux network stack. NetChannel's design is motivated by observations and insights from this work. i10 [28] and blk-switch [29] are recent enhancements to the Linux storage stack. They make use of the unmodified Linux network stack for remote storage access, and could reap the benefits of NetChannel if run on top of it.

Userspace network stacks. There has been a significant amount of recent work on designing userspace network stacks [7, 21, 34, 36–38, 49–51, 54, 56, 57, 68], many of which are built on top of low-level frameworks such as DPDK and netmap [60]. As discussed in §6, NetChannel's architecture has the potential to provide benefits to userspace network stacks as well.

Hardware network stacks. There has also been a lot of recent work on both partially and fully offloading host network stacks to hardware [6, 9, 58, 62]. At a conceptual level, FlexTOE [62] is the closest to our work. Similar to NetChannel, it enables parallelization of the packet processing pipeline. However, it focuses on parallelizing a specific transport protocol implementation (TCP), unlike NetChannel which considers the end-to-end packet processing pipeline from the application to the NIC, and enables parallelization in a transport-agnostic manner. In general, as discussed in §6, we believe that NetChannel's architectural ideas can be applied to hardware-offloaded network stacks as well.

8 CONCLUSION

We have demonstrated that today's host network stacks are unable to fully exploit the capabilities of modern hardware due to their dedicated, tightly integrated, and static packet processing pipelines. Our core contribution is NetChannel, a new disaggregated host network stack architecture that rearchitects the stack into three loosely-coupled layers. We have implemented NetChannel in the Linux kernel, and evaluated it to demonstrate that it enables new operating points that were previously unachievable, including saturation of a Terabit ethernet link with a single application core, independent scaling of network layer processing, and performance isolation between latency-sensitive and throughput-bound applications.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Jon Crowcroft, and the anonymous SIGCOMM reviewers for their insightful feedback. We would also like to thank Saksham Agarwal and Abhishek Vijay for many helpful discussions. This work was supported in part by NSF grants CNS-2047283 and CNS-1704742, a Google faculty research award, and a Sloan fellowship. This work was also supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2022R1A2C1011090). Christos Kozyrakis was supported by the Stanford Platform Lab and its affiliate members. This work does not raise any ethical issues.

REFERENCES

[1] 2017. CFQ (Complete Fairness Queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[2] 2017. Kyber multiqueue I/O scheduler. https://lwn.net/Articles/720071/.

[3] 2019. BFQ (Budget Fair Queueing). https://www.kernel.org/doc/Documentation/block/bfq-iosched.txt.

[4] 2022. Terabit Ethernet. https://en.wikipedia.org/wiki/Terabit_Ethernet.

[5] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *ACM SIGCOMM*.

[6] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *USENIX NSDI*.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI*.

[8] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *ACM SYSTOR*.

[9] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *USENIX OSDI*.

[10] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. 2022. dcPIM: Near-Optimal Proactive Datacenter Transport. In *ACM SIGCOMM*.

[11] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *ACM SIGCOMM*.

[12] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. https://github.com/Terabit-Ethernet/NetChannel.

[13] H. K. Jerry Chu. 1996. Zero-Copy TCP in Solaris. In *USENIX ATC*.

[14] Jonathan Corbet. 2012. TCP small queues. https://lwn.net/Articles/507065/.

[15] Jonathan Corbet. 2017. Zero-copy networking. https://lwn.net/Articles/726917/.

[16] Jonathan Corbet. 2018. Zero-copy TCP receive. https://lwn.net/Articles/752188/.

[17] Jonathan Corbet. 2021. Zero-copy network transmission with io_uring. https://lwn.net/Articles/879724/.

[18] Gregory Detal and Sebastien Barre. 2022. MultiPath TCP - Linux Kernel implementation. https://multipath-tcp.org/.

[19] Jon Dugan, John Estabrook, Jim Ferbuson, Andrew Gallatin, Mark Gates, Kevin Gibbs, Stephen Hemminger, Nathan Jones, Gerrit Renker Feng Qin, Ajay Tirumala, and Alex Warshavsky. 2021. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

[20] Eric Dumazet. 2012. The Path To TCP 4K MTU and RX ZeroCopy. https://legacy.netdevconf.info/0x14/pub/slides/62/ImplementingTCPRXzerocopy.pdf.

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *USENIX OSDI*.

[22] Peter Gao, Akshay Narayan, Sagar Karandikar, Joǎo Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *USENIX OSDI*.

[23] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. pHost: Distributed near-optimal datacenter transport over commodity network fabric. In *ACM CoNEXT*.

[24] Google. 2022. gRPC: A high performance, open source universal RPC framework. https://grpc.io/.

[25] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *USENIX OSDI*.

[26] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *USENIX ATC*.

[27] Alex Hultman. 2020. io_uring is slower than epoll. https://github.com/axboe/liburing/issues/189.

[28] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *USENIX NSDI*.

[29] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *USENIX OSDI*.

[30] Intel. 2012. Intel® Data Direct I/O Technology. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf.

[31] Intel. 2022. https://github.com/spdk/spdk/tree/master/examples/nvme/perf.

[32] Intel. 2022. SPDK: NVMe over Fabrics Target. https://spdk.io/doc/nvmf.html.

[33] Intel. 2022. Storage Performance Development Kit. https://spdk.io/.

[34] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems.. In *USENIX NSDI*.

[35] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *USENIX NSDI*.

[36] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *USENIX NSDI*.

[37] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable Low Latency for Data Center Applications. In *ACM SoCC*.

[38] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *ACM Eurosys*.

[39] Kernel. 2001. https://man7.org/linux/man-pages/man8/tc.8.html.

[40] Kernel. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[41] Collin Lee and Yilong Li. 2021. Homa DPDK Implementation. https://github.com/PlatformLab/Homa.

[42] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *ACM ASPLOS*.

[43] Linux. 2002. Qdisc: Pfifo Fast Scheduling Policy. https://man7.org/linux/man-pages/man8/tc-pfifo_fast.8.html.

[44] Linux. 2021. epoll: I/O event notification facilit. https://man7.org/linux/man-pages/man7/epoll.7.html.

[45] Linux. 2021. Socket. https://man7.org/linux/man-pages/man2/socket.2.html.

[46] Linux. 2022. Linux Kernel CFS Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[47] Linux. 2022. MPTCP Upstream Implementation. https://github.com/multipath-tcp/mptcp_net-next/wiki.

[48] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs using iPipe. In *ACM SIGCOMM*.

[49] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *ACM SIGCOMM*.

[50] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. 2017. Disk|Crypt|Net: Rethinking the Stack for High-Performance Video Streaming. In *ACM SIGCOMM*.

[51] 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 Kokonov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: a Microkernel Approach to Host Networking. In *ACM SOSP*.

[52] Mellanox. 2019. Mellanox Technologies: Dynamically-Tuned Interrupt Moderation (DIM). https://support.mellanox.com/s/article/dynamically-tuned-interrupt-moderation--dim-x.

[53] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *ACM SIGCOMM*.

[54] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*.

[55] John Ousterhout. 2021. A Linux Kernel Implementation of the Homa Transport Protocol. In *USENIX ATC*.

[56] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *USENIX OSDI*.

[57] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *ACM SOSP*.

[58] Quoc-Thai V Le, Jonathan Stern, and Stephen M Brenner. 2017. Fast memcpy with SPDK and Intel I/OAT DMA Engine. https://software.intel.com/content/www/us/en/develop/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.html.

[59] Redis. 2022. Redis: an in-memory data structure store. https://redis.io.

[60] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*.

[61] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. 2011. It's Time for Low Latency. In *USENIX HotOS*.

[62] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *USENIX NSDI*.

[63] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *USENIX OSDI*.

[64] u/T0p_H4t. 2021. https://tinyurl.com/iouring-reddit.

[65] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. 2011. The Case for VOS: The Vector Operating System. In *USENIX HotOS*.

[66] Yahoo. 2019. YCSB: Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/wiki.

[67] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *USENIX ATC*.

[68] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynykr, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *ACM SOSP*.