Understanding Host Network Stack Latency

Tianyu Zuo
University of Virginia

Jaehyun Hwang
Sungkyunkwan University

Ao Tang
Cornell University

Rachit Agarwal
Cornell University

Qizhe Cai
University of Virginia

Abstract

One of the most frequently cited flaws of Linux host network stacks is their high latency—recent studies show that the Linux network stack suffers from millisecond-scale tail latency. This has motivated clean-slate solutions such as userspace stacks and specialized host networking hardware, but without understanding the true root causes of Linux tail latency, these efforts risk repeating the same pitfalls.

This paper studies the root causes of tail latency in the Linux network stack and reaches a surprising conclusion: the dominant bottlenecks lie not in packet processing itself, but in how the host manages CPU resources for networking. Through extensive measurements, we identify three key factors: misattribution of processing time by CPU schedulers, limitations of CPU runtime as a fairness abstraction, and the inefficiency of interrupt tuning under unpredictable packet arrivals. We show that addressing these factors improves performance by up to 5.3× while maintaining similar throughput. Together, these results suggest that achieving low-latency networking requires rethinking host CPU scheduling and host–NIC interaction, providing new design directions for operating systems, network stacks, and future host networking 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:
Tianyu Zuo, Jaehyun Hwang, Ao Tang, Rachit Agarwal, and Qizhe Cai . 2026. Understanding Host Network Stack Latency. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3789240.3829115

1 Introduction

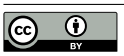
Today's most widely deployed host network stacks (*e.g.*, Linux) are seriously deficient along one or more dimensions. One of the most frequently cited flaws of these stacks is their high tail latency—recent research papers have demonstrated that Linux network stack can suffer from millisecond-scale tail latency, *e.g.*, [20, 22, 28, 35, 41, 44, 51, 53, 54, 57, 58, 60], to name a few[1]. While many of these studies reproduce the high tail latency phenomenon, they do not answer a basic question: "*Why* does the Linux network stack have high tail latency?"

Building an in-depth understanding of the answer to the above question is important due to at least three reasons. First, the observation of Linux network stack having high tail latency has spurred a large and active body of research in networking, systems, and computer architecture communities on designing clean-slate userspace network stacks and even specialized host network hardware that can achieve μs-scale tail latency; as these solutions start to mature, they may benefit from explicitly avoiding the mistakes made by the Linux network stack. Second, Linux is still one of the most widely deployed and used operating systems; understanding the root causes of high tail latency may allow us to explore improvements in Linux that may be useful for a large class of applications that use Linux. The third, and perhaps the most important, reason is also the simplest: pedagogical purposes.

We present in-depth measurements and insights into the root causes of high tail latency in the Linux kernel network stack[2]. For the problems we identify, we either propose concrete solutions or demonstrate the potential benefits of addressing them. Our key findings are:

In an isolated scenario, P99.9 latency is extremely low. Our evaluation suggests that, for a single-threaded application running in isolation, Linux network stack achieves P99.9 latency of ~14μs using the polling mode [26], and ~19μs in the non-polling CPU-efficient mode. This is comparable to the state-of-the-art userspace TCP stack TAS [41], which achieves ~9μs latency on its fast path (polling, no congestion) and falls back to a slower full-TCP path under congestion. Thus, we conclude that the root cause of high tail latency for Linux must be contention at host resources.

Are you sure the core of the problem is within the network stack? It's not! Most prior studies evaluate Linux using a setup with a large number of connections/threads [28, 41, 53] to conclude that Linux network stack fails to achieve μs-scale tail latency. Our measurements confirm the inflation but show it stems from the Linux CPU scheduler—not the network stack itself. The CPU scheduler aims to fairly share CPU time among same-priority threads by tracking each thread's virtual runtime, which reflects the CPU service it has received. Scheduling decisions are then based on virtual progress derived from this runtime—directly in CFS (i.e.,



SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829115

[1] As we discuss in §5, these works are complementary to the problem of designing CPU-efficient network stacks [24, 32, 36, 59] that primarily focus on improving throughput-per-core for individual network connection.
[2] We provide our measurement tools and an extended report covering more configurations and applications at https://github.com/Terabit-Ethernet/understand_latency.

Completely Fair Scheduler) and indirectly in EEVDF (i.e., Earliest Eligible Virtual Deadline First) via lag and virtual deadlines, and optionally in recent operator-defined eBPF schedulers [10]. For example, CFS prioritizes the thread with the smallest virtual runtime to ensure fairness. As a result, fair scheduling critically depends on accurate per-thread CPU runtime accounting. However, we find that the Linux scheduler charges softIRQ network processing time to the currently running thread, even when the packets are unrelated. This misattribution distorts CPU accounting, leads to unfair scheduling decisions, and inflates tail latency. We design a concise patch that corrects IRQ time accounting and reduces tail latency by up to 2.7× compared to the default Linux scheduler.

Runtime can lie—pick the right accounting unit as needed. Measuring runtime is widely regarded as the gold standard for ensuring fairness in CPU schedulers. However, runtime reflects not only executed instructions but also data locality—specifically, whether data comes from cache or main memory. As a result, even threads executing identical instructions can accumulate different runtimes, inflating tail latency. We observe persistent runtime gaps even with accurate IRQ accounting, because softIRQ processing pollutes a thread's execution state, causing threads that handle more softIRQ work to accumulate higher runtime. Fundamentally, fairness should rely on accounting units that abstract away noise and reflect meaningful progress among threads. For network-intensive applications, such units may include packets or requests/responses. We show that choosing appropriate accounting units reduces tail latency by up to 1.5× beyond accurate IRQ accounting and by up to 4.1× over the default Linux scheduler. While this does not constitute a complete scheduling solution, it demonstrates the potential benefit of supporting flexible accounting units in programmable scheduling frameworks, such as those based on eBPF. Such support could allow operators to explore fairness metrics that better align with application-level objectives.

Predictability is the missing piece in interrupt tuning. Beyond scheduling, the software–NIC interaction on packet arrival remains a long-standing challenge. Interrupt-based schemes often incur higher latency, while user-space stacks reduce latency via busy polling at the cost of CPU efficiency. Recent approaches such as DIM dynamically tune interrupt rates to balance latency and CPU efficiency based on current network load, but predicting future traffic from past history remains fundamentally limited due to bursty traffic patterns. Rather than asking how to better predict traffic, we ask whether traffic can be designed to be predictable. We show that predictable traffic fundamentally changes the effectiveness of interrupt tuning, enabling 1.28× extra latency reductions while maintaining throughput. This points to receiver-driven transport protocols [18, 23, 29, 33, 50] as a potential direction for addressing the interrupt-tuning problem, since receivers have full control of future packet arrivals.

Choose your parallelism carefully! Distributed applications seek both μs-scale tail latency and high throughput [27, 35, 44, 60, 66]. Achieving high throughput requires parallelism, either through many connections or multiple in-flight requests per connection. While prior work demonstrates tail-latency inflation using many connections, we find that maintaining multiple in-flight requests per connection better achieves both high throughput and low latency. As an example, our measurements suggest that with a single thread/connection and large number of in-flight requests, Linux is able to sustain 1.28 million 64B RPCs/second on a single core while maintaining P99.9 tail latency of 90μs; this is comparable to state-of-the-art userspace stacks [28, 41, 53] that achieve roughly the same throughput and ~100μs tail latency. We explain why this choice of parallelism matters in §3.3. Briefly, (1) using one thread/connection per core mitigates the unfairness discussed above, and (2) multiple in-flight requests on a single connection better exploit batching (*e.g.*, TSO/GRO), reducing per-request overhead. In practice, systems should use only a few connections per core whenever possible; when many connections are unavoidable, only a small subset should be active at any time to improve locality and reduce contention. Receiver-driven transport protocols [18, 23, 29, 33, 50] have the potential to provide this capability, as they give the receiver visibility into incoming traffic; together with local CPU availability, the receiver can precisely regulate the number of active flows.

Our measurements confirm these findings across a wide range of settings, varying core counts, connection counts, in-flight requests, and kernel versions, as well as real-world applications. We discuss the implications for transport protocols, operating systems, and hardware design throughout the paper. Before we dive deeper, we make an important note. There is a rich and active body of research on the design of userspace network stacks to achieve microsecond-scale tail latency and/or high throughput [22, 28, 32, 36, 38, 41, 53, 58, 59]. Our aim is not to compete with userspace stacks in terms of performance, but to deepen our understanding and push the performance limits of a well-established and widely used network stack. Given that a significant portion of cloud services still rely on Linux, we believe this is a worthy objective.

This work does not raise any ethical issues.

2 Measurement Methodology

This section outlines our measurement setup and methodology used to assess application performance on the Linux network stack.

Testbed. Since our measurements focus on host-side latency, we use a testbed with two servers directly connected via a 400Gbps link, each equipped with dual-socket Intel Xeon Gold 6530 CPUs (two NUMA nodes per socket), 32 cores per socket, 1024GB DRAM, 48KB/2MB/160MB L1 (data)/L2/L3 caches, and a ConnectX-7 NIC.

Linux kernel configuration. Both servers run Ubuntu 22.04. Kernel network stacks evolve rapidly, so studies like ours must fix specific versions to ensure consistent results. We use Linux kernel 5.10 with its default scheduler, CFS. While the Linux community has recently introduced a new scheduler, EEVDF [11], it is still under development, and our evaluation reveals fairness issues (§3.7). Accordingly, we use CFS (kernel 5.10) as the default throughout the paper, and include complementary EEVDF results using a recent kernel (version 6.12) in §3.7. We enable standard Linux network stack optimizations—TCP Segmentation Offload (TSO), Generic Receive Offload (GRO), jumbo frames (9000B), and accelerated Receive Flow Steering (aRFS), which steers IRQs to the application core to improve L3 cache locality and per-core performance [24, 63]. As in prior production-system work [48], we also enable hyper-threading.

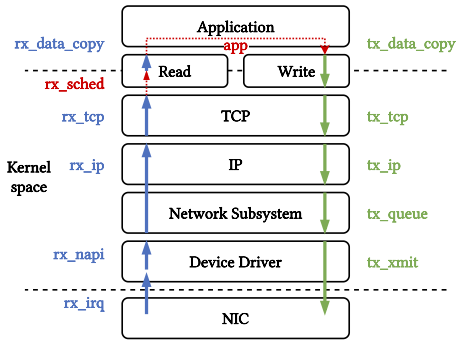


Figure 1: We measure per-request latency "breakdown" in the Linux network stack using 12 components, representing end-to-end data path. See §2 for more detailed description.

Component	Description	From	To
rx_irq	IRQ handling	NIC	skb alloc.
rx_napi	Rx NAPI processing	skb alloc.	Rx NAPI end
rx_ip	Rx IP processing	Netdev start	Rx IP end
rx_tcp	Rx TCP processing	Rx TCP start	Rx TCP end
rx_sched	App. waking-up	Wake-up signal	Read call resume
rx_data_copy	Rx data copy	Data read start	Data read end
app	App. proc. + syscall	Return to user	Write call
tx_data_copy	Tx data copy	Data write start	Data write end
tx_tcp	Tx TCP processing	Tx TCP start	Tx TCP end
tx_ip	Tx IP processing	Tx IP start	Tx IP end
tx_queue	Tx packet scheduling	qdisc proc. start	qdisc proc. end
tx_xmit	Packet transmitting	xmit start	xmit end

Table 1: Latency breakdown taxonomy. Each of the 12 components is mapped into one of the network stack layers along the end-to-end data path in Figure 1.

To minimize experimental noise, we disable `irqbalance`, which redistributes hardware interrupts among CPU cores [12], and load balancing, which moves applications based on CPU load.

DIM configuration. Modern NICs include Dynamic Interrupt Moderation (DIM), which adapts Rx interrupt rates to traffic load [52]. Enabling DIM reduces interrupts and improves throughput at high load but often hurts tail latency by delaying interrupts. Disabling DIM improves tail latency at low load with similar throughput, but reaches CPU bottlenecks earlier as load increases. Since we focus on peak achievable performance, across all latency–throughput curves in §3, we select the best-performing data points with DIM either enabled or disabled.

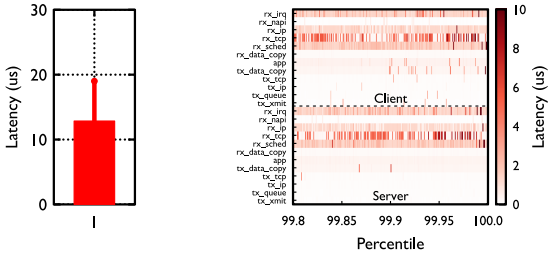
Measurement infrastructure. We generate a ping-pong style RPC workload using a customized version of Netperf [9], where each client–server thread pair maintains a dedicated TCP connection to send and receive 64B RPCs. While measuring the end-to-end latency, we break down the latency into 12 components as shown in Figure 1 and Table 1. We then carefully change the Linux kernel to keep and record timestamps within each socket buffer, `skb` (that represents each packet), to determine the time spent by the end-to-end request/response in each layer of the network stack. To enable per-`skb` measurement, our breakdown starts from `rx_irq` where a new `skb` is created for Rx processing. To obtain the NIC-side Rx timestamps, we enable hardware time stamping using `hwstamp_ctl` [8] and also use `phc2sys` [16] to synchronize the hardware and CPU clocks. To track the entire life cycle of the ping-pong RPC time breakdown, we need to retain both the Rx `skb` corresponding to the response and the Tx `skb` corresponding to the request on both the client and server sides. To achieve this, we store all the Rx timestamps in each Rx `skb` and copy them into the corresponding socket data structure before returning back to the userspace (before `app`). The time stamping resumes when the application enters the kernel through the `write` syscall and a new `skb` is created for Tx processing; we then copy all Rx timestamps stored in the socket into the newly created Tx `skb`, and record the subsequent Tx timestamps in the same Tx `skb`. When the packet is successfully transmitted (`tx_xmit`), we profile the latency breakdown with the timestamps obtained from Tx `skb`. To minimize the profiling overheads, (1) we sample the timestamps at regular intervals (*e.g.*, every 100 requests, a frequency that does not affect performance in our experiments) and (2) use `ftrace` [1], Linux's built-in kernel tracing framework, which logs profiling events into per-CPU buffers and avoids expensive synchronization or frequent user-kernel data transfers during execution. We perform this breakdown measurement at the client and server separately while generating the ping-pong workload.

To obtain per-RPC end-to-end latency breakdowns, we collect sample IDs and source/destination ports, enabling us to match client and server profiles for the same RPC even with multiple concurrent connections. We use the standard kernel APIs to obtain precise nanosecond-scale timestamps [13]. Despite our significant effort, some fundamental overheads (*e.g.*, writing timestamps to the `ftrace` ring buffer) are still inevitable in our profiling, leading to a mismatch (albeit, not significant) between the end-to-end latency reported by the application and the total sum of our breakdown components, especially when multiple RPC connections perform the latency profiling concurrently. Nevertheless, our latency breakdown is useful to identify the key bottlenecks with a fairly high accuracy. To give both clear intuition and correct latency performance, we present the latency breakdown result along with their end-to-end latency reported by the application in the paper.

3 Understanding Linux Latency

We now evaluate the application performance on top of Linux network stack for a variety of scenarios and present detailed insights on observed performance. We start by looking at the isolated case where a single application runs on a single CPU core[3], generating at most one request in flight at any time to measure the baseline latency (§3.1). Considering the application goal is to achieve low latency and high throughput, we increase the number of threads (§3.2) and the number of in-flight requests per thread (§3.3) to improve the throughput. We then repeat these scenarios on a multi-core environment (§3.4 and §3.5). Finally, we evaluate the application performance under the EEVDF scheduler (§3.7).

[3]In this paper, we use the terms "application" and "thread" interchangeably, and a CPU core indicates a physical core or its two logical cores.



(a) End-to-end latency **(b) Tail latency breakdown (P99.8–P100)**

Figure 2: Latency with no contention. (a) Linux achieves an average end-to-end latency (shown by bar) of 12.9μs and P99.9 tail latency (shown by top whisker) of 19μs; enabling busy polling reduces them to $\sim$ 11μs and $\sim$ 14μs at the cost of $\sim$ 2.28× higher CPU utilization. (b) Tail latency is dominated by receive-side processing and scheduling, including `rx_irq`, `rx_tcp`, `rx_ip`, and `rx_sched`. See §3.1 for more details.

3.1 No Contention

We start with the isolated case of a single thread that pingpongs a 64B request/response between the two servers.

An isolated thread can achieve low P99.9 tail latency. Figure 2 shows the end-to-end average and P99.9 tail latency, along with the tail latency breakdown (P99.8–P100) on both client and server sides, using a heatmap to attribute latency to individual components. The percentiles in the heatmap are ordered by the sum of all component contributions, rather than by any individual component. Therefore, each component's latency is not necessarily monotonic. Note that in this section we analyze the full tail region (P99.8–P100) rather than a single tail point. As component contributions vary substantially across samples (Figure 2(b)), a single point (*e.g.*, only P99.9) can be misleading and may not capture all sources of tail latency.

Overall, the P99.9 end-to-end tail latency achieved by Linux is 19μs, which aligns with a recent report [20], with a CPU utilization of 44.09% at the client and 43.56% at the server; we observe that the dominant components are (1) `rx_irq`, the delay from when a packet is received by the NIC to when its corresponding `skb` is allocated, which includes the overhead of the Rx interrupt handling, (2) `rx_ip`, the delay from when `skb` enters the Netdevice subsystem [24] to when the IP layer processing is completed, (3) `rx_tcp`, the delay for TCP stack processing, and (4) `rx_sched`, the delay from when a thread sleeping on a `read` call is woken up after data is received to when it is scheduled and starts executing.

For this isolated case, tail latency can be further improved by using busy-poll sockets [26] with 100% core utilization on both sides. Our experimental results (not shown in figures) demonstrate that busy-polling allows Linux to achieve P99.9 end-to-end latency of 14μs by effectively reducing the overhead from the `rx_irq` latency; in comparison, the state-of-the-art userspace stack, TAS, achieves P99.9 end-to-end tail latency of 9μs in the same isolated case for its fast path (that bypasses multiple stages within the packet processing pipeline, essentially assuming no network congestion and/or drops), merely 5μs lower than Linux. These measurements suggest that, in isolation, the Linux kernel stack already achieves low P99.9 latency; while further optimizations are possible, its contribution is only a small fraction of the latency introduced by the fabric.

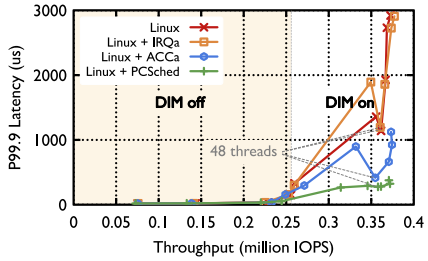


Figure 3: Application performance with increasing number of threads on a single core. We enable DIM for all four schemes after CPU becomes their bottleneck; all four schemes reach their performance knee points with 48 threads. Linux shows a P99.9 latency of 1139μs at the knee point. ACCa reduces it by ~2.7×, and using PCSched further lowers it to 276μs (~4.1× better than Linux). We also observe that P99.9 latency first increases and then decreases, which relates to the DIM tuning. After throughput reaches its peak, adding more threads may reduce throughput due to increased thread contention. See §3.2 for more discussion.

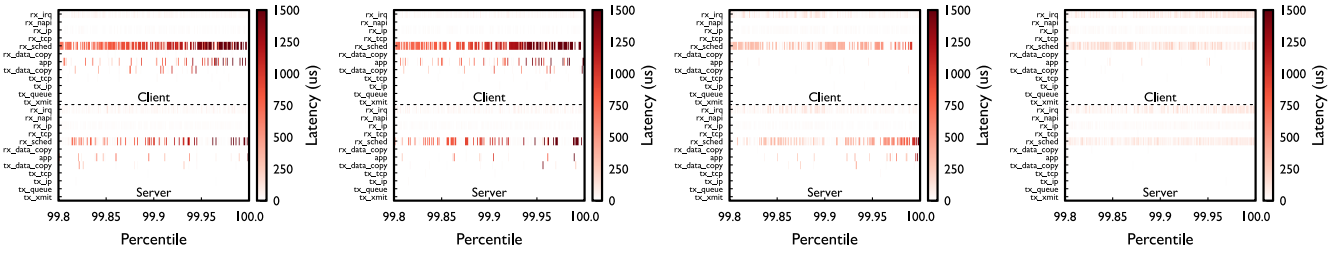
3.2 Contention on A Single Core

Next, we gradually increase the number of threads co-located on a single CPU core (i.e., across two logical cores evenly), generating one in-flight request per thread. This creates competition among the threads for the host resources.

`rx_sched` is the dominating factor in tail latency. Figure 3 shows that Linux experiences a sharp increase in the P99.9 tail latency ($\sim$ 1139μs) as the number of threads increases to 48, reaching the knee point with DIM enabled. Figure 4(a) shows that the main contributor to tail latency on both client and server is `rx_sched`—the delay between a sleeping `read` being woken by incoming data and the thread being scheduled and executing[4]. Moreover, the contribution of `rx_sched` to tail latency is orders of magnitude more than any other component.

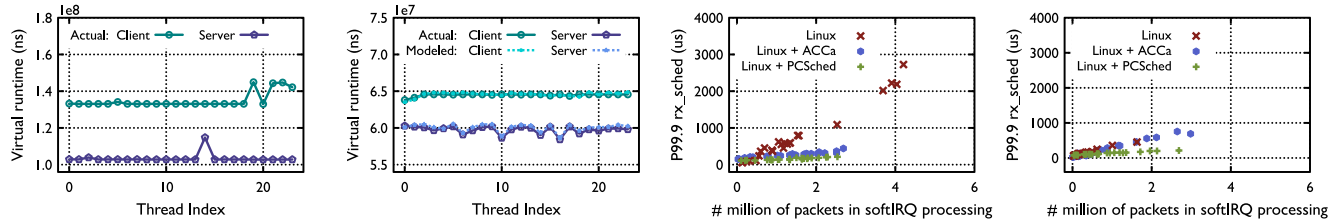
Inaccurate time accounting in CPU scheduler, not network stack, is the reason for high `rx_sched`. All prior studies that observe the high tail latency in the kernel network stack use the default Linux CPU scheduler (CFS or the latter EEVDF), designed to allocate equal CPU time among threads. We found that the interaction between the CPU scheduler and network stack processing leads to high `rx_sched` latency. In the scheduler, each thread maintains a virtual runtime that reflects the amount of CPU service it has received. Scheduling decisions are then made based on a notion of virtual progress derived from this virtual runtime (*e.g.*, directly in CFS, or via lag and virtual deadlines in EEVDF). In CFS (used in the experiment), it prioritizes the thread with the smallest virtual runtime for fair scheduling. This implies that threads with larger virtual runtime experience longer scheduling delays even after their data reaches TCP, resulting in high `rx_sched` latency. They are also more likely to be preempted, which inflates the latency observed in `app` and `tx_data_copy`. In fact, with 48 threads on the same

[4]Besides `rx_sched`, the `app` and `tx_data_copy` components also appear large, but this is mainly due to scheduler-induced preemption (*e.g.*, from interrupts or timers) rather than their own execution time. Thus, the high latency across all three components is still driven by CPU scheduling behavior. We still include these times to better align the breakdown with end-to-end tail latency.



(a) Linux (b) Linux + IRQa (c) Linux + ACCa (d) Linux + PCSched

Figure 4: P99.8–P100 latency breakdown at the knee point. (a) and (b) Scheduling latency (rx_sched), particularly on the client side, contributes substantially to tail latency in Linux, even when IRQ time accounting (IRQa) is enabled. (c) Our accurate time accounting patch (ACCa) substantially reduces rx_sched latency. (d) By scheduling packets based on the number of requests, PCSched attenuates nearly all rx_sched's impact on tail latency.



(a) Linux virtual runtime (b) Linux + ACCa virtual runtime (c) rx_sched vs. #processed, Client (d) rx_sched vs. #processed, Server

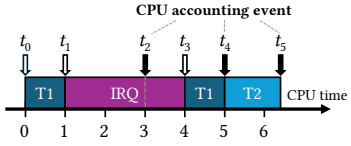
Figure 5: Understanding the causes of high rx_sched latency. (a) Under default Linux, virtual runtime gaps appear across threads; those with the largest virtual runtime also have the highest rx_sched latency. The client accumulates more runtime due to extra latency-record logic. (b) ACCa reduces the virtual runtime gap between threads; however, the gap still exists, which is related to the actual thread processing time differences in data copying and application processing across threads due to processor-side overheads. (c) and (d) show how the number of packets processed in softIRQ during a thread's execution correlates with rx_sched latency on the client and server. See §3.2 for more details.

core (24 per logical core), we observe large virtual-runtime gaps across threads after the experiment (Figure 5(a)). The four threads with the highest virtual runtime experience over 2000μs rx_sched latency on the client side. This is counterintuitive, since all threads run the same ping-pong workload, achieve similar throughput, and start with negligible runtime differences, so their virtual runtimes should converge.

We delve deeper to understand the cause of the large virtual runtime differences among the threads. We discover that inaccurate CPU accounting in the scheduler during softIRQ processing is one of the main reasons; the scheduler charges the CPU time of softIRQ processing to the current thread by default, irrespective of whether the processed packet belongs to that specific thread, unfairly increasing its virtual runtime. The problem is that some threads may experience more softIRQ events than the others as packet arrival events do not follow a uniform distribution. As a result, such *victim* threads with more softIRQ events (hence with larger virtual runtime) are likely to be treated less fairly. To make matters even worse, this runtime gap creates a positive feedback loop that amplifies the virtual runtime gap and can eventually become a sustained phenomenon: the thread with the largest virtual runtime runs only after all other threads finish. In a ping-pong workload, this means that while this max-runtime thread is running, peer threads on the other server—serving the remaining connections—are still actively processing and sending packets back. As a result, the victim thread receives bursts of incoming packets, handles more softIRQ work, and unfairly accumulates even more virtual runtime.

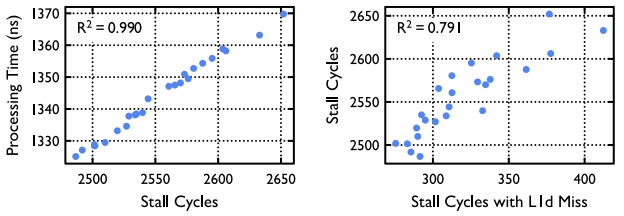
To validate our finding, we record the number of packets each thread processes during softIRQ handling and its corresponding tail rx_sched latency—which dominates end-to-end latency—using our modified Netfilter module. Our analysis shows that threads processing more packets within their allocated CPU time experience higher tail latency on both the client and server sides (Figure 5(c) and 5(d)). This observation aligns with prior work reporting similar CPU accounting issues in multi-container environments [42, 48].

Our patch with accurate time accounting reduces rx_sched latency by 2.91×. We note that the Linux community has acknowledged this issue and introduced an IRQ time accounting option in the scheduler, which is disabled by default [3]. The idea is to exclude IRQ time from CPU accounting—i.e., the scheduler reduces a thread's virtual runtime by the amount of softIRQ processing time. However, we find that this mechanism does not work as intended. As shown in Figure 3, enabling it (Linux + IRQa) actually increases P99.9 end-to-end latency by 4.7% with 48 threads compared to the original scheduler (red curve). Our investigation shows that IRQ time is accounted only after softIRQ processing completes, while CPU accounting events can occur in the middle of softIRQ execution. When this happens, the current thread misses the chance to exclude IRQ time and its virtual runtime still increases unfairly. The remaining unaccounted IRQ time is then deducted from the next



t	Linux	+IRQa	+ACCa
t_1	0	0 (0)	0 (0)
t_2	3	3 (0)	1 (0)
t_3	3	3 (3)	1 (1)
t_4	5	3 (1)	2 (0)

Figure 6: Linux's IRQ time accounting option can misattribute CPU time. The figure (left) shows how the CPU scheduler accounts for network-interrupt processing time; the table (right) reports thread T_1's virtual runtime, with accounted IRQ time awaiting subtraction shown in parentheses. Under the default scheduler (Linux), IRQ time is charged to the running thread: the IRQ time ($t_3 - t_1 = 3$) inflates T_1's virtual runtime to 5 at t_4. The IRQ-accounting option (+IRQa) aims to exclude this time, but a timing mismatch exists: the kernel updates the IRQ-time counter only after softIRQ processing completes, whereas it updates the virtual runtime at every accounting event. At t_2, the partial IRQ time ($t_2 - t_1 = 2$) is not yet recorded and is wrongly charged to T_1, raising its virtual runtime to 3. The IRQ-time counter reaches 3 at t_3 and is applied at the next CPU accounting event (i.e., t_4). T_1's virtual runtime at t_4 should be ($5 - 3 = 2$). However, because virtual runtime must increase monotonically, it remains ($5 - 2 = 3$); the leftover unit is instead deducted from the next thread T_2 at t_5. IRQa thus overestimates T_1 while underestimating T_2. Our patch (ACCa) fixes this by updating the IRQ-time counter at every accounting event (*e.g.*, t_2), correctly excluding IRQ time from the running thread's virtual runtime.



(a) Processing time vs. stall cycles **(b) Contribution from L1d miss**

Figure 7: CPU stall cycles inflate the processing time of threads. We measure the processing time of `rx_data_copy`, `app`, and `tx_data_copy`, along with the CPU stall cycles over the same period. (a) shows that processing time is almost perfectly linear with total stall cycles. Going one step further, we measure stall cycles with at least one outstanding L1d cache miss, which closely track total stall cycles as shown in (b), indicating that cache-related stalls are a major contributor. More discussion in §3.2.

woken threads, effectively benefiting them and further skewing scheduling fairness. As a result, tail latency does not improve—and can even increase—with Linux IRQ time accounting. An example is illustrated in Figure 6.

We have implemented a concise patch to resolve this issue, enabling accurate time accounting for every scheduling event. Specifically, whenever a CPU time accounting event occurs, the IRQ time is properly accounted for and excluded. In addition, we fix the time accounting issue during the `schedule` call so that the next woken-up thread's start runtime does not include the time spent in `schedule`. As shown in Figure 3, our patch (Linux + ACCa) yields significant improvements with 48 threads at its knee point, achieving 2.74× and 2.91× enhancements in terms of P99.9 end-to-end latency (from ~1139μs to ~415μs) and P99.9 `rx_sched` latency (from ~1071μs to ~367μs), respectively.

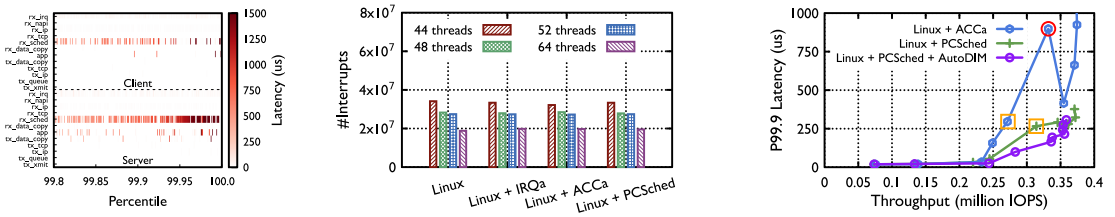
Virtual runtime can lie about fairness. To verify that our patch successfully removes IRQ time from CPU time accounting, we compare the measured virtual runtime of each thread with a modeled virtual runtime obtained by summing latency breakdown components that exclude IRQ time and context switches (mainly `rx_data_copy`, `app`, and Tx-side processing), as shown in Figure 5(b) (Actual vs. Modeled). Despite measurement overheads, the modeled virtual runtime matches the measured virtual runtime within 0.8% for most threads, and it accurately captures the virtual runtime gap among threads. This indicates that the virtual runtime differences driving scheduling decisions stem from actual processing time rather than time spent in softIRQ processing.

On the other hand, we still observe a virtual runtime gap even after excluding softIRQ time, especially on the server side. This is surprising, since each thread executes nearly identical code. According to our breakdown result, the largest variance across threads comes from `rx_data_copy`, `tx_data_copy`, and `app`. We find this is related to stalling cycles, rather than the actual computation time difference among threads. We use `rdpmc` [17] to measure stall cycles (`CYCLE_ACTIVITY.STALLS_TOTAL` [15]) during these three stages. Figure 7(a) shows that, on the client side, stall cycles and the corresponding processing times have a linear relationship across the three stages, which also correlates with the data-cache misses (`CYCLE_ACTIVITY.STALLS_L1D_MISS`) shown in Figure 7(b)[5].

We attribute the server-side virtual runtime gap to differences in per-packet processing time, which trigger a positive feedback loop similar to the one observed under inaccurate time accounting. On the client side, although our patch resolves the fairness issue, victim threads that handle more softIRQ work now experience cache pollution from frequent softIRQ handling, increasing `rx_data_copy`, `tx_data_copy`, and `app` processing times. Consequently, these victim threads need more CPU time to sustain the same throughput as other threads. Because the client is CPU-bottlenecked, this leads to a small throughput skew across threads (up to 1.58%) even when our patch aligns threads' virtual runtimes. Some server threads therefore receive relatively more packets; because the server is less CPU-bottlenecked, higher packet arrival rates translate into higher virtual runtime, reinforcing the feedback loop even with accurate time accounting. Processor-side overheads (*e.g.*, TLB/cache misses, branch mispredictions) further increase per-packet processing time (up to 2%). Combined, these effects account for up to a 3% virtual runtime gap among server threads.

This indicates that virtual runtime may not always be the ideal abstraction for scheduling applications, as it is influenced not only by computation but also by hardware state, such as cache behavior. To demonstrate the potential benefits of a more appropriate abstraction, we schedule applications based on the number of sent/received requests/responses, as a proof-of-concept (referred to as PCSched). Even with this simple scheme, we show that the P99.9 latency can be further reduced by 1.5× (Figure 3), reaching 276μs, with a significant improvement in `rx_sched` latency (Figure 4(d)). Identifying the

[5]The server side also exhibits a linear relationship between stall cycles and processing time, but the stalls are not directly correlated with data-cache misses. They may instead be caused by other factors, such as branch mispredictions or TLB misses. This discrepancy may be because the client records end-to-end latency, which requires a hotter cache state.



(a) Linux + ACCa, 44 threads **(b) Number of interrupts** **(c) Tuning interrupts according to traffic**

Figure 8: Understanding the impact of interrupts moderation. (a) With Linux+ACCa at 44 threads (red circle in Figures 8c), enabling DIM significantly increases `rx_sched` latency. (b) DIM does not reach the optimal tuning point at 44 threads. (c) Using the characteristics of ping-pong workloads, AutoDIM with PCSched achieves the lowest P99.9 latency (~215μs) with comparable throughput. Orange squares mark when DIM is enabled for ACCa and PCSched.

appropriate scheduling abstraction for different application classes is a challenging task; the ideal accounting unit should capture application progress while abstracting away noise such as data locality. For network-bound applications, such units may include packets or requests/responses, and we leave a systematic exploration of this space to future work. One promising direction is to integrate flexible accounting units into user-space scheduling frameworks (*e.g.*, eBPF), enabling operators to define fairness in ways that better align with application-level objectives.

Asymmetric behavior between server and client. We also observe asymmetric latency and runtime-gap behavior between the client and server, even though both run the same ping-pong application under the default Linux scheduler and ACCa. The key reason is that the client typically becomes the CPU bottleneck due to the additional application logic required to record end-to-end latency. Under the default Linux scheduler, IRQ time is not properly accounted, so higher CPU utilization on the client leads to more frequent interruptions of application threads and more misattributed CPU time. This enlarges the virtual runtime gap among client threads, making the client the initial bottleneck. After fixing IRQ time accounting, virtual runtime more accurately reflects actual execution time (though it is still an imperfect fairness metric). Because the client remains CPU-bound, the scheduler has more runnable threads to choose from and thus more opportunities to reduce runtime gaps there. In contrast, the server has fewer scheduling opportunities to close these gaps, so larger virtual runtime differences persist. As these gaps grow, the server ultimately becomes the bottleneck for tail latency.

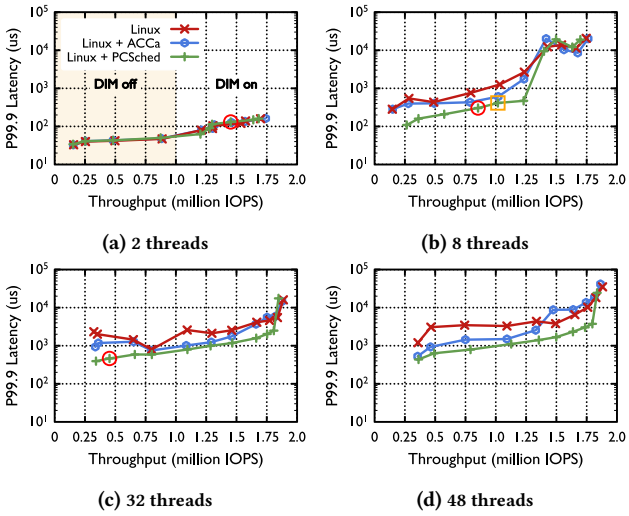
Leveraging traffic predictability can further reduce latency. Another observation from Figure 3 is that, before reaching the knee point, tail latency first increases and then decreases across all curves (including PCSched). For Linux + ACCa, comparing the latency breakdown at the knee point (Figure 4(c)) with that at threads = 44 (Figure 8(a)), we observe a decrease in `rx_sched`. One possible explanation is DIM behavior: as shown in Figure 8(b), the number of interrupts decreases as the thread count increases across all scheduling schemes. With fewer interrupts, waiting threads are more likely to wake up, instead of the scheduler prioritizing interrupt-woken threads with lower runtime, resulting in lower `rx_sched` latency. More importantly, this suggests that DIM is unable to find the optimal configuration when the traffic is bursty. To verify our hypothesis, we apply a very simple proof-of-concept configuration by exploiting the predictability of the workload. Since the total number of in-flight packets is known beforehand (equal to the number of threads), we set the minimum packet threshold and the maximum timeout before triggering an interrupt (i.e., `rx_frames` and `rx_usecs`) to half of the total in-flight packets across threads on a single logical core and the corresponding total processing time. This effectively assumes that half of the traffic is on the client side and the other half is on the server side. Figure 8(c) shows that, with this simple proof-of-concept scheme (referred to as AutoDIM), the latency can be further improved by 22% (to ~215μs) at the knee point, and the unusual latency–throughput curve (first increasing and then decreasing) disappears.

The gains from our heuristic tuning point to a broader insight: from the receiver's perspective, deterministic and predictable workloads can potentially simplify DIM tuning and improve performance. Achieving this is difficult with today's sender-driven protocols, where the receiver has little control over near-term traffic arrival. A promising direction is to co-design DIM with emerging receiver-driven transport protocols [18, 23, 29, 33, 50], in which the receiver actively shapes incoming traffic. By making traffic more predictable, DIM can identify suitable parameters more effectively and further reduce tail latency.

3.3 Multi-threaded Considered Harmful

Performance can improve by either increasing the number of threads or the number of in-flight requests. After studying threads, we now examine in-flight requests while keeping a single core fixed. With multiple in-flight requests, batching mechanisms (*e.g.*, TSO/GRO and TCP Nagle) break the one-to-one mapping between RPCs and `skbs`, making per-RPC latency breakdown difficult because requests and `skbs` may be merged at different stages on both sides. Consequently, in this subsection we focus on capturing `rx_sched` latency at RPC granularity to reduce measurement overhead while still capturing the dominant source of latency. As we show next, increasing the number of in-flight requests yields better performance than increasing the number of threads (§3.2).

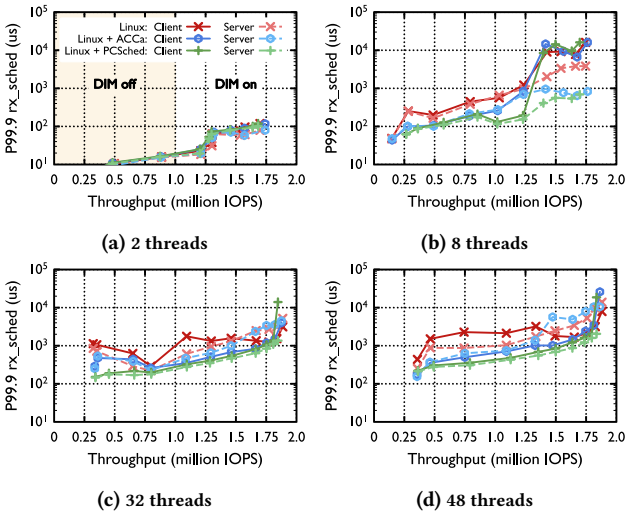
`rx_sched` latency still dominates the end-to-end latency in high I/O depths. From Figures 9 and 10, we observe that in high I/O depths, the dominant component of end-to-end latency still comes from the `rx_sched` latency. For example, for Linux with 32 threads, the ratio of server-side P99.9 `rx_sched` latency to P99.9 end-to-end latency increases to 60.24%, without even accounting for client-side



(a) 2 threads (b) 8 threads

(c) 32 threads (d) 48 threads

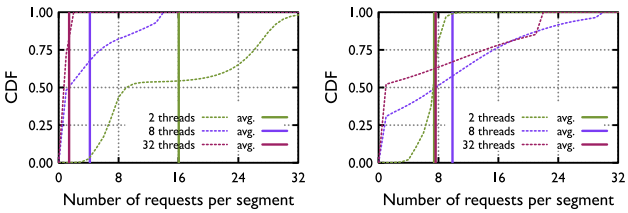
Figure 9: Application performance with increasing number of in-flight requests and a fixed number of threads. With two threads, each on a different logical core, the Linux network stack achieves low P99.9 latency and high throughput comparable to user-space stacks [28]. Note red circles mark data points with 128 in-flight requests. In (b), with PCSched, optimal performance occurs with DIM disabled below 24 in-flight requests and enabled afterward (orange squares). See §3.3 for more discussion.



(a) 2 threads (b) 8 threads

(c) 32 threads (d) 48 threads

Figure 10: `rx_sched` still dominates the end-to-end P99.9 latency with increasing in-flight requests. When there are two threads per core (one per logical core), `rx_sched` increases only mildly with increasing in-flight requests, primarily due to additional queuing in the TCP receive buffer rather than scheduling contention. More discussion in §3.3.



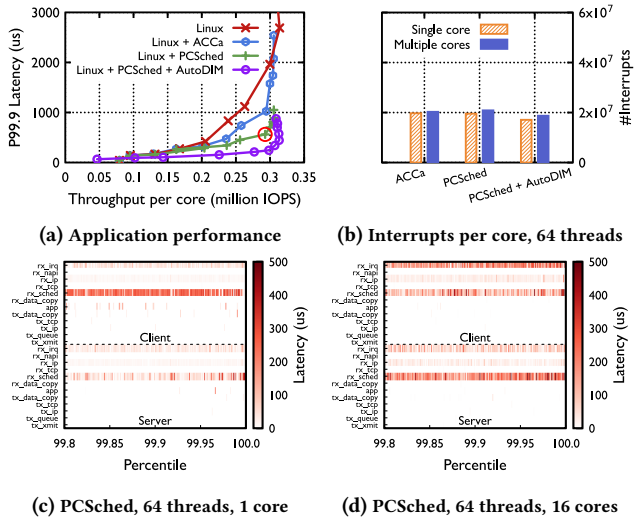
(a) #In-flight request = 128 (b) Throughput = 1.25 mIOPS

Figure 11: CDF: the number of requests per segment. With the same in-flight requests (a), increasing thread count reduces requests per segment, degrading batching efficiency. With the same throughput (b), average segment sizes are similar, but additional in-flight requests accumulate in the receive buffer, leading to increased `rx_sched` latency. Discussion in §3.3.

`rx_sched` latency. Thus, resolving IRQ time accounting (ACCa) and finding the right scheduling unit/abstraction (PCSched) continue to help reduce end-to-end latency, except when the number of threads is 2. With 2 threads per core, there is only one thread per logical core, so performance is similar across scheduling schemes. When the number of threads is 8, we observe that the end-to-end latency in PCSched suddenly increases once the throughput exceeds 1.25 million IOPS. This is because the client-side CPU starts to become the bottleneck. When the number of threads is 48, ACCa shows higher tail latency than the default Linux scheduler. Upon closer inspection, we find two reasons. First, the Linux scheduler reduces the gap among threads when the virtual runtime gap becomes too large; this mechanism is triggered in the default Linux scheduler but not in ACCa. When we disable this feature, the latency of the default Linux scheduler increases by ~1.78×. Second, compared to low I/O depth, under high I/O depth in ACCa, thread virtual runtimes become more dispersed, even though the absolute gap does not change. This can increase the waiting time for threads with larger virtual runtime, whereas previously they did not need to be preempted by threads with similar virtual runtime. Note that we cannot show AutoDIM results despite predictable workloads because batching makes packet arrivals nondeterministic and current NICs trigger interrupts based on packet counts rather than bytes. A promising direction is for NICs to expose more flexible interrupt-generation interfaces to better exploit workload predictability.

A single thread per logical core with increasing number of in-flight requests achieves low latency and high throughput. In Figure 9(a), it is observed that when there are two threads, each running on a different logical core, increasing the number of in-flight requests does not result in a significant rise in P99.9 latency since there is no contention between the threads. As shown in Figure 10(a), `rx_sched` increases only slightly, mainly because higher I/O depth introduces additional queuing delay in the TCP receive buffers. With increasing number of in-flight requests, the optimization techniques such as TSO/GRO and TCP Nagle algorithm reduce the per-request processing overhead by combining multiple requests into a single TCP segment. This allows Linux + PCSched to achieve 90μs P99.9 latency and 1.28 million IOPS with 32 in-flight requests per thread as seen in Figure 9(a), comparable to Caladan [28], a state-of-the-art user-space solution that achieves 0.94 million IOPS per core and ~100μs latency.

Good locality and bounded in-flight requests are essential for optimal performance. Now we consider cases where each core serves more than two threads. We make two comparisons across different thread counts: (1) fixing the total number of in-flight requests at 128, and (2) fixing the total throughput to be roughly



(a) Application performance (b) Interrupts per core, 64 threads

(c) PCSched, 64 threads, 1 core (d) PCSched, 64 threads, 16 cores

Figure 12: Throughput gains from multi-core scaling come at the cost of increased tail latency. We mirror our experiment in §3.2 on 16 cores within a single NUMA node. As shown in (a), PCSched reaches a knee point at 64 threads per core (the red circle), where P99.9 latency increases by 1.74× compared to the single core case. Breakdown results in (c) and (d) attribute this inflation to delayed interrupt handling (`rx_irq`) and increased scheduling latency (`rx_sched`). Interestingly, (b) shows only little change in interrupt rate. After throughput reaches its peak, adding more threads may reduce throughput due to increased thread contention. See §3.4 for further discussion.

the same. For the first case, as the number of threads increases from 2 to 32, the throughput degrades by 3.4×, while the tail latency inflates by 4.6×. This trend is consistent across all scheduling schemes. This escalation is primarily attributed to the interleaving of packets among multiple threads, which adversely affects both spatial and temporal locality. As depicted in Figure 11(a), for the same total number of in-flight requests, each segment (skb) carries fewer requests as the number of threads increases. This highlights the inefficiency of batching in scenarios with higher thread counts. In practical terms, the network stack must process a larger number of segments to handle the same volume of requests, leading to increased latency and higher per-request overhead. For the second case, we observe that while throughput remains roughly the same across different numbers of threads, the latency increases significantly. From Figure 11(b), we observe that the average segment sizes are very similar, resulting in comparable throughput. However, as the number of threads increases, more in-flight requests are generated, and as the CPUs approach saturation, these extra packets are simply buffered in the Rx TCP buffer, which leads to higher `rx_sched` latency on the Rx side.

From these observations, we draw two insights: 1) one or two connections per core are sufficient to achieve high throughput and low latency with effective batching. 2) increasing the number of connections inflates in-flight bytes and hence latency. This implies that, in practice, optimal performance is achieved by maintaining only a few threads per core whenever possible, which enables both high throughput and low latency. When this is not feasible—such as in servers that must maintain a large number of connections—an interesting direction is to keep only a small subset of connections active at any given time (*e.g.*, using receiver-driven protocols). Limiting the number of active connections can improve batching efficiency and reduce tail latency. However, this approach may cause some connections to wait too long. Understanding and managing this trade-off is an interesting direction for future work.

3.4 When DIM Meets Many Cores

In this section, we examine performance in a multi-core Linux network stack. By default, we enable DIM because it delivers the best performance. Surprisingly, when using multiple cores (16 physical cores within a single NUMA node) while keeping the same number of threads per core and one in-flight request per thread, tail latency degrades compared to the single-core case.

Packet interleaving among multiple cores complicates DIM tuning. Compared to the single-core case, maximum throughput scales by about 13.21×, roughly proportional to the number of cores. Fixing IRQ time accounting and choosing the right accounting abstraction still reduce P99.9 latency: at the knee point, tail latency drops by 1.9× and 3.5×, respectively, compared to the default Linux scheduler.

However, the tail latency of both ACCa and PCSched increases compared to the single-core case at the knee point. To understand this, we focus on PCSched and analyze the latency breakdown. In the multi-core setting, the knee point occurs at 64 threads per core, so we compare the breakdown between single-core and multi-core configurations at this point. Figures 12(c) and 12(d) show that the increase is mainly from `rx_sched` and `rx_irq`, each rising by about 100μs. However, Figure 12(b) indicates that the number of interrupts per core remains largely unchanged. We attribute this increase to packet interleaving across multiple cores, which makes DIM difficult to tune effectively. If DIM overreacts to bursty traffic and generates unnecessary interrupts, `rx_sched` latency rises because the scheduler may prioritize interrupt-woken threads due to their lower runtime. Conversely, if DIM underreacts and generates fewer interrupts, `rx_irq` latency increases. To validate this hypothesis, we run AutoDIM experiments using the same setup as in §3.2. We observe that the tail latency at the knee point remains largely unchanged under AutoDIM. This result further highlights the need to make traffic more predictable.

3.5 Best Case: Multiple Cores, High Load, Few Threads per Core

Finally, we maintain a constant thread count and increase the number of in-flight requests. The outcomes of this investigation are illustrated in Figure 13.

The application performance in a multiple-core environment is comparable to that of the single-core case. After scaling to multiple cores, performance remains comparable to the single-core case (§3.3), and the overall trends are similar. As before, fixing IRQ time accounting and using the right packet abstraction consistently reduce tail latency by up to 6.4× and 18.6×, respectively, compared to default Linux, since `rx_sched` remains a dominant latency source (not shown). Similarly, Linux achieves low P99.9 latency and high

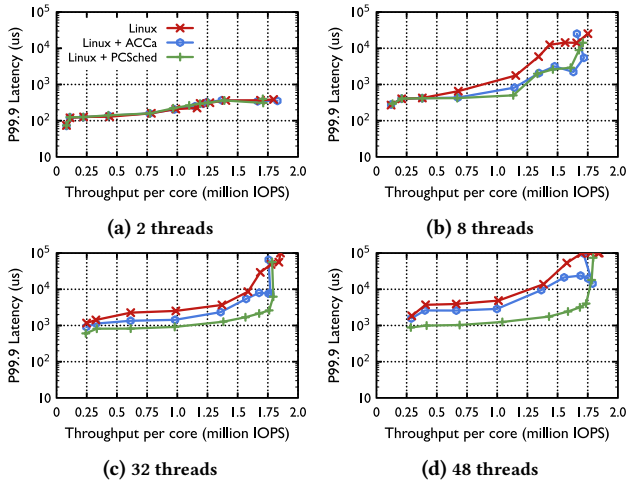


Figure 13: As the number of in-flight requests and the number of threads increase in the multi-core case, the threads achieve comparable performance to the single-core scenarios. More discussion in §3.5.

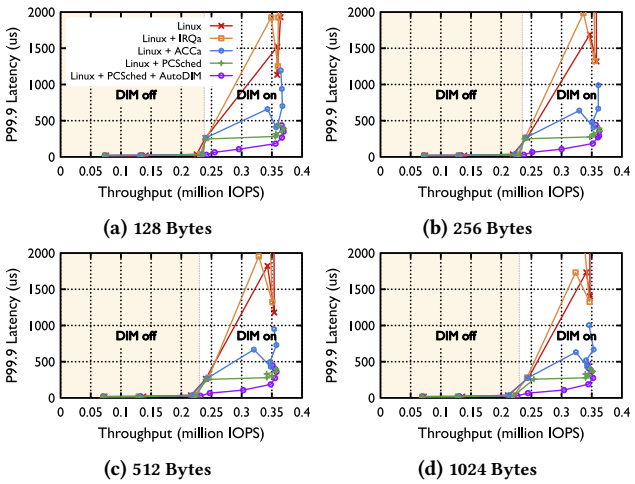


Figure 14: Our insights on host network stack latency generalize to different RPC sizes. More discussion in §3.6.

throughput with 2 threads per core. For example, with 48 in-flight requests, it reaches ~ 1 million IOPS per core with a P99.9 latency of 210μs, comparable to the single-core case.

One trend we observe is that, compared to the single-core case, using two threads per core leads to higher P99.9 latency when achieving similar throughput in multi-core configurations. To understand this behavior, we examine operating points where both configurations achieve ~ 1.25 mIOPS. In the single-core case, this throughput requires only 32 in-flight requests, whereas the multi-core case requires 96 in-flight requests. With more in-flight requests, the corresponding tail latency increases. A natural follow-up question is why, given the same number of in-flight requests, the multi-core configuration cannot match the throughput of the single-core case. We observe that with 32 in-flight requests, the per-core interrupt rate in the single-core configuration is 1.85× higher than that observed when using 16 cores. This indicates that in the multi-core case, DIM is unable to find the optimal interrupt triggering rate, potentially leading to higher latency in `rx_irq`, which in turn reduces throughput.

3.6 Impact of RPC Size, Communication Pattern, and Streams

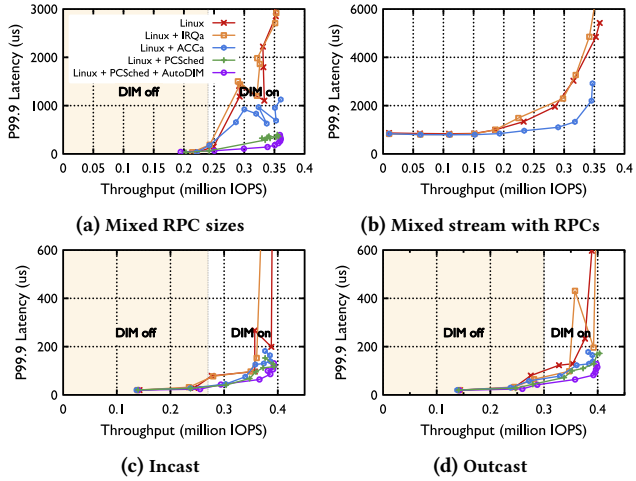
In this subsection, we examine whether our insights continue to hold across a broader range of workloads, including different RPC sizes, communication patterns such as incast and outcast, mixtures of RPC flow sizes, and RPC traffic coexisting with streaming traffic. Following the setup in §3.2, we run all threads on a single core and gradually increase the number of threads.

Performance across uniform and mixed RPC sizes. We increase the RPC size from 64 bytes to 128, 256, 512, and 1024 bytes and measure P99.9 latency and throughput as the number of threads increases under different schemes. As shown in Figure 14, the results exhibit trends similar to those observed in §3.2 across all RPC sizes. At the knee point, ACCa improves performance by 2.78, 3.22, 2.73, and 3.24× over default Linux. As proofs of concept, PCSched and AutoDIM still show that the benefits of a more accurate accounting unit and efficient interrupt tuning persist across RPC sizes, improving performance by 3.79, 4.36, 3.94, and 4.56× and 6.19, 7.12, 6.30, and 7.41× over default Linux, respectively.

We also evaluate performance under a mixture of RPC sizes, with half of the threads sending 64B RPCs and the other half sending 1024B RPCs. As shown in Figure 15(a), the overall trends are similar to those observed in §3.2. PCSched remains effective despite the heterogeneous RPC sizes because, at these sizes, CPU cost is still dominated by per-RPC overhead; as shown in Figures 3 and 14(d), 64B and 1024B RPCs achieve similar throughput at the knee point, approximately 0.35 million IOPS. We also observe that under ACCa and PCSched, throughput becomes less stable as the number of threads increases than with a uniform RPC size; AutoDIM eliminates this instability, highlighting the importance of efficient interrupt tuning, which we leave for future work.

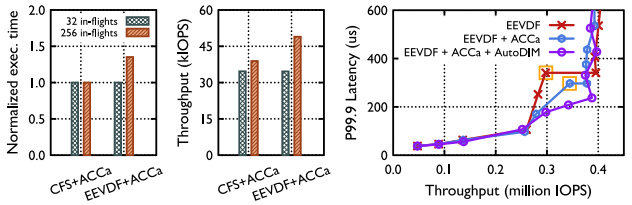
Mix stream with RPCs. We combine 64B RPC traffic, whose load increases with the number of threads, with a bulk byte stream generated by `iperf3` to create a mixed RPC-and-stream workload. We omit PCSched in this experiment because per-RPC and per-byte overheads are interleaved, making it difficult to identify an appropriate accounting unit; we also omit AutoDIM because stream traffic perturbs both the arrival patterns and processing times of RPC packets, making it unclear how to configure the interrupt threshold to optimize RPC performance. We observe that even at low load, RPC tail latency is high because RPC packets experience head-of-line blocking behind stream traffic, consistent with prior work [25]. However, as the number of threads increases, ACCa's latency remains relatively stable because it corrects IRQ-time misaccounting, whereas default Linux exhibits substantial latency inflation.

Incast and outcast workloads. We evaluate different schemes under incast and outcast workloads. In the incast case, client threads are spread across different cores within the same NUMA node, and therefore have low client-side (scheduling) latency, while the server



(a) Mixed RPC sizes (b) Mixed stream with RPCs (c) Incast (d) Outcast

Figure 15: Our insights generalize to different communication patterns as well. See §3.6 for more details.



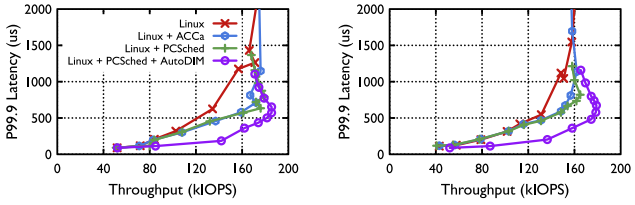
(a) Lag evaporation behavior (b) EEVDF performance

Figure 16: Performance under EEVDF. (a) shows that lag evaporation in the current EEVDF implementation leads to unfair CPU service across threads, inadvertently reducing virtual runtime gaps and thereby incidentally improving tail latency. Nevertheless, our optimizations continue to exhibit measurable improvements on tail latency. Orange squares mark when DIM is enabled for EEVDF and ACCa. See §3.7 for more discussion.

threads remain pinned to a single physical core; the outcast case reverses this setup. We observe that, at the knee point, both Linux and Linux + ACCa achieve less than half of the P99.9 latency compared to the case in Figure 3. The reason is that, in incast/outcast workloads, low client/server-side latency creates more scheduling opportunities on the bottleneck side, allowing the CPU scheduler to prioritize threads with lower virtual runtime and reduce runtime imbalance. In the default one-to-one case (§3.2), both client and server are actively involved in a ping-pong pattern, so the two threads are not always simultaneously runnable, which limits the scheduler's ability to correct virtual runtime imbalances. However, inaccurate IRQ-time accounting and inappropriate accounting units still affect performance in the short run. As a result, we still observe the benefits of ACCa, PCSched, and AutoDIM: they improve performance by 1.42, 1.70, and 1.88× for incast, and 1.69, 1.81, and 2.19× for outcast, compared to default Linux.

3.7 Performance under EEVDF

EEVDF is a recently deployed scheduling policy in the Linux kernel. Similar to CFS, it aims to distribute CPU time equally among



(a) 100% read (b) 95% read + 5% write

Figure 17: Our insights enable Linux to achieve up to 2.2× lower latency for Redis. After throughput reaches its peak, adding more threads may reduce throughput due to increased thread contention. Details in §4.

runnable tasks with the same priority [11]. However, we find that the current EEVDF implementation fails to achieve this goal. We construct a scenario in which one thread has a higher I/O depth than others while the CPU is fully bottlenecked. As shown in Figure 16(a), under CFS, threads have similar execution time despite different I/O depths. In contrast, under EEVDF, the high–I/O-depth thread receives significantly more CPU time (up to 1.35×), leading to higher throughput despite identical priorities. Note in CFS, threads with higher I/O depth achieve higher throughput simply due to more in-flight I/O. We attribute this behavior to *lag evaporation* in the current implementation of EEVDF. When a thread sleeps, EEVDF records its *lag*—the difference between its virtual runtime and the average virtual runtime of runnable threads—to preserve relative ordering upon wakeup. However, in workloads with frequent sleep–wakeup cycles, this mechanism shrinks runtime differences. When the thread with the largest virtual runtime runs while others are sleeping, its lag collapses to zero as its virtual runtime is the same as the run-queue average, effectively resetting its position on subsequent wakeups. Although recent kernels introduce mitigation [7], such unfairness persists as shown in Figure 16.

Due to lag evaporation, EEVDF shrinks the virtual runtime gap between threads. Because lag and virtual deadlines are both derived from virtual runtime, similar virtual runtime values lead to similar scheduling decisions. As a result, EEVDF can inadvertently mask virtual runtime inaccuracies, such as those caused by inaccurate IRQ time accounting. As shown in Figure 16(b), EEVDF incidentally improves tail latency compared to CFS. However, even under lag evaporation, we still observe that accurate IRQ time accounting (ACCa) provides up to 1.15× performance gains by improving runtime measurement (we omit PCSched, as its performance is similar to ACCa under lag evaporation). In addition, AutoDIM continues to deliver up to 1.25× improvement, indicating that interrupt tuning remains important even in the presence of lag evaporation. In the long term, we expect that once the lag evaporation issue is resolved, the benefits of accurate IRQ time accounting and appropriate accounting units will become more pronounced, as virtual runtime gaps will be preserved across scheduling epochs.

4 Validation with Real-world Application

In this section, we verify that our insights hold in real-world applications using Redis on the Linux kernel network stack, a widely adopted in-memory key–value store. We use a setup similar to prior work [25]: specifically, we run the standard YCSB workload [5] with

both a 100% read workload and a 95%/5% read/write mix. Each RPC request is 4KB in size. To increase the load, we increase the number of threads per CPU core on the client side, while keeping other configurations the same as in §2. Figure 17 shows that, for both workloads, accurate IRQ time accounting improves tail latency by up to 2× at the knee points compared to default Linux. With PCSched, tail latency further improves by up to 2.2× relative to default Linux. We also evaluate the performance of AutoDIM. As shown in Figure 17(b), AutoDIM further improves tail latency by up to 1.42× relative to PCSched and increases throughput by up to 1.08×. These results show that our key insights hold for real-world applications.

5 Related Work

Our objective is to gain a comprehensive understanding of latency within the Linux network stack, thereby providing valuable insights and recommendations for optimizing forthcoming systems and hardware to achieve superior latency performance. Our research seamlessly aligns with the ongoing endeavors within the field aimed at mitigating network stack overhead, as exemplified by prior work [4, 41, 47, 57, 64]. These initiatives primarily focus on minimizing data copy operations [64], streamlining system call overhead [4], and enhancing the overall performance of the TCP stack [41, 47]. By contributing to this body of knowledge, our work contributes to the ongoing pursuit of optimizing network systems for reduced latency and enhanced efficiency.

Improving the latency performance of network stack. There are numerous initiatives underway to improve the latency of applications, such as optimizing kernel data paths [3, 32, 46, 55], bypassing the kernel [22, 28, 36, 40, 41, 48, 53, 56, 57, 65], isolating network stack processing from applications [25, 41, 48, 53] and utilizing hardware offloading [2, 19, 51, 61].

While some prior research has indicated that Linux performance falls behind state-of-the-art user-space solutions by a significant factor, typically in the range of 5–10 times slower [20, 28, 53], it is essential to consider that this disparity might stem from the inherent complexity of the Linux stack and the vast array of system and hardware parameters at play. Earlier studies may not have unearthed the optimal configuration settings for minimizing latency within the Linux stack effectively. For instance, earlier research [43] has shown that employing thread pinning can notably enhance the performance of applications like memcached [14]. Curiously, this aspect seems to have been somewhat overlooked in many prior investigations [28, 53]. Furthermore, when it comes to increasing the number of in-flight requests, which can potentially enhance performance, many studies either rely on time-arrival-based workloads [28, 53, 54] or maintain limited in-flight requests while handling a large number of connections [41]. These approaches may not fully exploit Linux's potential for performance improvement through batch processing.

Uncovering latency bottlenecks. While previous research, such as Nsight [31], has made strides in comprehending the sources of latency within the network stack, our work delves deeper into this issue. We, like Nsight, identify blocking caused by threads scheduling and interrupt as a significant contributor to latency inflation in applications. This work goes beyond this observation by carefully identifying the root causes of high latency—namely inaccurate IRQ time accounting, the limitations of CPU runtime as a fairness metric, and the impact of bursty traffic on interrupt tuning—and by systematically demonstrating the performance benefits that arise from addressing each of these bottlenecks.

User interrupts. Modern processors provide hardware support for user interrupts [6], allowing user-space threads to receive interrupt notifications directly via dedicated instructions (*e.g.*, Intel's `SENDUIPI`). Built on user interrupts, recent work [21, 30, 45] is able to match or exceed the performance of existing mechanisms. We believe that building on one of the key insights of this work—leveraging traffic predictability to optimize interrupt tuning—can further benefit research on user interrupts.

CPU schedulers. Designing efficient CPU schedulers and policies has been widely studied [10, 28, 34, 37–39, 42, 45, 48, 49, 53]. Similar to our work, Iron [42] observes CPU accounting issues in multi-container environments and shows that they can lead to cross-container interference. eBPF scheduler [10] and Enoki [49] safely extend the kernel scheduler with custom policies; other work delegates policy enforcement to user-space agents [34, 39] or implements scheduling entirely within application runtimes [28, 37, 38, 45]. We believe the key insights of this work—accurate IRQ time accounting and identifying the right scheduling abstraction or unit—have the potential to inform and strengthen these scheduling systems and policies.

Cache Isolation. Cache isolation can reduce runtime interference. Intel CAT provides this capability but primarily isolates application data, leaving interference from network data cached in the LLC [63]. Its coarse cache-way granularity also provides only a limited number of partitions [24, 63], reducing its effectiveness under high consolidation. ResQ constrains working sets to DDIO capacity, but our results show interference even at low overall miss rates, suggesting an important role for conflict misses. A promising direction is page coloring: assigning applications and DMA traffic to disjoint cache sets, similar to Sepia [62].

6 Conclusion

Our study shows that tail latency in the Linux network stack is driven not by packet processing itself, but by how the host manages CPU resources for networking. We identify three key factors: misattributed softIRQ time, the limits of CPU runtime as a fairness metric for network workloads, and inefficient interrupt tuning under bursty traffic. Addressing these factors improves tail latency by up to 5.3× while maintaining comparable throughput. Overall, our findings show that achieving low-latency networking requires rethinking host CPU scheduling and host–NIC interaction, pointing to new directions for OSes, network stacks, and future host networking hardware.

Acknowledgments

We thank our shepherd and anonymous SIGCOMM reviewers for their insightful feedback. This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2024-00405128, RS-2026-25507250) and by NSF grant 2133403.

References

[1] 2008. ftrace. https://www.kernel.org/doc/html/v5.10/trace/ftrace.html.
[2] 2009. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/.
[3] 2010. Proper Kernel IRQ time Accounting. https://lwn.net/Articles/405889/.
[4] 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.
[5] 2019. Yahoo! Cloud Serving Benchmark (YCSB). https://github.com/brianfrankcooper/YCSB/wiki.
[6] 2021. User-space interrupts. https://lwn.net/Articles/871113/.
[7] 2024. Completing the EEVDF scheduler. https://lwn.net/Articles/969062/.
[8] 2025. hwstamp_ctl – set time stamping policy at the driver level. https://linux.die.net/man/8/hwstamp_ctl.
[9] 2025. Netperf. https://github.com/HewlettPackard/netperf.
[10] 2026. eBPF Tutorial: Introduction to the BPF Scheduler. https://eunomia.dev/tutorials/44-scx-simple/.
[11] 2026. EEVDF Scheduler. https://docs.kernel.org/scheduler/sched-eevdf.html.
[12] 2026. irqbalance(1) - Linux man page. https://linux.die.net/man/1/irqbalance/.
[13] 2026. ktime accessors. https://www.kernel.org/doc/html/latest/core-api/timekeeping.html.
[14] 2026. Memcached:in-memory key-value store. https://memcached.org.
[15] 2026. PerfMon Events Documentation. https://perfmon-events.intel.com/.
[16] 2026. phc2sys – synchronize two clocks. https://linux.die.net/man/8/phc2sys.
[17] 2026. RDPMC — Read Performance-Monitoring Counters. https://www.felixcloutier.com/x86/rdpmc.
[18] Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2024. Harmony: A congestion-free datacenter architecture. In *USENIX NSDI*.
[19] 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*.
[20] Shinichi Awamoto and Michio Honda. 2025. Opening Up Kernel-Bypass TCP Stacks. In *USENIX ATC*.
[21] Berk Aydogmus, Linsong Guo, Danial Zuberi, Tal Garfinkel, Dean Tullsen, Amy Ousterhout, and Kazem Taram. 2025. Extended User Interrupts (xUI): Fast and Flexible Notification without Polling. In *ACM ASPLOS*.
[22] 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*.
[23] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. 2022. dcPIM: Near-Optimal Proactive Datacenter Transport. In *SIGCOMM*.
[24] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *ACM SIGCOMM*.
[25] 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*.
[26] Julie Cummings and Eliezer Tamir. 2013. Open source kernel enhancements for low latency sockets using busy poll. *Intel white paper* (2013).
[27] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.
[28] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *USENIX OSDI*.
[29] 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*.
[30] Linsong Guo, Danial Zuberi, Tal Garfinkel, and Amy Ousterhout. 2025. The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling. In *USENIX NSDI*.
[31] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. 2022. How to diagnose nanosecond network latencies in rich end-host stacks. In *USENIX NSDI*.
[32] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *USENIX OSDI*.
[33] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*.
[34] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghost: Fast & flexible user-space delegation of linux scheduling. In *ACM SOSP*.
[35] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. 2018. Perfiso: Performance isolation for commercial latency-sensitive services. In *USENIX ATC*.
[36] 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*.
[37] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. 2024. Skyloft: A General High-Efficient Scheduling Framework in User Space. In *ACM SOSP*.
[38] 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*.
[39] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-defined scheduling across the stack. In *ACM SOSP*.
[40] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *USENIX NSDI*.
[41] 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*.
[42] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. 2018. Iron: Isolating Network-Based CPU in Container Environments. In *USENIX NSDI*.
[43] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *ACM EuroSys*.
[44] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *ACM SoCC*.
[45] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G Edward Suh, Kostis Kaffes, and Christina Delimitrou. 2024. Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling. In *IEEE HPCA*.
[46] 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*.
[47] Ilias Marinos, Robert NM Watson, and Mark Handley. 2014. Network stack specialization for performance. In *ACM SIGCOMM*.
[48] 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*.
[49] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2024. Enoki: High velocity linux kernel scheduler development. In *ACM EuroSys*.
[50] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM*.
[51] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *USENIX NSDI*.
[52] NVIDIA. 2026. Dynamically-Tuned Interrupt Moderation (DIM). https://enterprise-support.nvidia.com/s/article/dynamically-tuned-interrupt-moderation--dim-x.
[53] 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*.
[54] John Ousterhout. 2021. A Linux Kernel Implementation of the Homa Transport Protocol. In *USENIX ATC*.
[55] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore Systems. In *ACM EuroSys*.
[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] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *USENIX OSDI*.
[59] Luigi Rizzo. 2012. Revisiting Network I/O APIs: The netmap Framework: It is possible to achieve huge performance improvements in the way packet processing is done on modern operating systems.. In *USENIX ATC*.
[60] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. 2011. It's Time for Low Latency. In *HotOS*.
[61] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *USENIX NSDI*.
[62] Changwoo Song, Sanghyun Kim, Jinhyeok Oh, Qizhe Cai, Joonsung Kim, and Jaehyun Hwang. 2026. When DDIO Meets Page Coloring: Revisiting DDIO Performance with Sepia. In *USENIX OSDI*.
[63] 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*.
[64] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *USENIX ATC*.

[65] 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*.

[66] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU performance isolation for shared compute clusters. In *ACM Eurosys*.