PL2: Towards Predictable Low Latency in Rack-Scale Networks

Yanfang Le [†], Radhika Niranjan Mysore [††], Lalith Suresh [††], Gerd Zellweger[††],
Sujata Banerjee[††], Aditya Akella[†], Michael Swift[†]
University of Wisconsin-Madison[†], VMware Research[††]

1 Introduction

Rack-scale data center solutions like Dell-EMC VxRail [57] and Intel RSD [52] have emerged as a new building block for modern enterprise, cloud, and edge infrastructure. These rack-scale networks, that extend between NICs of rack-units and the top-of-rack (ToR) switch, need to satisfy the key requirements of **uniform low latency** and **high utilization**, irrespective of where applications reside, and which accelerators they access (e.g., FPGA vs. CPU vs. GPU). However, a key obstacle stands in the way of achieving these goals: Ethernet is not a lossless fabric, and our experiments on a 100G testbed confirm that **drops, not queueing**, are the largest contributor to tail latency pathologies.

In this paper, we present *Predictable Low Latency* or PL2, a rack-scale lossless network architecture that uses programmable network hardware to achieve low latency and high throughput in a transport-agnostic and workload-oblivious manner. PL2 reduces NIC-to-NIC latencies by proactively avoiding losses. PL2 supports a variety of message transport protocols and gracefully accommodates increasing numbers of flows, even at 100G line rates. It neither requires a-priori knowledge of workload characteristics nor depends on rate-limits or traffic priorities to be set based on workload characteristics (e.g., by configuring PFC classes).

To achieve these goals, senders in PL2 explicitly request a switch buffer reservation, for a given number of packets, a *packet burst*, and receive notification as to when that burst can be transmitted without facing any cross traffic from other senders. PL2 achieves this form of centralized scheduling even at 100G line rates by overcoming the key challenge of carefully partitioning the scheduling responsibility between hosts in the rack and the Top-of-Rack (ToR) switch. In particular, the end-host protocol is kept simple enough to accommodate accelerator devices and implementations within NICs, whereas the timeslot allocation itself is performed in the ToR switch at line rate (as opposed to doing so on a host, which is prone to software overheads).

In this short paper, we present a brief overview of PL2 design and the main result from our work. A longer version of our work with motivating experiments, complete design, implementation and evaluation is presented in the Appendix sections B,D,E and F.

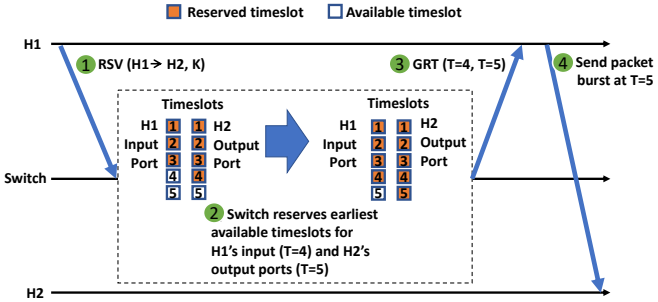


Figure 1: Scheduling example in PL2, with host H1 sending a packet burst to H2. (1) H1 sends an *RSV* to the switch to make a reservation. (2) The switch maintains timeslot reservations for the input and output ports connected to every hosts. It reserves the earliest available timeslots on H1's input port ($T = 4$) and H2's output port ($T = 5$). (3) The switch notifies H1 of these timeslots through a *GRT* message. (4) To avoid queuing, H1 then transmits at the maximum of the two timeslots indicated in the GRT, which is $T = 5$.

2 PL2 Design

The heart of PL2 is an algorithm for scheduling packet bursts at line rate using the *switch dataplane*, where a packet burst is simply a bounded number of Ethernet frames. Each packet burst is transmitted at a timeslot reserved by the scheduling algorithm, reducing cross-traffic.

Timeslot reservation. Conceptually, our switch maintains a list of *timeslots* per input and output buffer for each port. In our current implementation, we define a timeslot to be the time it takes to transmit an MTU sized packet out of a buffer. To transmit a packet burst from host h to h', we seek to reserve a timeslot t on the switch input port corresponding to h, and a timeslot t' on the output port corresponding to h'. Host h then transmits at a 'chosen timeslot' which is the greater of timeslots t and t' to avoid a collision. The astute reader will observe that we could instead let the switch choose the transmission time in a centralized manner rather than pairwise, but hardware constraints prevent us from doing so (§D.2).

Note, with the hosts choosing the transmission times, there is a risk of collision. With perfect scheduling, we would have no collisions, and need near zero buffering at the switch. However, PL2 uses a small amount of buffer space (less than 200KB in our 100 Gbps testbed) to accommodate occasional collisions.

To run the above-mentioned scheme at line rate and within the constraints of switching hardware (outlined in §D.1.1), we designed an algorithm illustrated in Figure 1, which divides scheduling logic between switch and hosts.

Algorithm 1 Switch Scheduling Algorithm.

```
1:  INIT:
2:  inReservation[port 1..port n] ← {0}
3:  outReservation[port 1..port n] ← {0}
4:  INPUT: packet
5:  src ←source port of RSV
6:  dst ←destination port requested
7:  if packet is a RSV then
8:      packet.sendTimeslot ← inReservation[src]
9:      inReservation[src] += packet.demand
10:     packet.recvTimeslot ← outReservation[dst]
11:     outReservation[dst] += packet.demand
12:     send GRT
13: else
14:     outReservation[dst] -= 1
15:     inReservation[src] -= 1
16: end if
```

Switch Logic. Algorithm 1 shows the scheduling logic at the switch. The switch creates a schedule of input and output reservations for every port, in terms of timeslots. Each highlighted gray box represents logic that can be implemented with a single P4 operation.

At switch start up, the input and output reservations are initialized to zero (lines 2-3). In response to RSV packets, the switch sends back the next available input and output timeslots for the requested transmissions (lines 8,10,12). It also reserves enough timeslots for each RSV request (lines 9,11). In Figure 1, the switch has reserved timeslot 4 at the source port (connected to Host 1) and timeslot 5 at the destination port (connected to Host 2) for transmission. Note that the reservation timeslots have not lined up exactly at the two ports. We describe how the sending host uses these timeslots next. Lines 13-15 describe switch logic for regular packets, in which the timeslot reservations for the packet are removed.

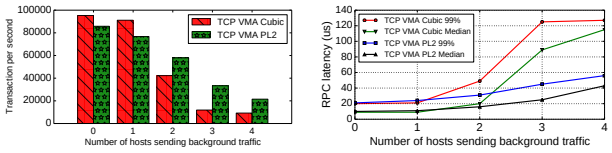
Host Logic. The host side conservatively chooses a timeslot to transmit that is available on both the relevant switch input and output ports using the equation $chosenTimeslot \leftarrow max(sendTimeslot, recvTimeslot)$. The sender then transmits packets at the `chosenTimeslot` by waiting for a period `waitingTime` calculated using the equation

$$waitingTime = timeslotWait - rsvGrtDelay - pendingDataDelay, \quad (1)$$

where $timeslotWait = chosenTimeslot * MTU/linerate$ and `pendingDataDelay = bytes/linerate`.

The timeslot chosen for transmission is `timeslotWait` seconds into the future from when the reservation is made at the switch. This reservation is conveyed back to the sender after `rsvGrtDelay/2` seconds. The first packet of the transmission would again take approximately `rsvGrtDelay/2` seconds to reach the switch, after transmitting all the previously scheduled packets at the sender NIC (which would take `pendingDataDelay` seconds)[1]. The sender therefore waits only for the remaining fraction of time to ensure that the first packet of the burst arrives in time at the switch.

3 Main result



(a) Throughput. **(b) Latencies.**

Figure 2: Memcached competes with TCP traffic.

When `memcached` competes with heavy (incast) background traffic with congestion control support, PL2 can ensure up to 2.3x lower rpc-latencies, and 1.8x corresponding improvement in application throughput compared to VMA TCP Cubic. Figures 2a and 2b show `memcached` transaction throughput and response latencies with TCP background traffic. The graphs shown are with 64B keys and 4 KiB values. The red bar in Figure 2a shows throughputs with memcached over TCP over PL2 (TCP VMA PL2), and the green bar shows throughputs with memcached over TCP with Cubic congestion control (TCP VMA Cubic).

When there is no background traffic (bars for 0 hosts sending background traffic), PL2 slows down memcached by a small amount(8%), even though the RPC latencies are similar to TCP VMA Cubic (Figure 2b). PL2 RSV-GRT exchange overheads contribute to reduced throughput seen with TCP VMA PL2. The same effect is seen in the case where the `memcached` host also sends background traffic that utilizes around 50% of the downlink to the `memcached` client (50 Gbps). Once we add background traffic load from additional servers, the load on the receiving link increases to 80-90% and the `memcached` latencies go up much faster for TCP VMA Cubic than TCP VMA PL2 as seen in Figure 2b; consequently `memcached` has 0.5x lower throughput with TCP VMA Cubic. This is despite the fact that the competing background traffic that runs using TCP VMA PL2 has 1-5 Gbps more load than the background traffic that runs on VMA TCP Cubic. The 99th-percentile RPC latency with VMA TCP Cubic is 127 µs as opposed to 56 µs with PL2 with 4-way TCP incast background traffic.

[1]We ensure that RSV-GRT packets do not wait behind data packets (if any) by prioritizing them using 801.1q

References

[1] Precision Time Protocol (PTP). http://linuxptp.sourceforge.net/.

[2] Workloads Traces Source. https://github.com/PlatformLab/HomaSimulation/tree/omnet_simulations/RpcTransportDesign/OMNeT%2B%2BSimulation/homatransport/sizeDistributions.

[3] Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.

[4] Dell emc vxrail. https://www.dellemc.com/resources/en-us/asset/data-sheets/products/converged-infrastructure/vxrail-datasheet.pdf.

[5] Gen-z. https://genzconsortium.org/.

[6] An introduction to ccix white paper. https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf.

[7] Mellanox Messaging Accelerator (VMA). http://www.mellanox.com/page/software_vma.

[8] Memcached. https://memcached.org/.

[9] Omni-path. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html.

[10] Open Network Linux. https://opennetlinux.org/.

[11] Very deep convolutional networks for large-scale image recognition. http://www.image-net.org/challenges/LSVRC/2014/results.

[12] Enabling programmable transport protocols in high-speed nics. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, February 2020. USENIX Association.

[13] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, 2010.

[14] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[15] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[16] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, 2013.

[17] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[18] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, 2011.

[19] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 267–280, New York, NY, USA, 2010. Association for Computing Machinery.

[20] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the ACM SIGCOMM 1994 Conference*, SIGCOMM '94, 1994.

[21] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data centers. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, 2014.

[22] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, 2017.

[23] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. *SIGCOMM Comput. Commun. Rev.*, 45(4):551–564, August 2015.

[24] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[25] Field programmable gate array over fabric. https://cdrdv2.intel.com/v1/dl/getContent/608298.

[26] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, 2015.

[27] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[28] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5), July 2008.

[30] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, SIGCOMM '17, 2017.

[31] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *SIGCOMM*, 2016.

[32] InfiniBand Trade Association. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1 Annex A17: RoCEv2. https://cw.infinibandta.org/document/dl/7781, 2014.

[33] Intel rack scale design v2.5: Architecture specification. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design/architecture-spec-v2-5.html.

[34] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 2015.

[35] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, 2014.

[36] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. Eyeq: Practical network performance isolation at the edge. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[37] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019.

[38] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with FlexNIC. In *ASPLOS*, 2016.

[39] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. Gpunet: Networking abstractions for GPU programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 201–216, Broomfield, CO, October 2014. USENIX Association.

[40] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, and et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 1–14, New York, NY, USA, 2015. Association for Computing Machinery.

[41] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. Picnic: Predictable virtualized nic. In *Proceedings of the ACM SIGCOMM 2019 Conference*, SIGCOMM '19, 2019.

[42] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *PProceedings of the ACM SIGCOMM 2019 Conference*, SIGCOMM '19, 2019.

[43] Radhika Mittal, Terry Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[44] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[45] Max Noormohammadpour and Cauligi Raghavendra. Datacenter traffic control: Understanding techniques and trade-offs. *IEEE Communications Surveys & Tutorials*, 20:1492 – 1525, 05 2018.

[46] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019.

[47] Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. Flowtune: Flowlet control for datacenter networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, 2017.

[48] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM*, 2014.

[49] 802.1qbb. http://1.ieee802.org/dcb/802-1qbb/.

[50] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, 2012.

[51] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 123–137, New York, NY, USA, 2015. Association for Computing Machinery.

[52] Intel Rack Scale Design Architecture. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf.

[53] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 255–270, Boston, MA, February 2019. USENIX Association.

[54] David Sidler, Zsolt István, and Gustavo Alonso. Low-latency tcp/ip stack for data center applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[55] R. Sivaram. Some measured google flow sizes (2008). google internal memo, available on request.

[56] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, 2012.

[57] Dell-EMC VxRail. https://www.dellemc.com/en-us/converged-infrastructure/vxrail/index.htm.

[58] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, 2011.

[59] Jackson Woodruff, Andrew W Moore, and Noa Zilberman. Measuring burstiness in data center applications. In *Proceedings of the 2019 Workshop on Buffer Sizing*, BS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[60] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference*. ACM, August 2015.

A Introduction

Rack-scale data center solutions like Dell-EMC VxRail [57] and Intel RSD [52] have emerged as a new building block for modern enterprise, cloud, and edge infrastructure. These rack-units have three key characteristics; First is the increasing use of *resource disaggregation and hardware accelerators* within these rack-units like GPUs and FPGAs [52], in addition to high-density compute and storage units. Second, *Ethernet* is by far the dominant interconnect of choice within such racks, even for communication between compute units, storage units and accelerators (e.g., Ethernet-pooled FPGA and NVMe in Intel RDS [33]). Third, these racks are deployed in a wide range of enterprise and cloud customer environments, running a heterogeneous mix of modern (e.g., machine learning, graph processing) and legacy applications (e.g., monolithic web applications), making it *impractical to anticipate traffic and workload patterns*.

Rack-scale networks[2] need to satisfy the key requirements of **uniform low latency** and **high utilization**, irrespective of where applications reside, and which accelerators they access (e.g., FPGA vs. CPU vs. GPU). However, three key obstacles stand in the way of achieving these goals because of the above-mentioned characteristics. First, the

[2]The network extending between NICs of such rack-units across the top-of-rack (ToR) switch.

rack-scale network **must be transport-agnostic**, a necessity in environments with *(a)* heterogeneous accelerator devices that have different characteristics[3] than CPU network stacks [25, 38, 39], and *(b)* increasing use of CPU-bypass networking [32, 35, 37]. Second, Ethernet is not a lossless fabric, and yet, our experiments (§B) on a 100G testbed confirm that **drops, not queueing**, are the largest contributor to tail latency pathologies. Third, the design must be **workload-oblivious** – given that we cannot anticipate traffic and workload patterns across a broad range of customer environments, it is impractical to rely on state-of-the-art proposals (§C) that hinge on configuring rate limits or priorities using a-priori knowledge of the workload.

In this paper, we present *Predictable Low Latency* or PL2, a rack-scale lossless network architecture that achieves low latency and high throughput in a transport-agnostic and workload-oblivious manner. PL2 reduces NIC-to-NIC latencies by proactively avoiding losses. PL2 supports a variety of message transport protocols and gracefully accommodates increasing numbers of flows, even at 100G line rates. It neither requires a-priori knowledge of workload characteristics nor depends on rate-limits or traffic priorities to be set based on workload characteristics.

To achieve these goals, senders in PL2 explicitly request a switch buffer reservation for a given number of packets, a *packet burst*, and receive notification as to when that burst can be transmitted without facing any cross traffic from other senders. PL2 achieves this form of centralized scheduling even at 100G line rates by overcoming the key challenge of carefully partitioning the scheduling responsibility between hosts in the rack and the Top-of-Rack (ToR) switch. In particular, the end-host protocol is kept simple enough to accommodate accelerator devices and implementations within NICs (§D.4), whereas the timeslot allocation itself is performed in the ToR switch at line rate (as opposed to doing so on a host, which is prone to software overheads).

In summary, our contributions are:

- The PL2 design that embodies novel yet simple algorithms for lossless transmissions and near-zero queuing within a rack
- A PL2 implementation using a P4 programmable switch and an end-host stack that leverages Mellanox's state-of-the-art VMA message acceleration library [7]
- A comprehensive PL2 evaluation on a 100 Gbps prototype, supporting three different transports (TCP, UDP and Raw Ethernet), all benefiting from near-zero queueing in the network. Compared to VMA, we demonstrate up to 2.2x improvement in the 99th percentile latency for the Memcached application; a 20% improvement to run a VGG16 machine learning workload; and near-optimal latency and throughput in experiments using trace-based workload generators.

[3]For example, FPGA stacks will not be connection-oriented due to scaling issues [54] and GPUs will not have a single receiver stack [39]).

B Motivation

The primary goal of PL2 is to provide uniformly low-latency across Ethernet rack-fabrics, while achieving high-utilization. We take inspiration from prior work around low and predictable latency within data center networks [13, 15, 16, 18, 20, 22, 26, 27, 29–31, 34, 36, 41–44, 47, 48, 56, 58, 60], but find that rack-scale networks provide a rich set of new challenges.

Rack-scale characteristics and implications

1. Even as line-rates increase, intra-rack RTTs are not getting smaller. [44] measured 5 µs end-to-end RTT on a 10 Gbps testbed with a single ToR switch, inclusive of software delays on the servers. A 64B packet still has an RTT of 5 µs in our 100 Gbps rack-scale PL2 prototype. Even though the network transmission times reduce proportionally with increasing line-rates, switching-delays, NIC hardware delays, and DMA transfer delays at end-hosts have remained about the same, and these delays together dominate transmission delays in a rack. Forward-error-correction delays increase as line-rates increase, and can add variability of up to 1-2 µs in a 100 Gbps network. This implies that rack-scale networks are able to transfer more data in the same RTT as interconnect speeds increase. Flows up to 61 kB can complete within 1 RTT with a 100 Gbps backplane as opposed to 6 kB for a 10 Gbps backplane. For congestion control protocols and predictable latency schemes to be effective for flows below these sizes, they will need to converge in sub-RTT timeframes.

2. Even as rack-densities go up, network buffering in ToR switches is not getting bigger. Shallow buffers are even more critical to a disaggregated rack, because buffering adds latencies to network transfers. However, the implication of this trend is that microbursts can over-run shared output switch-buffers and cause drops. For instance, a 2 MB buffer in a ToR switch with 100 Gbps ports provides buffering for just 160 µs which means only 8 simultaneous transfers of 2Mbits can be sustained before the switch starts dropping packets. Today's rack-scale networks support up to 64 rack-units [4], where each end-system can have tens of thousands of ongoing transfers. At that scale a 2 MB can be overrun with only 6 simultaneous 5 µs (1 RTT) network transfers per rack-scale unit. In short, as rack-densities go up, drops due to microbursts can be frequent. Therefore, assumptions made by congestion protocols like [30, 44] that the network-core (ToR switch in the case of racks) is lossless, no longer holds.

3. Rack-scale traffic is ON/OFF traffic [19] We believe this trend will continue with network traffic generated by accelerators. Measuring traffic-demands in such environments is hard, let alone learning about workload-characteristics; traffic demands at ns-timescales will be different compared to μs timescales and ms-timescales [59]. Workload churn and different application mixes adds to the unpredictability.

Coming up with effective rate-limits [18, 27, 34, 36, 41], and readjusting these rate-limits with changing traffic-conditions in time (i.e., less than an RTT) is impossible; so is setting relative packet priorities [44] effectively [40] so that important traffic is never lost or delayed. In our experience neither rate-limits nor priority prescription is an answer to congestion-control within a rack.

Drops cause the most noticeable tails

Based on the above three observations, we believe that new techniques are required to ensure low-latency and high-utilization within a rack-scale network. We hinge the PL2 design on the observation that drops, not queuing cause the most noticeable tails.

We illustrate this point with an experiment that introduces microbursts in our 100 Gbps testbed even when the network is partially loaded, by using 5-way incast of roughly 18 Gbps traffic per sender. All messages transmitted in our experiment can be transmitted within a single RTT. As shown in Figure 3a, the 99%ile latencies experienced by a receiver-driven scheme (RDS-baseline) based on Homa (described in detail in Section F) correspond to the maximum output-buffering available in the ToR switch (around 120 µs in Figure 3b), while the 99.9%ile latencies correspond to delays due to drops, and are two orders of magnitude higher. Reducing the drop-timeout in our implementation only increases drops, while only slightly reducing the 99.9%ile latencies.

In contrast, PL2's 99%ile and 99.9%ile latencies are similar to its median latencies, and it does not experience drops. PL2 is not impacted by microbursts; it keeps buffer occupancy low (maximum of 200 KiB). Figure 3c shows the drops (around 0.1%) experienced by RDS-baseline over time.

C Related Work

Priority Flow Control (PFC) [49] PFC is a closely related L2 mechanism that can also counter loss in a rack-scale network. Configuring PFC for correct operation is notoriously hard, even at rack-scale (see PXE booting issues in [28]), and turning on PFC in a rack that is part of a larger data center can be disruptive. Even within a single rack, PFC's coarse-grained mechanisms of providing losslessness across less than 8 traffic classes requires operators to choose between high utilization and lossless behavior because congestion in one downstream class can result in multiple unrelated senders receiving PAUSE frames due to HOL blocking [45].

Rack-scale interconnects Several custom designs for rack scale interconnects have been proposed; [23] propose direct-connect topologies, and [53] proposes circuit switched connectivity within a rack. [5, 6, 9] propose cache-coherent interconnects at rack scale to enable new computation models at rack-scale. PL2 differs fundamentally from all of these in vision. Even though other designs might perform better, PL2's goal is to allow traditional and commercially available rack-scale architectures to continue to avail the cost and operational ease benefits of Ethernet interconnects, but also to get predictable latency benefits;

Predictable latency [18, 27, 34, 36, 41] provide predictable latency by resource isolation among tenants or applications of a data center. All of these use rate-limit based network admission control to ensure isolation, and require a-priori knowledge of traffic characteristics (application mixes, demands). They typically dynamically readjust rate-limits based on new demand, but require considerable time to do so. For example, [41] requires a few RTTs for convergence. Often these systems can rely on inputs from applications, tenant requirements or systems like Bwe [40] to determine rate-limits. As described in Section B, PL2 cannot leverage these ideas in the rack-scale context.

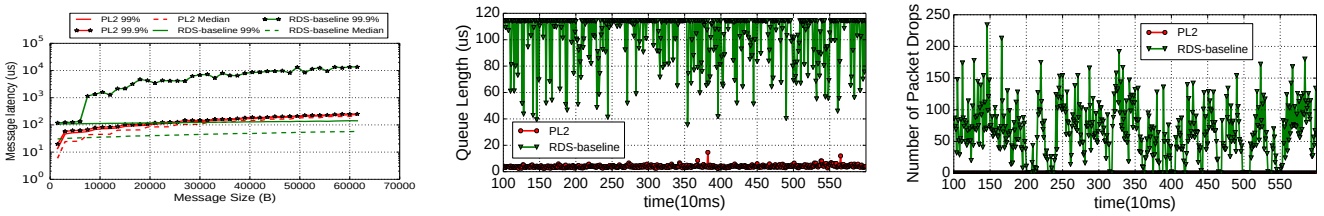
Congestion-control Our work follows the long history of low-latency, high-throughput congestion-control mechanisms. [22, 26, 30, 44] propose software-based receiver-side scheduling targeted towards 10 Gbps data center networks. Some of these schemes [30, 44] rely on the assumption that the network core does not experience loss; an assumption that is invalid in the rack-scale context (Section B).

Recent proposals suggest that starting new flows at line-rates [30, 44], or over-committing downlinks [44] could speed up network transfers; the experiment described in Figure 3 verifies that this idea trades off tail-latency for improved minimum and median latencies, which may not be beneficial [24].

Homa, pFabric, and others [15, 16, 27, 44, 56, 58] depend on prior knowledge of application traffic characteristics for providing benefits over other schemes; something that may be difficult due to shifting or short-lived workloads [50].

Most congestion control schemes proposed [13, 20, 29, 31, 42, 43, 60] rely on layer 3 and above to remedy congestion after it is observed either by way of delay, ECN marks, buffer occupancy or packet loss [21]. They require at least an RTT to respond to congestion and are too slow to prevent drops in a rack-scale network.

Fastpass [48] and Flowtune [47] proactively request permission to send packets and are the closest prior schemes to PL2. Fastpass decides when to send a burst of packets while Flowtune decides the rate to send a burst of packets. Their centralized arbiters, however, are host-based and cannot keep



(a) Message latency **(b) Switch queuing delay** **(c) Packets drop per 10ms**

Figure 3: Messages latency, switch queuing delay and packets drop during microburst

up with 100 Gbps line rates because they are bottlenecked both by scheduling software latencies and the downlink to the arbiter. Control packets to the centralized arbiter can also be dropped arbitrarily depending on NIC polling rates; these issues make centralized scheduling at an end host too fallible to be efficient and effective at 100 Gbps.

Most of these schemes (except for Fastpass and Flowtune) do not tackle the problem of being transport-agnostic; they rely on all traffic using the same end-host-based congestion control[4]. These schemes do not interact well with traffic that does not have congestion control. In PL2, the rack-scale interconnect intercepts traffic from all higher layers, and is therefore well suited to offer the properties we seek.

D PL2 Design

PL2 transforms the rack-scale network into a high-performanc lossless low-latency interconnect. At the heart of PL2 is an algorithm for scheduling packet bursts at line rate using the *switch dataplane*, where a packet burst is simply a bounded number of Ethernet frames.

PL2 is designed to be losslessness via proactive congestion control, while at the same time, being both transport-agnostic and, workload-oblivious.

We achieve losslessness via the scheduling algorithm, which implements a timeslot reservation scheme where each sender transmits only at its assigned timeslot, reducing cross-traffic. Since the scheduling function is placed in the switch dataplane in the network layer, it can schedule for packet bursts corresponding to all transports. The switch is one hop away from all hosts; therefore hosts can access the scheduling function at less than end-to-end RTT; we further eliminate scheduling overheads where possible. PL2 does not assume a-priori knowledge of traffic patterns or workloads.

In the following sections, we explain our scheduling algorithm first, followed by practical hardware constraints that inform its design. It is worth mentioning, for readers familiar with Fastpass, that we are unable to borrow the timeslot allocation scheme in Fastpass due to these hardware constraints; PL2 switch scheduling algorithm trades off optimal scheduling for speed. We elaborate on this towards the end of the next subsection.

[4]This is true of Homa also, which reorders traffic based on message size and would not interact well with TCP.

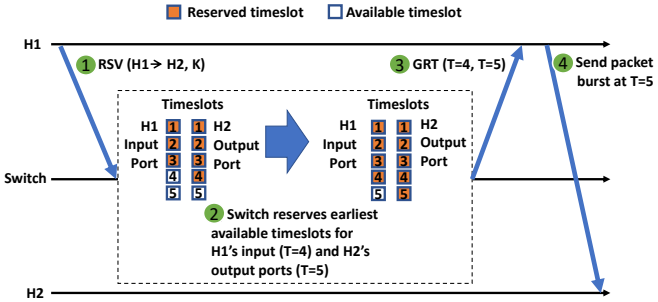


Figure 4: Scheduling example in PL2, with host H1 sending a packet burst to H2. (1) H1 sends an *RSV* to the switch to make a reservation. (2) The switch maintains timeslot reservations for the input and output ports connected to every hosts. It reserves the earliest available timeslots on H1's input port ($T = 4$) and H2's output port ($T = 5$). (3) The switch notifies H1 of these timeslots through a *GRT* message. (4) To avoid queuing, H1 then transmits at the maximum of the two timeslots indicated in the GRT, which is $T = 5$.

D.1 PL2 scheduling algorithm

Timeslot reservation overview Conceptually, our scheme maintains a list of *timeslots* per input and output buffer for each port. In our current implementation, we define a timeslot to be the time it takes to transmit an MTU sized packet out of a buffer[5]. To transmit a packet burst from host h to h', we seek to reserve a timeslot t on the switch input port corresponding to h, and a timeslot t' on the output port corresponding to h'. Host h then transmits at a 'chosen timeslot' which is the greater of timeslots t and t' to avoid a collision[6]. The astute reader will observe that we could instead let the switch choose the transmission time in a centralized manner

[5]The ideal duration for a timeslot is system and workload dependent. It should be chosen as a function of common (or minimum) message sizes and link speeds to ensure high utilization; transmitting small messages in large timeslots is wasteful. It's choice is also determined by the transmission granularity in a system; for e.g, a timeslot that is in picoseconds is useless because current hardware cannot transmit at such a fine granularity.

[6]Our design choice to always choose the greater of timeslots t and t' can create gaps in time when no packet is scheduled.

rather than pairwise, but hardware constraints prevent us from doing so (§D.2).

Note, with the hosts choosing the transmission times, there is a risk of collision. With perfect scheduling, we would have no collisions, and need near zero buffering at the switch. However, PL2 uses a small amount of buffer space (less than 200KB in our 100 Gbps testbed) to accommodate occasional collisions.

To run the above-mentioned scheme at line rate and within the constraints of switching hardware (outlined in §D.1.1), we designed an algorithm with the following division of logic between the switch and the hosts: *(i)* to transmit a packet burst of size K[7], hosts send *reserve* or RSV packets to the switch to reserve timeslots, *(ii)* the switch grants a reservation of size K and responds to the host with a *grant* or GRT packet, which specifies the earliest available timeslot for the corresponding input port and output port, *(iii)* the host then picks the maximum of the two timeslots to avoid a collision, *(iv)* finally, the host converts the timeslot into a waiting time after which it transmits the packet burst. In doing so, the timeslot reservation logic is divided between the switch and the host. Importantly, both the switch and host logic stays simple and in line with switching hardware constraints, which we describe below.

Algorithm 2 Switch Scheduling Algorithm. Each highlighted block is a single P4 operation.

```
1:  INIT:
2:  inReservation[port 1..port n] ← {0}
3:  outReservation[port 1..port n] ← {0}

4:  INPUT: packet
5:  src ←source port of RSV
6:  dst ←destination port requested
7:  if packet is a RSV then
8:      packet.sendTimeslot ← inReservation[src]
9:      inReservation[src] += packet.demand

10:     packet.recvTimeslot ← outReservation[dst]
11:     outReservation[dst] += packet.demand

12:     send GRT
13: else
14:     outReservation[dst] -= 1
15:     inReservation[src] -= 1
16: end if
```

D.1.1 Switch logic Algorithm 2 shows the scheduling logic at the switch. To stay ahead of packet transmissions, despite the delay of each RSV-GRT exchange, the switch creates a schedule of input and output reservations for every port, in terms of timeslots. Each highlighted gray box represents logic that can be implemented with a single P4 operation.

At switch start up, the input and output reservations are initialized to zero (lines 2-3). In response to RSV packets, the switch sends back the next available input and output timeslots for the requested transmissions (lines 8,10,12). It also reserves enough timeslots for each RSV request (lines 9,11). In Figure 4, the switch has reserved timeslot 4 at the source port (connected to Host 1) and timeslot 5 at the destination port (connected to Host 2) for transmission. Note that the reservation timeslots have not lined up exactly at the two ports. We describe how the sending host uses these timeslots next. Lines 13-15 describe switch logic for regular packets, in which the timeslot reservations for the packet are removed.

D.2 Hardware constraints

There are two key hardware constraints that Algorithm 2 satisfies. First, currently available programmable switching hardware cannot access more than one stateful memory object (SMO) at a time in an operation, per packet, at line-rates. This is why `inReservation` and `outReservation` timeslot counters in Algorithm 2 are accessed and updated in two separate operations. Second, all pipelined network hardware can only read/modify/write SMOs once during the processing of packets. If a SMO is updated or accessed twice during the same pipeline, it results in race-conditions across packets. This is why `inReservation` and `outReservation` have to be read and modified in a single atomic operation in Algorithm 2.

In the absence of the first limitation, the switch could update both `inReservation` and `outReservation` to max(`inReservation`, `outReservation`). If it could also cache the computed timeslot in packet metadata, it could send this timeslot information in response to a RSV message. Since the chosen timeslot increases monotonically, this scheme removes any possibility of collisions. In the current implementation, the switch relays `inReservation` and `outReservation` to the host, which computes the chosen timeslot to send at.

Why not implement Fastpass in a switch instead? Fastpass [48] is able to perform timeslot allocation with maximal matching because the scheduler processes a list of all demands in the network at once; implementing such a scheme is impossible in the switch dataplane at line-rate, because switch pipelines cannot compare multiple RSV packets in flight.

In addition, Fastpass performs timeslot allocation using a bitmap table, that has a sender and receiver bitmap to track multiple timeslots. Allocation requires a bitwise `AND` of the sender and receiver bitmap, followed by a 'set' on the first available timeslot. Supporting such an algorithm would require hardware to be able to access and set at least two

[7] K is technically the number of timeslots a sender is allowed to reserve at a time.

stateful memory objects in a single operation. Maintaining a sliding window of timeslots is similarly hard to achieve with dataplanes today because they expose minimal timing API that are restricted to timestamping — converting these timestamps to sliding windows requires accessing multiple stateful memory objects, one that maintains timestamp information, and another that maintains timeslot information, and updating them at line-rates for every RSV packet.

D.2.1 Host logic On the host side, for lossless transmission, senders must ensure that their packet bursts do not collide with other transmissions at *both* the corresponding input and output ports on the switch. In fact, input and output ports at a switch might be shared by multiple hosts (like when a 100G link is divided into four 25G links and connected to four different hosts). End hosts in PL2 therefore conservatively choose a timeslot to transmit that is available on both the relevant switch input and output ports using the equation $chosenTimeslot \leftarrow max(sendTimeslot, recvTimeslot)$.

The sender then transmits packets at the `chosenTimeslot` by waiting for a period `waitingTime` calculated using the equation

$$waitingTime = timeslotWait - rsvGrtDelay - pendingDataDelay, \quad (2)$$

where $timeslotWait = chosenTimeslot * MTU/linerate$ and `pendingDataDelay = bytes/linerate`.

The explanation for this calculation is as follows: The timeslot chosen for transmission is `timeslotWait` seconds into the future from when the reservation is made at the switch. This reservation is conveyed back to the sender after `rsvGrtDelay/2` seconds. The first packet of the transmission would again take approximately `rsvGrtDelay/2` seconds to reach the switch, after transmitting all the previously scheduled packets at the sender NIC (which would take `pendingDataDelay` seconds)[8]. The sender therefore waits only for the remaining fraction of time to ensure that the first packet of the burst arrives in time at the switch.

The aforementioned logic has some leeway with regards to where on the host it runs. In our current userspace stack implementation, one instance of the PL2 host logic runs per application thread and each thread maintains at most one outstanding RSV-GRT exchange. We believe a better implementation would be one where the host logic runs inside a NIC. In that case, the NIC can allow sending threads to have one outstanding RSV-GRT exchange per destination mac address, so that a thread sending messages simultaneously to multiple destinations can do so without encountering head-of-line blocking for messages to unrelated destinations.

[8]We ensure that RSV-GRT packets do not wait behind data packets (if any) by prioritizing them using 801.1q

D.3 Setting the packet-burst size, K

A key parameter in PL2 is the packet-burst size, K. PL2 ensures stable queuing by proactively scheduling transmissions such that when these packet bursts are transmitted, they encounter almost no queuing. However, the timeslot reservations for the relevant input and output ports for a transmission determine when packet bursts are transmitted. The switch reserves as many timeslots as needed for a transmission based on the demand from the host, which is capped by K.

A small value of K limits the amount of head-of-line blocking a burst introduces at its input and output ports and ensures that PL2 supports a large number of concurrent transmissions at any point in time. However, when K is too small, the overhead of the RSV-GRT exchange dominates, lowering throughput and effective utilization. Similarly, large values of K help amortize the cost of the RSV-GRT exchange delay, but increase head-of-line blocking because that causes the switch to reserve a burst of timeslots for the same host, potentially starving other hosts.

We find that $K = 4$ works best for our 100G testbed prototype and in our simulation for all the workloads we tested, based on a parameter sweep. §E details the configurations in our testbed and simulations.

D.4 Reducing scheduling overheads

Each RSV-GRT exchange enables senders to determine the waiting time before transmitting on a given input/output port pair. Under light loads, the waiting times for a sender will mostly be 0ns (or nominal at best), providing an opportunity to reduce the number of exchanges required to send messages. Such a reduction has the potential to reduce the minimum end-to-end message latencies in PL2, which are otherwise impacted by RSV-GRT exchange overheads.

We therefore design an optimization that enables senders to send *unsolicited* packet bursts immediately after sending a RSV packet to the switch, without waiting for the GRT. Unsolicited bursts are allowed only when the following two conditions are satisfied: (i) the timeslots at the input and output port known from a previous GRT are *both* below a threshold t and (ii), the information about the reservation is deemed to be recent, i.e., obtained within a certain interval of time (e.g., comparable to the time for a RSV-GRT exchange). Condition (ii) is met when senders send consecutive bursts to the same destination. Packets that are sent unsolicited are marked by using a spare packet header field.

Algorithm 3 shows the sender side scheduling logic. As usual before sending a burst, RSV packets are sent (line 7). However, if the `lastChosenTimeslot` was within `t` and the previous GRT was recent, an unsolicited packet burst is sent even before the next GRT arrives (line 8-9). The switch schedule is modified to pass through unsolicited bursts, unless the

Algorithm 3 Scheduling at sender

1: **PARAMETERS:** t, K
2: **INIT:**
3: $lastChosenTimeslot \leftarrow \{-1\}$
4: $lastResponseTime \leftarrow \{0\}$

5: **Function scheduleBurst**
6: **INPUT:** packet burst with K or fewer packets
7: Send RSV for burst
8: **if** $lastChosenTimeslot < t$ and $lastResponseTime$ is current **then**
9: Send *unsolicited* packet burst
10: **else**
11: Call receiveGRT
12: Send packet burst after $waitingTime$
13: **end if**

14: **Function receiveGRT**
15: **INPUT:** GRT
16: **if** $chosenTimeslot > t$ **then**
17: **if** $lastChosenTimeslot < t$ **then**
18: Resend packet burst corresponding to GRT
19: **end if**
20: **end if**
21: $lastChosenTimeslot \leftarrow chosenTimeslot$
22: $lastResponseTime \leftarrow now()$

reservation queues are larger than a threshold $T >> t$, in which case unsolicited bursts are dropped (not shown in algorithm 2). When the sender receives a GRT back, it determines whether the new `chosenTimeslot > t` (where `chosenTimeslot` is calculated using the equation from Section D.2.1) and if so, resends the packet burst corresponding to the GRT (lines 16-20) at the right timeslot. This ensures that the packet burst arrives at the time it is expected at the switch, even if the unsolicited burst was dropped. However since `T >> t`, it is also possible that both the unsolicited and scheduled packets arrive at the receiver, and the receiver has to deal with a small number of duplicate packets.

In general, the threshold `t` is set to a sufficiently low value, so that the optimization only applies at very low loads. In our testbed implementation, we found `t` to be robust across a broad range of values and workloads. All our testbed experiments are run on a setting of `t=15`. Sweeping through `t=10` up to `t=25` does not show statistically significant changes to loss rates or latency (of course, setting `t` to large values like 35 does lead to loss).

This optimization results in some transmissions arriving at the switch at the same time under low load. We find that when there are only a few senders to a port, and there are continuous transmissions at a low rate, the optimization helps reduce the minimum message latencies; this is because the queuing caused by such simultaneous transmissions is smaller than the RSV-GRT message exchange delay under low loads.

Tonic [12] and Sidler et. al. [54] have demonstrated that it is possible to place complex congestion control logic into NIC hardware. Since algorithm 3 requires only a minimal subset of the supported logic (timed transmit), we believe it will be easier to implement in NIC hardware. Such an implementation will allow GPU and FPGAs (apart from CPUs) to access PL2 logic directly.

D.5 Other design considerations

Implications for intra-rack transports. We find that PL2 is able to provide congestion control to all traffic within the rack. One key advantage with TCP over PL2 is that transmissions with PL2 rely on current knowledge of network demand (using RSV-GRT), rather than TCP's window estimate, which might become stale depending on the time of the last transmission within a flow. We are able to completely turn off TCP congestion control when using PL2 underneath in our 100G rack prototype(§E).

Handling failures. When a PL2 ToR fails, the entire rack fails, as is the case with non-PL2 rack. When a PL2 sender fails, its reservations on the ToR switch (up to K outstanding for each connection) will need to be removed using external detection and recovery logic.

One way to detect a failed sender is to have GRT packets be additionally sent to receivers, and have receivers track pending transmissions. This adds a small overhead comparable to an ACK packet for every K packets, and therefore trades-off a small amount of receiver bandwidth for better failure tolerance. When a sender crashes, or misbehaves, the receiver can detect the failure and reset the pending reservations at the switch. This scheme has an additional advantage of informing the receiver of upcoming transmissions; receivers can use this information to schedule the receiving process in time to receive the transmission to further drive down the end-to-end message delay with systems like [46]. We aim to look at this issue in the future and study its overheads.

E Implementation

We have built a 100 Gbps solution that uses PL2 in a rack. The switch scheduling algorithm (Alg 2) is implemented using a P4 program and runs on a ToR switch' programmable dataplane ASIC [3]. The switching delay with PL2 enabled is measured to be between 346 ns (min) to 508 ns (99.99%ile), with a median delay of 347 ns and standard deviation 3 ns. `inReservation` and `outReservation` are 32-bit register arrays updated in one switch pipeline stage; Since the switch has 64 ports, each array consists of 64 registers.

We use the Mellanox Messaging Accelerator (VMA) [7] to prototype the sender side scheduling support; we choose

VMA instead of DPDK [2] and the Linux network stack because VMA provides lower latencies on Mellanox NICs. The TCP/IP library integrated in VMA allows us to compare TCP and UDP performance with and without PL2. We are able to turn off congestion control support in TCP when using PL2 underneath. We also augment RDMA raw Ethernet with PL2 scheduling to mimic PL2 support for traffic generated by accelerators that might lack congestion control.

PL2 prototype topology and configuration PL2 hardware prototype is a rack with 6 servers, each equipped with a Mellanox ConnectX-5 100 Gbps NIC. Each server has two 28-core Intel Xeon Gold 5120 2.20 GHz CPUs, 196 GiB of memory, and runs Ubuntu 18.04 with Linux Kernel version 4.15 and Mellanox OFED version 4.4. The servers connect to a 6.5 Tbps programmable dataplane switch [3], with 64 physical 100 Gbps ports. The switch runs OpenNetworkLinux [10] with Kernel version 3.16. The network MTU is 1500 bytes.

Our servers also connect to a Mellanox SN2700 switch using a second port on the Mellanox ConnectX-5 NIC. The servers synchronize over this out-of-band network using IEEE 1588 Precision Time Protocol (PTP) [1] using hardware NIC timestamps and boundary clock function at the Mellanox switch. Using more precise time synchronization will improve PL2 scheduling accuracy.

RSV-GRT exchange delay We implement RSV and GRT packets as 64-byte Ethernet control packets. PL2 continuously measures RSV-GRT delays using hardware and software timestamping and uses these measures to correctly estimate packet transmission times (equation 2). We find that RSV-GRT exchanges can have variable delay even in an unloaded network; the exchanges take between 1 µs (min), 1.06 µs (median) to 14us (max) NIC-to-NIC, using hardware timestamping and 1.98 µs (min), 2.05 (median) and 22.25 µs (max) using software timestamping in a network in an idle network. Interestingly we see similar variance and tails when we transmit data close to line rate using PL2. We anticipate that the performance that PL2 offers will get better if this variability is remedied.

Interfacing with application `send` PL2 is prototyped on a user-space stack, where application `send` and `receive` is implemented in the same thread (as opposed to `send` and `receive` being handled in separate threads). We intercept function calls within the VMA library [7] that executes the actual `send` (`send_lwip_buffer` and `send_ring_buffer`) and send RSV packets for every K or fewer packets. Because the receive executes in the same thread, we also wait for the GRT before sending each packet burst; i.e., the application `send` call blocks until transmission completes. When employing the low-load optimization discussed in §D.4, even though we send packet bursts together with the RSV, we wait to receive the GRT before transmitting the next burst. As such, we have not been able to eliminate the overheads due to RSV-GRT exchange to the degree that the optimization design permits in our current implementation. A more favorable implementation of PL2 would execute RSV-GRT receives in a separate thread or offload RSV-GRT exchanges into the NIC.

F Evaluation

We evaluate PL2 100 Gbps prototype for the desired properties of losslessness, low latency and high utilization across various transports.

F.1 Experiment setup and methodology

We use burst size $K = 4$ and threshold $t = 15$ in all experiments, configured as described in §D.3 and §D.4. We next describe the applications, transports, loads and traffic patterns we have evaluated.

`memcached` [8] We evaluate the impact of worst-case background loads on latencies and throughput of single `memcached` client-server instances that reside on separate hosts. The key are 64B with 1 KiB and 4 KiB values. `memcached` uses TCP communication. The client executes reads (GET) continuously and the responses compete with background traffic. We use reads rather than a mix of reads and writes because reads stress both the forward and reverse paths. Without background traffic,`memcached` introduces 2-3 Gbps network load.

Machine learning with vgg16 [11] VGG16 is a popular convolutional neural network model used for large-scale image recognition. We emulate the network communication for VGG16 training, where gradients from each neural network layer are transferred over the network. Each parameter server in our set up receives messages from 4 workers at a total load of 70-89 Gbps.

Workload traces We evaluate PL2 with traces generated from message-size distributions collected from various real-world modern application environments (W1-W5) [14, 17, 51, 55] also used in Homa [44]. W1 is generated from a collection of memcached servers at Facebook, W2 from a search application at Google, W3 from aggregated RPC-workloads from a Google datacenter, W4 is from a Hadoop cluster at Facebook and W5 is a web search workload used for DCTCP. The traces are generated in an open-loop with Poisson arrivals. These workloads represent modern database and analytics workloads that we expect rack-scale networks to run. The workload generator that replays these traces does not prioritize messages from one thread over the other. We believe this accurately mimics several user-space applications competing for the network independently across several cores.

Transport protocols We evaluate the performance of raw Ethernet, UDP and VMA TCP over PL2. raw Ethernet

and UDP represent accelerator transports that do not have end-host-based congestion control support. When we use PL2 underneath VMA TCP we turn off VMA's congestion control support[9]. We compare PL2 congestion control with VMA TCP Cubic and a receiver-driven congestion control scheme (RDS) based on Homa.

Our RDS implementation achieves close to 100 Gbps line-rates by separating receiver-driven scheduling from data-processing; we schedule up to 4 packets when possible and use a lock-free mechanism for communication between the scheduling and data-processing threads. When packets are lost, the receiver can detect these losses by monitoring out-of-order packet arrivals. In these cases, the receiver notifies the sender of the lost packets immediately (rather than waiting for a timeout [44]). This helps reduce delays due to a majority of lost packets. Sometimes an entire burst of packets is lost, and in these cases, the receiver cannot effectively determine if the packets are lost or delayed. Instead they timeout after 1 ms and send a lost-packet notification to the sender. Senders send up to 61 KiB, the bandwidth-delay product in our system blindly as fast as they can (similar to RTTbytes in [44]). If these initial packets are lost, the sender times out after 1 ms, and resends the packets again.

Server-server raw Ethernet provides an optimal baseline for comparison with PL2. raw Ethernet is unhindered by congestion control, and transmits messages as fast as they arrive. No congestion control scheme can achieve smaller delays than raw Ethernet. Recently [30, 44] showed that RDS-like mechanisms can provide significantly smaller latencies than available congestion control schemes. Therefore we choose to implement and compare PL2 with RDS on the testbed. We also compare PL2 with VMA TCP Cubic, because VMA touts impressive latency improvements.

We are unable to make a fair-comparison with DCTCP [13] and other available congestion control schemes in the linux kernel because PL2 is only implemented at user-space (using VMA), and does not experience the overheads of the linux kernel.

Traffic patterns, link loads and latency evaluation We evaluate PL2 under incast, outcast, and shuffle traffic patterns. Incast helps demonstrate the lossless behavior of PL2. Outcast is PL2's worst-case traffic pattern, because it stresses senders that not only transfer data but also have to do RSV-GRT signalling.

We control the network load at each server by injecting traffic using multiple threads pinned to different cores. We experiment with both persistent congestion caused by multiple long running background flows, and with microbursts caused by replaying W1-W5 traces from multiple threads.

[9]VMA only provides Cubic and Reno congestion control options, of which we have chosen Cubic

We present results of 99%ile and median latencies measured in user-space; for memcached, we measure two-way request-response delays; for other workloads, we measure one-way latencies, i.e., the time from message arrival at the sending thread to the time when the message is delivered at the receiving thread.

The rest of the evaluation section summarizes our findings.

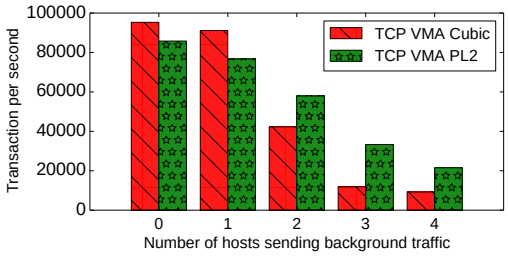


Figure 5: Memcached throughput with competing TCP traffic.

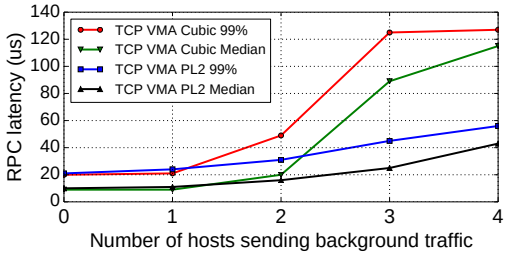


Figure 6: Memcached latencies with competing TCP traffic.

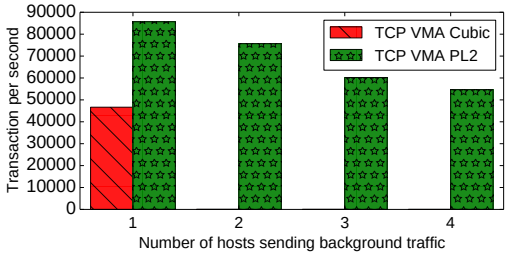


Figure 7: Memcached throughput with competing UDP traffic.

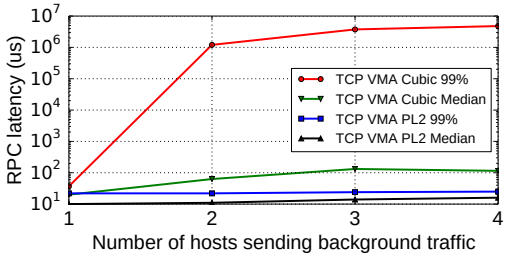


Figure 8: Memcached latencies with competing UDP traffic.

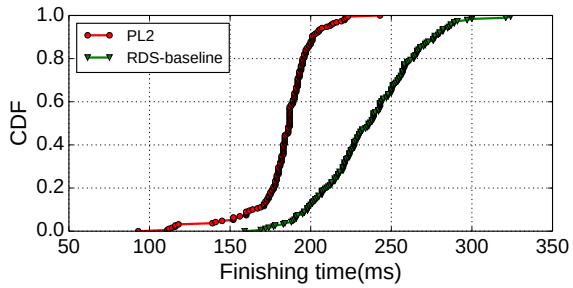


Figure 9: CDF of latency for all iterations in VGG16 model training using PL2 and RDS.

F.2 End-to-end application performance

We experiment with two real-world applications, memcached and vgg16 training. When memcached competes with heavy (incast) background traffic with congestion control support, PL2 can ensure up to 2.3x lower rpc-latencies, and 1.8x corresponding improvement in application throughput compared to VMA TCP Cubic. When it competes with traffic without congestion control(the kind of traffic we expect from accelerators), memcached over PL2 still sees similar latencies and throughput; whereas without PL2, memcached rpcs fail to complete due to severe losses. This demonstrates that PL2 is able to provide fabric-level congestion control for all transports that use it.

Experiments with VGG-16 training show that PL2 can reduce network transfer times per iteration by 30% compared to RDS (222ms vs 321ms 99%ile latency per iteration); we achieve a 20% speed up for 100 iterations (9.5s versus 12s).

F.2.1 PL2 keeps memcached 99%ile latencies below 60 µs **even with 400% offered load** Figures 5-6 show memcached transaction throughput and response latencies with TCP background traffic. The graphs shown are with 4 KiB values; the results for the same experiment with 1 KiB values show the same trend. Figure 5 shows how memcached throughput (in transactions per second) reduces with increasing TCP background traffic. The red bar shows throughputs with memcached over TCP over PL2 (TCP VMA PL2), and the green bar shows throughputs with memcached over TCP with Cubic congestion control (TCP VMA Cubic).

When there is no background traffic (bars for 0 hosts sending background traffic), PL2 slows down memcached by a small amount(8%), even though the RPC latencies are similar to TCP VMA Cubic (Figure 6). PL2 RSV-GRT exchange overheads contribute to reduced throughput seen with TCP VMA PL2. The same effect is seen in the case where the memcached host also sends background traffic that utilizes around 50% of the downlink to the memcached client (50 Gbps). Once we add background traffic load from additional servers, the load on the receiving link increases to 80-90% and the memcached latencies go up much faster for TCP VMA Cubic than TCP VMA PL2 as seen in Figure 6; consequently memcached has 0.5x lower throughput with TCP VMA Cubic. This is despite the fact that the competing background traffic that runs using TCP VMA PL2 has 1-5 Gbps more load than the background traffic that runs on VMA TCP Cubic. The 99th-percentile RPC latency with VMA TCP Cubic is 127 µs as opposed to 56 µs with PL2 with 4-way TCP incast background traffic.

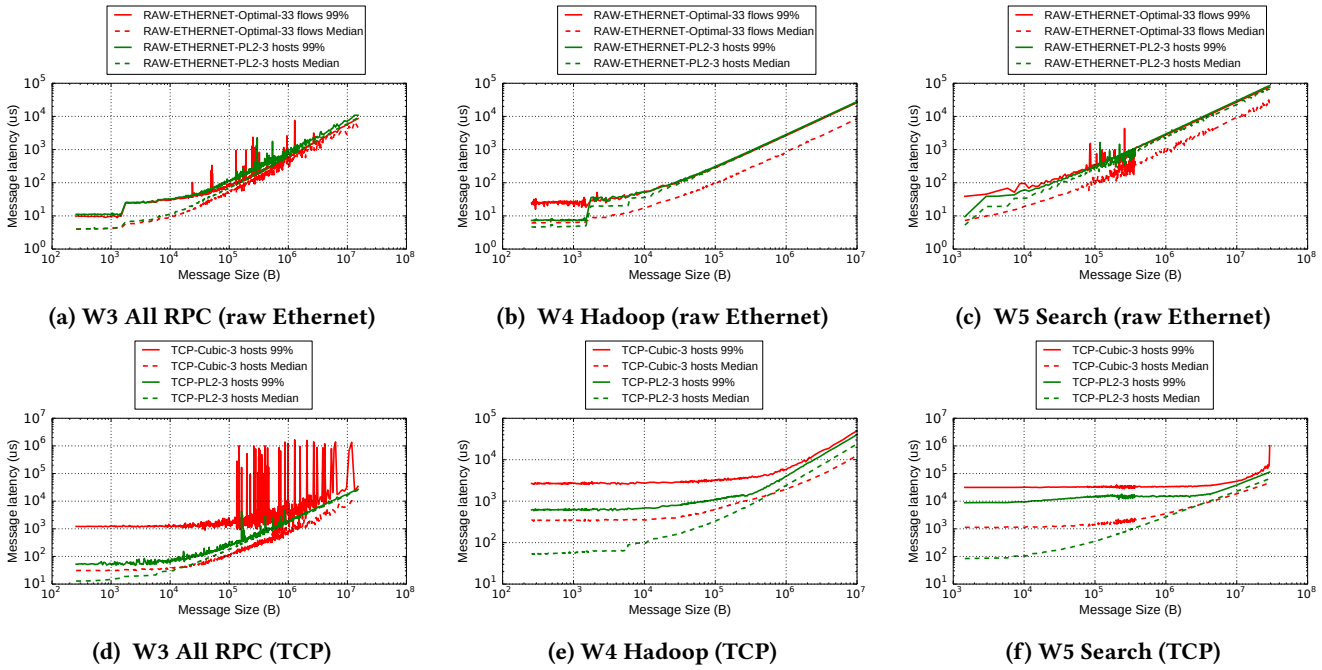
F.2.2 PL2 keeps memcached latencies low (25 µs**) even when competing with traffic with no end-host-based congestion control** Figures 7-8 show memcached transaction throughput and response latencies with UDP background traffic (memcached traffic still uses TCP). Since UDP has no congestion control, it presents particularly severe competition to memcached traffic. Our goal with this experiment is to show how PL2 enables multiple transports with or without congestion control to co-exist within a rack.

Figure 7 shows the throughput of memcached as we increase UDP background load. Without PL2, the memcached benchmark does not complete due to severe losses. Figure 8 shows the 99%ile and median memcached RPC latencies with VMA TCP Cubic and PL2. Memcached sees 5s 99th-percentile latency without PL2 (due to drops) as opposed to 25 µs with PL2 with 4-way UDP incast background traffic.

Once we introduce background traffic from 2 or more hosts, we find that memcached incurs severe losses without PL2, and its throughput drops to 3-7 transactions per second, while the tail latency shoots up to several seconds (Figure 8). However, with PL2, the memcached RPC latencies do not degrade with such severity (even the tail latencies remain steady, with some increase in the median), and memcached continues to have throughputs similar to our experiment with TCP background traffic. This is because PL2 keeps the UDP background traffic from 2-4 hosts within 75-90 Gbps.

F.2.3 PL2 improves training latencies for vgg16 by 30% The communication pattern for exchanging gradient updates in VGG16 is inherently a shuffle process. We have 4 hosts sending data as workers and 2 hosts as parameter servers, which receives gradients from all the workers. The gradient data (500 MiB) is partitioned according to VGG16 architecture. The aggregate incoming rate to each parameter server is less than the line rate to ensure the network is not a bottleneck. We repeat this process 100x and measure the finishing time of transferring the entire gradient set at each iteration.

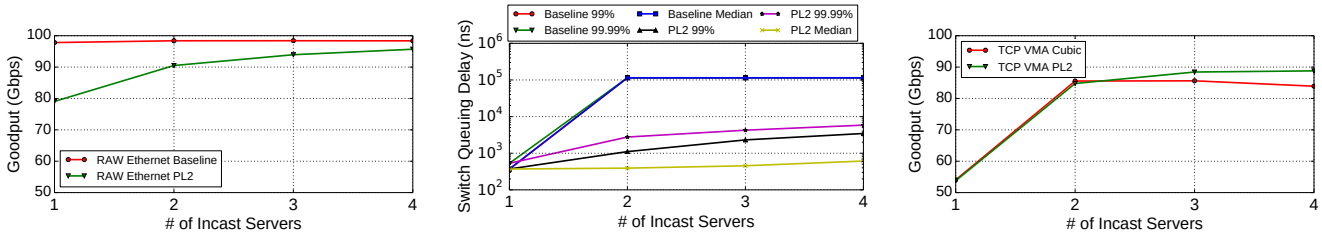
Figure 9 shows the CDF of iteration times using the receiver-driven scheme and PL2. We did not prioritize small messages with the receiver-driven scheme like Homa [44] because it interferes with the ordering of messages that the training set expects. As such, we find that the 99th percentile finishing time of each iteration in the receiver-driven scheme



(a) W3 All RPC (raw Ethernet) (b) W4 Hadoop (raw Ethernet) (c) W5 Search (raw Ethernet)

(d) W3 All RPC (TCP) (e) W4 Hadoop (TCP) (f) W5 Search (TCP)

Figure 10: Message latencies of workloads W3-W5 for raw Ethernet and TCP



(a) Throughput over Ethernet (b) Switch queuing delays for raw Ethernet and PL2 (c) Throughput over TCP

Figure 11: Incast evaluation comparing against raw Ethernet and TCP.

is 1.45x than that of PL2. To finish 100 gradients sets from each worker, PL2 takes 9.5s in total while the receiver-driven scheme takes 12s. The receiver-driven approach is slower because the receiver is unaware of traffic from the sender to other hosts: it frequently allocates time slots when the sender is already sending to another receiver. This leads to lower link utilization. PL2 does not have this issue because PL2 counts the queuing both at the sender and receiver.

F.3 W1-W5: Near-optimal p99 latencies

We compare how close message latencies for W1-W5 are to optimal with 3-way incast using raw Ethernet over PL2. We do this by using a baseline where W1-W5 are run using just raw Ethernet between two servers. Since the baseline encounters no contention, and raw Ethernet transmits messages as soon as they arrive (with no congestion control), the message latencies it encounters are close to optimal. We call this scheme raw Ethernet (optimal). When comparing 3-way incast results with raw Ethernet over PL2 to raw Ethernet (optimal), we try to keep the network load equivalent. We do so by using the same number of threads in total to run the workload across the two schemes (11 threads per server, 33 threads in total).

Figs. 10a-10c present the message latencies for workload traces W3-W5 over raw Ethernet. For these workloads, 3-way incast achieves 40-90 Gbps throughput. PL2 over Ethernet has 99%ile latencies close to raw Ethernet (optimal), and the throughputs obtained with and without PL2 are similar (except in the case of Figure 10b where PL2 over Ethernet

gets slightly lower throughput compared to raw Ethernet (optimal) for W4 (86 Gbps vs. 89 Gbps) due to RSV-GRT exchange overhead). The impact of RSV-GRT exchanges are more prominent in the median latencies in Figs. 10b and 10c.

We see latency spikes in Figures 10a and 10c; these are outliers, when the message arrivals exceeded the capacity of our workload generator when it ran out of available worker threads to transmit the next message.

F.3.1 W1-W5 over TCP: 10x lower 99%ile latencies We also compare 3-way incast of W1-W5 with TCP-over-PL2 to 3-way incast with TCP Cubic. Figures 10d, 10e, 10f show that TCP-Cubic has worse median and 99%ile latencies than TCP over PL2, except in the case of Figure 10d. VMA TCP Cubic has better latencies with W3, because it undergoes a congestion collapse and achieves only 43 Gbps, as compared to 81 Gbps with TCP over PL2. In all other cases, we find that VMA TCP Cubic achieves throughputs close to TCP with PL2 while having retransmissions due to message loss (TCP with PL2 has no losses). We believe these graphs show the stable-queuing effect of proactive congestion control.

Our workload generator is unable to generate greater than 5 Gbps traffic for W1 and W2 even in an incast scenario because the master thread cannot keep up with assigning messages to connections within the inter-arrival times. We find that when workloads impose such light loads, PL2 does not give significant benefits over VMA TCP Cubic, nor does it cause significant degradation; we omit these results for space and refer the reader to our results with memcached with no background traffic in Figure 5.

F.4 Throughput implications

Figure 11a shows the aggregate throughput achieved by PL2 over Ethernet in comparison to raw Ethernet, with incast, while not having loss. The x-axis shows the number of hosts that are sending out persistent traffic to the same receiver and the y-axis shows the throughput in Gbps. Each host starts 12 flows (pinned to separate cores on the sender), and sends 24 million 6 KiB messages (4 MTU sized packets). raw Ethernet achieves line rate for the case of one host sending traffic to another host without any packets drop. However, as the number of sending hosts n increases, we find that only only $1/n$ packets are delivered to the receiver (all senders are able to send at line-rate). PL2 over Ethernet has no losses but caps server-to-server throughput to 80 Gbps. The throughput depends on the number of flows in PL2 (12), because each flow has at most one outstanding RSV when the network is loaded.

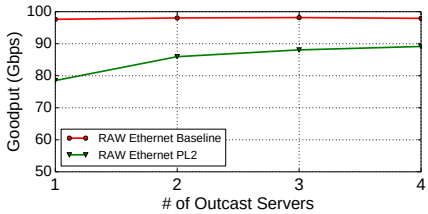


Figure 12: Outcast throughput comparison raw Ethernet vs. PL2 over Ethernet

Figure 11b shows the switch queuing delays for the same experiment. As can be seen raw Ethernet has uncontrolled queue lengths, whereas PL2 over Ethernet achieves stable queueing with low variance.

Figure 11c shows PL2's aggregate throughput in comparison to TCP cubic, with incast traffic. The experiment settings mimic the raw Ethernet experiment. TCP cubic experiences 233, 313, 404 retransmissions for 2, 3, and 4 host incasts. In the case of congestion controlled TCP traffic, traffic over PL2 sees the same (or higher) throughput as the baseline, i.e., PL2's proactive scheduling is comparable to TCP's reactive scheduling, while preventing losses.

Outcast traffic pattern stresses PL2 senders to the maximum extent. With raw Ethernet outcast, as shown in Figure 12 (with the same settings as above), the maximum throughput PL2 can achieve is capped at 90 Gbps (as opposed to 97 Gbps in the case of incast). This drop (10%) is due to the overhead of RSV-GRT exchanges at the sender; it is the price PL2 pays for it's proactive scheduling design.

G Limitations

Utilization PL2 cannot fully utilize the Ethernet capacity available, although gets close (up to 96 Gbps utilization with incast, and up to 90 Gbps with outcast). RSV and GRT signalling overheads take up some spare capacity; in our implementation PL2 adds a minimum of 2% bandwidth overhead (128B of overhead for $K = 4$ MTU (1500B) packets). In the worst case when all network packets are at 64B, the overheads are comparable to the demand in the network. However, this is not the common-case for which PL2 is designed. Variation in hardware delays due to DMA transfer latencies can introduce scheduling inefficiencies. PL2 also leans towards preventing losses in selecting the time duration to wait before sending packets, and thus can leave some capacity unused.

Fairness Like Homa [44], PL2 is unfair but not to large messages. PL2 is unfair because the switch scheduler schedules bursts in FIFO order of receiving RSV packets. The scheduling size K limits this unfairness; under loaded conditions, until the scheduled K packets are sent, the next RSV packet is not sent. We believe that PL2 scheduler design should change to eliminate this unfairness once switch-hardware is able to handle richer reservation logic.

In-network priorities PL2 does not implement in-network priorities; this design also stems from the limitations of

switch-hardware processing RSV packets in FIFO order. However, given that PL2 does not drop packets, and ensures that at any point only a few (K) packets from different senders are in flight to a receiver, we anticipate that in-network priorities will not have a big role to play in further reducing latencies with PL2. Rather the order in which RSV packets are admitted into the network would play a bigger role, which in turn is determined by (i) external policies that govern rate at which RSV packets can be sent by an application; (ii) process scheduling prioritization within the rack; for e.g., how many cores the communicating processes are given, how often applications are scheduled in comparison to others; and (iii) application's internal logic. We aim to study these implications in our future work.

Handling inter-rack traffic To the PL2 scheduler, traffic leaving the rack using a port on the ToR is no different from intra-rack traffic; the traffic exiting will also be scheduled using the timeslot reservation scheme. However this traffic will not be co-ordinated with other inter-rack traffic destined to an external rack. With PL2's current design, static bandwidth reservations will be required to ensure that traffic entering a PL2 rack will not disrupt PL2 guarantees for intra-rack traffic, or be dropped altogether; e.g., 10% of rack bandwidth is reserved for ingress traffic. PL2 can reflect such reservations either in the length of timeslot or the reservation increments.

H Conclusion

In this paper, we present the PL2 rack-scale network architecture designed to convert Ethernet into a reliable, predictable latency, high-speed interconnect for high density racks with accelerators by leveraging new capabilities in programmable switches. Our hardware prototype demonstrates that PL2 does so by providing practical and tightly coordinated congestion control. Further, we achieve our goals without any knowledge of workload characteristics or any assumptions about future hardware stacks. We believe that the design presented in this paper will spur new ideas around switch and NIC hardware, how they interface with and simplify the network and applications stack.

This work does not raise any ethical issues.