High Performance Network Multiplexing with IX++

Gregory D. Hill Albert H. Chen

Stanford University

1 Abstract

IX++ is an extension of the IX Dataplane Operating System that uses PCI Single Root I/O Virtualization (SR-IOV) to run multiple networked applications concurrently with minimal interference. The original IX ran alongside Linux and required a dedicated physical network interface card (NIC) for each multi-threaded IX application, as well as one for Linux. In IX++, each multi-threaded application is instead given its own virtual NIC through SR-IOV, and Linux can run concurrently on the physical NIC. In brief, the original IX required a dedicated NIC for Linux and one NIC per IX application while IX++ uses NIC virtualization to allow Linux and multiple IX applications to multiplex a single physical NIC. In this paper we describe the implementation of IX++ and demonstrate its significant performance advantages over Linux when running concurrent networked applications. We also show that using different Linux Traffic Control Queuing Disciplines can help, but not close, this performance advantage.

2 Introduction and background

IX is an experimental operating system designed for high networking performance being developed at Stanford University [1]. IX was designed to achieve high I/O performance without sacrificing the protection of a traditional operating system. IX uses hardware virtualization to separate management and scheduling functions of the kernel (control plane) from network processing (dataplane). IX does not expose the traditional POSIX networking API to applications. Instead, it has a custom API that gives applications closer, but not raw, access to the NIC.

Currently, IX requires a dedicated physical network interface controller (NIC) for each multi-threaded application it runs, as it uses a modified NIC driver which exploits low level hardware features. We extended IX to enable multiplexing of a single NIC between multiple applications through the use of PCI Single Root I/O Virtualization (SR-IOV) which allows a single NIC to appear as multiple virtual NICs [2].

We demonstrate that with IX++ multiple IX applications could share a single physical NIC concurrently and with Linux applications. We also benchmarked the performance of running multiple IX applications simultaneously against running those applications in a standard Linux environment.

3 Implementation

IX++ implementation consists of two main parts: First, we instantiate and configure PCIe Virtual Functions (VFs) using Linux system commands and sysfs interface. Second, we modify IX to work with these Virtual Functions which appear as physical devices with limited capabilities.

3.1 Virtual Function Instantiation and Configuration

Below are the steps we used to instantiate and configure Virtual Functions. Unlike original IX, we do not remove the Linux NIC driver (IXGBE) which drives the Physical Function. This allows the Linux host to retain concurrent network access through the NIC.

1. Set kernel parameter "pci=assign-busses" to ask the kernel to always assign PCI bus numbers regardless of firmware/BIOS initialization.
2. Instantiate VFs by writing the number of VFs desired into "/sys/devices/<pci address of physical function>/sriov_numvfs".
3. Configure VF's MAC address using "ip link set" command. For example "ip link set p1p1 vf 0 mac e6:14:c6:39:a5:d7".
4. Turn off VF's MAC spoof check using "ip link set" command. For example "ip link set p1p1 vf 0 spoofchk off".
5. Use "lspci" to identify VF's PCI address and pass it to IX. For example "./ix --dev 0000:02:10.0 --cpu 0 -- /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 ../apps/memcached-1.4.18-ix/memcached -u nobody -m 8192".

3.2 SR-IOV Integration

Below is a summary of modifications we implemented for IX to work with Virtual Functions.

1. Disable PF specific functionalities such as EEPROM validation, redirection table initialization and disabling promiscuous mode.
2. Query initialization parameters such as MAC address and queue count from the PF using mailbox registers.
3. Merge changes from Data Plane Development Kit (DPDK) [3] such as Rx/Tx queue initialization.

4 Limitations of IX++

Our implementation of IX++ is experimental and currently severely limited because each IX application can only use a single CPU core. This limitation is in our implementation, and is something that is definitely theoretically possible in the future. The main problem stems from the incomplete integration of SR-IOV with the Intel 82599 NIC chipset IX is currently designed to run on. The original IX used receive side scaling (RSS) on the NIC to direct network flows to multiple queues that different CPU cores could access independently. However the 82599 has only one RSS table and doesn't take into account if different hardware queues are assigned to different virtual NICs [4]. Thus if RSS is used, it would mix packets between virtual NICs, making RSS essentially unusable. In the newer generation

Intel XL710 NICs there is a separate RSS table per VF which solves this problem, but we did not have access to those NICs and they would require a significant programing effort for IX to support.

5 Benchmark

In additional to using existing benchmark tools such as Mutilate [5], we've implemented a TCP upload benchmarking tool in both Linux and IX for throughput comparison and multi-application interference characterization. Our TCP upload benchmark tool consists of the following components.

Server

- Linux and IX application
- Sends a stream of bytes to each client connection as fast as TCP allows

Client

- Linux application
- Download bytes from server as fast as TCP can deliver
- Multiple concurrent download streams using Pthread
- Reports Rx throughput on key press or process signal.
  The Rx throughput is calculated by dividing the number of bytes received on the VF interface (from /proc/net/dev) by the elapsed time.

6 Evaluation Setup

Figure 1 shows the high level space of our evaluation. We benchmark baselines of running single applications on IX and Linux and compare these results to each other as well as to running multiple applications concurrently on both operating systems.

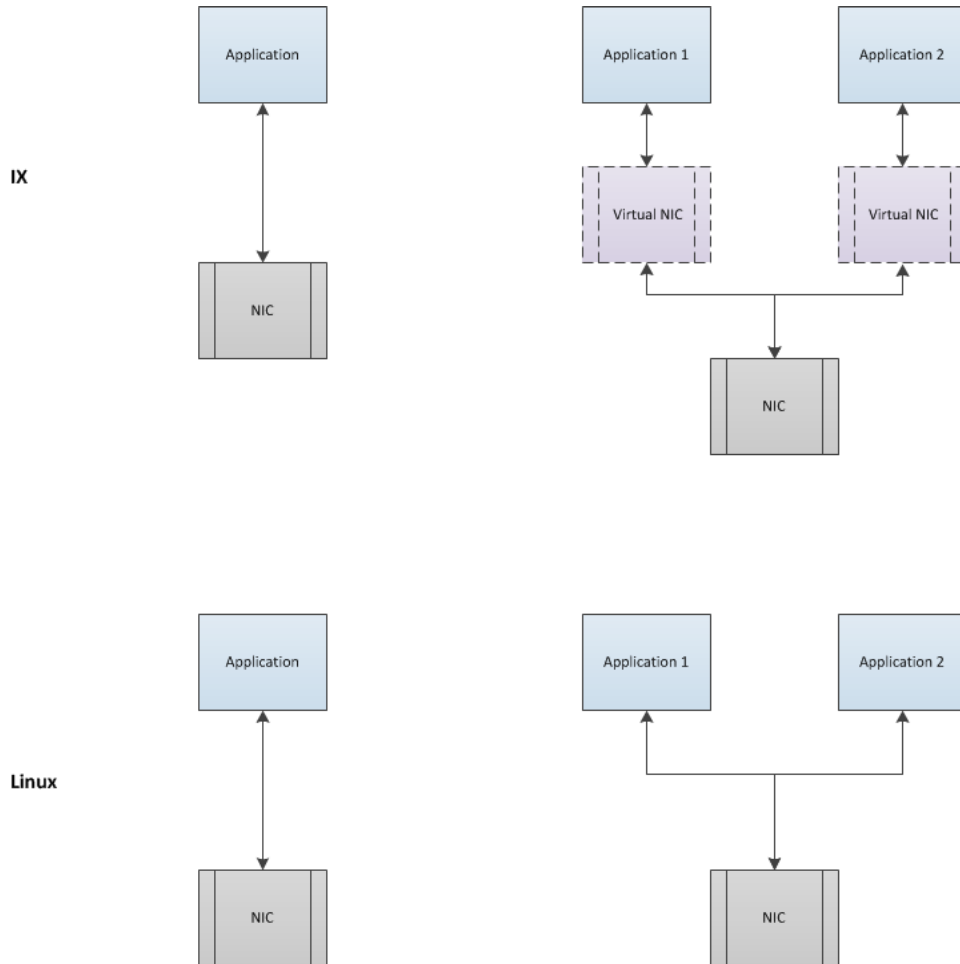


Figure 1 – Our evaluation setup for both Linux and IX

The servers used for our primary IX/Linux benchmark as well as the receiver for the TCP upload app have two 6 core, 12 thread, Intel Xeon E5-2630 (2.30GHz) CPUs, 64 GB RAM, and Intel 82599ES 10 Gigabit Ethernet. The Memcached master client server had the same specs except for a Solarflare SFC9020 10 Gigabit NIC instead of the Intel one. Load generation for Memcached came from 26 servers with a 4 cores, 8 threads, Intel Xeon E5335 (2.00GHz) CPU, 16 GB RAM, and Broadcom BCM5721 Gigabit Ethernet. All machines are running Ubuntu LTS 14.04 with the 3.13.0 Linux kernel.

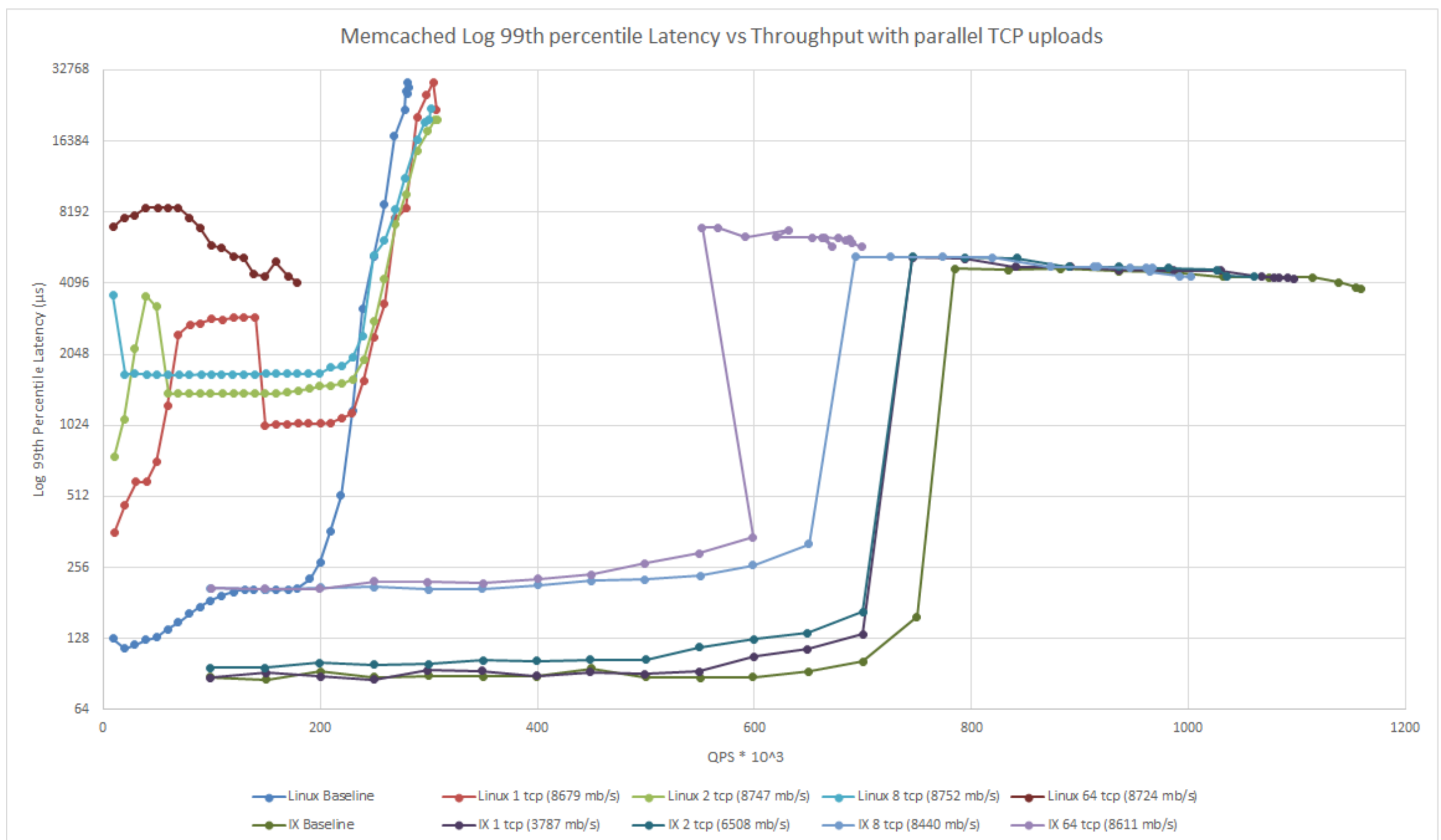
Memcached [6] benchmarks were taken using the Mutilate benchmark [5]. The master Mutilate client coordinated 26 Mutilate agents to generate target QPS loads while taking 1000 latency samples per second. These samples are where the 99th percentile latency numbers are taken from. Samples

lasted 3 seconds per QPS target. All mutilate clients opened 4 connections to the Memcached server and pipelined requests with a depth of 4 (-c 4 –d 4 flags).

7 Results

7.1 Performance of Memcached

First we look at the performance of running Memcached with a varying number of parallel TCP streams on both Linux and IX++. The 'baseline' measurement is Memcached running alone. The average throughput of the concurrent TCP uploads is included in the legend. Note that the TCP upload application for IX achieves a lower throughput for the TCP Upload app (significantly lower for 1 and 2



streams). The TCP Upload performance is directly graphed in the following section.

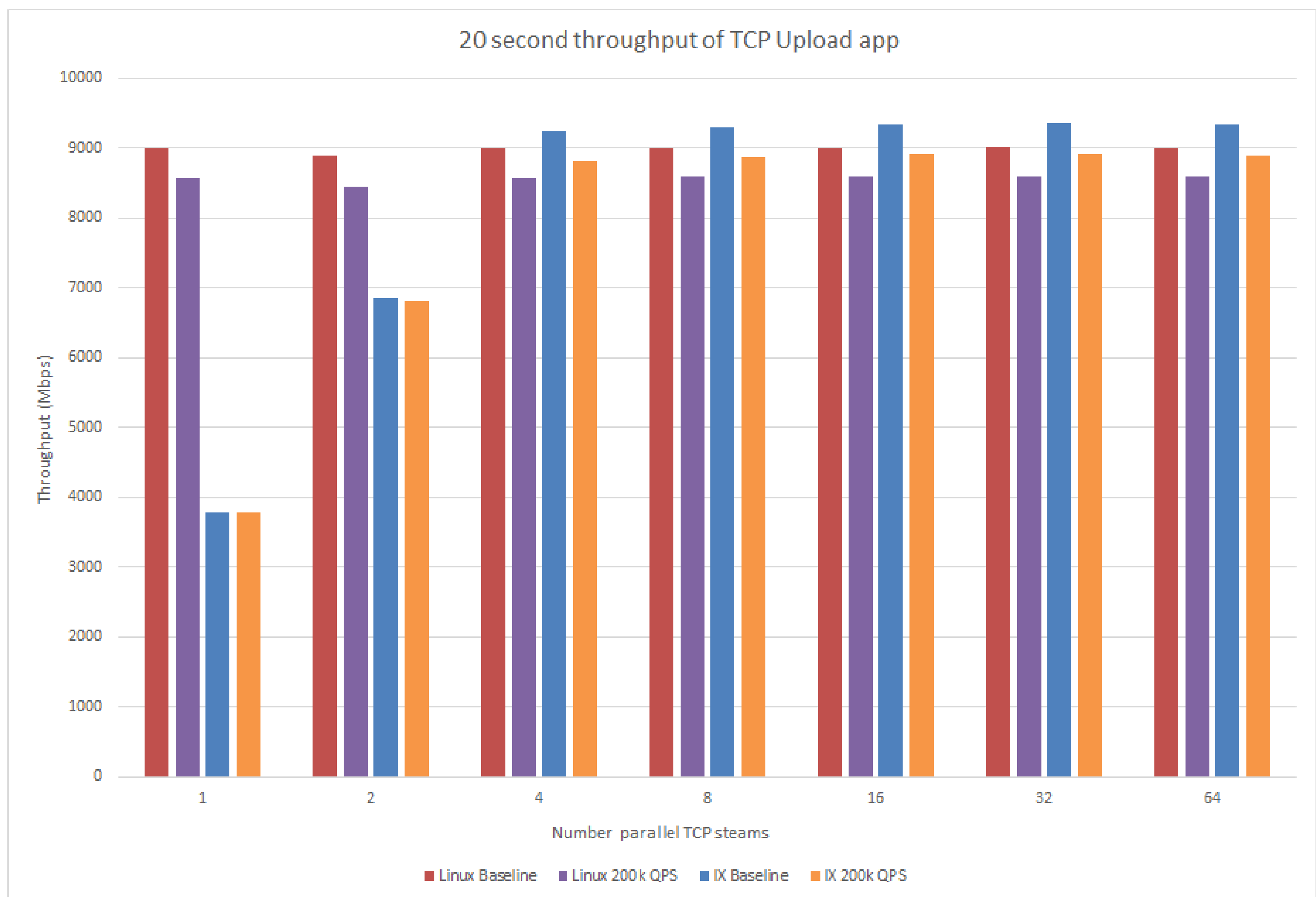
Comparing baselines recreates the work done in the original IX paper, though this time using only one core for IX and one thread to process incoming requests for Linux Memcached (-t 1 flag). IX is capable of a 4x improvement of request throughput over Linux with lower 99th percentile latency. When running the TCP upload application alongside Memcached, the performance advantages of IX++ are even more pronounced. The Linux latency is more than an order of magnitude worse than that of

IX++. The latency of running Memcached on IX++ at worst is around double that of the baseline. In contrast, a single core writing to 64 TCP sockets on Linux causes a concurrently running Memcached to suffer an over 20x latency increase.

Traditionally, cloud operators will refuse to schedule any applications alongside latency critical applications [7]. This first result showing that IX++ can reduce interference latency overhead from 20x to 2x for a low latency application shows that this practice is not fundamentally required. Further research into reducing interference could allow datacenters to achieve higher resource utilization by running more concurrent workloads.

7.2 Performance of TCP Upload app

Next, we looked at the average throughput of running the TCP Upload app on Linux and IX++. We compare both the Upload app by itself, as well as running alongside Memcached serving 200k queries per second (close to the maximum throughput for Linux Memcached). The TCP connections are opened and run for 20 seconds and the average network throughput over this period is reported.

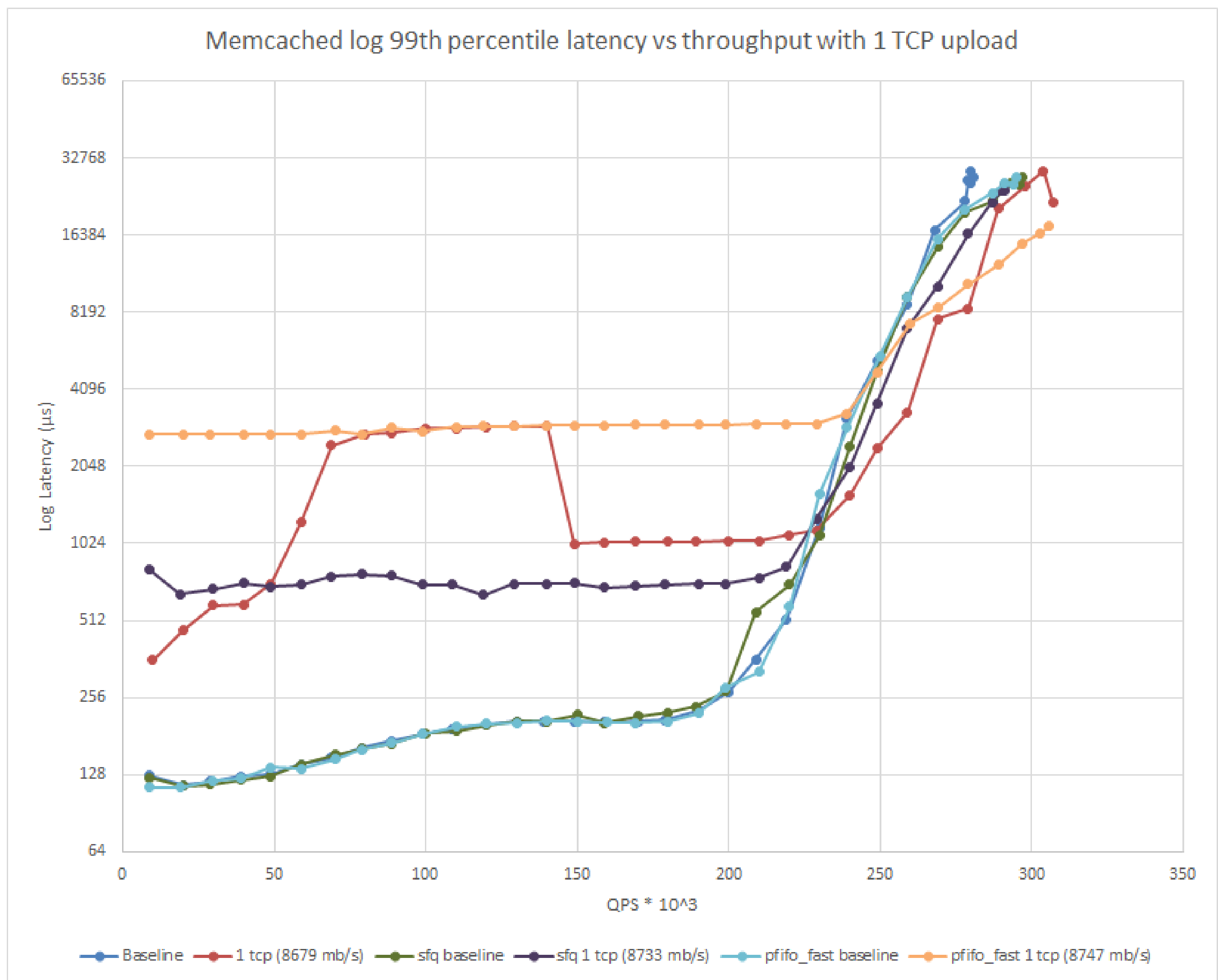


As noted earlier, IX has greatly reduced performance for 1 and 2 TCP streams. After consulting with Adam Belay, one of the original authors of IX, we believe the problem could be a bug in how TCP

acknowledgement is processed within the lwIP TCP/IP stack [8]. We look forward to tracking down and fixing this bug in future work.

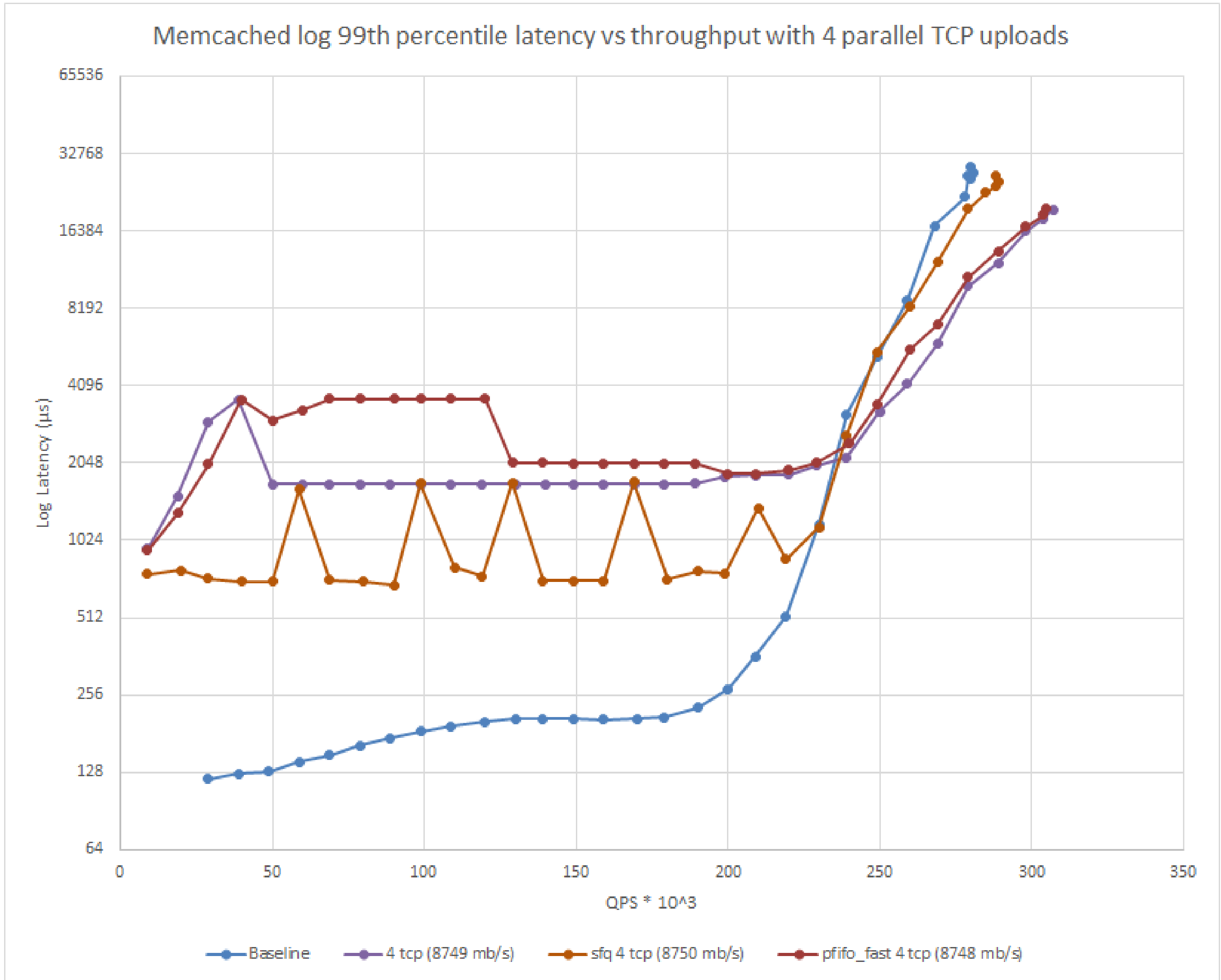
7.3 Linux Traffic Control Queueing Disciplines

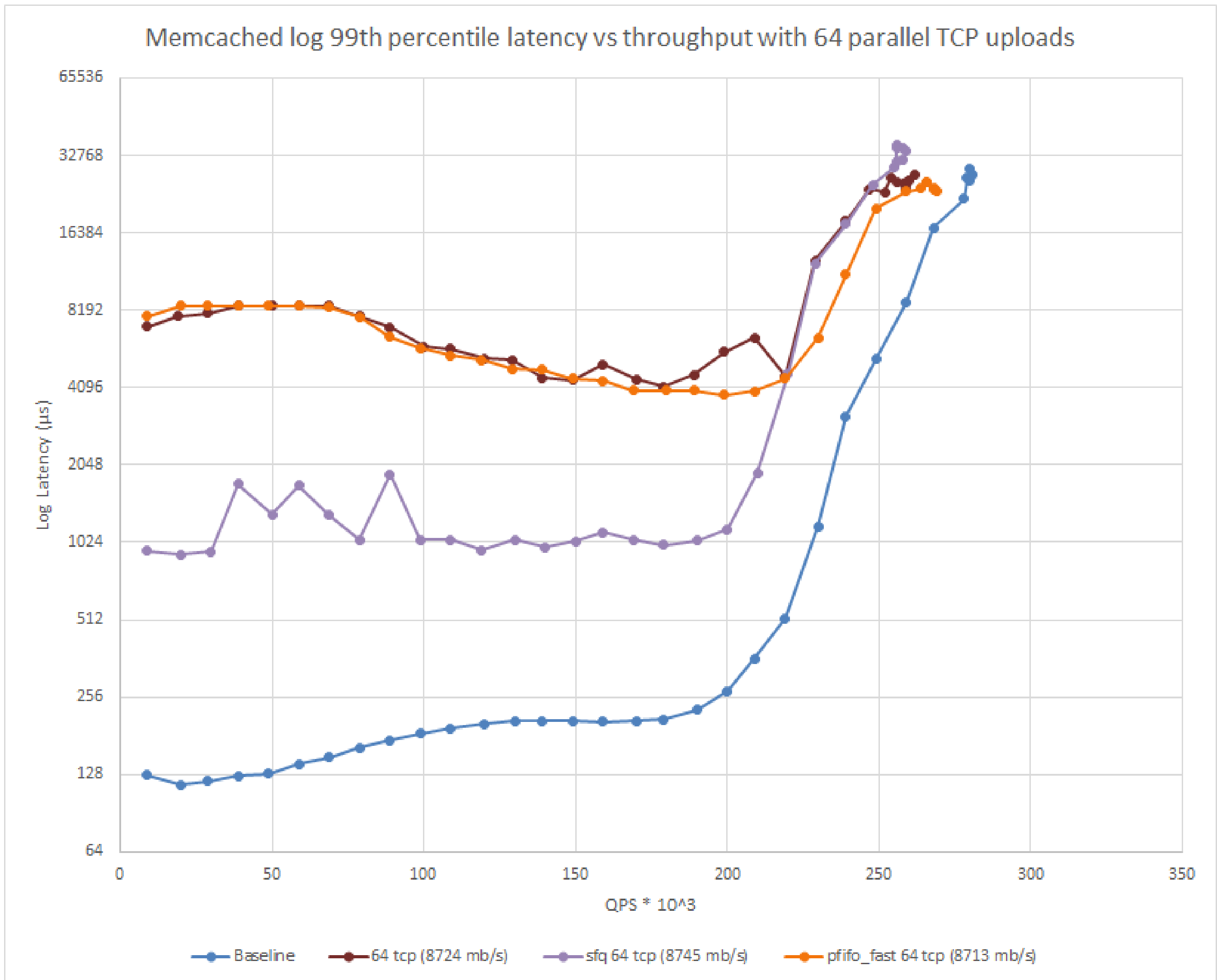
Lastly, to explore if we could reduce the interference from parallel TCP uploads on Memcached with Linux, we experimented with different queuing disciplines in Linux's TC utility, which can configure how the kernel shapes and schedules network traffic. In the figures below, we compare running the standard Multi-Queue network queueing discipline with both Stochastic Fairness Queueing and pfifo_fast. We chart the Memcached 99th percentile latency vs throughput for 1, 4, and 64 parallel TCP uploads in the following three figures. The average throughput of the TCP uploads is included in the legend.



In this chart, we see that all three queuing disciplines perform equally well for Memcached when there is no other traffic running. Running Memcached concurrent with the TCP upload

benchmark, we see that Stochastic Fairness Queueing achieves a lower latency than the other two queuing disciplines with the same throughput capability while pfifo_fast has even higher latency than the default. The following two figures show the same comparison as above but with 4 and 64 parallel TCP upload streams. In both we observe that Stochastic Fairness Queueing performs the best while pfifo_fast performs worse than or as poorly as the default Linux multi-queue.





Stochastic Fairness Queueing performs the best while pfifo_fast performs worse than or as poorly as the default Linux multi-queue. Future work could be done to look at the performance of other queuing disciplines, including classful ones (which classify and treat traffic differently depending on filters).

8 Related Work

IX++ is an extension of the IX Dataplane Operating System which in term leveraged DPDK, lwIP TCP/IP stack [8] and the Dune library [9]. Please refer to the related work section in the IX paper for more context on the operating system techniques. IX was developed at the same time as the Arakis operating system from the University of Washington [10]. Arakis shares IX's separation of the dataplane and control plane as a high level design principle, but the implementations differed significantly. Like IX++, Arakis also uses SR-IOV to give applications more direct access to NIC hardware.

9 Conclusion

IX++ closed a major limitation in IX: that it required multiple physical NICs to run multiple applications. Our solution used Single Root I/O Virtualization to multiplex the NIC at the hardware level, minimizing overall overhead as well as network interference between applications. IX++ significantly reduces interference between applications compared to Linux and in some cases yields an order of magnitude reduction to interference overheads.

10 References

[1] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, 2014.

[2] PCI-SIG, "Single Root I/O Virtualization and Sharing 1.1 Specification," January 2010. [Online]. Available: Specification https://www.pcisig.com/members/downloads/specifications/iov/sr-iov1_1_20Jan10_cb.pdf.

[3] "DPDK: Data Plane Development Kit," [Online]. Available: http://dpdk.org.

[4] Intel, "Intel® 82599 10 Gigabit Ethernet Controller: Datasheet," [Online]. Available: http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html.

[5] J. Leverich, "Mutilate: High-Performance Memcached Load Generator," [Online]. Available: https://github.com/leverich/mutilate.

[6] "memcached - a distributed memory object caching system," [Online]. Available: http://memcached.org.

[7] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale".

[8] A. Dunkels, "Design and Implementation of the lwIP," [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.1795&rep=rep1&type=pdf.

[9] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières and C. Kozyrakis, "Dune: Safe User-level Access to Privileged CPU Features," in *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI '12)*.

[10] S. Peter, J. Li, I. Zhang, D. Ports, D. Woos, A. Krishnamurthy, T. Anderson and T. Roscoe, "Arrakis:

The Operating System is the Control Plane," in *11th USENIX Symposium on Operating Systems Design and Implementation*.