Understanding Host Interconnect Congestion

Saksham Agarwal
Cornell University

Rachit Agarwal
Cornell University

Behnam Montazeri
Google

Masoud Moshref
Google

Khaled Elmeleegy
Google

Luigi Rizzo
Google

Marc Asher de Kruijf
Google

Gautam Kumar
Google

Sylvia Ratnasamy
Google & UC Berkeley

David Culler
Google

Amin Vahdat
Google

ABSTRACT

We present evidence and characterization of *host congestion* in production clusters: adoption of high-bandwidth access links leading to emergence of bottlenecks within the host interconnect (NIC-to-CPU data path). We demonstrate that contention on existing IO memory management units and/or the memory subsystem can significantly reduce the available NIC-to-CPU bandwidth, resulting in hundreds of microseconds of queueing delays and eventual packet drops at hosts (even when running a state-of-the-art congestion control protocol that accounts for CPU-induced host congestion). We also discuss implications of host interconnect congestion to design of future host architecture, network stacks and network protocols.

CCS CONCEPTS

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

KEYWORDS

Congestion control, datacenter transport, network hardware

ACM Reference Format:
Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. 2022. Understanding Host Interconnect Congestion. In *The 21st ACM Workshop on Hot Topics in Networks (HotNets '22), November 14–15, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3563766.3564110

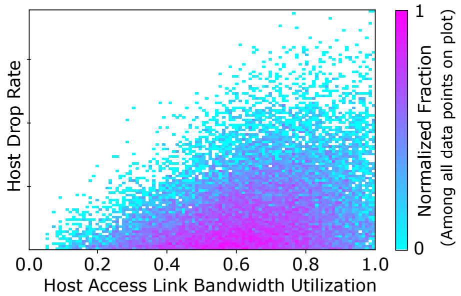

Figure 1: Host congestion in a large-scale Google production cluster. The figure shows scatter plot of link utilization (normalized by access link bandwidth) and packet drop rates at hosts. The cluster runs both the Linux kernel and SNAP [22] network stacks, with TCP and Swift [15] congestion control protocols, respectively[0]. Precise values for packet drop rates are hidden due to these being sensitive. Packet drop rates are positively correlated with host access link utilization; also, packet drops happen even when host link utilization is low.

1 INTRODUCTION

The conventional wisdom in the systems and networking communities is that congestion happens primarily within the network fabric (that is, at network switches). This may have been true for the Internet and early-generation datacenter networks; however, we need to revisit this conventional wisdom for modern clusters. This paper presents evidence and characterization of *host congestion* in large-scale production clusters: we demonstrate that adoption of high-bandwidth access links, combined with relatively stagnant technology trends for other host resources, have led to emergence of bottlenecks within the host interconnect, that is, the data path between the NIC and the CPU. Our goals are three-fold: (1) to characterize host congestion originally observed in a large-scale Google production cluster that runs Swift [15]—a state-of-the-art congestion control protocol (see Figure 1); (2) to build an in-depth understanding of the root causes of host congestion, and of the impact of host congestion on application-level performance; and (3) to explore implications of host congestion to design of next-generation host architecture, network stacks, and network protocols.

We study two root causes for host congestion. The first one, discussed in §3.1, is due to the need for memory protection from I/O devices. When such memory protection is

[0] Data collected over a 24-hour period, and binned at a 10-minute granularity. All drops in the figure are due to host congestion induced by host interconnect bottlenecks (confirmed using counters within our infrastructure).

HotNets '22, November 14–15, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9899-2/22/11.
https://doi.org/10.1145/3563766.3564110

enabled, for every DMA request initiated by the NIC, one must translate the NIC-visible virtual address to host physical address; when the address translation (page) table does not fit into the cache, one or more memory accesses are required for the translation. The resulting increase in per-DMA latency directly impacts the rate at which NIC can transfer data to CPU. The challenge is that, with increasing access link bandwidths, we expect larger number of actively used addresses (to accommodate increase in bandwidth-delay product), larger cache misses (since cache sizes have not increased at the same rate), and larger per-DMA latency (since memory access latency has largely been stagnant); thus, this problem is likely to worsen over time.

The second root cause of host congestion builds upon a rather surprising observation of packet drops at hosts even when host access link bandwidth is far from fully utilized (Figure 1). We find that this is due to bottlenecks within the host memory subsystem. In particular, CPUs reading/writing data to main memory share the memory bus bandwidth with the NIC performing DMA operations; when memory bus is contended, CPUs are able to acquire a larger fraction of memory bus bandwidth than NIC. As a result, in-flight packets result in NIC buffers building up before congestion control protocols can react; this, in turn, results in large queueing delays and eventual packet drops at the host even when host is receiving data at a rate lower than the access link bandwidth. We study this phenomenon in more depth in §3.2. Given that memory bus bandwidth per core is increasing much more slowly than NIC and PCIe bandwidths, this problem is also likely to worsen over time.

In terms of application-level performance, host congestion is no different from congestion within the network fabric—it can lead to hundreds of microseconds of tail latency, significant throughput drop, and violation of isolation properties due to packet drops. Unfortunately, host congestion due to host interconnect bottlenecks is only going to become worse—even though a 10× increase in access link bandwidth is anticipated within next few years (100Gbps NICs are already commodity, and 400–800Gbps NICs are already standardized [32]), technology trends for essentially all other resources along the host interconnect—CPU speeds, cache sizes, memory access latency, memory bus bandwidth per core, NIC buffer sizes, etc.—are largely stagnant. To that end, we outline several interesting avenues for research in alleviating host congestion. In particular, we discuss in §4 that resolving host interconnect congestion problem requires a multi-pronged approach: rethinking host architecture (new mechanisms for memory protection from I/O devices, for sharing of memory bandwidth, etc.), new system design (enabling new congestion signals from "outside" the network stack), and new network protocols (rethinking sub-RTT response to host congestion).

Our study—that focuses on host congestion due to host interconnect bottlenecks—complements recent works [8–10, 34] that focus on CPU efficiency and/or network stack performance for regimes where hosts operate at maximum loss-free rate (that is, hosts observe no persistent congestion to begin with). Our study also presents a more complete picture on host congestion when compared to recent works that focus on understanding the impact of high-bandwidth links on individual hardware components (*e.g.*, PCIe [25], memory bandwidth [20, 30], and IOMMU [4, 5, 10, 19, 21, 29])—as we will discuss throughout the paper, host congestion turns out to be a result of imperfect interaction (and resource imbalance!) between multiple components within the host interconnect. Finally, unlike most prior work, our work studies the behavior of congestion control protocols and resulting application-level performance under host congestion due to bottlenecks within the host interconnect.

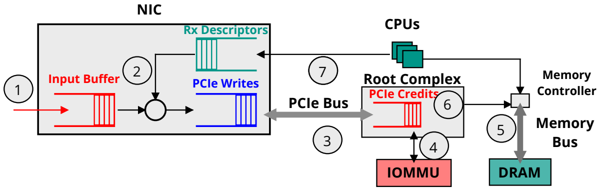

Figure 2: The host datapath between the NIC and the CPU.

2 HOST DATAPATH

Figure 2 shows the life of a packet at a receiver host[1].

1. The NIC first enqueues all arriving packets into its input buffer, which is typically a small SRAM (typically 1 − 2 MBs for commodity NICs [14, 31]).
2. The NIC then fetches an Rx descriptor from a queue (the NIC driver periodically replenishes these descriptors). When memory protection from I/O devices is enabled, the descriptor provides the virtual address in the host memory where the packet should be DMA'ed.
3. To DMA the packet to host memory, the NIC then instantiates PCIe write transactions using the address in the packet descriptor. PCIe uses a lossless interconnect using credit-based flow-control, implemented via a fixed (hardware-specific) number of credits [25]. When PCIe does not have enough credits to serve a request, requests are enqueued in the NIC input buffer (resulting in queueing and eventual packet drops) until requisite number of credits become available.
4. The write transaction requests are then handled by the PCIe root complex (the other end of the PCIe). The root complex first performs virtual to physical memory address translation using a special device called input/outout memory management unit (IOMMU). IOMMU uses a page table that primarily resides in host memory; these translations can be sped up using a cache called I/O Translation Lookaside buffer (IOTLB). An IOTLB miss requires a page table walk using one or more accesses to host memory.

[1] Sender-side datapath has mechanisms in place (*e.g.*, backpressure from NIC to CPU) that ensure that sender-side typically does not experience host congestion [8]. Thus, we primarily focus on the receiver side.

5. Once the physical memory address is available, the PCIe root complex moves the data to the host memory over the memory bus (shared by CPUs)[2].
6. The root complex, upon completion of the memory write, replenishes PCIe credits that can be used to service subsequent write requests. Any delays in the NIC-to-memory datapath (*e.g.*, due to IOTLB misses, memory access latency, etc.) result in a backpressure to the NIC input buffer, until the root complex can replenish the credits.
7. Finally, the NIC interrupts the CPU to initiate packet processing and/or replenishing of Rx descriptors.

Note that, if memory protection is not enabled, no address translation is needed as the descriptor provides the physical memory address for the NIC to DMA the packet.

3 HOST CONGESTION

We now characterize host congestion. We use the following testbed: our machines have Intel Skylake CPUs, 2 NUMA nodes each with 28 cores, 100Gbps NICs, PCIe 3.0 x16 lanes per NIC, 128 size IOTLB per IOMMU, and a theoretical maximum memory bus bandwidth of 115.2GBps per NUMA node (6 DDR4 channels per NUMA node with maximum data rate of 2400MT/s per channel). We extract a minimalistic workload from our production cluster that leads to host congestion: 40 sender machines and one receiver machine exchange traffic using SNAP [22] network stack with Swift [15] congestion control (CC) protocol. The receiver machine runs one or more threads, each on a dedicated core in the same NUMA node as the NIC; each receiver thread issues 16KB remote reads using one connection per sender.

We primarily focus on application-level throughput (payload bytes received per unit time) and packet drop rate (ratio of the number of packets dropped and the number of packets transmitted). Drop rate serves as a proxy for violation of isolation properties—all applications use a shared NIC buffer where drops end up occuring. For our cluster, when using 4K MTUs, the throughput is upper bounded by ∼92Gbps due to protocol header overheads. To demonstrate IOTLB contention, we measure IOTLB misses per packet. To demonstrate memory bandwidth contention, we measure total memory bandwidth across all memory channels connected to the NIC-local NUMA node. Ideally, we want to achieve maximum throughput with near-zero drop rates.

3.1 IOMMU induced host congestion

Intuition. Almost all modern hosts enable memory protection using an input/output memory management unit (IOMMU) [19, 29]—for every DMA request initiated by the NIC, IOMMU translates the NIC-visible virtual address to host physical address using a page table that primarily resides in memory; these address translations are sped up using a special translation lookaside buffer (IOTLB) cache. An IOTLB miss requires a page table walk using one or more memory accesses, resulting in larger per-DMA latency. For instance, an IOTLB hit typically takes a few nanoseconds of latency; a miss, however, can trigger one or more memory accesses (depending on what page entry level was already cached in IOTLB) [5, 19], thus incurring additional latency of few hundreds of nanoseconds to up to a microsecond [25]. Given that NIC can have only a small number of fixed-size DMA transactions in flight, the increase in per-DMA latency directly impacts the rate at which NIC can transfer data to memory (as we discuss below, this follows immediately from Little's Law [18]). Since PCIe is only nominally faster than the line rate for 100Gbps NICs (using PCIe 3.0 with a maximum 128Gbps theoretical capacity, the achievable PCIe goodput is only ∼110Gbps due to the PCIe transaction and link layer header overheads [25]), such reduced PCIe transfer rates result in NIC queue build up and eventual packet drops.

Setup. We compare scenarios with IOMMU OFF and ON (see §2 for the two datapaths). Our network stack uses IOMMU in so-called loose mode: each thread registers upfront a fixed amount of memory with IOMMU (using 4K/2MB mappings), and keeps the mapping alive throughout its operations, so there are no software IOTLB invalidations at run time (other modes supported by Linux, *e.g.*, dynamically deleting IOMMU mappings at run time are known to cause even worse IOTLB misses [19, 29]). Thus, by varying number of threads/cores, we can control the number of active pages registered to IOMMU (this number increases linearly with number of cores). By default, hugepages are enabled—2MB IOMMU mappings are used for data transfers; descriptors and other control packets use the standard 4KB mappings.

NIC-to-CPU throughput reduces with increasing IOMMU contention. Figure 3 demonstrates host congestion due to IOMMU-induced host interconnect bottleneck. For fewer than 8 cores, the CC protocol is bottlenecked by CPU cycles; thus, the throughput almost linearly increases with number of cores reaching the maximum possible throughput of 92Gbps (accounting for protocol headers discussed earlier, shown by green line) at × = 8. Beyond that point, we observe a difference between the IOMMU OFF and ON cases. When IOMMU is OFF, throughput stays at 92Gbps. When IOMMU is ON, throughput reduces with increasing number of receiver threads (Figure 3(left)); corresponding increase in IOTLB misses per packet (Figure 3(right)) demonstrates that the host interconnect is bottlenecked in this regime.

Increasing number of IOTLB misses are due to increase in larger number of entries in IOMMU (due to each receiver thread being allocated a fixed memory region) and due to lack of locality in IOMMU access patterns—since there are multiple active flows (recall from our setup that we create a single flow per receiver thread per sender), subsequent packets do not necessarily lie in contiguous memory regions. Note that we are increasing both the working set size and

[2] If Direct Cache Access (*e.g.*, DDIO) is enabled, data is first moved to the CPU cache [8]; this may result in eviction of existing cache contents to the host memory over the same memory bus.

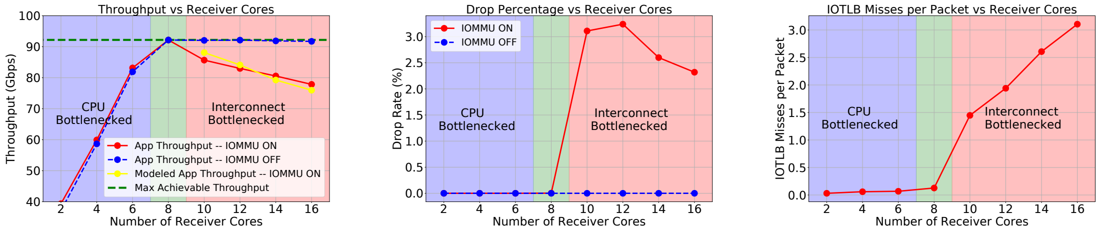

Figure 3: For our baseline setup, IOMMU induced congestion can result in up to 15% throughput degradation compared to the no-IOMMU case, and up to 3% packet drops. With increasing number of cores, the page entries registered to IOMMU increases and the host interconnect soon becomes the bottleneck due to large number of IOTLB misses.

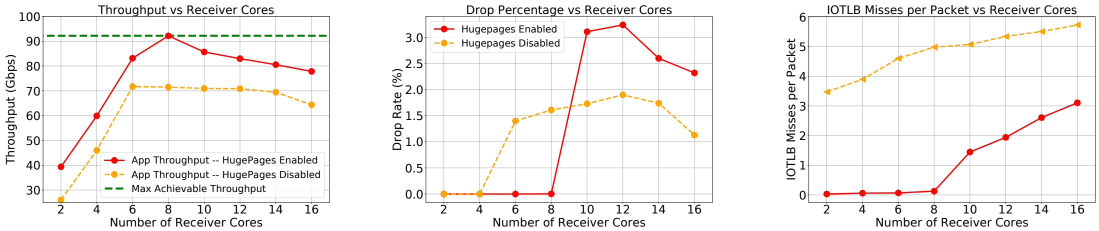

Figure 4: Disabling hugepages results in more than 30% throughput degradation due to much more increased IOMMU contention. The drop rates can still be as high as 2% at such low network utilization.

the number of flows in Figure 3; we also measured that the impact of increase in number of concurrent flows is minimal. Thus, the key reason for degradation here is in fact increasing working set size. There is a sudden increase of IOTLB misses per packet above 8 threads (see Figure 3(right)), indicating that the total number of IOMMU entries exceed the IOTLB size (128 for our setup) for more than 8 threads. Recall that each packet can incur multiple IOTLB misses[3].

As outlined earlier, IOTLB misses create a hard limit to the maximum achievable NIC-to-CPU throughput: PCIe credits allow at most C packets in flight, each PCIe write experiences a latency $T_{base} + M \cdot T_{miss}$ where the T_{base} is the latency with no IOTLB misses, M is the IOTLB miss rate per packet, and T_{miss} is latency accounting for IOTLB misses. As a result, the throughput is bounded by $(C \cdot \text{pkt\_size})/(T_{base} + M \cdot T_{miss})$. The observed throughput closely matches the above model—we show the simulated results of this model in Figure 3 (only for the scenarios with threads ≥ 10 where this model applies, since PCIe credits are the bottleneck).

Existing transport designs cannot react to the host congestion effectively. Figure 3(center) shows significant packet drops ($\geq 2\%$) in the steady state, suggesting that the net average arrival rate of packets at the NIC exceeds the NIC-to-CPU throughput. This may be surprising given that our CC protocol was designed to handle delays within the host. The reason is as follows. Our CC protocol uses a target host delay value of 100μs to account for inflation in host delays due to CPU bottlenecks, queueing delay at the NIC buffer and NIC-to-memory DMA latency [15] (when host is not a bottleneck, we measure the delay to be almost always $\leq 10\mu$s). However, ~1MB NIC buffer size in our testbed corresponds to NIC queueing delay of less than 90μs when NIC can transfer data to the CPU at rates greater than or equal to 1MB/90μs= 88.8Gbps (~81Gbps application-level throughput); thus, when throughput is greater than 81Gbps, the CC protocol does even react to host congestion resulting in queueing and packet drops at the NIC.

We observe precisely the above phenomenon in our experiments — until × = 12, throughput is greater than 81Gbps and packet drops continue to increase; once the throughput drops below 81Gbps, CC kicks in, reducing the rate and packet drops. Note that, even when CC kicks in, packet drop rate does not reduce to zero; this is because of two reasons: (1) CC protocol reduces the rate when the NIC buffer is already almost full, and the corresponding in-flight packets experience drops upon arrival at the NIC; and (2) like any classical CC protocol, our protocol also demonstrates the sawtooth behavior—upon reducing the rate, the host delay reduces, resulting in a corresponding increase in rate, leading to subsequent host congestion and drops.

Disabling hugepages causes larger IOMMU contention. Figure 4 presents results for hugepage disabled case (using 4K pages instead of 2M pages). We observe that the threshold when the host interconnect becomes the bottleneck arrives with fewer receiver threads. This is because: (1) the number of registered pages increase by 512×, resulting in more IOTLB misses (as evident from Figure 4(right)); (2) DMAing data for each 4K MTU packet requires fetching two pages instead of just a single hugepage. This results in worse NIC-to-CPU

[3] In the worst-case, a packet can result in 6 IOTLB misses when hugepages are enabled, one for each of the 6 PCIe transactions necessary—DMA of payload, completion queue entry and packet descriptor for the packet, and the corresponding ACK packet sent back to the sender. The observed miss rates, of course, depend on address locality.

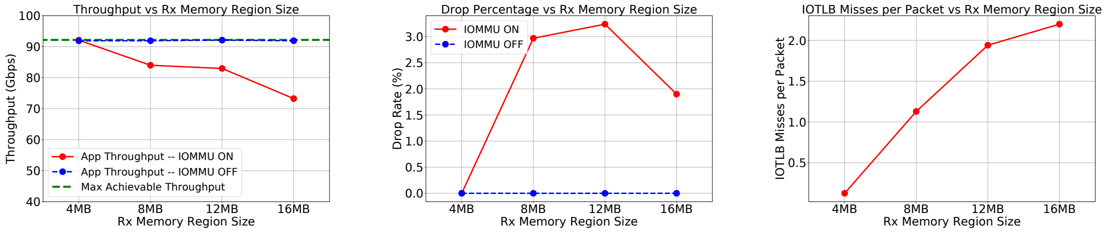

Figure 5: Provisioning networks for larger BDPs makes the host congestion problem worse – hosts require at least BDP worth of memory to store arriving packets. Larger memory can significantly degrade the application throughput due to IOMMU quickly becoming the bottleneck, since larger number of pages per core are registered to the IOMMU leading more IOTLB misses.

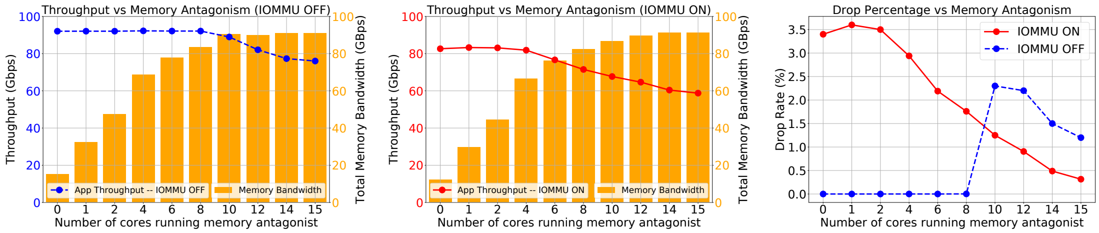

Figure 6: Adding memory bandwidth contention can degrade network utilization (∼ 15% even without any IOMMU contention). Adding IOMMU contention in conjunction lead to even larger degradation (∼ 25% degradation with 15 stream cores).

throughput degradation when hugepages are disabled. The drop percentage, as discussed above, is lower—while CC kicks in earlier (since throughput < 81Gbps even with 6 threads), it lacks effectiveness due to small NIC buffer sizes resulting in drops even at 65% network utilization. Hence, disabling hugepages can provide smaller drop rates, but at the expense of degraded throughput with IOMMU contention.

Provisioning networks for larger BDPs leads to larger IOMMU contention. For larger bandwidths, BDP will be larger; thus, larger memory regions must be provisioned to ensure one BDP worth of buffer pools per receive queue to fully utilize the network. Figure 5 shows the effect of increasing memory region sizes. As expected, this leads to larger IOTLB misses per packet, since the same number of concurrent requests could potentially access larger number of pages, thus swapping out more IOTLB entries, resulting in lower NIC-to-CPU throughput. Drop rates follow similar trend as previous scenarios—for the baseline case of 12MB memory region size, the host delay before drops was 98.7μs; with 16MB size, it increases to 110.5μs, hence leading to the CC protocol kicking in and reducing the drop rate.

3.2 Memory bus induced host congestion

Intuition. Another root cause of host congestion is the rapidly reducing gap between access link bandwidth and memory bus bandwidth: in large servers, applications that perform large volumes of memory operations can lead to starvation of memory requests coming from the NIC. Specifically, within the host interconnect, CPUs reading/writing data to memory share the memory bus with the NIC performing DMA operations; when memory bus is contended, per-DMA latency increases for memory requests coming from the NIC. This increase leads to an effect similar to the IOMMU case discussed in the previous subsection: increase in latency eventually results in PCIe bandwidth underutilization, and in-flight packets resulting in NIC buffers quickly building up. This, in turn, results in large host delays and packet drops even when host is receiving data at rate lower than the access link bandwidth; these delays and drops can result in subsequent rate reduction and link underutilization.

Setup. We use Stream [7] benchmark to antagonise the memory bus, with one instance per physical core, up to 15 cores. We use the same setup of §3.1, with 40 senders and 12 receiver threads, and two scenarios: IOMMU on and off. The maximum achievable bandwidth by Stream per NUMA node on our machine is ∼90GB/s (65GB/s for reads and 25GB/s for writes).

NIC-to-host throughput degrades with higher memory bus contention. Figure 6 shows throughput degradation and packet drops with increasing memory bus contention. Without any Stream cores, we achieve maximum throughput, with ∼11.8GB/s write bandwidth (due to PCIe writes for arriving packets), along with a small read bandwidth (∼3.3GB/s)—this is because of data copy from receiver threads to application threads. When memory bus is close to saturation, starting with 10 stream cores, we see that NIC-to-CPU throughput reduces. This is because in such a scenario, the latency in performing memory writes increases. When the memory controller receives read/write requests from CPUs/NIC, they are typically served in a first-come-first-serve manner [16, 24], irrespective of the source of request. Hence as the offered load to the memory bus reaches closer to the maximum achievable memory bandwidth, similar to any load-latency curve for a closed-loop system, the service times for PCIe write requests will also increase.

Larger PCIe latencies further degrade the throughput. Figure 6(center) shows scenario where IOMMU is contended in conjunction to the memory bus. Here, NIC-to-CPU throughput is already low with no antagonists due to additional latency from IOTLB misses. Throughput now starts degrading with only 6 stream cores—this is because, starting with 6 stream cores, the increase in memory bus bandwidth utilization per core is sublinear; at this point, the memory bus is already close to saturation and latencies for PCIe write requests will start increasing further[4]. With 15 stream cores, the throughput reduces to ~60Gbps. The drop rates follow trends similar to discussed previously in §3.1—the CC protocol observes delays increasingly more than 100μs with reducing arrival rates, hence reducing drops.

4 LOOKING FORWARD

Technology trends suggest that the problem of host congestion is only going to get worse with time. As discussed earlier, while host access link bandwidths are likely to increase by 10× over the next few years, technology trends for essentially all other host resources—*e.g.*, NIC buffer sizes [30], the ratio of access link bandwidth to PCIe bandwidth [26–28], IOTLB sizes [4, 25], memory access latencies [17, 32], and memory bandwidth per core [23]—are largely stagnant.

One root cause of host congestion, that we did not focus on much, is the inability of the host software to consume packets at the rate at which NIC receives packets [8–10, 22] either due to lack of available CPUs or simply due to inefficient host software. The challenges introduced by inefficient host software are important, but do admit a solution: dynamically changing cores allocated to software processing [9, 15]. Our results, in §3, show that state-of-the-art CC protocols can handle host congestion introduced by host software using precisely such solutions. However, as we have shown in the previous section, the case of host congestion due to host interconnect bottlenecks is very different: increasing the number of cores can in fact increase host congestion due to inflating host interconnect bottlenecks. Simply using a lower host target delay would not resolve the problem: with CC protocols taking at least one RTT to respond to congestion, in-flight bytes can exceed NIC buffer sizes even with a small number of senders[5]. Indeed, similar reasoning also applies for TCP-like protocols [2, 12]: the total in-flight bytes can still exceed NIC buffer capacity. Thus, different forms of host congestion—host software and host interconnect—require different responses. To alleviate host congestion, and to design CC protocols that respond efficiently to host congestion, we believe the following directions are worth exploring:

Rethinking host architecture for future-generation datacenter networks. Given our results, we believe the following are interesting directions to explore: (a) alternative architectures to enable memory protection from the NIC, *e.g.*, efficient offload of I/O address translation as in technologies like ATS [1]; (b) alternatives to PCIe link layer protocol, *e.g.*, CXL [33] might alleviate host-congestion problems to some degree via potentially reducing PCIe latency or via expanding memory bandwidth over PCIe channels; and perhaps more importantly, (c) mechanisms to more "fairly" share the memory bandwidth between compute and network traffic, *e.g.*, emerging technologies like Intel MBA [13] and ARM MPAM [6] enable enforcing QoS guarantees for memory bus, and we can utilize existing ideas in QoS-scheduling [3, 11] to share memory bandwidth.

Rethinking congestion signals. Host congestion due to bottlenecks within the host interconnect brings a new twist to the classical CC problem: in addition to congestion signals that originate within the network (*e.g.*, queuing and drops at switches, ECN, delay, etc.), future CC protocols should both incorporate new congestion signals that come from "outside the network" (*e.g.*, CPU utilization, memory bandwidth contention, memory fragmentation, etc.), and new mechanisms to react to these signals.

Rethinking congestion response. Traditionally, different components within the host have been responsible for allocation of resources (*e.g.*, network stack for network resources, CPU schedulers for compute resources, memory controller for memory bandwidth, etc.) with little or no coordination. Thus, CC protocols typically take a view of allocating network resources: responding to congestion by reducing the rate of data transmission. Our study suggests at least two directions where we should rethink congestion response to fundamentally resolving host congestion. First, we need a more coordinated approach to balanced allocation of resources across all components (compute, memory bandwidth, network bandwidth); for instance, rather than reducing rate for network transfers upon congestion at the NIC, one could trigger CPU rescheduling to reduce memory bus bottleneck (*e.g.*, scheduling applications on NUMA nodes different from the one where the NIC is connected). Second, we need to rethink the timescale of congestion response: while RTT-level response may be sufficient for fabric congestion, emergence of Terabit Ethernet and stagnant NIC buffer sizes may necessitate a sub-RTT response for host congestion.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Srinivasan Seshan, and HotNets reviewers for insightful feedback. We would also like to thank Nandita Dukkipati and Jeff Mogul for feedback on early versions of this paper. This research was in part supported by NSF grants CNS-2047283 and CNS-1704742, a Google faculty research award, and a Sloan fellowship.

[4] Similar phenomenon will also occur for Figure 6(left), but there is no visible degradation for 6-8 cores due to available headroom in PCIe bandwidth.

[5] Even if we assume a 20μs RTT and 100G line rate, in-flight packets for just 8 concurrent senders can exceed 1MB threshold.

REFERENCES

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

[2] 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*.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM*.

[4] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: efficient IOMMU emulation. In *USENIX ATC*.

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

[6] Arm. 2021. Memory system resource partitioning and monitoring (MPAM). (2021). https://developer.arm.com/documentation/ddi0598/latest

[7] Lars Bergstrom. 2011. Measuring NUMA effects with the STREAM benchmark. *arXiv:1103.3225* (2011).

[8] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *ACM SIGCOMM*.

[9] 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*.

[10] 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*.

[11] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. 2015. Queues don't matter when you can jump them!. In *USENIX NSDI*.

[12] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. In *ACM SIGOPS OSR*.

[13] Intel. 2019. Introduction to Memory Bandwidth Allocation. (2019). https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html

[14] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *USENIX NSDI*.

[15] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM*.

[16] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N Patt. 2008. Prefetch-aware DRAM controllers. In *IEEE/ACM MICRO*.

[17] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-direct: High-performance in-memory key-value store with programmable nic. In *ACM SOSP*.

[18] John DC Little and Stephen C Graves. 2008. Little's law. In *Building intuition*. Springer.

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

[20] Ilias Marinos, Robert NM Watson, Mark Handley, and Randall R Stewart. 2017. Disk| Crypt| Net: rethinking the stack for high-performance video streaming. In *ACM SIGCOMM*.

[21] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. 2018. DAMN: Overhead-free IOMMU protection for networking. In *ACM ASPLOS*.

[22] 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*.

[23] Hassan Mujtaba. 2020. Intel Sapphire Rapids Xeon Scalable CPUs. (2020). https://wccftech.com/intel-sapphire-rapids-xeon-scalable-cpus-volume-ramp-rumored-for-2023/

[24] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: enhancing both performance and fairness of shared DRAM systems. In *IEEE ISCA*.

[25] 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*.

[26] NVIDIA. 2022. ConnectX-5. (2022). https://www.nvidia.com/en-us/networking/ethernet/connectx-5/

[27] NVIDIA. 2022. ConnectX-6. (2022). https://www.nvidia.com/en-us/networking/ethernet/connectx-6/

[28] NVIDIA. 2022. ConnectX-7. (2022). https://nvdam.widen.net/s/srdqzxgdr5/connectx-7-datasheet

[29] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. 2015. Utilizing the IOMMU scalably. In *USENIX ATC*.

[30] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The benefits of general-purpose on-NIC memory. In *ACM ASPLOS*.

[31] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: scalable NIC for end-host rate limiting. In *USENIX NSDI*.

[32] Shelby Thomas, Geoffrey M Voelker, and George Porter. 2018. Cachecloud: towards speed-of-light datacenter communication. In *USENIX HotCloud*.

[33] Stephen Van Doren. 2019. HOTI 2019: Compute Express Link. In *IEEE HOTI*.

[34] Yimeng Zhao, Ahmed Saeed, Mostafa Ammar, and Ellen Zegura. 2021. Scouting the path to a million-client server. In *PAM*.