Host Congestion Control

Saksham Agarwal
Cornell University

Arvind Krishnamurthy
Google & University of Washington

Rachit Agarwal
Cornell University

ABSTRACT

The conventional wisdom in systems and networking communities is that congestion happens primarily within the network fabric. However, adoption of high-bandwidth access links and relatively stagnant technology trends for resources within hosts have led to emergence of host congestion—that is, congestion within the host network that enables data exchange between NIC and CPU/memory. Such host congestion alters the many assumptions entrenched within decades of research and practice of congestion control.

We present hostCC, a congestion control architecture to handle both host and network fabric congestion. hostCC embodies three key ideas. First, in addition to congestion signals that originate within the network fabric, hostCC collects host congestion signals that capture the precise time, location, and reason for host congestion. Second, hostCC introduces a sub-RTT granularity host-local congestion response that uses congestion signals to allocate host resources between network traffic and host-local traffic. Finally, hostCC uses both host and network congestion signals to allocate network resources at an RTT granularity.

We realize hostCC within the Linux network stack. Our hostCC implementation requires no modifications in applications, host hardware, and/or network hardware; moreover, it can be integrated with existing congestion control protocols to handle both host and network fabric congestion. Evaluation of Linux DCTCP with and without hostCC suggests that, in the presence of host congestion, hostCC significantly reduces queueing and packet drops at the host, resulting in improved performance of networked applications in terms of throughput and tail latency.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Data center networks**; • **Software and its engineering** → **Operating systems**;

KEYWORDS

Congestion control, datacenter transport, network stack

ACM Reference Format:
Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. 2023. Host Congestion Control. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3603269.3604878

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604878

1 INTRODUCTION

Classical literature in datacenter congestion control takes a narrow view of "end-to-end", often interpreting the end as the point of presence of Ethernet (the network interface card, or NIC). This view precludes a network that every datacenter server has—the *host network*, that is, the network comprising processor, memory and peripheral interconnects that enables the exchange of data between CPU, memory and peripheral devices. The host network provides many desirable properties—minuscule probability of failures and packet corruption, ample bandwidth, and losslessness guarantees; thus, the conventional wisdom in systems and networking communities is that congestion happens primarily within the network fabric (that is, at network switches).

Several recent studies from large-scale production clusters [1, 24, 25] demonstrate that the above conventional wisdom is merely an appeal to tradition fallacy: adoption of high-bandwidth access links, coupled with relatively stagnant technology trends for resources within the host—CPU speeds, cache sizes, memory access latency, memory bandwidth per core, NIC buffer sizes, etc.—has led to the emergence of host congestion, that is, congestion within the processor, memory and peripheral interconnects of the host network. For instance, a recent study from Google [1] demonstrates that host congestion in their production clusters leads to significant queueing and packet drops at hosts, resulting in application-level performance degradation in terms of latency and throughput. We reproduce the host congestion phenomenon from [1] using Linux DCTCP; we observe that host congestion can lead to as much as 1% packet drops at the host, $35-55\%$ throughput degradation, and $120-5000\times$ tail latency inflation (§2).

The regime of host congestion forces us to revisit the many fundamental assumptions entrenched within decades of research and practice of congestion control. For instance, classical congestion control literature assumes that packet drops happen at the congestion point; in contrast, host congestion results in queueing and drops away from the actual congestion point (since the host network is lossless). Thus, we must rethink congestion signals to capture the precise time, location, and reason for host congestion. As another example, an unspoken assumption in classical congestion control literature is that all competing traffic adheres to the congestion control protocol; such is not the case in the host congestion regime where traffic from "outside the network" (*e.g.*, applications generating CPU-to-memory traffic) does not employ congestion control mechanisms, is much closer to the congestion point, and can thus change dramatically at sub-RTT granularity. This has powerful implications in terms of rethinking congestion response: existing congestion control protocols that operate at RTT granularity may achieve performance far from optimal in the host congestion regime. Thus, host congestion provides us an opportunity to revisit intellectually intriguing, decades-old, fundamental questions related to congestion control architecture and protocols.

We present hostCC, a congestion control architecture that takes the ultimate end-to-end view: it handles both host congestion and network fabric congestion by allocating both host and network resources among competing traffic. The ethos of host *and* network resource allocation is the core that drives the three key technical ideas embodied within hostCC. First, in addition to congestion signals from within the network fabric, hostCC generates host-local congestion signals at processor, memory, and peripheral interconnects at sub-microsecond timescales. These host congestion signals enable hostCC to precisely capture the time, location, and reason for host congestion. The second key technical idea in hostCC is a sub-RTT granularity host-local congestion response: at both the sender and the receiver, hostCC uses host-local congestion signals to allocate host resources between network traffic and host-local traffic. At the sender, hostCC uses host-local congestion response to ensure that network traffic is not starved, even at sub-RTT granularity; at the receiver, hostCC uses host-local congestion response to minimize queueing and packet drops at the host: it modulates host resources allocated to the network traffic at sub-RTT granularity to ensure that NIC queues are drained at the same rate at which network traffic arrives at the NIC. Finally, the third key technical idea in hostCC is to use both host and network congestion signals to perform efficient network resource allocation at RTT timescales.

hostCC admits efficient realization within existing host network stacks, without any modifications in applications, host hardware, and/or network hardware; moreover, hostCC can be integrated with existing congestion control protocols to efficiently handle both host and network fabric congestion. To demonstrate this, we perform an end-to-end implementation of hostCC in the Linux kernel using ~800LOC, and evaluate it along with unmodified Linux DCTCP. Our evaluation demonstrates that, in the presence of host congestion, hostCC reduces queueing and packet drops at the host to a bare minimum, resulting in near-optimal network utilization and tail latency for networked applications. The end-to-end implementation of hostCC, along with all the documentation needed to reproduce our results, is available at https://github.com/Terabit-Ethernet/hostCC.

2 HOST CONGESTION

We start with a brief primer on the host network, with a particular focus on potential congestion points within the host network (§2.1). We then reproduce the host congestion phenomenon from [1] for Linux DCTCP, and provide insights on the root causes of performance degradation in the host congestion regime (§2.2).

2.1 Background: the host network

Figure 1 illustrates the host network datapath. We discuss below the life of a packet from the time it arrives at the NIC until it is transferred to CPU/memory. For brevity, we exclude certain architectural details that are not necessary to understand the hostCC architecture (*e.g.*, DRAM architecture). We primarily focus on the receiver side since host congestion is more prominent at the receiver [1, 22, 24].

The host network datapath is best described in two components—one between the NIC (one end of the PCIe interconnect) to the Integrated IO Controller (IIO, the other end of the PCIe interconnect), and the other between the IIO to memory. We discuss below the steps involved in each component.

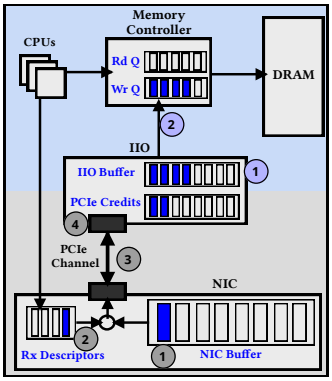


Figure 1: Illustration of host network datapath for the Intel architecture (AMD architecture is conceptually similar) between NIC and CPU/memory for the DDIO disabled case. Discussion in §2.1.

NIC to IIO datapath. NIC and IIO sit at the two ends of the PCIe interconnect.

(1) Upon a packet arrival, the NIC enqueues the packet into its input buffer (typically in a small SRAM [1, 21]).

(2) Next, the NIC fetches a descriptor that provides a host memory address for the NIC to Direct Memory Access (DMA) the packet; the NIC driver periodically replenishes these descriptors.

(3) Importantly, PCIe is a lossless interconnect that uses a credit-based flow control mechanism implemented via a fixed (hardware-specific) number of credits [33]. When credits are available, the NIC instantiates a DMA request over the PCIe (executed using PCIe transactions); DMA'ing a single packet may require multiple PCIe credits [1, 33]. PCIe being a lossless interconnect, the packet can be safely removed from the NIC buffer as soon as DMA is initiated. If PCIe runs out of credits (we will discuss potential reasons below), DMAs cannot be initiated until credits are replenished.

(4) The IIO intercepts each PCIe transaction and initiates writes to memory (discussed below); importantly, a PCIe credit is replenished only when the IIO has successfully issued a write to the memory[1].

IIO to memory datapath. Modern hosts have support for direct cache access (*e.g.*, using DDIO [14]) that allows NICs to DMA packets directly into the last-level cache (LLC). The precise IIO to memory datapath depends on whether DDIO is enabled or disabled. We first describe the datapath with DDIO disabled; we then discuss the case when DDIO is enabled.

(1) Upon receiving a PCIe transaction, IIO enqueues the request in an IIO buffer.

(2) IIO issues the requests from its buffer to the memory controller buffer, and the controller executes the final write to DRAM. Importantly, IIO to memory controller datapath also traverses a lossless interconnect that uses a credit-based flow control mechanism; the IIO can issue a write request to the memory

[1]PCIe transactions are executed at the granularity typically 256 − 512 byte-sized PCIe Transaction Layer Packets (TLPs); however, IIO to memory transactions are executed at the granularity of 64 byte-sized cachelines. Thus, each PCIe transaction requires multiple IIO to memory writes before completion. We will largely ignore this detail since it is not necessary to understand the phenomenon of host congestion.

controller only if the write queue at the memory controller is not full. If this write queue is full (*e.g.*, due to other memory requests from CPUs), the request remains enqueued in the IIO buffer. Once the write queue becomes free, IIO transfers the request to the memory controller write queue; this incurs a cacheline worth of memory write bandwidth. Since IIO to memory controller datapath is lossless, the request can be safely removed from the IIO buffer as soon as it is admitted into the memory controller write queue. At this point, IIO also replenishes the PCIe credit, as discussed above.

If DDIO is enabled, IIO transfers the cacheline to the LLC. This may require evicting an already existing cacheline to the memory controller [9, 14]. Therefore, if DDIO is enabled and does not lead to evictions, it reduces the latency of an IIO write request, since the speed-of-light delay from IIO to LLC is smaller than from IIO to DRAM. However, if it does lead to evictions, we are back to the DDIO disabled case: each eviction not only incurs a cacheline worth of memory write bandwidth, but also higher latency since IIO to LLC write can only be executed after the eviction has completed.

Host congestion. Host congestion may occur due to one or more bottlenecks along the datapath, *e.g.*, memory interconnect [1, 24], peripheral interconnect [33], or hardware components required for memory protection from peripheral devices [1, 6, 9, 28, 33]. To gain intuition on host congestion, consider the memory interconnect bottleneck. When the memory controller write queue is full, we observe a domino effect due to latency inflation for IIO-to-memory requests: as IIO-to-memory latency increases, more requests get queued at the IIO buffer, PCIe credit replenishing incurs larger delays, and PCIe may run out of credits. As a result, PCIe bandwidth remains underutilized, resulting in packet queueing and eventual drops at the NIC buffer.

2.2 Understanding impact of host congestion

Existing network protocols and stacks are not designed to handle host congestion—they primarily target either network fabric congestion only (*e.g.*, [4, 8, 26, 31]), or compute bottlenecks due to inefficient software (*e.g.*, [9, 10, 12, 22, 29]). Indeed, the Google study [1] demonstrates that, even with state-of-the-art congestion control protocol Swift [22] and userspace network stack Snap [29], their production clusters suffer from host congestion resulting in significant queueing and packet drops at the host, throughput degradation, and tail latency inflation. In this subsection, we reproduce the host congestion phenomenon from [1] using Linux DCTCP, and provide insights on root causes for queueing, packet drops, and performance degradation in the host congestion regime.

Setup. To build an understanding of performance degradation in the host congestion regime, we use a setup with two servers connected via a single switch. These are 4-socket (NUMA node) servers with 8 Intel Cascade lake CPU cores per socket, 100Gbps Mellanox CX5 NIC connected to one of the sockets over 128Gbps PCIe 3.0, and DDR4 DIMMs connected to 2 memory channels (with a total maximum theoretical capacity of 375Gbps = 46.9GBps). In terms of resource balancing, our servers are state-of-the-art: the latest commercially available Intel servers have the same set of resources but scaled by a factor (that is, a server with PCIe 4.0 with 256 Gbps

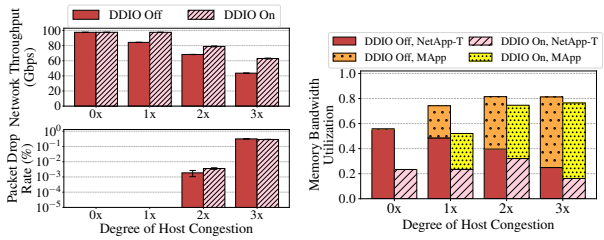


Figure 2: (left) Host congestion leads to significant performance degradation in terms of packet drop rates and throughput (even with no network congestion). DDIO helps a little but observes similar performance degradation. (right) `MApp` **is able to acquire a large fraction of memory bandwidth, leaving little room for** `NetApp-T`**. Discussion in §2.2.**

capacity would also have 4× more cores, 2× more NIC bandwidth, 4× more memory bandwidth, etc.) [1, 13]. Our servers run Linux kernel 5.4.0 and, by default, use 4K MTUs (similar to [1]), and all available optimizations (segmentation offloads like TSO and GRO, accelerated request flow steering, etc.) that have been shown to achieve best-possible Linux performance [9].

We use three applications—(a) a `NetApp-T` that generates 4 long flows, each flow from one sender-side CPU core to one receiver-side CPU core on the NIC-local NUMA node (DCTCP needs a minimum of 4 cores to saturate 100Gbps NIC in an uncongested scenario); (b) a `NetApp-L` that generates latency-sensitive RPCs (of sizes varying from 128B to 32KB) between a sender-side and receiver-side CPU core on the NIC-local NUMA node; and (c) an `MApp` that generates CPU-to-memory traffic with 1 : 1 read-write ratio and sequential memory access pattern; we increase the offered load to the memory interconnect by the `MApp` from 1× to 3×, by increasing the number of `MApp` CPU cores, and therefore increasing the number of in-flight memory requests (in the absence of any other source of memory traffic, using 1× to 3× `MApp` cores results in a total observed memory bandwidth of 16.0GBps, 28.7GBps, and 34.8GBps, respectively). We use standard benchmark tools for these applications—iperf [17] for `NetApp-T`, netperf [20] for `NetApp-L`, and MLC [16] for `MApp`.

[Figure 2] Host congestion leads to degraded network throughput (>35%), even when DDIO is enabled. Figure 2 shows throughput, packet drops, and memory bandwidth utilization with increasing degrees of host congestion, with and without DDIO. The case of 0× is easy: there is no host congestion, and thus network traffic is able to saturate the access link bandwidth. Enabling DDIO reduces memory bandwidth utilization since the CPU is able to consume the data before it is evicted; nevertheless, memory bandwidth utilization is non-zero because evictions do happen due to cache pollution—since LLC is shared across all cores, one cannot guarantee a perfect cache hit rate [9]. The case of 1× is more interesting; let us start with the DDIO disabled case. Here, as shown in the right figure, memory bandwidth utilization is now close to saturation[2]; thus, memory access latency (and thus, the number of CPU cycles per memory access) starts to increase. As a result, network throughput is now compute bottlenecked (more precisely,

[2]Memory bandwidth utilization at saturation depends upon DRAM hardware as well as application workload (read-write ratio, access patterns, etc.) [18], and is typically lower than the maximum theoretical bandwidth.

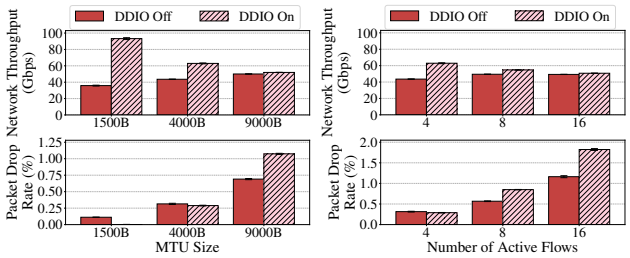


Figure 3: The impact of host congestion worsens with larger MTU size and number of flows, with 3× degree of host congestion. DDIO enabled case, in particular, suffers more than DDIO disabled case.

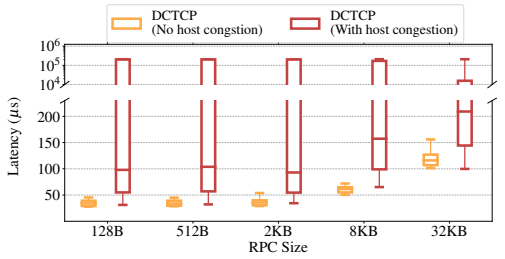


Figure 4: Host congestion leads to orders of magnitude tail latency inflation. The figure shows {P50, P90, P99, P99.9, P99.99} latencies (represented by whiskers) for `NetApp-L`, with and without host congestion, when `NetApp-T`, `NetApp-L`, and `MApp` are run together with DDIO disabled. DDIO enabled results, shown in §5.2, are identical. Discussion in §2.2.

by receiver-side buffers) and is unable to saturate the access link bandwidth. DDIO shines in such a scenario: lower memory bandwidth utilization results in lower memory access latency and fewer CPU cycles per memory access, allowing network traffic with DDIO to continue to saturate the access link bandwidth.

For 2× and 3×, memory bandwidth is now saturated, resulting in inflated memory access latency. We now see the domino effect discussed earlier: PCIe becomes underutilized, and packets get queued (and eventually dropped) at the NIC; even in this simple setup, we observe 0.3% drops. Interestingly, while DDIO can be slightly helpful in improving network throughput, it has negligible impact on packet drop rate; this is because of the reasons discussed earlier—as shown in the right figure, memory bandwidth utilization is similar for both DDIO enabled and disabled (that is, majority of cachelines are evicted from LLC before the CPU can consume them). We will discuss in §5 that at 2× and 3×, DCTCP is operating in the AIMD regime: senders keep on increasing sending rate, resulting in queue build-up and drops at the NIC when total sending rate exceeds the instantaneous host interconnect capacity; packet drops leads to rate reduction, followed by subsequent sawtooth behavior.

Figure 2(right) shows that, as we increase the number of `MApp` cores, network traffic is allocated a decreasing fraction of memory bandwidth. More work is needed to understand the precise reasons; our evaluation suggests that memory bandwidth allocation is essentially proportional to the load generated by individual entities (IIO or CPU); this implies that, as `MApp` cores increase, `MApp` generates increasingly larger load[3] but the maximum number of requests issued by IIO remains the same (dependent on the PCIe credit limit). As a result, CPUs are able to quickly acquire a larger fraction of memory bandwidth, creating even more host contention for network traffic.

[Figure 3] Impact of host congestion worsens with increasing MTU sizes and a larger number of flows. Larger MTU sizes and a large number of connections have been shown to improve network throughput when compute at the host is bottlenecked due to inefficient software [1, 9]. Figure 3 shows that, in the presence of host congestion, these optimizations can, in fact, hurt performance—we see a significant increase in packet drop rates for both DDIO enabled and disabled cases; and, while we observe a slight improvement in throughput for the DDIO disabled case (due to reduction in compute cycles required to process each packet), we also observe significant throughput reduction for the DDIO enabled case.

The trends for the DDIO disabled case are easy to understand: for AIMD-style congestion control protocols, it is known that packet drop rates increase with MTU size [30] and the number of flows [2, 32]. The trends for the DDIO enabled case are related to the inefficacy of DDIO observed in [9]—with an increase in MTU size and/or number of flows, DDIO observes an increase in cache eviction rates (and thus, increase in memory bandwidth utilization); as discussed in §2.1, each such eviction not only incurs a cacheline worth of memory bandwidth as in DDIO disabled case, but also higher latency than DDIO disabled case since IIO to LLC write can only be executed after the eviction has completed. As a result, DDIO enabled case at 3× host congestion observes even higher packet drop rates than the DDIO disabled case for large MTU sizes and large numbers of flows.

[Figure 4] Host congestion can cause orders of magnitude tail latency inflation for latency-sensitive applications. We now run the three applications—`NetApp-T`, `NetApp-L`, and `MApp`—together, with 3× host congestion. Since `NetApp-L` introduces a tiny amount of additional traffic, `NetApp-T` and `MApp` performance is similar to previous experiments; we thus focus on `NetApp-L` results. We observe significant latency inflation for `NetApp-L` in the host congestion regime. This latency inflation is caused due to three reasons (a) queueing delay at the NIC buffer; (b) retransmission and timeout delays due to drops at the NIC buffer; and (c) larger CPU processing delays due to inflated in CPU cycles in memory accesses caused by host congestion. Building upon previous observations (close to 0.3% drops), P99 latency is dominated by (a) and (c), and P99.9 is dominated by (b)—for both DDIO enabled and disabled (that observe similar drop rates), P99 latency inflation is roughly $60 - 100\mu s$ which is close to the worst-case queueing delay at the NIC buffer; at P99.9, latency inflation is close to 200ms, which is the default Linux minimum retransmission timeout (RTO) value. Smaller RPCs suffer from higher tail latency inflation because any packet drop necessitates a timeout; for larger RPCs, Linux Tail Loss Probe (TLP) [34] mechanism is effective (with a smaller timeout) when there is more than one in-flight packet. Isolating NIC buffers does not solve this problem: a smaller NIC buffer size would incur a larger number of drops, increasing (b); a larger NIC buffer size, on the other hand, would increase (a).

[3]This is because each CPU core can maintain a hardware-specific maximum number of in-flight memory requests, equal to so-called Line Fill Buffer (LFB) size. On our servers, this limit is typically $10 - 12$.

3 hostCC

Figure 5 illustrates the end-to-end hostCC architecture. In this section, we provide details on the three key technical ideas embodied within the architecture—host congestion signals (§3.1), host-local congestion response (§3.2), and network resource allocation (§3.3).

3.1 Host congestion signals

hostCC, in addition to classical congestion signals from within the network fabric, generates host-local congestion signals. More precisely, hostCC uses IIO buffer occupancy as a congestion signal.

To understand why, recall the host datapath discussed in §2.1. Let $\mathcal{R}$ be the rate at which NIC receives data, $\mathcal{P}$ be the maximum in-flight bytes that PCIe can maintain (a fixed hardware-dependent constant that depends on the maximum number of credits, and on TLP size), ℓ_p be the latency between the NIC to the IIO (a fixed hardware-dependent constant), and ℓ_m be the latency between the IIO and the memory controller. As discussed in §2.1, ℓ_m depends upon multiple factors, including the memory controller write queue size, load on the memory controller, whether DDIO is enabled (and whether an eviction is triggered), and the speed-of-light-latency between the IIO and memory [7, 14, 23]. We will refer to $\ell_m^{\min}$ and $\ell_m^{\max}$ as the minimum and the maximum value of ℓ_m.

Given the above, PCIe throughput is given by $\mathcal{P}/\max\{\ell_p, \ell_m\}$, that is, PCIe utilization is dominated by the maximum latency among all links along the path from the NIC to memory[4]. The IIO buffer occupancy is equal to $\mathcal{R} \times \ell_m$, that is, the maximum number of bytes that can be received by the IIO while it is waiting for credits to be replenished (if it had credits, IIO would issue the requests).

In the regime of *no* host congestion, $\ell_m \approx \ell_m^{\min} \ll \ell_p$ by hardware design, making ℓ_p the dominant factor in PCIe utilization. Thus, by hardware design, IIO replenishes PCIe credits at a rate no lower than PCIe consumes credits, and PCIe bandwidth utilization matches the rate at which NIC receives data; that is, $\mathcal{P}/\ell_p \geq \mathcal{R}$. However, in the host congestion regime, the IIO buffer occupancy can be anywhere between $\mathcal{R} \times \ell_m^{\min}$ and $\min\{\mathcal{R} \times \ell_m^{\max}$, maximum number of PCIe credits$\}$, where the second expression in the upper bound is achieved when IIO is unable to replenish PCIe credits due to large ℓ_m. Our measurements in Figure 8 provide an empirical confirmation.

We are now ready to describe the benefits of using IIO occupancy as the host congestion signal. First, IIO occupancy provides accurate information about time, location, and reason for host congestion: IIO occupancy increases immediately upon the memory controller becoming congested (accuracy in time and location) and it increases *only if* memory controller is congested (accuracy in reason). Second, IIO occupancy can be combined with another statistic—IIO insertion rate, defined as the rate at which PCIe inserts data into the IIO buffer—to measure various other useful metrics; for instance, instantaneous PCIe throughput (capturing the rate at which NIC buffers are drained) is equal to instantaneous IIO insertion rate times the cacheline size, host delay ($\ell_p + \ell_m$) can be computed using Little's Law [27]), etc. Third, IIO occupancy and IIO insertion rates can be measured using two registers typically available on commodity hardware, allowing hostCC to work without any hardware modifications/support. Finally, IIO measurements are done at the processor interconnect, outside the NIC-to-memory datapath; thus, IIO occupancy measurements are not impacted by host congestion. We provide more details in §4.1, including details on how hostCC measures IIO occupancy and IIO insertion rates at sub-μs granularity using existing hardware.

[4]Here is one way to visualize this intuitively: if $\ell_p \to \infty$ or $\ell_m \to \infty$, PCIe utilization tends to 0; furthermore, if $\ell_m \gg \ell_p$, PCIe utilization is bottlenecked by ℓ_m (host congestion) and if $\ell_p \gg \ell_m$, PCIe utilization is bottlenecked by ℓ_p. This follows almost immediately using the analysis in [19].

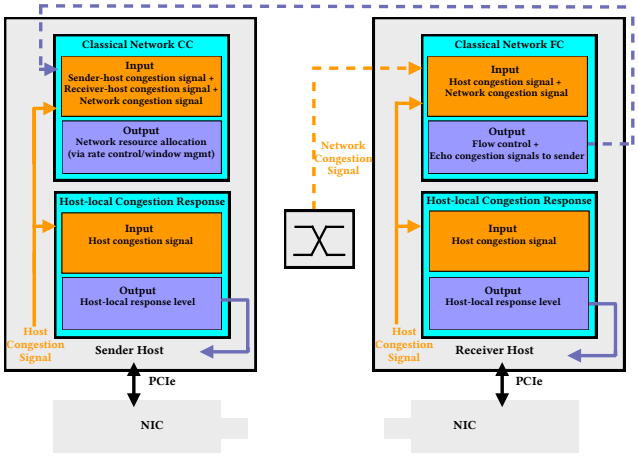


Figure 5: hostCC architecture overview. Discussion in §3.

3.2 Host-local congestion response at sub-RTT granularity

A conceptual interpretation of classical congestion control protocols is that, to handle congestion within the network, these protocols (along with network switches) allocate network resources across entities competing at the congestion point. The second key technical idea in hostCC architecture is motivated by this conceptual view: to handle *both* host and network congestion, hostCC allocates *both* host and network resources among entities competing at the congestion point. To achieve this, hostCC introduces a host-local congestion response—at both the sender and the receiver host—that uses host congestion signals discussed in the previous subsection to allocate host resources across network traffic and host-local traffic.

Resource allocation depends on the underlying policy; hostCC architecture does not dictate the precise resource allocation policy—just like different network resource allocation mechanisms use different network allocation policies (max-min fairness, weighted max-min fairness, prioritization, etc.), we envision hostCC to embody various host resource allocation policies and respective implementation. For the following discussion, we assume that the policy periodically computes a target network bandwidth (B_T), and feeds it as input to hostCC. In addition, the host-local congestion response takes as input IIO occupancy I_S as the host congestion signal (using a threshold I_T, where $I_S > I_T$ indicates host congestion) and PCIe bandwidth utilization B_S (computed using IIO insertion rates, as discussed in the previous subsection). The congestion response mechanism operates on a per-packet basis, and makes a decision whether to increase or decrease the resource allocation to both network traffic and host-local traffic.

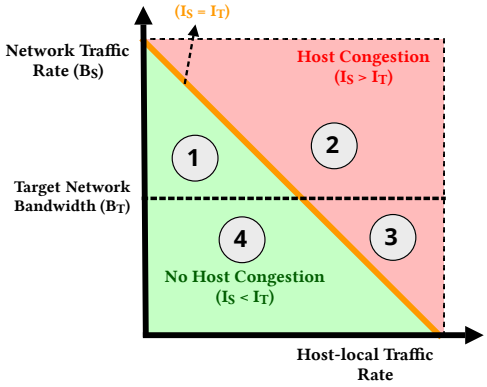


Figure 6: Resource allocation decisions made by hostCC under different regimes of operation. Discussion in §3.2.

Given the above, hostCC's host-local congestion response mechanism is best described in terms of four possible regimes of operation, depicted in Figure 6. We describe individual regimes and corresponding host-local congestion response below.

① **[No host congestion, network traffic has met the target network bandwidth].** In this regime, host is not congested ($I_S < I_T$) and the network traffic is using more resources than what is needed to meet the target bandwidth (that is, $B_S > B_T$). Thus, the host-local congestion response mechanism increases the resources allocated to the host-local traffic. This is the right action to take since, in absence of host congestion, more host resources can be allocated to either network traffic or host-local traffic; moreover, since the network traffic has already met the target network bandwidth, we want to ensure that host-local traffic is not backpressured unnecessarily. Thus, the host-local congestion response mechanism increases resources allocated to the host-local traffic. It is possible that host-local traffic does not need additional resources; hostCC handles this case by relying on the AIMD-style mechanisms used in network congestion control protocols—since host is not congested, network traffic does not get marked with congestion signals at the host allowing network traffic to increase its rate and acquire unused host resources (if network fabric is not congested).

② **[Host congestion, network traffic has met the target network bandwidth].** In this regime, host is congested and the network traffic is using more resources than what is needed to meet the target bandwidth. Thus, the right action in this regime is to reduce resources allocated to the network traffic and to not reduce resources allocated to the host-local traffic. To achieve this, hostCC again relies on AIMD-style mechanisms used in network congestion control protocols: it echoes the host congestion signal to the network congestion control protocol resulting in reduction in network traffic rate.

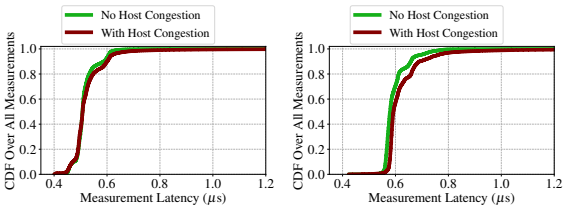
③ **[Host congestion, network traffic has not met the target network bandwidth].** In this regime, host is congested but the network traffic is allocated fewer resources than what is needed to meet the target bandwidth. Since there is host congestion, we must reduce the allocated resources; since network traffic has not met the target bandwidth, hostCC first reduces the resources allocated to the host-local traffic (this happens at sub-RTT granularity). However, this is not sufficient to avoid NIC buffer build up and packet drops. To see why, recall that the PCIe bandwidth utilization B_S and target network bandwidth B_T may be much lower than the rate $\mathcal{R}$ at which NIC is currently receiving traffic. By reducing resources allocated to host-local traffic in order to accommodate B_T network traffic bandwidth, the host-local congestion response merely ensures that NIC buffers build up at a rate no faster than $\mathcal{R} - B_T$. Without any explicit congestion signal, the network traffic will have no reason to reduce $\mathcal{R}$, resulting in increasingly more queueing at the NIC and eventual packet drops. To avoid this, the host-local congestion response also echoes the host congestion signal to the network congestion control protocol resulting in reduction in network traffic rate. We note that, if $\mathcal{R} < B_T$, the host-local congestion response may lead to inefficient resource allocation due to reducing resources allocated to both network and host-local traffic; nevertheless, hostCC takes the above conservation decision temporarily to minimize NIC buffer buildup and packet drops.

④ **[No host congestion, network traffic has not met the target network bandwidth].** In this regime, host is not congested and the network traffic has fewer resources than what is needed to meet the target bandwidth. Thus, the host-local congestion response allocates more resources to network traffic; this allocation is again implicit, in that, it again relies on the AIMD-style mechanisms—since host is not congested, network traffic does not get marked with congestion signals at the host allowing network traffic to increase its rate and acquire unused host resources (if network fabric is not congested). Increasing resources allocated to the network traffic implicitly may take multiple RTTs, or may not even be feasible (*e.g.*, due to network congestion); nevertheless, the host-local congestion response makes the conservation decision to not increase resources allocated to the host-local traffic in this regime to avoid host congestion before reaching the target network bandwidth.

The host-local congestion response in hostCC uses host congestion signals (that are generated at sub-microsecond granularity) and is purely local to the host; thus, it can be done at sub-RTT granularity. As a result, even if host-local traffic changes at sub-RTT granularity, the host-local congestion response can ensure high host resource utilization while maintaining target network bandwidth according to any given policy.

3.3 Network resource allocation at RTT granularity

Consider the regime of host congestion ($I_S > I_T$) and network traffic transmitting at rate $\mathcal{R} > B_T$ (which will lead to $B_S > B_T$). In this scenario, the right action is for the network traffic to reduce its rate. To achieve this, hostCC's host-local congestion response mechanism does not take any action; instead, hostCC simply echos the host congestion signal back to the network congestion control protocol (in addition to any network congestion signal). This has two benefits. The first benefit is conceptual: it enables a clean separation of concerns, where the host-local congestion response handles host congestion at sub-RTT granularity, and network congestion control continues to handle network congestion at RTT



(a) CDF for I_S read latency (b) CDF for B_S read latency

Figure 7: hostCC generates host congestion signals that are not on the NIC-to-memory datapath; it can thus measure both I_S (left) and B_S (right) at sub-μs timescales independent of host congestion.

granularity as they do today. Second, hostCC can be integrated with any network congestion control protocol: the only difference is that the protocol will now use both host and network congestion signals for host resource allocation.

End-to-end hostCC behavior. We provide an intuitive description of hostCC's end-to-end behavior. Suppose the network traffic is operating at rate $\mathcal{R} > B_T$. Suppose severe host congestion is introduced abruptly; then, as evaluated in §2.2, B_S will reduce to a small value below B_T, I_S will grow beyond I_T, and the host-local congestion response will kick in quickly to increase the host resources allocated to network traffic to accommodate $\sim B_T$ bandwidth (potentially by reducing resources allocated to host-local traffic causing host congestion). However, for a few RTTs, the arrival rate of network traffic $\mathcal{R}$ at receiver NIC will still be higher than B_T, resulting in hostCC echoing host congestion signals back to the sender. The sender will eventually reduce $\mathcal{R}$ until it converges to the target network bandwidth B_T.

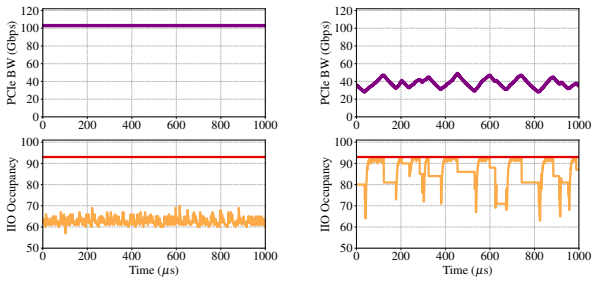
4 hostCC IMPLEMENTATION

We now provide details on how we incorporate the three key technical ideas from the previous section in an end-to-end hostCC implementation—generating host congestion signals (§4.1), using host congestion signals for host-local congestion response at sub-RTT granularity (§4.2) and using both host and network congestion signals for network resource allocation at RTT granularity (§4.3). We implement hostCC as a loadable Linux kernel module using ~800 LOC; hostCC works out-of-the-box with various existing congestion control protocols, without requiring any modifications to applications, host hardware, and/or network hardware.

4.1 Host congestion signals

hostCC collects host congestion signals at sub-microsecond granularity. To maintain brevity, we describe an implementation atop Intel architectures (AMD architecture is conceptually very similar).

Most hardware counters are exposed using model specific registers (MSRs) [15]. The MSR for the IIO occupancy value at time t (denoted by $R_{OCC}(t)$) maintains the cumulative value of the occupancy, incremented at IIO clock frequency (denoted by F_{IIO}); for example, F_{IIO} = 500MHz for our servers. The average IIO occupancy I_S between any two time instants t_1 and t_2 is computed using: $I_S = (R_{OCC}(t_2) - R_{OCC}(t_1))/((t_2 - t_1) * F_{IIO})$. The time difference $(t_2 - t_1)$ is measured by the standard method of reading



(a) No Host Congestion (b) With Host Congestion

Figure 8: Variation of IIO occupancy I_S and PCIe bandwidth utilization B_S with time for 1ms period for the experiment of Figure 2 without host congestion (left) and with 3× host congestion (right). In absence of host congestion, $B_S \approx$ 103Gbps (line-rate bandwidth including PCIe overheads with 4K MTUs) and $I_S \approx$65, which corresponds to hardware-specific IIO-DRAM bandwidth-delay product (§3.1). During host congestion, I_S increases to a maximum value of ~93 (shown in red), resulting in a reduction of B_S, queueing and packet drops at the NIC, and network CC reducing the rate.

the TSC register, which provides nanosecond level time accuracy. To minimize read latency, we used inline assembler code to read the TSC register. The read latency is bottlenecked by the read call to the IIO occupancy MSR register. On our servers, we measured that each TSC read took < 2ns, and each MSR read call took <~600ns. Thus, we are able to collect host congestion signal (IIO occupancy I_S) at sub-μs timescales. Similarly, to measure PCIe bandwidth utilization B_S, we read another MSR counter (denoted by R_{INS}) that stores cumulative IIO insertions. Thus, similar to IIO occupancy, we compute the average rate of IIO insertions I between time instants t_1 and t_2 as $I = (R_{INS}(t_2) - R_{INS}(t_1))/(t_2 - t_1)$. The PCIe bandwidth utilization between t_1 and t_2 is thus I times the cacheline size. Both congestion signals—I_S and B_S—require reading CPU registers, which does not overlap with the NIC-to-memory datapath; thus, hostCC is able to measure these signals at a sub-μs timescale, independent of host congestion (as shown in Figure 7).

We note a low-level detail. Similar to existing network congestion control protocols [22, 31], hostCC uses an exponentially weighted moving averaging (EWMA) for its congestion signals, I_S and B_S, rather than their instantaneous values. The EWMA weight used in hostCC has a standard tradeoff in terms of aggressiveness in hostCC's response to host congestion and delayed reaction—using a large weight will quickly trigger host-local congestion response as well as network congestion control response (the latter because congestion signals will be generated more quickly, and echoing them to network congestion control protocol will trigger its response) which could lead to overreaction in presence of temporary burst of host congestion; a small weight, on the other hand, will delay congestion response. hostCC uses a default weight value of 1/8 for I_S and 1/256 for B_S (that is, last 8 IIO occupancy values and last 256 PCIe bandwidth utilization values are dominant). An important note here is that we use the same weight for I_S for detecting host congestion and echoing congestion to the network congestion control protocol; this works because network congestion control protocols typically maintains EWMA of their parameters (*e.g.*, α in

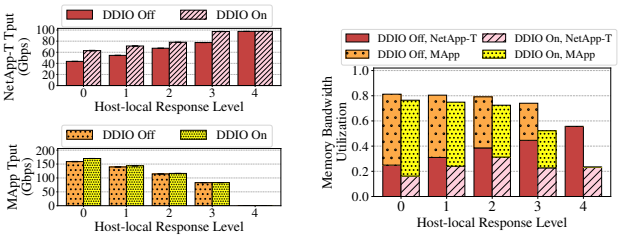


Figure 9: Evaluating MBA efficacy: `NetApp-T` is able to acquire more host resources with higher host-local response levels (that is, more backpressure on `MApp`). As a result, `NetApp-T` gets higher throughput (left) and a larger fraction of memory bandwidth (right). Discussion in §4.2 and §5.

DCTCP) used to trigger network congestion response. We show in Figure 8 the IIO occupancy and PCIe write bandwidth with time for the baseline setup in §2, with and without host congestion.

4.2 Host-local congestion response

The host-local congestion response in hostCC implementation takes as input host congestion signals (I_S and B_S) and corresponding thresholds (I_T and B_T) and triggers the response discussed in §3.2. Any resource allocation mechanism must operate using the interface provided by the system that offers resources; we describe our implementation using the Intel Memory Bandwidth Allocation (MBA) tool. This interface uses a simple multi-level backpressure mechanism to the host-local traffic (recall, host interconnect is lossless; thus, backpressure is a natural mechanism to reduce CPU to memory traffic). Higher levels mean more backpressure; that is, higher levels result in fewer resources allocated to host-local traffic. Internally, MBA alters the rate at which any CPU core can generate memory traffic by introducing additional latency to every read/write request that observes an L2 cache miss on that core. Therefore, average traffic generated by a core to memory is inversely proportional to the introduced additional latency: (LFB size × cacheline size)/per-access latency, where LFB size is as discussed in §2.2. The current MBA interface allows 10 levels, with higher levels introducing higher latency resulting in lower CPU to memory traffic [37]. AMD's Memory Bandwidth QoS control tool uses a similar interface [5].

The desired rate level is realized by performing a single MSR write to MBA-specific registers. For each socket (NUMA node), MBA maintains 8 MSR registers, one for each "class-of-service" (COS), which can include any number of CPU cores within a particular socket. The assignment of CPU cores to COS can also be changed dynamically using an MSR write to another control register. Therefore, a single MSR write can simultaneously alter the CPU to memory traffic for any number of CPU cores within a socket. Our current implementation uses 5 local response levels $\ell = \{0, 1, 2, 3, 4\}$, where each successive level ℓ introduces increasingly larger latency—level 0 corresponds to no backpressure, and level 4 corresponds to maximum backpressure. We separate out network traffic cores from host-local traffic cores using different COS and introduce backpressure only to the latter.

Figure 9(left) shows network throughput when we hard code each individual host-local response level ℓ. We observe that, with each level ℓ, throughput increases as expected: with more aggressive backpressure on host-local traffic, network throughput increases from 43Gbps at level 0 to ∼100 Gbps at level 4[5]. To understand Figure 9(right), that shows corresponding memory bandwidth utilization, we must understand the difference between application-level throughput and corresponding load on memory bandwidth. In particular, `NetApp-T` and `MApp` use ∼2.1× and ∼1.33× memory bandwidth per unit of application-level throughput (due to data copy and processor interconnect overheads, respectively). When DDIO is enabled, the network application achieves higher throughput at lower response levels (for *eg.*, ∼100Gbps at level 3 instead of 4) because `NetApp-T` utilizes smaller amount of memory bandwidth per unit throughput (since DDIO cache eviction rate is typically less than 100%). Consequently, it requires smaller amount of backpressure to the `MApp` to achieve the same network throughput. Our measurements also suggest that it takes ∼22μs to perform a write to today's MBA MSR registers due to MBA limitations; to verify that this was not an hostCC implementation artifact, we performed MSR writes using inline assembler core such that we only execute a single assembly-level instruction for the write and still incurred 22μs latency (2× smaller than our network RTT). We discuss this limitation of MBA in §6.

4.3 Network resource allocation

hostCC can be integrated with many existing network congestion control protocols. We focus here on hostCC's implementation with ECN-based protocols; we discuss extensions to integrate hostCC with other protocols in §6. hostCC requires no modification to existing network congestion control protocol implementations: hostCC simply generates ECN markings on the ACKs sent back to the sender if $I_S > I_T$ (if the packet was already marked by the switch, no modifications are made). The current hostCC implementation performs ECN marking at the IP layer using 2 out of 6 DSCP bits (as in RFC 3168 [35]) by exploiting a hook to the `ip_recv` function provided by the widely used NetFilter kernel module [38] in Linux. hostCC's host-local congestion response does exactly what today's switches do—mark both these bits as 1 to indicate congestion—before delivering the IP datagram to the transport layer. The transport layer then processes the ECN marked datagram in exactly the same manner as it would for any ECN marked packet at today's switches.

5 hostCC EVALUATION

We now evaluate hostCC performance. Our goals are three-fold:

- Understanding benefits of hostCC's core ideas—host congestion signals, host-local congestion response, and performing network resource allocation using both host and network congestion signals—to application-level performance;

[5]The host-local response level 3 corresponds to the maximum possible latency that can be introduced using the current MBA implementation to all `MApp` cores. However, this added latency does not provide sufficient backpressure to `MApp` to allow the network application to reach line rate throughput (as shown in Figure 9, `NetApp-T` only achieves ∼77Gbps throughput at level 3). In order to emulate an MBA response with larger added latency, we introduced a level 4 in current hostCC implementation—when emulating level 4, we pause the execution of the `MApp` process using the SIGSTOP signal, and when switching back from level 4 to lower levels, we resume the `MApp` process using the SIGCONT signal.

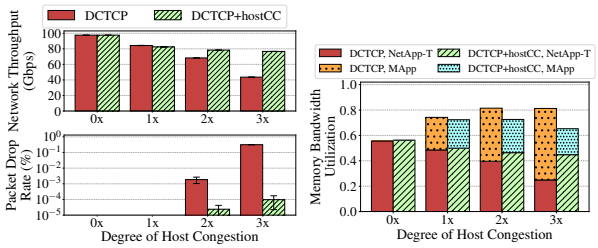


Figure 10: (left) hostCC allows network traffic to achieve its target network bandwidth of B_T = 80Gbps, while simultaneously reducing packet drop rates by orders of magnitude, even with a high degree of host congestion. (right) with hostCC, MApps no longer acquire a large fraction of memory bandwidth, even with high degree of host congestion.

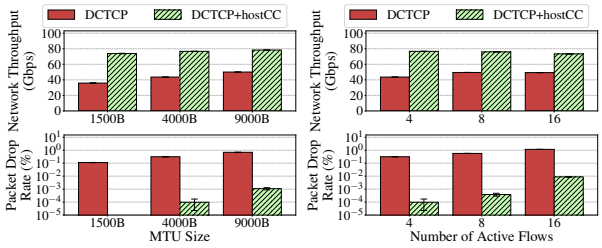


Figure 11: Even with high degree of host congestion (3× in this experiment), hostCC consistently maintains its benefits across MTU sizes and number of flows.

- Deep dive into hostCC microscopic behavior (capturing host congestion signals, host-local congestion response, and network resource allocation) in the host congestion regime;
- Develop lessons for the host congestion regime that would be useful to design future host hardware and network stacks that can better enable the reaction to host congestion.

Throughout this section, we use Linux DCTCP as our network congestion control protocol (network CC) since Linux DCTCP is a stable open-sourced implementation that works with commodity hardware. We primarily focus on the DDIO disabled case since results are easier to explain (as discussed earlier, performance for DDIO enabled case depends on cache eviction policies); hostCC evaluation for the DDIO enabled case is presented in §5.2. Unless mentioned otherwise, we use IIO occupancy threshold I_T = 70 and network target bandwidth B_T = 80Gbps for hostCC (we present sensitivity analysis of hostCC performance with I_T and B_T in §5.3); for DCTCP, we use default parameters from [4]. We do not perform any experiment-specific parameter optimization.

5.1 hostCC benefits

We first evaluate hostCC on the same setup as in §2.2—this setup does not have any network congestion, allowing us to gain insights about hostCC performance in the host congestion regime. We then extend the setup to the one used in [1]; this setup includes network congestion and allows us to gain insights on hostCC performance in the presence of network congestion (with and without host congestion).

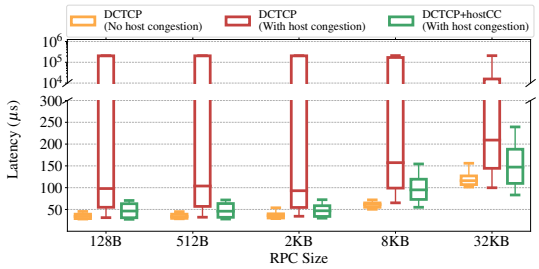


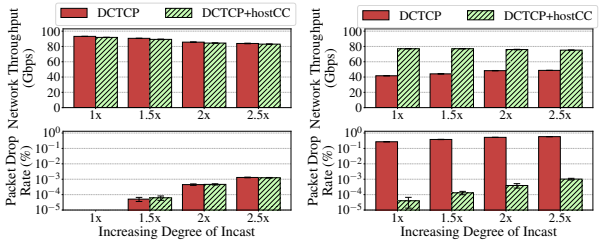
Figure 12: Even with a high degree of host congestion (3× in this experiment), hostCC incurs minimal latency inflation compared to no host congestion, significantly improving tail latency across all RPC sizes.

hostCC avoids throughput degradation for network traffic while simultaneously reducing packet drops at the host by orders of magnitude. Figure 10(left) shows that, when the degree of host congestion is so low that NetApp-T can reach its target network bandwidth without creating bottlenecks within the host network (0× and 1× cases), hostCC has negligible impact on NetApp-T throughput (which is bandwidth bottlenecked for the 0× case and CPU bottlenecked for the 1× case). More interestingly, in the presence of host congestion, hostCC allows NetApp-T to achieve throughput close to the desired target network bandwidth (80Gbps in this experiment), even with a high degree of host congestion. Essentially, using the host congestion signals, the sub-RTT host-local congestion response promptly reduces the resources allocated to the MApp traffic whenever the network throughput falls below B_T in presence of host congestion (hostCC steady-state behavior illustrated later in §5.4) Moreover, Figure 10(right) shows that these benefits to NetApp-T do not come at the cost of starving MApp traffic: hostCC's host-local congestion response also increases resources allocated to the MApp traffic whenever NetApp-T is able to sustain the target network bandwidth. Figure 10(left) also shows that hostCC reduces packet drop rates to a bare minimum, since the host-local congestion response and network CC (using host congestion signals) work in tandem to keep the NIC buffer occupancy low for a larger fraction of time.

We observe that, with a high degree of host congestion, the total memory bandwidth utilization is slightly reduced. We believe this behavior is not due to hostCC's architecture but rather due to the coarse granularity of host resource allocation using existing tools (Intel MBA, in this case). Due to such coarse granularity of the allocation, the MApp sometimes gets backpressured much more than than it needs to, in order to accommodate additional NetApp-T traffic. Figure 9 shows an example of this behavior: when we switch from host-local response level 3 to 4, NetApp-T gains 5.2GBps of memory bandwidth, while MApp loses 13.8GBps of memory bandwidth. We discuss potential avenues for future hardware support for improved host-local congestion response in §6.

Figure 11 shows that hostCC consistently achieves benefits in terms of maintaining target network bandwidth and reduced packet drop rates across all evaluated MTU sizes and number for flows.

hostCC observes minimal tail latency inflation for latency-sensitive traffic, even with high degree of host congestion. Figure 12 shows the observed latency under the same multi-tenant



(a) Network congestion (b) Host + Network congestion

Figure 13: (left) In the presence of network congestion and absence of host congestion, hostCC achieves performance similar to network CC indicating minimal overheads; (right) in the presence of both host and network congestion, hostCC consistently provides benefits similar to Figure 10.

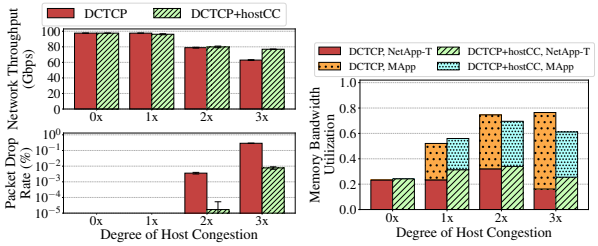


Figure 14: hostCC, with DDIO enabled, provides benefits similar to the DDIO disabled case in Figure 10.

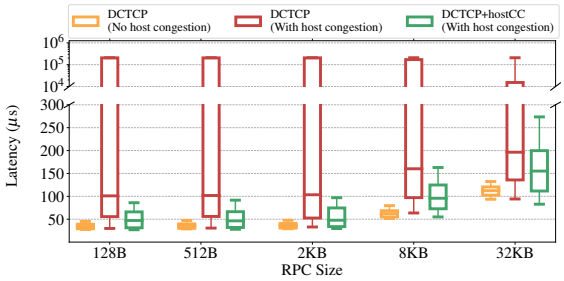


Figure 15: hostCC, with DDIO enabled, provides benefits in terms of latency improvements similar to the DDIO disabled case in Figure 12.

evaluation setup as in Figure 4 (where we use all apps NetApp-T, NetApp-L and MApp together). hostCC observes minimal latency inflation in the regime of host congestion due to two reasons: (1) hostCC's host-local congestion response ensures minimal queueing delay at the host; and (2) hostCC significantly reduces packet drop rates, avoiding retransmission and timeout delays. Results for small-sized RPCs provide evidence for the first reason: recall, from §2.2, that P99 latency in this experiment is dominated by NIC queueing delays; the figure shows that, despite the high degree of host congestion, hostCC incurs a minuscule latency inflation of 13μs for 128B RPCs (when compared to no host congestion scenario). All results provide evidence for the second reason: we observe no timeouts even at P99.9 percentile.

hostCC maintains its benefits even in the presence of both host and network congestion. Figure 13 evaluates hostCC performance in the presence of network congestion, with and without host congestion. For this experiment, we use an incast workload with two senders and a single receiver, directly connected to a switch. We vary the degree of network congestion by varying the degree of incast (the total number of active concurrent flows at the receiver) from 4 to 10 (1× to 2.5× degree of incast). We observe that, in the absence of host congestion, network CC without hostCC observes increased packet drop rates with an increase in the degree of network congestion (as one would expect); since there is no host congestion, hostCC performance is near-identical to the network CC performance indicating that hostCC has minimal overheads in the absence of host congestion. In the presence of both host and network congestion, however, network CC performance without hostCC suffers from high packet drops rates and reduced throughput; in this scenario, hostCC provides significant benefits using all three of its core ideas: it collects host congestion signals at sub-μs timescales, it is able to modulate host resources allocated to the network traffic so as to maintain target network bandwidth, and incurs minimal packet drops rates by ensuring that network CC converges to a rate that matches available network and host resources. This experiment demonstrates that hostCC interpolates well with network CC even in the presence of both host and network congestion.

5.2 hostCC results with DDIO enabled

Figure 14 shows hostCC results using the same setup as in Figure 10, but with DDIO enabled. We use I_T = 50 here because the observed IIO occupancy value when there is no host congestion is smaller when DDIO is enabled (~45, compared to ~65 when disabled). This is due to smaller average IIO-to-memory latency with DDIO enabled as discussed in §2.1. We observe similar trends as in Figure 10—hostCC ensures that network traffic is able to achieve the target network bandwidth, while reducing packet drop rates to a bare minimum. With large degree of host congestion, the absolute packet drop rate with hostCC is slightly higher than other evaluated cases (hostCC still helps reduce packet drop rates by ~37× compared to the case without hostCC); identifying the precise reasons for this observation requires more visibility into DDIO-related hardware operations like cache eviction policies (as also noted in [9, 11]).

We also observe that, with DDIO enabled, MApp is able to acquire larger fraction of memory bandwidth (compared to DDIO disabled case in Figure 10) for any given degree of host congestion. Figure 9 helps explain this behavior—when DDIO is enabled, NetApp-T is able to sustain higher average throughput at a lower host-local response level; hence, MApp experiences smaller amount of backpressure to achieve the target network bandwidth.

Figure 15 shows benefits of hostCC in terms of latency for NetApp-L using the same setup as in Figure 4, but with DDIO enabled. We observe that latency inflation without and with hostCC is identical to that of Figure 12; this is because both DDIO enabled and disabled cases observe similar level of packet drop rates for 3× degree of host congestion (as shown in Figure 2), leading to similar tail latency inflation for NetApp-L (as discussed in §2.2).

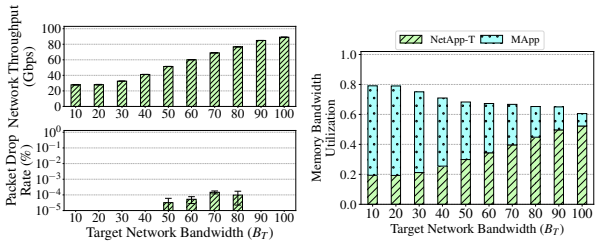


Figure 16: hostCC consistently maintains its benefits across varying network target bandwidth B_T. Discussion in §5.3.

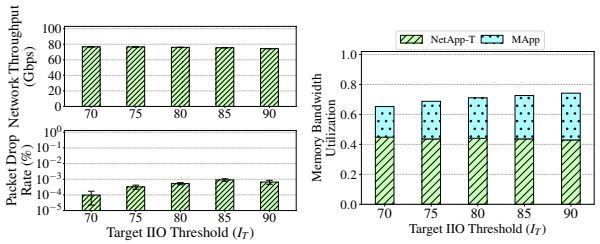


Figure 17: (left) Increasing I_T values lead to increasing drop rates due less aggressive hostCC reaction to congestion. (right) MApp acquires larger memory bandwidth with larger I_T due to less aggressive backpressure.

5.3 hostCC sensitivity analysis

hostCC has only two parameters B_T and I_T. Figure 16 shows that hostCC consistently achieves benefits in terms of maintaining target network bandwidth and minimal packet drop rates for all values of B_T (while only applying as much backpressure on MApp as needed to maintain target network bandwidth). The drop rates are particularly low for small values of B_T; this is because the arrival rate of packets at the NIC is smaller than the PCIe bandwidth utilization (that is, the rate at which packets are drained from the NIC buffer). To see this, recall from Figure 2 that, even without hostCC, network traffic achieves ~43Gbps throughput at 3× degree of host congestion; thus, average PCIe utilization must be at least 43Gbps. Here, hostCC maintains average network throughput less than 40Gbps, resulting in NIC buffers rarely filling up and no packet drops. We also observe low drop rates with large values of B_T. As discussed in §3.2, NIC buffer build up depends on $\mathcal{R}-B_T$, larger values of B_T gives network traffic more time to converge to the right rate; since hostCC ensures the average PCIe bandwidth utilization remains close to B_T, the time it takes for NIC buffer to fill up reduces with increasing B_T.

Figure 17 shows hostCC performance with varying I_T values: increasing I_T leads to an increasingly delayed reaction to the onset of host congestion, leading to larger packet drops and higher MApp throughput.

5.4 Deep dive into hostCC performance

We now provide more insights into hostCC's performance.

Necessity of the three hostCC ideas. Figure 18 demonstrates that each of the three key technical ideas in the hostCC architecture—generating host congestion signals at sub-μs granularity, sub-RTT host-local congestion response, and network resource allocation based on both host and network congestion signals—contribute to hostCC's performance.

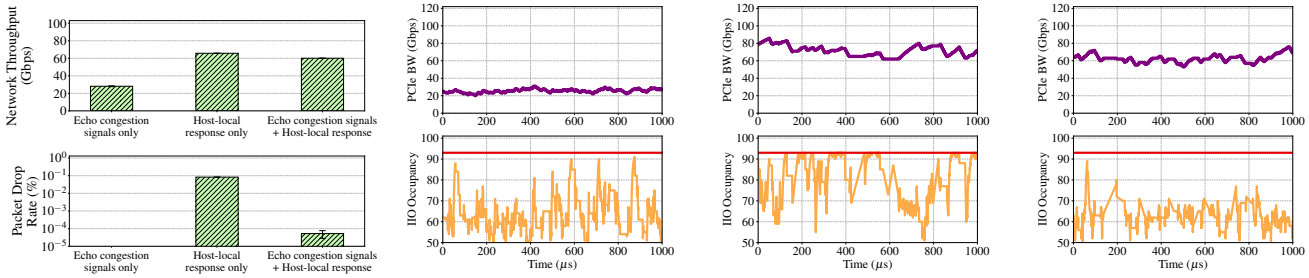
In particular, without host-local congestion response, it is possible to minimize packet drop rates but only at the cost of degraded throughput: network traffic achieves merely ~28Gbps of throughput. To explain the root cause for this observation, we plot in Figure 18(b) measured IIO occupancy and PCIe bandwidth utilization (including the PCIe-level overheads that turn out to be ~5% with 4K MTU and hardware default TLP size) with time for a 1000μs horizon for the case of 3× memory contention. We observe that IIO occupancy often increases beyond $I_T = 70$, indicating a possible onset of host congestion (whenever the network traffic increases using AIMD; network CC reacts to this by reducing rate, thus bringing down the host congestion and drop rate, but also suffering from low throughput. On the other hand, without performing network resource allocation based on both host and network congestion signals, it is possible to achieve high throughput but at the cost of large packet drop rates. Figure 18(c) demonstrates the reason for this observation: IIO occupancy I_S frequently saturates to the maximum value of ~93, indicating NIC buffer build-up and subsequent packet drops at the host.

Figure 18(d) shows that, by carefully allocating both host resources (using host-local congestion response) and network resources (using both host and network congestion signals), hostCC is able to simultaneously achieve high throughput and low packet drops rates. Using the host-local congestion response, hostCC is able to modulate host resources allocated to network traffic in a manner that NIC queue buffer buildup can be avoided (as suggested by smaller IIO occupancy immediately upon crossing the I_T threshold) until network traffic converges to the right throughput using both host and network congestion signals.

Understanding example hostCC steady-state behavior. Figure 19 shows a snapshot of hostCC's behavior over a 250μs time horizon. Figure 19(a) shows the measured PCIe bandwidth utilization with time for B_T = 80Gbps (including PCIe-level overheads, this amounts to 84Gbps, denoted by the green line). We note from Figure 9 that PCIe bandwidth utilization lies between the host-local response levels 3 and 4 (which provide ~77Gbps and 100Gbps throughput, respectively). Therefore, as expected, hostCC host-local congestion response oscillates between levels 3 and 4, ensuring that the measured PCIe bandwidth utilization remains close to B_T in Figure 19(a). The switches across levels happen in accordance with the host-local congestion response logic in §3.2: hostCC switches from level 3 to 4 when the IIO occupancy goes higher than I_T (denoted by red line in Figure 19(c)) and the PCIe bandwidth is still lower than B_T; and switches back to level 3 when PCIe bandwidth has increased beyond B_T and IIO occupancy is again lower than I_T.

6 DISCUSSION AND LESSONS LEARNT

Adoption of high-bandwidth access links and relatively stagnant technology trends for resources within hosts have led to emergence of host congestion—that is, congestion within the host network that enables data exchange between NIC and CPU/memory. hostCC is a congestion control architecture that handles both host and network fabric congestion. hostCC achieves this using three key ideas—generation of host congestion signals, a sub-RTT host-local



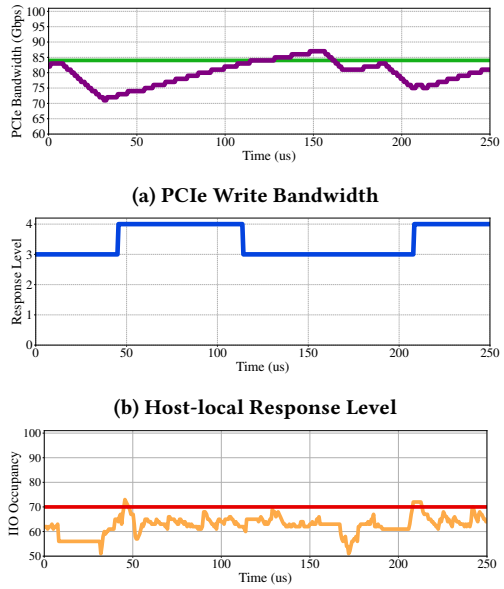
(a) Necessity of using both kinds of hostCC responses in tandem

(b) hostCC behavior when it only echoes congestion signals

(c) hostCC behavior with only host-local congestion response

(d) Default hostCC behavior with both responses in tandem

Figure 18: Each of the three key technical ideas in the hostCC architecture—generating host congestion signals at sub-μs granularity, sub-RTT host-local congestion response, and network resource allocation based on both host and network congestion signals—contribute to hostCC's performance. Discussion in §5.4.



(a) PCIe Write Bandwidth

(b) Host-local Response Level

(c) IIO Occupancy

Figure 19: In steady-state, hostCC keeps PCIe bandwidth close to the target network bandwidth, while ensuring that IIO occupancy remains smaller than the congestion threshold. Discussion in §5.4.

congestion response that uses host congestion signals to allocate host resources between network and host-local traffic, and using host and network congestion signals to perform network resource allocation at RTT granularity. We have realized hostCC within the Linux network stack without any modifications in applications, host hardware, and/or network hardware. We outline interesting avenues of future research based on our experience building hostCC.

Existing tools for host resource allocation are insufficient. We need more support from hardware to perform fine-grained host resource allocation. For instance, the tool currently used in hostCC—MBA—has two main limitations. First, while a write to a typical MSR register takes $< 1\mu$s, it takes $\sim 22\mu$s to write into the MBA MSR register, thus precluding finer-grained response. Second, MBA has non-linear performance: increasing latency using successive MBA levels results in a non-linear and coarse-grained response (also observed in [37]). We also need more tools to enable QoS at the memory controller.

Would new technologies help? Two important emerging technologies are RDMA [39] and CXL [36]. RDMA does not handle host congestion by itself [24]; and, the benefits of CXL to alleviate host congestion are unclear. For instance, consider the two use cases of CXL. First, reducing PCIe to IIO latency; our analysis in §3.1 suggests that reducing ℓ_p does not alleviate memory interconnect congestion (ℓ_m is the core problem). Second, CXL may enable memory expansion where CPUs can directly read from CXL-attached memory; this requires massive changes in the host infrastructure, and whether it will provide benefits to host congestion remains an interesting avenue of future research. Furthermore, as discussed in §2.1, host congestion may occur due to bottlenecks at any of the resources along the host network; one particularly interesting case is PCIe underutilization due to bottlenecks within hardware devices for memory protection (*e.g.*, IOMMU) [1, 9]. New technologies like ATS [3] can help IOMMU-induced host congestion, but we believe more work needs to be done to avoid IOMMU-induced host congestion.

Host congestion signals. hostCC can be easily extended to incorporate additional congestion signals. For instance, in §3.1, we discussed simple extensions in hostCC to generate delay-based congestion signals. Using this signal could allow hostCC to also work with delay-based CC protocols [22]. While commodity hardware does not provide NIC buffer occupancy, it would also be interesting to explore whether NIC buffer occupancy can provide accurate information on time, location and reason for host congestion. Finally, we need additional congestion signals to capture IOMMU-induced host congestion [1].

ACKNOWLEDGMENTS

We would like to thank SIGCOMM reviewers for insightful feedback that helped shape the final version of the paper. We would also like to thank Qizhe Cai and Midhul Vuppalapati for many useful discussions during this project. This research was in part supported by NSF grants CNS-2047283, a Google faculty research award, a Sloan fellowship, and a gift from Enfabrica. This paper does not raise any ethical concerns.

REFERENCES

[1] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, et al. 2022. Understanding Host Interconnect Congestion. In *ACM HotNets*.

[2] Amit Aggarwal, Stefan Savage, and Thomas Anderson. 2000. Understanding the Performance of TCP Pacing. In *IEEE INFOCOM*.

[3] Jasmin Ajanovic. 2008. PCI Express* (PCIe*) 3.0 Accelerator Features. (2008). https://www.intel.com.ec/content/dam/doc/white-paper/pci-express3-accelerator-white-paper.pdf

[4] 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*.

[5] AMD. 2020. AMD64 Technology Platform Quality of Service Extensions. (2020). https://developer.amd.com/wp-content/resources/56375.pdf

[6] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *ACM/IEEE ISCA*.

[7] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H Loh, and Onur Mutlu. 2012. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *ACM SIGARCH Computer Architecture News*.

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

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

[10] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μ s Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *ACM SIGCOMM*.

[11] 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*.

[12] Mario Gerla and Leonard Kleinrock. 1980. Flow control: A comparative survey. In *IEEE Transactions on Communications*.

[13] Giulia Guidi, Marquita Ellis, Aydin Buluç, Katherine Yelick, and David Culler. 2021. 10 Years Later: Cloud Computing Is Closing the Performance Gap. In *ACM/SPEC ICPE*.

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

[15] Intel. 2023. Intel® 64 and IA-32 Architectures Software Developer Manuals. (2023). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[16] Intel. 2023. Intel® Memory Latency Checker. (2023). https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html

[17] iperf. 2023. iPerf - The Ultimate Speed Test Tool for TCP, UDP and SCTP. (2023). https://iperf.fr/

[18] Bruce Jacob, David Wang, and Spencer Ng. 2010. *Memory Systems: Cache, DRAM, Disk*.

[19] Raj Jain. 1996. Congestion Control and Traffic Management in ATM Networks: Recent Advances and a Survey. In *Computer Networks and ISDN Systems*.

[20] Rick Jones. 2012. Netperf Benchmark. *http://www.netperf.org/* (2012).

[21] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs Can Be General and Fast. In *USENIX NSDI*.

[22] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. 2020. Swift: Delay Is Simple and Effective for Congestion Control in the Datacenter. In *ACM SIGCOMM*.

[23] Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N Patt. 2010. DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems. (2010). https://users.ece.cmu.edu/~omutlu/pub/dram-aware-caches-TR-HPS-2010-002.pdf

[24] Qiang Li, Qiao Xiang, Derui Liu, Yuxin Wang, Haonan Qiu, Gexiao Tian, Xiaoliang Wang, Lulu Chen, Ridi Wen, Jianbo Dong, et al. 2022. From RDMA to RDCA: Toward High-Speed Last Mile of Data Center Networks Using Remote Direct Cache Access. (2022). https://arxiv.org/abs/2211.05975

[25] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, et al. 2023. More Than Capacity: Performance-Oriented Evolution of Pangu in Alibaba. In *USENIX FAST*.

[26] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *ACM SIGCOMM*.

[27] John DC Little and Stephen C Graves. 2008. Little's Law. In *Building Intuition: Insights From Basic Operations Management Models and Principles*.

[28] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. 2015. rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers. In *ACM SIGPLAN Notices*.

[29] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. 2019. Snap: A Microkernel Approach to Host Networking. In *ACM SOSP*.

[30] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. 1997. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. In *ACM SIGCOMM CCR*.

[31] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM*.

[32] Robert Morris. 1997. TCP Behavior With Many Flows. In *IEEE ICNP*.

[33] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe Performance for End Host Networking. In *ACM SIGCOMM*.

[34] Mohammad Rajiullah, Per Hurtig, Anna Brunstrom, Andreas Petlund, and Michael Welzl. 2015. An Evaluation of Tail Loss Recovery Mechanisms for TCP. In *ACM SIGCOMM CCR*.

[35] K Ramakrishnan, Sally Floyd, and D Black. 2001. RFC3168: The Addition of Explicit Congestion Notification (ECN) to IP. (2001). https://datatracker.ietf.org/doc/html/rfc3168

[36] Debendra Das Sharma. 2022. Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy. In *IEEE MICRO*.

[37] Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. 2022. A Closer Look at Intel Resource Director Technology (RDT). In *RTNS*.

[38] Harald Welte. 2000. The Netfilter Framework in Linux 2.4. In *Linux-Kongress*.

[39] 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. In *ACM SIGCOMM*.