Understanding Host Network Stack Overheads

Qizhe Cai
Cornell University

Shubham Chaudhary
Cornell University

Midhul Vuppalapati
Cornell University

Jaehyun Hwang
Cornell University

Rachit Agarwal
Cornell University

ABSTRACT

Traditional end-host network stacks are struggling to keep up with rapidly increasing datacenter access link bandwidths due to their unsustainable CPU overheads. Motivated by this, our community is exploring a multitude of solutions for future network stacks: from Linux kernel optimizations to partial hardware offload to clean-slate userspace stacks to specialized host network hardware. The design space explored by these solutions would benefit from a detailed understanding of CPU inefficiencies in existing network stacks.

This paper presents measurement and insights for Linux kernel network stack performance for 100Gbps access link bandwidths. Our study reveals that such high bandwidth links, coupled with relatively stagnant technology trends for other host resources (*e.g.*, core speeds and count, cache sizes, NIC buffer sizes, etc.), mark a fundamental shift in host network stack bottlenecks. For instance, we find that a single core is no longer able to process packets at line rate, with data copy from kernel to application buffers at the receiver becoming the core performance bottleneck. In addition, increase in bandwidth-delay products have outpaced the increase in cache sizes, resulting in inefficient DMA pipeline between the NIC and the CPU. Finally, we find that traditional loosely-coupled design of network stack and CPU schedulers in existing operating systems becomes a limiting factor in scaling network stack performance across cores. Based on insights from our study, we discuss implications to design of future operating systems, network protocols, and host hardware.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Network performance analysis**; **Data center networks**; • **Hardware** → **Networking hardware**;

KEYWORDS

Datacenter networks, Host network stacks, Network hardware

ACM Reference Format:
Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3452296.3472888

SIGCOMM '21, August 23–27, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8383-7/21/08...$15.00
https://doi.org/10.1145/3452296.3472888

1 INTRODUCTION

The slowdown of Moore's Law, the end of Dennard's scaling, and the rapid adoption of high-bandwidth links have brought traditional host network stacks at the brink of a breakdown—while datacenter access link bandwidths (and resulting computing needs for packet processing) have increased by 4 – 10× over the past few years, technology trends for essentially all other host resources (including core speeds and counts, cache sizes, NIC buffer sizes, etc.) have largely been stagnant. As a result, the problem of designing CPU-efficient host network stacks has come to the forefront, and our community is exploring a variety of solutions, including Linux network stack optimizations [11, 12, 21, 24, 32, 41], hardware offloads [3, 6, 9, 16], RDMA [31, 34, 43], clean-slate userspace network stacks [4, 27, 30, 33, 36], and even specialized host network hardware [2]. The design space explored by these solutions would benefit from a detailed understanding of CPU inefficiencies of traditional Linux network stack. Building such an understanding is hard because the Linux network stack is not only large and complex, but also comprises of many components that are tightly integrated into an end-to-end packet processing pipeline.

Several recent papers present a preliminary analysis of Linux network stack overheads for short flows [21, 30, 32, 38, 40]. This fails to provide a complete picture due to two reasons. First, for datacenter networks, it is well-known that an overwhelmingly large fraction of data is contained in long flows [1, 5, 28]; thus, even if there are many short flows, most of the CPU cycles may be spent in processing packets from long flows. Second, datacenter workloads contain not just short flows or long flows in exclusion, but a mixture of different flow sizes composed in a variety of traffic patterns; as we will demonstrate, CPU characteristics change significantly with varying traffic patterns and mixture of flow sizes.

This paper presents measurement and insights for Linux kernel network stack performance for 100Gbps access link bandwidths. Our key findings are:

High-bandwidth links result in performance bottlenecks shifting from protocol processing to data copy. Modern Linux network stack can achieve ~42Gbps throughput-per-core by exploiting all commonly available features in commodity NICs, *e.g.*, segmentation and receive offload, jumbo frames, and packet steering. While this throughput is for the best-case scenario of a single long flow, the dominant overhead is consistent across a variety of scenarios—data copy from kernel buffers to application buffers (*e.g.*, > 50% of total CPU cycles for a single long flow). This is in sharp contrast to previous studies on short flows and/or low-bandwidth links, where protocol processing was shown to be the main bottleneck. We also observe receiver-side packet processing to become a bottleneck much earlier than the sender-side.

- *Implications.* Emerging zero-copy mechanisms from the Linux networking community [11, 12] may alleviate data copy overheads, and may soon allow the Linux network stack to process as much as 100Gbps worth of data using a single core. Integration of other hardware offloads like I/OAT [37] that transparently mitigate data copy overheads could also lead to performance improvements. Hardware offloads of transport protocols [3, 43] and userspace network stacks [21, 27, 30] that do not provide zero-copy interfaces may improve throughput in microbenchmarks, but will require additional mechanisms to achieve CPU efficiency when integrated into an end-to-end system.

The reducing gap between bandwidth-delay product (BDP) and cache sizes leads to suboptimal throughput. Modern CPU support for Direct Cache Access (DCA) (*e.g.*, Intel DDIO [25]) allows NICs to DMA packets directly into L3 cache, reducing data copy overheads; given its promise, DDIO is enabled by default in most systems. While DDIO is expected to improve performance during data copy, rather surprisingly, we observe that it suffers from high cache miss rates (49%) even for a single flow, thus providing limited performance gains. Our investigation revealed that the reason for this is quite subtle: host processing becoming a bottleneck results in increased host latencies; combined with increased access link bandwidths, BDP values increase. This increase outpaces increase in L3 cache sizes—data is DMAed from the NIC to the cache, and for larger BDP values, cache is rapidly overwritten before the application performs data copy of the cached data. As a result, we observe as much as 24% drop in throughput-per-core.

- *Implications.* We need better orchestration of host resources among contending connections to minimize latency incurred at the host, and to minimize cache miss rates during data copy. In addition, window size tuning should take into account not only traditional metrics like latency and throughput, but also L3 sizes.

Host resource sharing considered harmful. We observe as much as 66% difference in throughput-per-core across different traffic patterns (single flow, one-to-one, incast, outcast, and all-to-all) due to undesirable effects of multiple flows sharing host resources. For instance, multiple flows on the same NUMA node (thus, sharing the same L3 cache) make the cache performance even worse—the data DMAed by the NIC into the cache for one flow is polluted by the data DMAed by the NIC for other flows, before application for the first flow could perform data copy. Multiple flows sharing host resources also results in packets arriving at the NIC belonging to different flows; this, in turn, results in packet processing overheads getting worse since existing optimizations (*e.g.*, coalescing packets using generic receive offload) lose a chance to aggregate larger number of packets. This increases per-byte processing overhead, and eventually scheduling overheads.

- *Implications.* In the Internet and in early-generation datacenter networks, performance bottlenecks were in the network core; thus, multiple flows "sharing" host resources did not have performance implications. However, for high-bandwidth networks, such is no longer the case—if the goal is to design CPU-efficient network stacks, one must carefully orchestrate host resources so as to minimize contention between active flows. Recent receiver-driven transport protocols [18, 35] can be extended to reduce the number of concurrently scheduled flows, potentially enabling high CPU efficiency for future network stacks.

The need to revisit host layering and packet processing pipelines. We observe as much as ∼43% reduction in throughput-per-core compared to the single flow case when applications generating long flows share CPU cores with those generating short flows. This is both due to increased scheduling overheads, and also due to high CPU overheads for short flow processing. In addition, short flows and long flows suffer from very different performance bottlenecks—the former have high packet processing overheads while the latter have high data copy overheads; however, today's network stacks use the same packet processing pipeline independent of the type of the flow. Finally, we observe ∼20% additional drop in throughput-per-core when applications generating long flows are running on CPU cores that are not in the same NUMA domain as the NIC (due to additional data copy overheads).

- *Implications.* Design of CPU schedulers independent of the network layer was beneficial for independent evolution of the two layers; however, with performance bottlenecks shifting to hosts, we need to revisit such a separation. For instance, application-aware CPU scheduling (*e.g.*, scheduling applications that generate long flows on NIC-local NUMA node, scheduling long-flow and short-flow applications on separate CPU cores, etc.) are required to improve CPU efficiency. We should also rethink host packet processing pipelines—unlike today's designs that use the same pipeline for short and long flows, achieving CPU efficiency requires application-aware packet processing pipelines.

Our study[1] not only corroborates many exciting ongoing activities in systems, networking and architecture communities on designing CPU-efficient host network stacks, but also highlights several interesting avenues for research in designing future operating systems, network protocols and network hardware. We discuss them in §4.

Before diving deeper, we outline several caveats of our study. First, our study uses one particular host network stack (the Linux kernel) running atop one particular host hardware. While we focus on identifying trends and drawing general principles rather than individual data points, other combinations of host network stacks and hardware may exhibit different performance characteristics. Second, our study focuses on CPU utilization and throughput; host network stack latency is another important metric, but requires exploring many additional bottlenecks in end-to-end system (*e.g.*, network topology, switches, congestion, etc.); a study that establishes latency bottlenecks in host network stacks, and their contribution to end-to-end latency remains an important and relatively less explored space. Third, kernel network stacks evolve rapidly; any study of our form must fix a version to ensure consistency across results and observations; nevertheless, our preliminary exploration [7] suggests that the most recent Linux kernel exhibits performance very similar to our results. Finally, our goal is not to take a position on how future network stacks will evolve (in-kernel, userspace, hardware), but rather to obtain a deeper understanding of a highly mature and widely deployed network stack.

[1] All Linux instrumentation code and scripts along with all the documentation needed to reproduce our results are available at `https://github.com/Terabit-Ethernet/terabit-network-stack-profiling`.

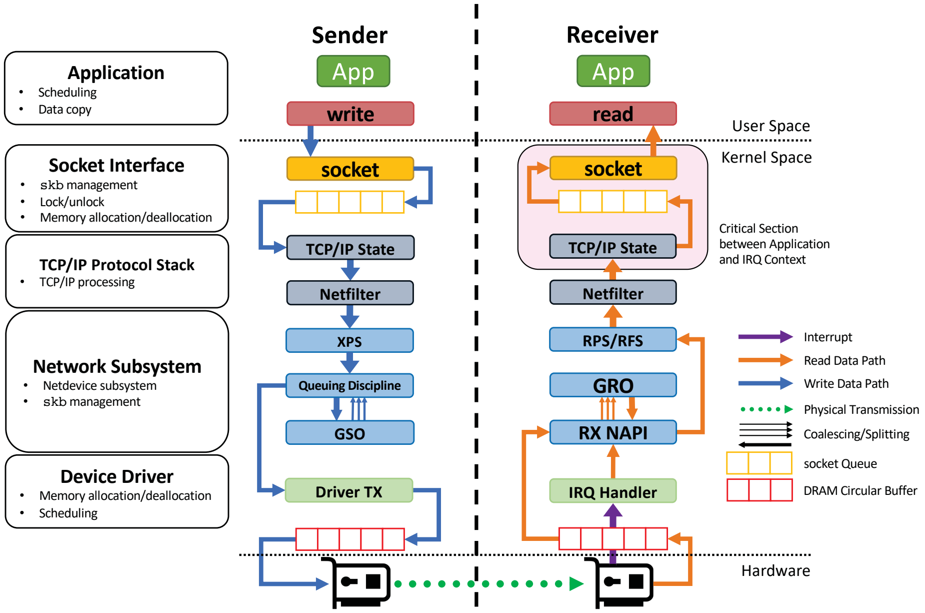


Figure 1: Sender and receiver-side data path in the Linux network stack. See §2.1 for description.

Component	Description
Data copy	From user space to kernel space, and vice versa.
TCP/IP	All the packet processing at TCP/IP layers.
Netdevice subsystem	Netdevice and NIC driver operations (*e.g.*, NAPI polling, GSO/GRO, qdisc, etc.).
skb management	Functions to build, split, and release skb.
Memory de-/alloc	skb de-/allocation and page-related operations.
Lock/unlock	Lock-related operations (*e.g.*, spin locks).
Scheduling	Scheduling/context-switching among threads.
Others	All the remaining functions (*e.g.*, IRQ handling).

Table 1: CPU usage taxonomy. The components are mapped into layers as shown in Fig. 1.

2 PRELIMINARIES

The Linux network stack tightly integrates many components into an end-to-end pipeline. We start this section by reviewing these components (§2.1). We also discuss commonly used optimizations, and corresponding hardware offloads supported by commodity NICs. A more detailed description is presented in [7]. We then summarize the methodology used in our study (§2.2).

2.1 End-to-End Data Path

The Linux network stack has slightly different data paths for the sender-side (application to NIC) and the receiver-side (NIC to application), as shown in Fig. 1. We describe them separately.

Sender-side. When the sender-side application executes a `write` system call, the kernel initializes socket buffers (`skb`s). For the data referenced by the `skb`s, the kernel then performs data copy from the userspace buffer to the kernel buffer. The `skb`s are then processed by the TCP/IP layer. When ready to be transmitted (*e.g.*, congestion control window/rate limits permitting), the data is processed by the network subsystem; here, among other processing steps, `skb`s are segmented into Maximum Transmission Unit (MTU) sized chunks by Generic Segmentation offload (GSO) and are enqueued in the NIC driver Tx queue(s). Most commodity NICs also support hardware offload of packet segmentation, referred to as TCP segmentation offload (TSO); see more details in [7]. Finally, the driver processes the Tx queue(s), creating the necessary mappings for the NIC to DMA the data from the kernel buffer referenced by `skb`s. Importantly, almost all sender-side processing in today's Linux network stack is performed at the same core as the application.

Receiver-side. The NIC has a number of Rx queues and a per-Rx queue page-pool from which DMA memory is allocated (backed by the kernel `pageset`). The NIC also has a configurable number of Rx descriptors, each of which contains a memory address that the NIC can use to DMA received frames. Each descriptor is associated with enough memory for one MTU-sized frame.

Upon receiving a new frame, the NIC uses one of the Rx descriptors, and DMAs the frame to the kernel memory associated with the descriptor. Ordinarily, the NIC DMAs the frame to DRAM; however, modern CPUs have support for Direct Cache Access (DCA) (*e.g.*, using Intel's Data Direct I/O technology (DDIO) technology [25]) that allows NIC to DMA the frames directly to the L3 cache. DCA enables applications to avoid going to DRAM to access the data.

Asynchronously, the NIC generates an Interrupt ReQuests (IRQ) to inform the driver of new data to be processed. The CPU core that processes the IRQ is selected by the NIC using one of the hardware steering mechanisms; see Table 2 for a summary, and [7] for details on how receiver-side flow steering techniques work. Upon receiving an IRQ, the driver triggers NAPI polling [17], that provides an alternative to purely interrupt-based network layer processing—the system busy polls on incoming frames until a certain number of frames are received or a timer expires[2]. This reduces the number of IRQs, especially for high-speed networks where incoming data rate is high. While busy polling, the driver allocates an `skb` for each frame, and makes a cross reference between the `skb` and the kernel memory where the frame has been DMAed. If the NIC has written enough data to consume all Rx descriptors, the driver allocates more DMA memory using the page-pool and creates new descriptors.

The network subsystem then attempts to reduce the number of `skb`s by merging them using Generic Receive Offload (GRO), or its corresponding hardware offload Large Receive Offload (LRO); see discussion in [7]. Next, TCP/IP processing is scheduled on one of the CPU cores using the flow steering mechanism enabled in the system (see Table 2). Importantly, with aRFS enabled, all the processing (the

[2] These NAPI parameters can be tuned via `net.core.netdev_budget` and `net.core.netdev_budget_usecs` kernel parameters, which are set to 300 and 2ms by default in our Linux distribution.

Mechanism	Description
Receive Packet Steering (RPS)	Use the 4-tuple hash for core selection.
Receive Flow Steering (RFS)	Find the core that the application is running on.
Receive Side Steering (RSS)	Hardware version of RPS supported by NICs.
accelerated RFS (aRFS)	Hardware version of RFS supported by NICs.

Table 2: Receiver-side flow steering techniques.

IRQ handler, TCP/IP and application) is performed on the same CPU core. Once scheduled, the TCP/IP layer processing is performed and all in-order skbs are appended to the socket's receive queue. Finally, the application thread performs data copy of the payload in the skbs in the socket receive queue to the userspace buffer. Note that at both the sender-side and the receiver-side, data copy of packet payloads is performed only once (when the data is transferred between userspace and kernel space). All other operations within the kernel are performed using metadata and pointer manipulations on skbs, and do not require data copy.

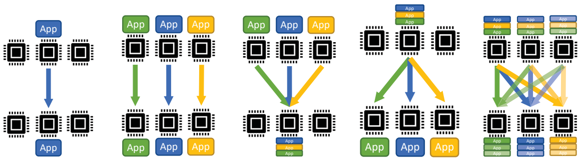
2.2 Measurement Methodology

In this subsection, we briefly describe our testbed setup, experimental scenarios, and measurement methodology.

Testbed setup. To ensure that bottlenecks are at the network stack, we setup a testbed with two servers directly connected via a 100Gbps link (without any intervening switches). Both of our servers have a 4-socket NUMA-enabled Intel Xeon Gold 6128 3.4GHz CPU with 6 cores per socket, 32KB/1MB/20MB L1/L2/L3 caches, 256GB RAM, and a 100Gbps Mellanox ConnectX-5 Ex NIC connected to one of the sockets. Both servers run Ubuntu 16.04 with Linux kernel 5.4.43. Unless specified otherwise, we enable DDIO, and disable hyperthreading and IOMMU in our experiments.

Experimental scenarios. We study network stack performance using five standard traffic patterns (Fig. 2)—single flow, one-to-one, incast, outcast, and all-to-all—using workloads that comprise long flows, short flows, and even a mix of long and short flows. For generating long flows, we use a standard network benchmarking tool, iPerf [14], which transmits a flow from sender to receiver; for generating short flows, we use netperf [22] that supports ping-pong style RPC workloads. Both of these tools perform minimal application-level processing, which allows us to focus on performance bottlenecks in the network stack (rather than those arising due to complex interactions between applications and the network stack); many of our results may have different characteristics if applications were to perform additional processing. We also study the impact of in-network congestion, impact of DDIO and impact of IOMMU. We use Linux's default congestion control algorithm, TCP Cubic, but also study impact of different congestion control protocols. For each scenario, we describe the setup inline.

Performance metrics. We measure total throughput, total CPU utilization across all cores (using sysstat [19], which includes kernel and application processing), and throughput-per-core—ratio of total throughput and total CPU utilization at the bottleneck (sender or receiver). To perform CPU profiling, we use the standard sampling-based technique to obtain a per-function breakdown of CPU cycles [20]. We take the top functions that account for ~95% of the CPU utilization. By examining the kernel source code, we classify these functions into 8 categories as described in Table 1.



(a) Single (b) One-to-one (c) Incast (d) Outcast (e) All-to-all

Figure 2: Traffic patterns used in our study. (a) Single flow from one sender core to one receiver core. **(b)** One flow from each sender core to a unique receiver core. **(c)** One flow from each sender core, all to a single receiver core. **(d)** One flow to each receiver core all from a single sender core. **(e)** One flow between every pair of sender and receiver cores.

3 LINUX NETWORK STACK OVERHEADS

We now evaluate the Linux network stack overheads for a variety of scenarios, and present detailed insights on observed performance.

3.1 Single Flow

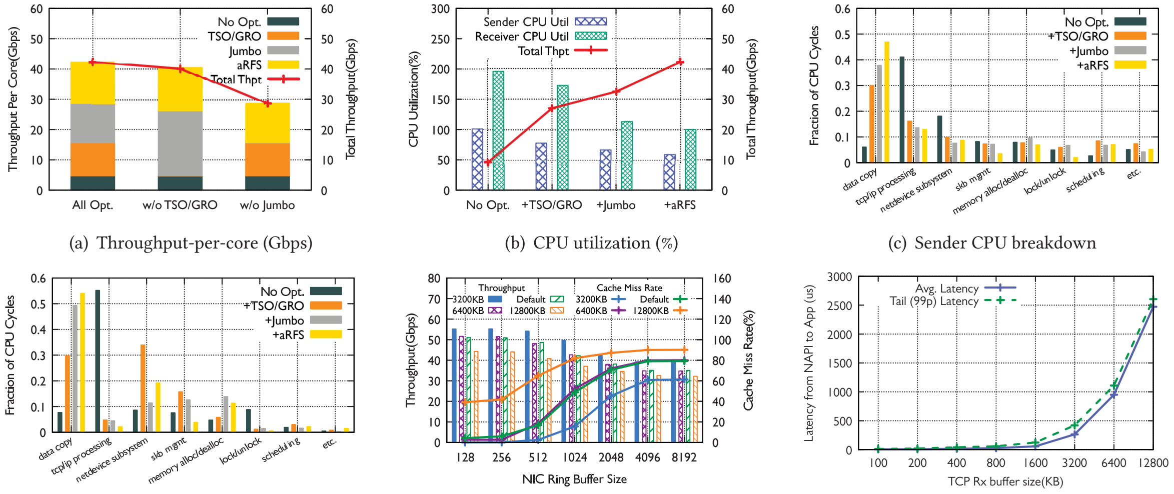
We start with the case of a single flow between the two servers, each running an application on a CPU core in the NIC-local NUMA node. We find that, unlike the Internet and early incarnations of datacenter networks where the throughput bottlenecks were primarily in the core of the network (since a single CPU was sufficient to saturate the access link bandwidth), high-bandwidth networks introduce new host bottlenecks even for the simple case of a single flow.

Before diving deeper, we make a note on our experimental configuration for the single flow case. When aRFS is disabled, obtaining stable and reproducible measurements is difficult since the default RSS mechanism uses hash of the 4-tuple to determine the core for IRQ processing (§2.1). Since the 4-tuple can change across runs, the core that performs IRQ processing could be: (1) the application core; (2) a core on the same NUMA node; or, (3) a core on a different NUMA node. The performance in each of these three cases is different, resulting in non-determinism. To ensure deterministic measurements, when aRFS is disabled, we model the worst-case scenario (case 3): we explicitly map the IRQs to a core on a NUMA node different from the application core. For a more detailed analysis of other possible IRQ mapping scenarios, see [7].

A single core is no longer sufficient. For 10 − 40Gbps access link bandwidths, a single thread was able to saturate the network bandwidth. However, such is no longer the case for high-bandwidth networks: as shown in Fig. 3(a), even with all optimization enabled, Linux network stack achieves throughput-per-core of ~42Gbps[3]. Both Jumbo frames[4] and TSO/GRO reduce the per-byte processing overhead as they allow each skb to bring larger payloads (up to 9000B and 64KB respectively). Jumbo frames are useful even when GRO is enabled, because the number of skbs to merge is reduced with a larger MTU size, thus reducing the processing overhead for packet aggregation in software. aRFS, along with DCA, generally

[3] We observe a maximum throughput-per-core of upto 55Gbps, either by tuning NIC Rx descriptors and TCP Rx buffer size carefully (See Fig. 3(e)), or using LRO instead of GRO (See [7]). However, such parameter tuning is very sensitive to the hardware setup, and so we leave them to their default values for all other experiments. Moreover, the current implementation of LRO causes problems in some scenarios as it might discard important header data, and so is often disabled in the real world [10]. Thus we use GRO as the receive offload mechanism for the rest of our experiments.

[4] Using larger MTU size (9000 bytes) as opposed to the normal (1500 bytes).



(a) Throughput-per-core (Gbps)

(b) CPU utilization (%)

(c) Sender CPU breakdown

(d) Receiver CPU breakdown

(e) Cache miss rate (%)

(f) Latency from NAPI to start of data copy

Figure 3: Linux network stack performance for the case of a single flow. (a) Each column shows throughput-per-core achieved for different combinations of optimizations. Within each column, optimizations are enabled incrementally, with each colored bar showing the incremental impact of enabling the corresponding optimization. **(b)** Sender and Receiver total CPU utilization as all optimizations are enabled incrementally. Independent of the optimizations enabled, receiver-side CPU is the bottleneck. **(c, d)** With all optimizations enabled, data copy is the dominant consumer of CPU cycles. **(e)** Increase in NIC ring buffer size and increase in TCP Rx buffer size result in increased cache miss rates and reduced throughput. **(f)** Network stack processing latency from NAPI to start of data copy increases rapidly beyond certain TCP Rx buffer sizes. See §3.1 for description.

improves throughput by enabling applications on the NIC-local NUMA node cores to perform data copy directly from L3 cache.

Receiver-side CPU is the bottleneck. Fig. 3(b) shows the overall CPU utilization at sender and receiver sides. Independent of the optimizations enabled, receiver-side CPU is the bottleneck. There are two dominant overheads that create the gap between sender and receiver CPU utilization: (1) data copy and (2) skb allocation. First, when aRFS is disabled, frames are DMAed to remote NUMA memory at the receiver; thus, data copy is performed across different NUMA nodes, increasing per-byte data copy overhead. This is not an issue on the sender-side since the local L3 cache is warm with the application send buffer data. Enabling aRFS alleviates this issue reducing receiver-side CPU utilization by as much as 2× (right-most bar in Fig. 3(b)) compared to the case when no optimizations are enabled; however, CPU utilization at the receiver is still higher than the sender. Second, when TSO is enabled, the sender is able to allocate large-sized skbs. The receiver, however, allocates MTU-sized skbs at device driver and then the skbs are merged at GRO layer. Therefore, the receiver incurs higher overheads for skb allocation.

Where are the CPU cycles going? Figs. 3(c) and 3(d) show the CPU usage breakdowns of sender- and receiver-side for each combination of optimizations. With none of the optimizations, CPU overheads mainly come from TCP/IP processing as per-skb processing overhead is high (here, skb size is 1500B at both sides[5]). When aRFS is disabled, lock overhead is high at the receiver-side because of the socket contention due to the application context thread (recv system call) and the interrupt context thread (softirq) attempting to access the same socket instance.

These packet processing overheads are mitigated with several optimizations: TSO allows using large-sized skb at the sender-side, reducing both TCP/IP processing and Netdevice subsystem overheads as segmentation is offloaded to the NIC (Fig. 3(c)). On the receiver-side, GRO reduces the CPU usage by reducing the number of skbs, passed to the upper layer, so TCP/IP processing and lock/unlock overheads are reduced dramatically, at the cost of increasing the overhead of the network device subsystem where GRO is performed (Fig. 3(d)). This GRO cost can be reduced by 66% by enabling Jumbo frames as explained above. These reduced packet processing overheads lead to throughput improvement, and the main overhead is now shifted to data copy, which takes almost 49% of total CPU utilization at the receiver-side when GRO and Jumbo frames are enabled.

Once aRFS is enabled, co-location of the application context thread and the IRQ context thread at the receiver leads to improved cache and NUMA locality. The effects of this are two-fold:

(1) Since the application thread runs on the same NUMA node as the NIC, it can now perform data copy directly from the L3 cache (DMAed by the NIC via DCA). This reduces the per-byte data copy overhead, resulting in higher throughput-per-core.
(2) skbs are allocated in the softirq thread and freed in the application context thread (once data copy is done). Since the two are co-located, memory deallocation overhead reduces. This is because page free operations to local NUMA memory are significantly cheaper than those for remote NUMA memory.

Even a single flow experiences high cache misses. Although aRFS allows applications to perform data copy from local L3 cache, we observe as much as 49% cache miss rate in this experiment. This is surprising since, for a single flow, there is no contention

[5] Linux kernel 4.17 onwards, GSO is enabled by default. We modified the kernel to disable GSO in "no optimization" experiments to evaluate benefits of skb aggregation.

for L3 cache capacity. To investigate this further, we varied various parameters to understand their effect on cache miss rate. Among our experiments, varying the maximum TCP receive window size, and the number of NIC Rx descriptors revealed an interesting trend.

Fig. 3(e) shows the variation of throughput and L3 cache miss rate with varying number of NIC Rx descriptors and varying TCP Rx buffer size[6]. We observe that, with increase in either of the number of NIC Rx descriptors or the TCP buffer size, the L3 cache miss increases and correspondingly, the throughput decreases. We have found two reasons for this phenomenon: (1) BDP values being larger than the L3 cache capacity; and (2) suboptimal cache utilization.

To understand the first one, consider an extreme case of large TCP Rx buffer sizes. In such a case, TCP will keep BDP worth of data in flight, where BDP is defined as the product of access link bandwidth and latency (both network and host latency). It turns out that large TCP buffers can cause a significant increase in host latency, especially when the core processing packets becomes a bottleneck. In addition to scheduling delay of IRQ context and application threads, we observe that each packet observe large queueing behind previous packets. We measure the delay between frame reception and start of data copy by logging the timestamp when NAPI processing for an skb happens, and the timestamp when the data copy of it starts, and measure the difference between the two. Fig. 3(f) shows the average and 99th percentile delays observed with varying TCP Rx buffer size. As can be seen, the delays rise rapidly with increasing TCP Rx buffer size beyond 1600KB. Given that DCA cache size is limited[7], this increase in latency has significant impact: since TCP buffers and BDP values are large, NIC always has data to DMA; thus, since the data DMAed by the NIC is not promptly copied to userspace buffers, it is evicted from the cache when NIC performs subsequent DMAs (if the NIC runs out of Rx descriptors, the driver replenishes the NIC Rx descriptors during NAPI polling). As a result, cache misses increase and throughput reduces. When TCP buffer sizes are large enough, this problem persists independent of NIC ring buffer sizes.

To understand the second reason, consider the other extreme where TCP buffer sizes are small but NIC ring buffer sizes are large. We believe cache misses in this case might be due to an imperfect cache replacement policy and/or cache's complex addressing, resulting in suboptimal cache utilization; recent work has observed similar phenomena, although in a different context [15, 39]. When there are a large number of NIC Rx descriptors, there is a correspondingly larger number of memory addresses available for the NIC to DMA the data. Thus, even though the total amount of in-flight data is smaller than the cache capacity, the likelihood of a DCA write evicting some previously written data increases with the number of NIC Rx descriptors. This limits the effective utilization of cache capacity, resulting in high cache miss rates and low throughput-per-core.

Between these two extremes, both of the factors contribute to the observed performance in Fig. 3(e). Indeed, in our setup, DCA cache capacity is ~3MB and hence TCP buffer size of 3200KB and fewer than 512 NIC Rx descriptors (512 × 9000 bytes ≈ 4MB) delivers the optimal single-core throughput of ~55Gpbs. An interesting observation here is that the default auto-tuning mechanism used in the Linux kernel network stack today is unaware of DCA effects, and ends up overshooting beyond the optimal operating point.

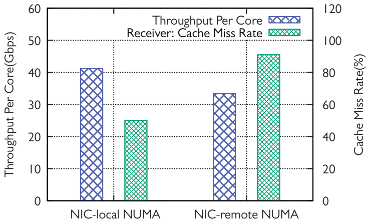


Figure 4: Linux network stack performance for the case of a single flow on NIC-remote NUMA node. When compared to the NIC-local NUMA node case, single flow throughput-per-core drops by ~20%.

DCA limited to NIC-local NUMA nodes. In our analysis so far, the application was run on a CPU core on the NIC-local NUMA node. We now examine the impact of running the application on a NIC-remote NUMA node for the same single flow experiment. Fig. 4 shows the resulting throughput-per-core and L3 cache miss rate relative to the NIC-local case (with all optimizations enabled in both cases). When the application runs on NIC-remote NUMA node, we see a significant increase in L3 cache miss rate and ~20% drop in throughput-per-core. Since aRFS is enabled, the NIC DMAs frames to the target CPU's NUMA node memory. However, because the target CPU core is on a NIC-remote NUMA node, DCA is unable to push the DMAed frame data into the corresponding L3 cache [25]. As a result, cache misses increase and throughput-per-core drops.

3.2 Increasing Contention via One-to-one

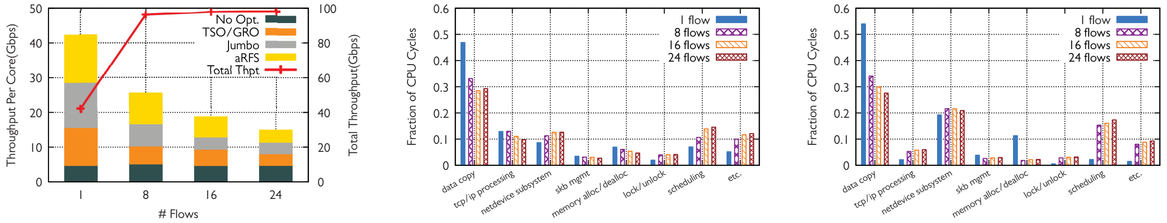
We now evaluate the Linux network stack with higher contention for the network bandwidth. Here, each sender core sends a flow to one unique receiver core, and we increase the number of core/flows from 1 to 24. While each flow still has the entire host core for itself, this scenario introduces two new challenges compared to the single-flow case: (1) network bandwidth becomes saturated as multiple cores are used; and (2) flows run on both NIC-local and NIC-remote NUMA nodes (our servers have 6 cores on each NUMA node).

Similar to §3.1, to obtain deterministic measurements when aRFS is disabled, we explicitly map IRQs for individual applications to a unique core on a different NUMA node.

Host optimizations become less effective with increasing number of flows. Fig. 5(a) shows that, as the number of flows increases, throughput-per-core decreases by 64% (i.e., 15Gbps at 24 flows), despite each core processing only a single flow. This is because of reduced effectiveness of all optimizations. In particular, when compared to the single flow case, the effectiveness of aRFS reduces by as much as 75% for the 24-flow case; this is due to reduced L3 cache locality for data copy for NIC-local NUMA node cores (all cores share L3 cache), and also due to some of the flows running on NIC-remote NUMA nodes (that cannot exploit DCA, see §3.1, Fig. 4). The effectiveness of GRO also reduces: since packets at the receiver are now interleaved across flows, there are fewer opportunities for aggregation; this will become far more prominent in the all-to-all case, and is discussed in more depth in §3.5.

[6] The kernel uses an auto-tuning mechanism for the TCP Rx socket buffer size with the goal of maximizing throughput. In this experiment, we override the default auto-tuning mechanism by specifying an Rx buffer size.

[7] DCA can only use 18% (~3 MB) of the L3 cache capacity in our setup.

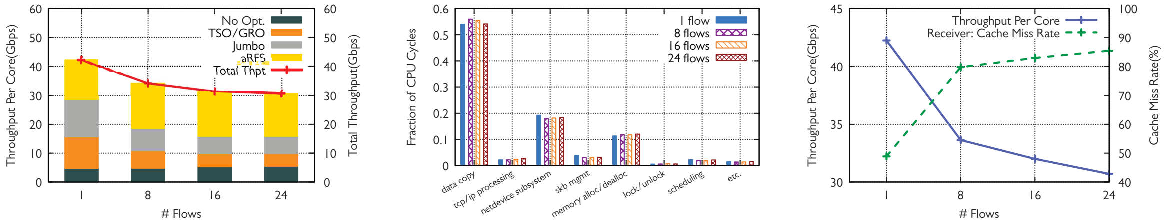


(a) Throughput-per-core (Gbps)

(b) Sender CPU breakdown

(c) Receiver CPU breakdown

Figure 5: Linux network stack performance for one-to-one traffic pattern. (a) Each column shows throughput-per-core achieved for different number of flows. At 8 flows, the network is saturated, however, throughput-per-core decreases with more flows. **(b, c)** With all optimizations enabled, as the number of flows increase, the fraction of CPU cycles spent in data copy decreases. On the receiver-side, network saturation leads to lower memory management overhead (due to better page recycling) and higher scheduling overhead (due to frequent idling). The overall receiver-side CPU utilizations for x= 1, 8, 16 and 24 cases are, 1, 3.75, 5.21 and 6.58 cores, respectively. See §3.2 for description.



(a) Throughput-per-core (Gbps)

(b) Receiver CPU breakdown

(c) L3 cache miss rate (%)

Figure 6: Linux network stack performance for incast traffic pattern. (a) Each column shows throughput-per-core for different number of flows (receiver core is bottlenecked in all cases). Total throughput decreases with increase in the number of flows. **(b)** With all optimizations enabled, the fraction of CPU cycles used by each component does not change significantly with number of flows. See [7] for sender-side CPU breakdown. **(c)** Receiver-side cache miss rate increases with number of flows, resulting in higher per-byte data copy overhead, and reduced throughput-per-core. See §3.3 for description.

Processing overheads shift with network saturation. As shown in Fig. 5(a), at 8 flows, the network link becomes the bottleneck, and throughput ends up getting fairly shared among all cores. Fig. 5(c) shows that bottlenecks shift in this regime: scheduling overhead increases and memory management overhead decreases. Intuitively, when the network is saturated, the receiver cores start to become idle at certain times—threads repeatedly go to sleep while waiting for data, and wake up when new data arrives; this results in increased context switching and scheduling overheads. This effect becomes increasingly prominent with increase in number of flows (Fig. 5(b), Fig. 5(c)), as the CPU utilization per-core decreases.

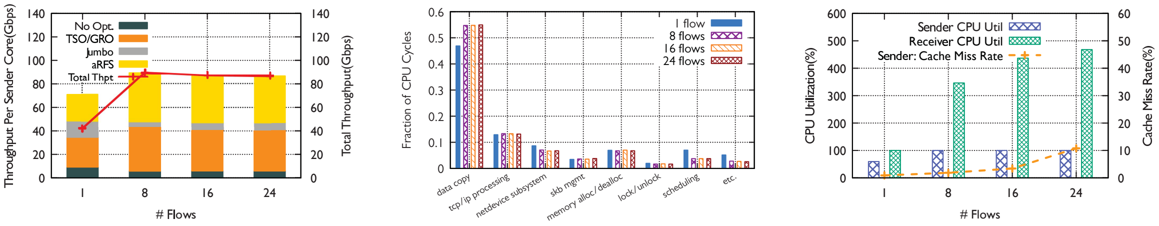
To understand reduction in memory alloc/dealloc overheads, we observe that the kernel page allocator maintains per-core `pageset` that includes a certain number of free pages. Upon an allocation request, pages can be fetched directly from the `pageset`, if available; otherwise the global free-list needs to be accessed (which is a more expensive operation). When multiple flows share the access link bandwidth, each core serves relatively less amount of traffic compared to the single flow case. This allows used pages to be recycled back to the `pageset` before it becomes empty, hence reducing the memory allocation overhead (Fig. 5(c)).

3.3 Increasing Receiver Contention via Incast

We now create additional contention at the receiver core using an incast traffic pattern, varying number of flows from 1 to 24 (each using a unique core at the sender). Compared to previous scenarios, this scenario induces higher contention for (1) CPU resources such as L3 cache and (2) CPU scheduling among application threads. We discuss how these changes affect the network processing overheads.

Per-byte data copy overhead increases with increasing flows per-core. Fig. 6(a) shows that throughput-per-core decreases with increase in number of flows, observing as much as ∼19% drop with 8 flows when compare to the single-flow case. Fig. 6(b) shows that the CPU breakdown does not change significantly with increasing number of flows, implying that there is no evident shift in CPU overheads. Fig. 6(c) provides some intuition for the root cause of the throughput-per-core degradation. As number of flows per core increases at the receiver side, applications for different flows compete for the same L3 cache space resulting in increased cache miss rate (the miss rate increases from 48% to 78%, as the number of flows goes from 1 to 8.). Among other things, this leads to increased per-byte data copy overhead and reduced throughput-per-core. As shown in Fig. 6(c), the increase in L3 cache miss rate with increasing flows correlates well with degradation in throughput-per-core.

Sender-driven nature of TCP precludes receiver-side scheduling. Higher cache contention observed above is the result of multiple active flows on the same core. While senders could potentially reduce such contention using careful flow scheduling, the issue at the receiver side is fundamental: the sender-driven nature of the TCP protocol precludes the receiver to control the number of active flows per core, resulting in unavoidable CPU inefficiency. We believe receiver-driven protocols [18, 35] can provide such control to the receiver, thus enabling CPU-efficient transport designs.



(a) Throughput-per-core (Gbps)

(b) Sender CPU breakdown

(c) CPU utilization (%)

Figure 7: Linux network stack performance for outcast traffic pattern. (a) Each column shows throughput-per-*sender*-core achieved for different number of flows, that is the maximum throughput sustainable using a single sender core (we ignore receiver core utilization here). Throughput-per-sender-core increases from 1 to 8 flows, and then decreases as the number of flows increases. **(b)** With all optimizations enabled, as the number of flows increases from 1 to 8, data copy overhead increases but does not change much when the number of flows is increased further. Refer to [7] for receiver-side CPU breakdown. **(c)** For 1 flow, sender-side CPU is underutilised. Sender-side cache miss rate increases slightly as the number of flows increases from 8 to 24, increasing the per-byte data copy overhead, and there is a corresponsing decrease in throughput-per-core. See §3.4 for description.

3.4 Increasing Sender Contention via Outcast

All our experiments so far result in receiver being the bottleneck. To evaluate sender-side processing pipeline, we now use an outcast scenario where a single sender core transmits an increasing number of flows (1 to 24), each to a unique receiver core. To understand the efficiency of sender-side processing pipeline, this subsection focuses on throughput-per-*sender*-core: that is, the maximum throughput achievable by a single sender core.

Sender-side processing pipeline can achieve up to 89Gbps per core. Fig. 7(a) shows that, with increase in number of flows from 1 to 8, throughput-per-sender-core increases significantly enabling total throughput as high as ∼89Gbps; in particular, throughput-per-sender-core is 2.1× when compared to throughput-per-receiver-core in the incast scenario (§3.3). This demonstrates that, in today's Linux network stack, sender-side processing pipeline is much more CPU-efficient when compared to receiver-side processing pipeline. We briefly discuss some insights below.

The first insight is related to the efficiency of TSO. As shown in Fig. 7(a), TSO in the outcast scenario contributes more to throughput-per-core improvements, when compared to GRO in the incast scenario (§3.3). This is due to two reasons. First, TSO is a hardware offload mechanism supported by the NIC; thus, unlike GRO which is software-based, there are no CPU overheads associated with TSO processing. Second, unlike GRO, the effectiveness of TSO does not degrade noticeably with increasing number of flows since data from applications is always put into 64KB size `skbs` independent of the number of flows. Note that Jumbo frames do not help over TSO that much compared to the previous cases as segmentation is now performed in the NIC.

Second, aRFS continues to provide significant benefits, contributing as much as ∼46% of the total throughput-per-sender-core. This is because, as discussed earlier, L3 cache at the sender is always warm: while cache miss rate increases slightly with larger number of flows, the absolute number remains low (∼11% even with 24 flows); furthermore, outcast scenario ensures that not too many flows compete for the same L3 cache at the receiver (due to receiver cores distributed across multiple NUMA nodes). Fig. 7(b) shows that data copy continues to be the dominant CPU consumer, even when sender is the bottleneck.

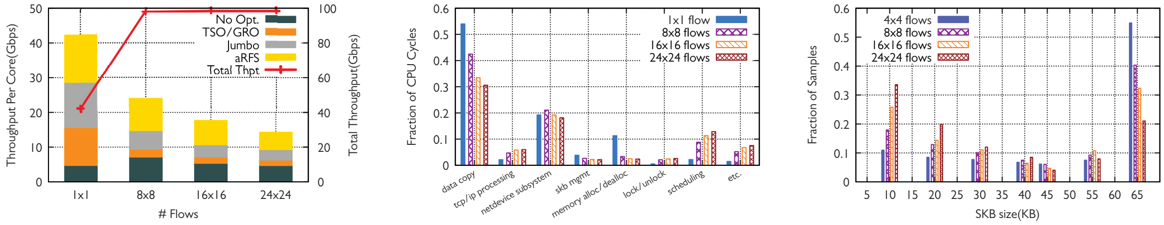
3.5 Maximizing Contention with All-to-All

We now evaluate Linux network stack performance for all-to-all traffic patterns, where each of x sender cores transmit a flow to each of the x receiver cores, for x varying from 1 to 24. In this scenario, we were unable to explicitly map IRQs to specific cores because, for the largest number of flows (576), the number of flow steering entries requires is larger than what can be installed on our NIC. Nevertheless, even without explicit mapping, we observed reasonably deterministic results for this scenario since the randomness across a large number of flows averages out.

Fig. 8(a) shows that throughput-per-core reduces by ∼67% going from 1 × 1 to 24 × 24 flows, due to reduced effectiveness of all optimizations. The benefits of aRFS drop by ∼64%, almost the same as observed in the one-to-one scenario (§3.2). This is unsurprising, given the lack of cache locality for cores in non-NIC-local NUMA nodes, and given that cache miss rate is already abysmal (as discussed in §3.2). Increasing the number of flows per core on top of this does not make things worse in terms of cache miss rate.

Per-flow batching opportunities reduce due to large number of flows. Similar to the one-to-one case, the network link becomes the bottleneck in this scenario, resulting in fair-sharing of bandwidth among flows. Since there are a large number of flows (*e.g.*, 24×24 with 24 cores), each flow achieves very small throughput (or alternatively, the number of packets received for any flow in a given time window is very small). This results in reduced effectiveness of optimizations like GRO (that operate on a per-flow basis) since they do not have enough packets in each flow to aggregate. As a result, upper layers receive a larger number of smaller `skbs`, increasing packet processing overheads.

Fig. 8(c) shows the distribution of `skb` sizes (post-GRO) for varying number of flows. We see that as the number of flows increase, the average `skb` size reduces, leading to our argument above about the reduced effectiveness of GRO. We note that the above phenomenon is not unique to the all-to-all scenario: the number of flows sharing a bottleneck resource also increase in the incast and one-to-one scenarios. Indeed, this effect would also be present in those scenarios, however the total number of flows in those cases is not large enough to make these effects noticeable (max of 24 flows in incast and one-to-one versus 24 × 24 flows in all-to-all).

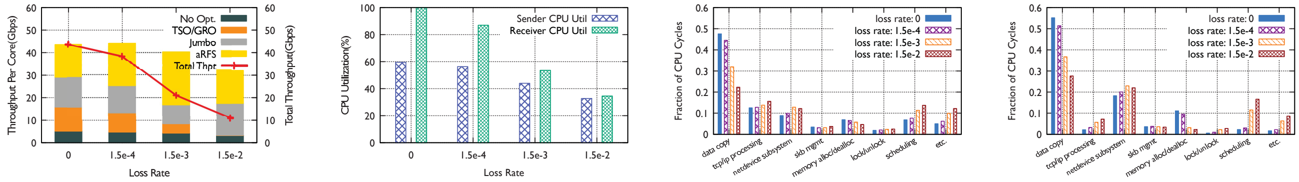


(a) Throughput-per-core (Gbps)

(b) Receiver CPU breakdown

(c) `skb` size distribution

Figure 8: Linux network stack performance for all-to-all traffic pattern. (a) Each column shows throughput-per-core achieved for different number of flows. With 8 × 8 flows, the network is fully saturated. Throughput-per-core decreases as the number of flows increases. **(b)** With all optimizations enabled, as the number of flows increase, the fraction of CPU cycles spent in data copy decreases. On the receiver-side, network saturation leads to lower memory management overhead (due to better page recycling) and higher scheduling overhead (due to frequent idling and greater number of threads per core.). TCP/IP processing overhead increases due to smaller `skb` sizes. The overall receiver-side CPU utilizations for x= 1 × 1, 8 × 8, 16 × 16 and 24 × 24 are 1, 4.07, 5.56 and 6.98 cores, respectively. See [7] for sender-side CPU breakdown. **(c)** The fraction of 64KB `skbs` after GRO decreases as the number of flows increases because the larger number of flows prevent effective aggregation of received packets. See §3.5 for description.



(a) Throughput-per-core (Gbps)

(b) CPU Utilization

(c) Sender CPU breakdown

(d) Receiver CPU breakdown

Figure 9: Linux network stack performance for the case of a single flow, with varying packet drop rates. (a) Each column shows throughput-per-core achieved for a specific packet drop rate. Throughput-per-core decreases as the packet drop rate increases. **(b)** As the packet drop rate increases, the gap between sender and receiver CPU utilisation decreases because the sender spends more cycles for retransmissions. **(c, d)** With all optimizations enabled, as the packet drop rate increases, the overhead of TCP/IP processing and netdevice subsystem increases. See §3.6 for description.

3.6 Impact of In-network Congestion

In-network congestion may lead to packet drops at switches, which in turn impacts both the sender and receiver side packet processing. In this subsection, we study the impact of such packet drops on CPU efficiency. To this end, we add a network switch between the two servers, and program the switch to drop packets randomly. We increase the loss rate from 0 to 0.015 in the single flow scenario from §3.1, and observe the effect on throughput and CPU utilization at both sender and receiver.

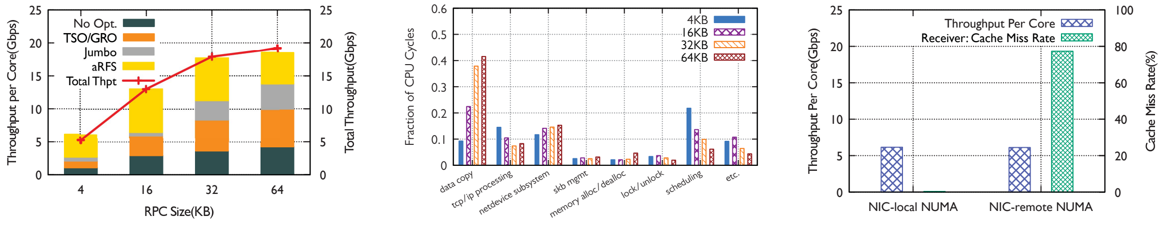
Impact on throughput-per-core is minimal. As shown in Fig. 9(a) the throughput-per-core decreases by ∼24% as the drop rate is increased from 0 to 0.015. Fig. 9(b) shows that the receiver-side CPU utilization decreases with increasing loss rate. As a result, the total throughput becomes lower than throughput-per-core, and the gap between the two increases. Interestingly, the throughput-per-core slightly increases when the loss rate goes from 0 to 0.00015. We observe that the corresponding receiver-side cache miss rate is reduced from 48% to 37%. This is because packet loss essentially reduces TCP sending rate, thus resulting in better cache hit rates at the receiver-side.

Figs. 9(c) and 9(d) show CPU profiling breakdowns for different loss rates. With increasing loss rate, at both sender and receiver, we see that the fraction of CPU cycles spent in TCP, netdevice subsystem, and other (etc.) processing increases, hence leading to fewer available cycles for data copy.

The minimal impact is due to increased ACK processing. Upon detailed CPU profiling, we found increased ACK processing and packet retransmissions to be the main causes for increased overheads. In particular:

- At the receiver, the fraction of CPU cycles spent in generating and sending ACKs increases by 4.87× (1.52% → 7.4%) as the loss rate goes from 0 to 0.015. This is because, when a packet is dropped, the receiver gets out-of-order TCP segments, and ends up sending duplicate ACKs to the sender. This contributes to an increase in both TCP and netdevice subsystem overheads.
- At the sender, the fraction of CPU cycles spent in processing ACKs increases by 1.45× (5.79% → 8.41%) as the loss rate goes from 0 to 0.015. This is because the sender has to process additional duplicate ACKs. Further, the fraction of CPU spent in packet retransmission operations increases by 1.34%. Both of these contribute to an increase in TCP and netdevice subsystem overheads, while the former contributes to increased IRQ handling (which is classified under "etc." in our taxonomy).

Sender observes higher impact of packet drops. Fig. 9(b) shows the CPU utilization at the sender and the receiver. As drop rates increase, the gap between sender and receiver utilization decreases, indicating that the increase in CPU overheads is higher at the sender side. This is due to the fact that, upon a packet drop, the sender is responsible for doing the bulk of the heavy lifting in terms of congestion control and retransmission of the lost packet.



(a) Throughput-per-core (Gbps)

(b) Server CPU breakdown

(c) NIC-remote NUMA effect (4KB)

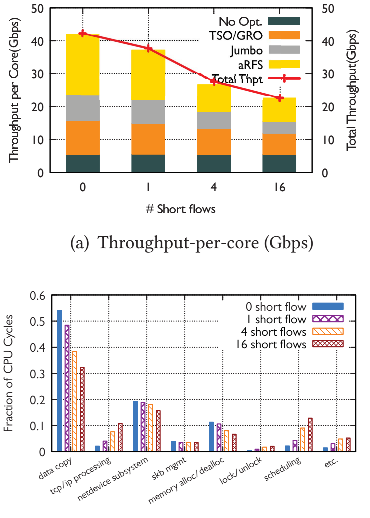
Figure 10: Linux network stack performance for short flow, 16:1 incast traffic pattern, with varying RPC sizes. (a) Each column shows throughput-per-core achieved for a specific RPC size. Throughput-per-core increases with increasing RPC size. For small RPCs, optimizations like GRO do not provide much benefit due to fewer aggregation opportunities. **(b)** With all optimizations enabled, data copy quickly becomes the bottleneck. The server-side CPU was completely utilized for all scenarios. See [7] for client-side CPU breakdown. **(c)** Unlike long flows, no significant throughput-per-core drop is observed even when application runs on NIC-remote NUMA node core at the server. See §3.7 for description.

3.7 Impact of Flow Sizes

We now study the impact of flow sizes on the Linux network stack performance. We start with the case of short flows: a ping-pong style RPC workload, with message sizes for both request/response being equal, and varying from 4KB to 64KB. Since a single short flow is unable to bottleneck CPU at either the sender or the receiver, we consider the incast scenario—16 applications on the sender send ping-pong RPCs to a single application on the receiver (the latter becoming the bottleneck). Following the common deployment scenario, each application uses a long-running TCP connection.

We also evaluate the impact of workloads that comprise of a mix of both long and short flows. For this scenario, we use a single core at both the sender and the receiver. We run a single long flow, and mix it with a variable number of short flows. We set the RPC size of short flows to 4KB.

DCA does not help much when workloads comprise of extremely short flows. Fig. 10(a) shows that, as expected, throughput-per-core increases with increase in flow sizes. We make several observations. First, as shown in Fig. 10(b), data copy is no longer the prominent consumer of CPU cycles for extremely small flows (*e.g.*, 4KB)—TCP/IP processing overhead is higher due to low GRO effectiveness (small flow sizes make it hard to batch `skbs`), and scheduling overhead is higher due to ping-pong nature of the workload causing applications to repeatedly block while waiting for data. Second, data copy not being the dominant consumer of CPU cycles for extremely short flows results in DCA not contributing to the overall performance as much as it did in the long-flow case: as shown in Fig. 10(c), while NIC-local NUMA nodes achieve significantly lower cache miss rates when compared to NIC-remote NUMA nodes, the difference in throughput-per-core is only marginal. Third, while DCA benefits reduce for extremely short flows, other cache locality benefits of aRFS still apply: for example, `skb` accesses during packet processing benefit from cache hits. However, these benefits are independent of the NUMA node on which the applications runs. The above three observations suggest interesting opportunities for orchestrating host resources between long and short flows: while executing on NIC-local NUMA nodes helps long flows significantly, short flows can be scheduled on NIC-remote NUMA nodes without any significant impact on performance; in addition, carefully scheduling the core across short flows sharing the core can lead to further improvements in throughput-per-core.



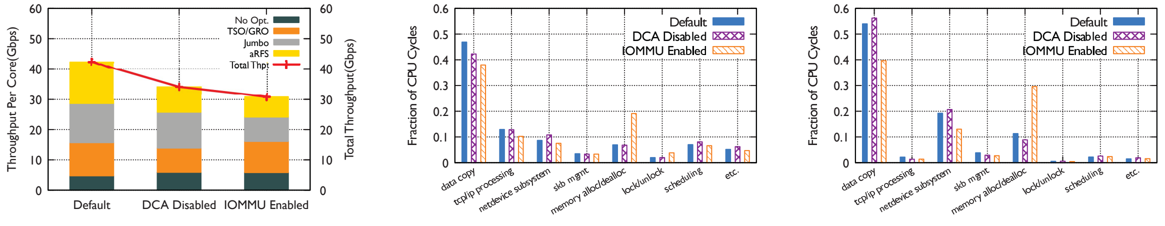
(a) Throughput-per-core (Gbps)

(b) Server CPU breakdown

Figure 11: Linux network stack performance for workloads that mix long and short flows on a single core. (a) Each column shows throughput-per-core achieved for different number of short flows colocated with a long flow. Throughput-per-core decreases with increasing number of short flows. **(b)** Even with 16 flows colocated with a long flows, data copy overheads dominate, but TCP/IP processing and scheduling overheads start to consume significant CPU cycles. The server-side CPU was completely utilized for all scenarios.; refer to [7] for client-side CPU breakdown. See §3.7 for description.

We note that all the observations above become relatively obsolete even with slight increase in flow sizes—with just 16KB RPCs, data copy becomes the dominant factor and with 64KB RPCs, the CPU breakdown becomes very similar to the case of long flows.

Mixing long and short flows considered harmful. Fig. 11(a) shows that, as expected, the overall throughput-per-core drops by ~43% as the number of short flows colocated with the long flow is increased from 0 to 16. More importantly, while throughput-per-core for a single long flow and 16 short flows is ~42Gbps (§3.1) and ~6.15Gbps in isolation (no mixing), it drops to ~20Gbps and ~2.6 Gbps, respectively when the two are mixed (48% and 42% reduction for long and short flows). This suggests that CPU-efficient network stacks should avoid mixing long and short flows on the same core.

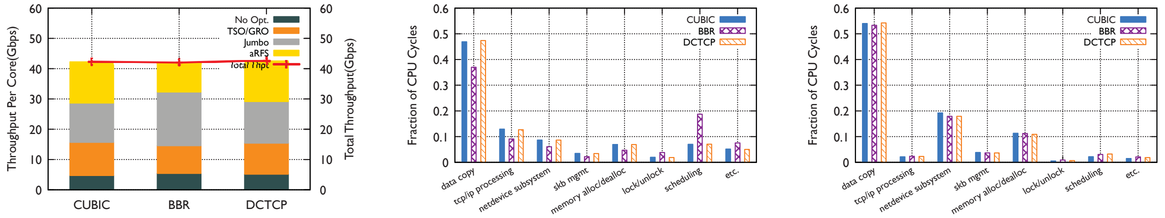


(a) Throughput-per-core (Gbps)

(b) Sender CPU breakdown

(c) Receiver CPU breakdown

Figure 12: Impact of DCA and IOMMU on Linux network stack performance. (a) Each column shows throughput-per-core achieved for different DCA and IOMMU configurations: Default has DCA enabled and IOMMU disabled. Either of disabling DCA or enabling IOMMU leads to decrease in throughput-per-core. **(b, c)** Disabling DCA does not cause a significant shift in CPU breakdown. Enabling IOMMU causes a significant increase in memory management overheads at both the sender and the recever. See §3.8 and §3.9 for description.



(a) Throughput-per-core (Gbps)

(b) Sender-side CPU breakdown

(c) Receiver CPU breakdown (IOMMU enabled)

Figure 13: Impact of congestion control protocols on Linux network stack performance. (a) Each column shows throughput-per-core achieved for different congestion control protocols. There is no significant change in throughput-per-core across protocols. **(b, c)** BBR causes a higher scheduling overhead on the sender-side. On the receiver-side, the CPU utilization breakdowns are largely similar. For all cases, receiver-side core is fully utilized for all protocols. See §3.10 for description.

3.8 Impact of DCA

All our experiments so far were run with DCA enabled (as is the case by default on Intel Xeon processors). To understand the benefits of DCA, we now rerun the single flow scenario from §3.1, but with DCA disabled. Fig. 12(a) shows the throughput-per-core without DCA relative to the scenario with DCA enabled (Default), as each of the optimizations are incrementally enabled. Unsurprisingly, with all optimizations enabled, we observe a 19% degradation in throughput-per-core when DCA is disabled. In particular, we see a ~50% reduction in the effectiveness of aRFS; this is expected since disabling DCA reduces the data copy benefits of NIC DMAing the data directly into the L3 cache. The other benefits of aRFS (§3.1) still apply. Without DCA, the receiver-side remains the bottleneck, and we do not observe any significant shift in the CPU breakdowns at sender and receiver (Figs. 12(b) and 12(c)).

3.9 Impact of IOMMU

IOMMU (IO Memory Management Unit) is used in virtualized environments to efficiently virtualize fast IO devices. Even for non-virtualized environments, they are useful for memory protection. With IOMMU, devices specify virtual addresses in DMA requests which the IOMMU subsequently translates into physical addresses while implementing memory protection checks. By default, the IOMMU is disabled in our setup. In this subsection, we study the impact of IOMMU on Linux network stack performance for the single flow scenario (§3.1).

The key take-away from this subsection is that IOMMU, due to increased memory management overheads, results in significant degradation in network stack performance. As seen in Fig. 12(a), enabling IOMMU reduces throughput-per-core by 26% (compared to Default). Figs. 12(b) and 12(c) show the core reason for this degradation: memory alloc/dealloc becoming more prominent in CPU consumption at both sender and receiver (now consuming 30% of CPU cycles at the receiver). This is because of two additional per-page operations required by IOMMU: (1) when the NIC driver allocates new pages for DMA, it has to also insert these pages into the device's pagetable (domain) on the IOMMU; (2) once DMA is done, the driver has to unmap those pages. These two additional per-page operations result in increased overheads.

3.10 Impact of Congestion control protocols

Our experiments so far use TCP CUBIC, the default congestion control algorithm in Linux. We now study the impact of congestion control algorithms on network stack performance using two other popular algorithms implemented in Linux, BBR [8] and DCTCP [1], again for the single flow scenario (§3.1). Fig. 13(a) shows that choice of congestion control algorithm has minimal impact on throughput-per-core. This is because, as discussed earlier, receiver-side is the core throughput bottleneck in high-speed networks; all these algorithms being "sender-driven", have minimal difference in the receiver-side logic. Indeed, the receiver-side CPU breakdowns are largely the same for all protocols (Fig. 13(c)). BBR has relatively higher scheduling overheads on the sender-side (Fig. 13(b)); this is because BBR uses pacing for rate control (with qdisc) [42], and repeated thread wakeups when packets are released by the pacer result in increased scheduling overhead.

4 FUTURE DIRECTIONS

We have already discussed several immediate avenues of future research in individual subsections—*e.g.*, optimizations to today's Linux network stack (*e.g.*, independent scaling of each processing layer in the stack, rethinking TCP auto-tuning mechanisms for receive buffer sizing, window/rate mechanisms incorporating host bottlenecks, etc.), extensions to DCA (*e.g.*, revisiting L3 cache management, support for NIC-remote NUMA nodes, etc.) and, in general, the idea of considering host bottlenecks when designing network stacks for high-speed networks. In this section, we outline a few more forward-looking avenues of future research.

Zero-copy mechanisms. The Linux kernel has recently introduced new mechanisms to achieve zero-copy transmission and reception on top of the TCP/IP stack:

- For zero-copy on the sender-side, the kernel now has MSG_ZEROCOPY feature [11] (since kernel 4.14), which pins application buffers upon a `send` system call, allowing the NIC to directly fetch this data through DMA reads.
- For zero-copy on the receiver-side, the kernel now supports a special `mmap` overload for TCP sockets [12] (since kernel 4.18). This implementation enables applications to obtain a virtual address that is mapped by the kernel to the physical address where the NIC DMAs the data.

Some specialized applications [13, 26] have demonstrated achieving $\sim$100Gbps of throughput-per-core using the sender-side zero-copy mechanism. However, as we showed in §3, receiver is likely to be the throughput bottleneck for many applications in today's Linux network stack. Hence, it is more crucial to eliminate data copy overheads on the receiver-side. Unfortunately, the above receiver-side zero-copy mechanism requires changes in the memory management semantics, and thus requires non-trivial application-layer modifications. Linux eXpress Data Path (XDP) [23] offers zero copy operations for applications that use AF_XDP socket [29] (introduced in kernel 4.18), but requires reimplementation of the entire network and transport protocols in the userspace. It would be interesting to explore zero-copy mechanisms that do not require application modifications and/or reimplementation of network protocols; if feasible, such mechanisms will allow today's Linux network stack to achieve 100Gbps throughput-per-core with minimal or no modifications.

CPU-efficient transport protocol design. The problem of transport design has traditionally focused on designing congestion and flow control algorithms to achieve a multi-objective optimization goal (*e.g.*, a combination of objectives like low latency, high throughput, etc.). This state of affairs is because, for the Internet and for early incarnations of datacenter networks, performance bottlenecks were primarily in the core of the network. Our study suggests that this is no longer the case: adoption of high-bandwidth links shifts performance bottlenecks to the host. Thus, future protocol designs should explicitly orchestrate host resources (just like they orchestrate network resources today), *e.g.*, by taking not just traditional metrics like latency and throughput into account, but also available cores, cache sizes and DCA capabilities. Recent receiver-driven protocols [18, 35] have the potential to enable such fine-grained orchestration of both the sender and the receiver resources.

Rearchitecting the host stack. We discuss two directions in relatively clean-slate design for future network stacks. First, today's network stacks use a fairly static packet processing pipeline for each connection—the entire pipeline (buffers, protocol processing, host resource provisioning, etc.) is determined at the time of socket creation, and remains unchanged during the socket lifetime, independent of other connections and their host resource requirements. This is one of the core reasons for the many bottlenecks identified in our study: when the core performing data copy becomes the bottleneck for long flows, there is no way to dynamically scale the number of cores performing data copy; even if short flows and long flows have different bottlenecks, the stack uses a completely application-agnostic processing pipeline; and, there is no way to dynamically allocate host resources to account for changes in contention upon new flow arrivals. As performance bottlenecks shift to hosts, we should rearchitect the host network stack to achieve a design that is both more dynamic (allows transparent and independent scaling of host resources to individual connections), and more application-aware (exploits characteristics of applications colocated on a server to achieve improved host resource orchestration).

The second direction relates to co-designing CPU schedulers with the underlying network stack. Specifically, CPU schedulers in operating systems have traditionally been designed independent of the network stack. This was beneficial for independent evolution of the two layers. However, with increasingly many distributed applications and with performance bottlenecks shifting to hosts, we need to revisit such a separation. For instance, our study shows that network-aware CPU scheduling (*e.g.*, scheduling applications that generate long flows on NIC-local NUMA node, scheduling long-flow and short-flow applications on separate CPU cores, etc.) has the potential to lead to efficient host stacks.

5 CONCLUSION

We have demonstrated that recent adoption of high-bandwidth links in datacenter networks, coupled with relatively stagnant technology trends for other host resources (*e.g.*, core speeds and count, cache sizes, etc.), mark a fundamental shift in host network stack bottlenecks. Using measurements and insights for Linux network stack performance for 100Gbps links, our study highlights several avenues for future research in designing CPU-efficient host network stacks. These are exciting times for networked systems research—with emergence of Terabit Ethernet, the bottlenecks outlined in this study are going to become even more prominent, and it is only by bringing together operating systems, computer networking and computer architecture communities that we will be able to design host network stacks that overcome these bottlenecks. We hope our work will enable a deeper understanding of today's host network stacks, and will guide the design of not just future Linux kernel network stack, but also future network and host hardware.

ACKNOWLEDGMENTS

We thank our shepherd, Neil Spring, SIGCOMM reviewers, Shrijeet Mukherjee, Christos Kozyrakis and Amin Vahdat for their insightful feedback. This work was supported by NSF grants CNS-1704742 and CNS-2047283, a Google faculty research scholar award and a Sloan fellowship. This work does not raise any ethical concerns.

REFERENCES

[1] 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*.

[2] Amazon. 2021. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. (2021).

[3] 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*.

[4] 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*.

[5] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *IMC*.

[6] Zhan Bokai, Yu Chengye, and Chen Zhonghe. 2005. TCP/IP Offload Engine (TOE) for an SOC System. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/dc/_3_3-2005_taiwan_3rd_chengkungu-web.pdf. (2005).

[7] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. https://github.com/Terabit-Ethernet/terabit-network-stack-profiling. (2021).

[8] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, September-October (2016), 20 – 53.

[9] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *IEEE/ACM MICRO*.

[10] Jonathan Corbet. 2009. JLS2009: Generic receive offload. https://lwn.net/Articles/358910/. (2009).

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

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

[13] Patrick Dehkord. 2019. NVMe over TCP Storage with SPDK. https://ci.spdk.io/download/events/2019-summit/(Solareflare)+NVMe+over+TCP+Storage+with+SPDK.pdf. (2019).

[14] 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/. (2021).

[15] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX ATC*.

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *USENIX NSDI*.

[17] The Linux Foundation. 2016. Linux Foundation DocuWiki: napi. https://wiki.linuxfoundation.org/networking/napi. (2016).

[18] 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*.

[19] Sebastien Godard. 2021. Performance monitoring tools for Linux. https://github.com/sysstat/sysstat. (2021).

[20] Brendan Gregg. 2020. Linux perf Examples. http://www.brendangregg.com/perf.html. (2020).

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

[22] HewlettPackard. 2021. Netperf. https://github.com/HewlettPackard/netperf. (2021).

[23] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *ACM CoNEXT*.

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

[25] 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. (2012).

[26] Intel. 2020. SPDK NVMe-oF TCP Performance Report. https://ci.spdk.io/download/performance-reports/SPDK_tcp_perf_report_2010.pdf. (2020).

[27] EunYoung Jeong, Shinae Woo, Muhammad 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*.

[28] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. 2009. The nature of data center traffic: measurements & analysis. In *IMC*.

[29] Magnus Karlsson and Björn Töpel. 2018. The Path to DPDK Speeds for AF XDP. In *Linux Plumbers Conference*.

[30] 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*.

[31] Yuliang Li, Rui Miao, Hongqiang Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*.

[32] 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*.

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

[34] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *ACM SIGCOMM*.

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

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

[37] 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. (2017).

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

[39] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *USENIX NSDI*.

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

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

[42] Neal Cardwell Yuchung Cheng. [n. d.]. Making Linux TCP Fast. "https://netdevconf.info/1.2/papers/bbr-netdev-1.2.new.new.pdf". ([n. d.]).

[43] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.