Universiteit Leiden

Master Computer Science

The Performance of Distributed Applications:
A Traffic Shaping Perspective

Name:	Jasper A. Hasenoot
Student ID:	s2619369
Date:	August 4, 2022
Specialisation:	Advanced Computing and Systems
1st supervisor:	Dr. A. Uta
2nd supervisor:	Dr. K.F.D. Rietveld

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Acknowledgements

Before you lies the result of six months of hard work: my Master's Thesis. While it is technically "mine", and I am proud of this achievement, I would like dedicate this part to thank some people for their insights and support during this journey.

First and foremost my supervisors: Alex Uta, who provided me with guidance for the general direction and layout of the thesis and who was a real asset by connecting me with the people and resources needed for the project, and Kristian Rietveld, who assisted me in setting up an FPGA, providing me access to the right tools and assisting me in solving issues related to programming it. An honourable mention here for Nele Mentens who also played a part in this.

Secondly, Robert Ricci, Mike Hibler and Leigh Stoller from CloudLab, whom Alex got me in touch with, for providing me access to the CloudLab environment to be able to execute my experiments. Not to mention the quick responses and fixes you provided once issues occurred.

Thirdly, Chris Neely from Xilinx, for providing the – as of that time – not yet released altered DPDK version to be used with the AU250 FPGA. Without you there would not have been *any* chance to experiment with FPGA-assisted network processing.

Finally, my friends, family and colleagues, for keeping me sane and providing some well needed distractions during this journey. However fun a topic may be, sometimes you just need a change of scenery and some fun to give you the energy to get back to work.

All of you, thank you so, so very much. Now all that remains is the *pièce de résistance* itself: the thesis. I hope you enjoy it.

Jasper

Abstract

Traffic shaping is a performance influencing factor which is often overlooked when benchmarking distributed applications, even though it is widely used in real-world cloud environments. While in theory it should allow for a fairer division of network resources, in practice it may also exacerbate or introduce new problems, which may influence both performance and performance consistency. By benchmarking a Key-Value Store, Big Data workloads and High Performance Computing tasks under the influence of token buckets and priority queues – present on combinations of an on-host vSwitch and a physical network switch – some key insights are obtained. In particular, the on-switch token bucket often caused high tail latencies and a general reduction in performance consistency, though applications sending relatively small packets saw the opposite results when compared to a baseline without traffic shaping. Additionally, compared to this baseline, priority queues generally increased performance consistency, though the combination with a token bucket sometimes exacerbated the negative influence of the token bucket. As these results show, the impact of traffic shaping needs to be taken into account when benchmarking distributed applications. Therefore, several practical implications and recommendations are provided to aid in repeatability and performance optimization.

Table of Contents

1 Introduction

Modern cloud infrastructure relies on resource sharing between tenants in order to maximize utilization of available hardware resources, such as servers or networks, while ensuring a minimum Service Level Agreement (SLA) is met. This allows for the maximization of profits at minimal cost. However, while a minimum SLA would be guaranteed, the consistency of the performance of the hardware resources is not [1, 2], as this may fluctuate depending on the resource usage of other tenants. Cloud providers can generally consistently provide RAM [2] and – to a lesser degree – (v)CPUs [3, 2], however, the available network bandwidth is only shown as a "maximum available" amount [4, 5], which is not guaranteed [5, 6, 2]. While it is theoretically possible to provide a minimum bandwidth guarantee, in practice this would impact over-subscription [7] and is likely therefore not specified by the cloud providers.

Variance in network performance due to multi-user resource sharing across servers and networks [2], combined with over-subscription on network links [8], can impact the performance of applications relying on the consistency of these resources [9]. This, in turn, will impact any performance evaluations or experiments, which may lead to unfair, unrepeatable or incorrect results. Evaluation of distributed applications *without* regard for the impact of variable network performance therefore may not be representative of the performance in a real-world cloud environment with shared resources. Since not all distributed applications would be impacted to the same degree – due to lower reliance on either latency, bandwidth or both [10] – providing the characteristics of distributed applications in this regard should prove valuable. Variance in network performance impacts not only distributed applications running in real-world contexts, but also research oriented contexts. The latter relies on the consistency of the network in order to be able to consistently measure other variables. Next to this, variance in network performance severely hampers reproducibility of experiments, which would then rely not only on the hardware used, but also on the host and network utilization profiles at the time of the experiment [11, 12, 13, 14].

In order to ensure a fair sharing of network resources between tenants, and thus ensuring a Quality of Service (QoS), cloud providers utilize traffic shaping techniques like priority queues and token buckets. The former allows for prioritizing certain network flows over others, causing an imbalance between flows of differing priorities. This could be used to ensure that latency-sensitive traffic flows (e.g. Voice over IP; VoIP) get processed with a lower latency, for example [15]. The introduction or removal of such flows may therefore cause sudden changes in network performance – provided the network is nearing congestion – when using a traffic flow with a lower priority. The latter allows for limiting the available bandwidth to each host, by limiting the amount of data able to be sent in a certain time frame. The token bucket "refills" at a preset rate, up to a maximum buffer [16, 17]. This allows for short bursts of high bandwidth traffic – which appear uncapped – provided there is little network usage in between, allowing for the bucket to refill. Since applications generally transmit traffic in bursts, this feature allows for a more consistent packet latency [10]. When a continuous flow of high bandwidth traffic is present, however, the token bucket will deplete its buffer and then continue to transmit at the refill rate. This being a fraction of the maximum available bandwidth achievable during the short (initial) bursts causes sudden drops in network performance. Combining the two techniques results in a situation in which the available network bandwidth and latency at any given point in time can only be determined through benchmarking, as the influence of other tenants and the unknown parameters of priority queues and token buckets (configured by the cloud provider) cause unpredictable behaviour. Earlier research has shown that the performance of distributed applications handling Big Data suffers with variance in network performance, which occurs despite – or even because of – deployment of QoS traffic shaping techniques [12].

In order to create an understanding of the impact of traffic shaping on distributed applications in a broader sense, some experiments will be executed. These require a setup similar to those found in cloud environments, with physical and virtualized resources (e.g. switches), as well as commonly used traffic shaping techniques, which were previously mentioned. The addition of a suite of distributed applications covering multiple domains – Key-Value Store, Big Data and High Performance Computing (HPC) – should allow for real-world representative results to be obtained.

The traffic shaping techniques can be enforced using either dedicated network equipment, such as a network switch, or on-host using a virtual appliance like a vSwitch. This should allow for a more granular

control over network flows [18]. In the latter case this can be achieved using e.g. a kernel-bypass network processor such as the Data Plane Development Kit (DPDK) [19]. In order to determine to what degree network traffic shaping techniques used in modern data centers, when applied using a switch and/or DPDK, affect the performance of a selection of distributed systems, the CloudLab [20] cloud-building infrastructure is used. This allows for creation of dedicated links between nodes as well as user-managed switches to apply the QoS traffic shaping measures. The links are created using a L1 switch such that no unintended interference from other tenants occurs.

The main goal of this thesis is to answer the following research question:
What is the effect of traffic shaping on distributed applications?
In addition to this, three supporting sub-questions have been formulated, which will be answered in the following Sections:

- What traffic shaping methods are used in modern data centers? (Sections 2 and 3)
- How can the impact of traffic shaping on distributed applications be quantified? (Section 4)
- What are the lessons learned from benchmarking distributed applications under the influence of traffic shaping? (Sections 5 and 6)

By answering these research questions, a better understanding of the impact of traffic shaping on distributed applications can be gained. In particular, this thesis presents the following key contributions. First, it provides novel insights into the effect of traffic shaping on distributed applications spanning different domains, for example key-value stores suffering from high tail latencies. Second, it gives recommendations for the benchmarking of distributed applications in the context of traffic shaping, as well as determining the optimal environment when deploying them. This covers experiment implications for research, and touches upon real-world use. Third, an application-independent benchmarking setup framework is provided – tailored for CloudLab but usable in a broader context – which allows for benchmarking distributed applications' performance while affected by on-host and/or on-switch traffic shaping. Fourth, it shows preliminary results using OVS-DPDK with FPGA network processing to possibly alleviate CPU usage, included in Appendix A

2 Background

In order to establish a shared baseline knowledge of terminology and technologies used in this thesis, in this chapter a brief description will be given for each. First Software Defined Networking (SDN) and its related technologies will be introduced, after which a more detailed description of Quality of Service (QoS) and traffic shaping techniques will be provided. Finally, a general classification of distributed applications will be given, as well as a more detailed description with regard to the distributed applications that will be used in the experiments.

2.1 Software Defined Networking

Software Defined Networking (SDN) [21] differs from traditional network processing in that it allows for separating the data plane, where the processing of network traffic occurs, from the control plane, which is responsible for the configuration of the data plane. This decoupling allows for centralized control of network flows, thus facilitating rapid deployment. While traditional network processing is done through hardware, e.g. routers and switches, the switches operating in the data plane of SDN can be either hardware-based or software-based, i.e. Virtual Network Functions (VNF). One such software-based switch is Open vSwitch [22]. The hardware-based switches differ from a traditional router or switch in that they do not require a local control plane. The software-based switches share this attribute and additionally may be virtualized instead of running bare-metal, and will work on commodity hardware.

Both hardware-based and software-based switches can be controlled from the control plane, which is centralized, though it may be virtualized and redundant or replicated. Through a protocol such as OpenFlow [23], the network switches can be programmed with flows. These flows describe (prioritized) rules the switches enforce based on switch-specific metadata (e.g. source or destination ports) as well as fields in the packet headers. Amongst this are source and destination IP addresses, port numbers, TCP headers, source and destination MAC addresses and VLAN ID [24]. Additionally, packet headers may be modified [25]. This is especially useful in the case of VLAN – to separate networks / broadcast domains – and Multi Protocol Label Switching (MPLS) in which packets are forwarded based on a label assigned at the network edge.

If MPLS is used throughout the network, the processing cost of traffic is lower, thus increasing capacity or allowing for the use of lower-cost hardware. In a similar fashion, by removing the local control plane found in traditional network equipment, the hardware requirements are lower, thus allowing for cheaper equipment to be used, lowering the cost-per-port. Next to this, the capability to run the software-based SDN switches in a virtualized environment is cost effective, though it may be detrimental to performance if multiple such VNFs are present on the same host, which may occur in cloud deployments [26].

2.1.1 Data Plane Development Kit

The Data Plane Development Kit (DPDK) [19] takes SDN a step further by allowing the processing of network packets to bypass the kernel entirely. It does this by pinning CPU cores that actively poll for packets to be sent, reducing latency due to e.g. scheduling. This has the upside of possibly greatly increasing efficiency, though it does come at the cost of reduced configurability. One solution to this is the integration of DPDK with Open vSwitch (OVS-DPDK) [22], which also supports communication with a data plane through OpenFlow [23] in order to modify flows at runtime.

2.2 Quality of Service & Traffic Shaping

Quality of Service (QoS) is a concept introduced to allow for distinguishing between applications based on their "communication requirements" [15, p. 6]. This allows for treating the communication of these applications over computer networks (i.e. flows) differently. This ensures each application receives flows with the appropriate characteristics. Flows have one source and one or more destinations, in the case of unicast and multicast traffic respectively.

QoS in the context of computer networks describes the quality of the connections or flows in the network. This can be determined by measuring certain characteristics, such as bandwidth, ping or jitter [15]. Since

not all applications and traffic flows have equal or as strict requirements regarding these characteristics, flows will be assigned a priority in an attempt to meet the requirements as close as possible. Several gradations in priority exist, as well as traffic shaping techniques to enforce them, which will be defined in more detail in Section 2.2.1.

Apart from a division of priority, traffic shaping also encapsulates the strict limiting of flows on bandwidth. In the context of public cloud infrastructure it would be useful to limit the maximum bandwidth usage of tenants, which increases fairness and allows for the creation of different bandwidth tiers [4, 5]. Such a limit is not automatically enforced when different flow priorities are used, as these still allow for the full utilization of available bandwidth for lower priorities, as long as no higher priority flows are present. In practice, the division of bandwidth between different traffic flows is done through the use of priority queues, while the enforcement of maximum bandwidth caps is done through token buckets [27, 28]. Both of which will be discussed in Sections 2.2.1 and 2.2.2 respectively.

Traffic shaping requires more resources with increased traffic and traffic types, which would lead to a greater number of QoS classes and queues. Traditional switches generally have a pre-determined number of both, while the latter may be variable in software-bases switches. In the case of on-host vSwitch use, increasing the amount of traffic and number of queues would require more CPU cycles for processing, as well as a greater amount of memory to keep track of the queues [29].

2.2.1 Priority Queue

Dividing available bandwidth between flows of differing priority is done through the use of queues. These queues broadly fall into two distinct classes, which may be used in conjunction: strict queues and (weighted) fair queues [30]. Both classes of queues will fill up the maximum available bandwidth if enough traffic is present. No hard-limit bandwidth cap is enforced.

A strict priority queue always allows the flow of the highest priority to use all available bandwidth [30]. Once this highest priority flow no longer has traffic to send, the second highest priority flow is allowed to send traffic. This continues for all gradations of priorities that may be specified. Note that this means that a high priority flow will drown out lower priority flows if it is over-used. Therefore, high priority in strict priority queues should be used sparingly, for example in the case of low-bandwidth traffic requiring low latency or jitter such as VoIP.

Fair queues on the other hand allow for sharing bandwidth between all flows equally. Next to this, they allow for the use of priorities, which are used in weighting the share of bandwidth per flow [31]. Such a weighted fair queue shapes the traffic on the different priority flows to divide the available bandwidth amongst currently transmitting flows. It allows higher priority flows to send more data than lower priority flows, but the share of bandwidth for a flow at a certain point in time is not set. If flows start or stop transmitting traffic, the share of bandwidth between them is adjusted accordingly [31]. This prevents high priority flows to completely stop lower priority queues from sending traffic, and allows for the maximal utilization of the available bandwidth. This is a desirable trait in cloud environments.

The priority for the available weights used in the fair queues are defined in the DiffServ (DS) field of the IP headers of the respective packets using a Differentiated Services Code Point (DSCP) [32], which is used to communicate QoS preferences between devices. This field allows for four different tiers in priority, as well as three tiers in "drop probability" (low, medium and high), according to which packets are dropped once the maximum available bandwidth would be exceeded [33].

2.2.2 Token Bucket

Strict bandwidth rate limiting of flows is achieved through the use of token buckets, which are defined by the Committed Burst Size (CBS) representing a maximum bucket size as well as the Committed Information Rate (CIR), which is the rate at which the bucket refills [16, 17]. Both parameters represent "tokens" and are defined as a number of (e.g. Mega/Giga)bits or packets within this context [16, 17]. The defined refill rate is added to the current bucket size at a set regular time interval, "refilling" the bucket.

Once the bucket is full – i.e. the current size equals the maximum size – any subsequent refilling will be discarded.

Traffic transmitted on a flow will not exceed the specified CIR on average [16, 17]. The filling of the bucket when no traffic is generated creates a buffer which allows flows to temporarily exceed the maximum allowed bandwidth. This is because transmitted traffic will cause the current size of the token bucket to be decreased proportionally to the packet sizes that are transmitted [16, 17]. If the rate of traffic exceeds the CIR, the built-up tokens in the bucket will first be depleted before the traffic will be shaped down to the CIR [16]. The size of the token bucket therefore determines the maximum magnitude or duration of allowed burst traffic the flow allows. If not enough tenants transmit at a high enough rate, the available bandwidth will not be fully utilized, since outside of the burst window the CIR determines the maximal transmission speed.

This token bucket concept can be extended by adding a second token bucket with a separate refill rate. The size of this bucket is defined by the Excessive Burst Size (EBS), with the respective refill rate being the Peak Information Rate (PIR) [17]. This second bucket allows for more granular shaping of network traffic. While the same principle regarding filling and depleting buckets applies, flows with a transmission rate exceeding the CIR but not exceeding the PIR will still be allowed to transmit, albeit with a higher chance for packets to be dropped, as marked using the DSCP values [17]. With a sufficiently high PIR, this allows single flows to fully utilize all available bandwidth on a link if no flows from other tenants are sending any traffic. Once flows from other tenants appear, the available bandwidth will be shaped down accordingly, with packets exceeding the CIR being dropped if necessary. The CIR remains as the minimal bandwidth that is guaranteed [17].

3 Related Work

While this work focusses on the effects of QoS traffic shaping on the performance of distributed applications, previous work has touched upon (1) the effect of other changes in network conditions on distributed applications, (2) ways to facilitate (on-host) congestion control in data center networks at minimal impact to tenant processes on those hosts, (3) repeatability of experiments in networked environments, (4) performance guarantees, and (5) the Data Plane Development Kit.

(1) Effect of Network Conditions on Distributed Applications Liechti et al. [34] present the THUNDERSTORM tool, which supports evaluating the performance and behaviour of distributed applications when exposed to sudden changes in network latency or bandwidth, in particular those incurred due to changes in WAN topologies. It uses traffic shaping mechanisms in the Linux kernel to emulate the network dynamics encountered in real-world scenarios.

Shea et al. [1] investigate the performance impact of virtualization on network performance, in particular when caused by CPU resource contention between tenants' processes and network queue processing in the hypervisor. The network buffer between the hypervisor and VMs may overflow due to scheduling between the two, which is shown to impact both bandwidth and latency.

(2) (On-Host) Congestion Control in Data center Networks He et al. [35] present AC/DC TCP, which allows for on-host, in-vSwitch congestion control and fairness between traffic flows of unmodified virtual machines, and reconfigurability with regard to congestion control algorithms and bandwidth allocation schemes. It provides low latency and high bandwidth traffic without incurring a corresponding high CPU cost to the host. This is achieved through the use of the TCP congestion control mechanism, though this does cause it to be ineffective on UDP or encrypted traffic.

Saeed et al. [29] created Carousel, which is able to shape both data center and WAN traffic flows on-host and is capable of scaling to thousands of flows and policies at minimal cost to CPU and memory. To do this, it utilizes a single time-based queue for each pinned CPU core, on which packets are enqueued based on their priority and allowed bandwidth.

Yang et al. [36] extend the QoS traffic shaping mechanism of the on-host vSwitch (OVS-DPDK) to ensure fair bandwidth sharing based on the number of CPU cycles needed to process traffic. This allows for fairness with varying packet sizes as well as with CPU contention, improving throughput in file-transfer workloads and reducing latency in web-serving workloads.

Jang et al. [10] introduce Silo, which attempts to guarantee latency in data center networks through VM placement and on-host policing, which allows for bandwidth, delay and burst allowance guarantees – three requirements for guaranteeing latency. Its design is queue-free, which makes it able to support many concurrent flows. On top of that, it shows the sensitivity to bandwidth, delay and latency of several applications.

(3) Repeating Experiments in Networked Environments Bulej et al. [13] present the procedure of Duet Benchmarking, in which relative performance is measured in parallel experiments – subjecting them to the same external influence – in order to increase accuracy. This should prevent the noise of performance variability due to resource contention or virtualization to reflect in the comparison, as simultaneous experiments are affected similarly, thus allowing the noise to be filtered out.

Abedi et al. [14] demonstrate that fair comparisons in cloud environments may be made, despite changing performance conditions, through the use of Randomized Multiple Interleaved Trials. Randomly alternating experiments should even out the external noise present through resource contention or hardware differences across experiments, allowing for a more accurate comparison of experiments.

Uta et al. [12] evaluate the reproducibility of experiments regarding big-data workloads in public cloud environments. In particular, it is mentioned that the QoS and fairness solutions used by the cloud provider may not achieve consistent performance and may even cause an increase in variability. This may lead to inaccuracies in result interpretation when not enough measures are taken to account for this.

(4) Performance Guarantees Kogias et al. [37] present Lancet, a tool that measures (tail) latency – which may have a large impact on distributed application performance – given a user-specified workload and a desired confidence interval, by generating RPC messages & measuring the response time. It uses standard kernel drivers and supports hardware timestamping in-NIC to do this. To ensure statistical certainty, statistical tests are executed on the gathered data, which also allows for determining whether the desired confidence interval can be reached.

Guruprasad et al. [38] use a combination of network emulation & simulation to accurately represent a network which may use real traffic loads in a repeatable manner. It supports scaling to larger (distributed) topologies than single emulation or simulation can, and additionally allows the use of real applications. To reduce the amount of required shared links between nodes of the same application, as well as reduce the congestion on those links, optimizations are done during mapping to physical hardware. This way, shared hardware may be used with automatic systems in place to minimize the amount of overloaded hardware, resulting in a guaranteed minimum performance over the network.

(5) Data Plane Development Kit Demoulin et al. [39] present Perséphone, which schedules packages based on expected processing time. This application-aware approach aims to reduce tail latency by reserving some pinned CPU cores used in packet processing for short requests. This way, requests that require more processing time do not block the head-of-line for the short requests. This approach does require the implementation of a classifier in the source application, however. Perséphone increases utilization this way, reducing the number of machines required to serve set workloads.

Pitaev et al. [26] compare virtual network function (VNF) throughput when using multiple VNFs on the same host. Network packet processing is compared between OVS-DPDK, SR-IOV and VF.io VPP. SR-IOV achieves the highest performance, though it is not as configurable. OVS-DPDK and VF.io VPP perform comparably, though when heavier VNF functionalities are used, OVS-DPDK falls behind at a higher number of VNFs.

Compared to the work described above, this work differs in a few notable points. **(1)** The work described above looks at effects of WAN deployment, topology changes, and network virtualization (which may suffer from resource contention) on distributed application performance. This work *also* looks at the effect of network variance on distributed application performance, though in this case it is specifically the impact of QoS/traffic shaping techniques on distributed applications. **(2)** The work described above alters on-host traffic shaping and/or rate limiters to improve CPU and memory efficiency, and extends VM placement algorithms to allow for fair link utilization. This work does not alter placement mechanisms nor on-host shaping methods, though it does look at hierarchical QoS – traffic shaping split up between hosts and switches – to reduce possible impact on distributed applications due to local processing (by reducing CPU and/or memory usage). **(3)** The work described above looks at reproducibility of comparative experiments and big-data related experiments in networked environments, or deals with variance in CPU, Memory, Disk and Network performance in general. This work additionally acknowledges the effect on experiment reproducibility when applications are affected by resource contention. It looks at repeatability and performance variance caused by network contention by QoS/traffic shaping in particular. **(4)** The work described above looks at both measuring performance to evaluate guarantees, as well as setting up networked (experiment) environments that guarantee enough available bandwidth over the shared physical links. While this work measures performance impact due to influences from the network in the form of traffic shaping – which can be used to provide performance fairness or a degree of guarantees – it does not look into ways of providing guarantees to end-users. Rather, the importance of guarantees in the context of variance caused by the use of traffic shaping can be derived from the results. **(5)** The work described above regards the implementation of different packet scheduler methods using DPDK as well as a comparative performance evaluation of OVS-DPDK itself. In this work, DPDK is used in combination with OVS to facilitate traffic processing with and without traffic shaping. No emphasis is put on the specific performance of OVS-DPDK, nor are new functionalities added to it. However, in Appendix A some preliminary experiments are described where packet processing is offloaded to a dedicated FPGA.

4 Experiment & Design

In order to answer the main research question "What is the impact of network variation on distributed application performance?", experiments measuring the performance of applications spanning the Key-Value Store, Big Data and High Performance Computing domains were executed. For each of them, benchmarking suites were picked such that multiple facets of the system could be benchmarked, while the system would be subjected to different traffic shaping settings. This Section will – amongst others – elaborate further on this.
To execute these experiments, the CloudLab [20] platform was used, which allows for the reservation of nodes and creation of private (user-managed) networks. The number and type of nodes used, as well as the connecting network and other resources such as storage can be described in a user-shareable profile. Next to this, each node can be provided with a configurable disk image, which facilitates the automatic setup of experiments. The combination of the profile and disk images causes the reproducibility of experiments to be less complicated, as identical files may be used – provided they are still publicly available.

In the following sections, first the creation of the Docker overlay network will be described, after which the interference traffic generation and traffic shaping settings will be discussed. Then, the used distributed applications will be described. Finally, for the Key-Value Store, Big Data and HPC experiments the setup will be broken down in greater detail. All three experiments will be conducted using Docker containers backed by OVS-DPDK as this allows for the use of traffic shaping both on-host and on-switch, both with (network-)resource contention present. Next to this, in all experiment setups the xl170 nodes – whose specifications are shown in Table 1 – are used. These use Mellanox ConnectX-4 NICs, which are supported by DPDK [40]. They are connected to a user-managed Mellanox MSN2410-BB2F switch using 10Gb/s Ethernet links. These links are allocated via a NetScout 3903 L1 switch managed by CloudLab, essentially creating direct connections between the nodes and the switch.

Node: xl170	
CPU	10-core Intel E5-2640v4 (2.4GHz)
RAM	64GB ECC DDR4-2400 (4x 16GB)
Disk	Intel DC S3520 480 GB 6G SATA SSD
NIC	Two Dual-port Mellanox ConnectX-4 25 GB NIC (PCIe v3.0, 8 lanes)

Table 1: Specifications of the xl170 node [41] on CloudLab [20].

4.1 Docker Overlay Network

The basis of each experiment is the Docker overlay network, which is backed by OVS-DPDK on each host. This virtual network is needed because it lays on top of the physical user-managed network provided by CloudLab, and therefore "extends" the Docker network over multiple hosts. This allows for the application of traffic shaping both on-host (on a per-container level), and on-switch, i.e. on a per-host level. Docker and this overlay network are used as a representative environment for a generic system allowing for virtualization of compute and network over multiple hosts.
The Docker overlay network consists of a single Consul [42] cluster store instance, which is used for tracking the network status across nodes, as well as a Docker and OVS-DPDK instance on each node [43]. These instances communicate with Consul over a separate control network, as the ports connected to the user-managed network bypass kernel processing due to them being controlled by OVS-DPDK. Figure 1 shows this setup. This creates a setup similar to those found in Cloud environments, where there is a combination of an on-host (kernel-bypass) and a physical network, with on each node a setup that allows for the creation of virtualized environments [10, 44].

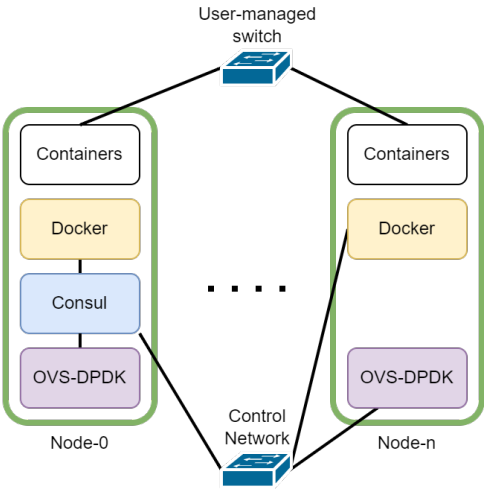

Figure 1: The Docker overlay network uses a single Consul instance as cluster store, with Docker and OVS-DPDK instances on each node communicating with it over a separate control network to track network status. Containers created in docker are added to the overlay network automatically.

When creating virtual networks or adding containers to Docker, all network-related events are passed to the OVS-DPDK instance through the use of an OVN Docker overlay driver [45] on each host. This ensures regular Docker commands can be used to manage the network. The driver translates the network events to OpenFlow, which is used to control OVS-DPDK. The overlay driver did require some modifications from its source, as it was neither compliant with current library versions nor Python3. Next to this, the driver was added to the system as a service such that no manual intervention would be needed on boot.

This setup allows containers to automatically receive IP addresses within the overlay network IP pool on creation, as well as communicate using the user-managed network. The only prerequisite is the creation of an OVS-DPDK backed network in Docker, and the specification of an external (named) network for each container in its `docker-compose` file. The setup including necessary changes will be included in Appendix B .

4.2 Interference Traffic

Since traffic in networked environments varies over time, the interference traffic used during the experiments should exhibit similar behaviour. Based on the traffic generated in the experiments simulating cloud environments ran by Ballani et al. at Microsoft Research [9], the interference traffic will be normally distributed around $\frac{\text{max\_bandwidth}}{2}$, as is shown in Figure 2. The standard deviation chosen is such that the distribution is slightly wider than the feasible values between $[0, \text{max\_bandwidth}]$, which ensures that the higher and lower interference bandwidths are used slightly more frequently. If the distribution were to exactly match the range of feasible values, the higher and lower values would likely not occur within the duration of the experiments.

The traffic generator utilizes Iperf3 [46] to generate network traffic with varying bandwidth sampled according to this distribution. Iperf3 batches the generated traffic such that small bursts in packets are present. Next to this, the Iperf3 traffic streams are bidirectional. Since Iperf3 requires some wind-up time when starting to generate traffic, each sampled bandwidth is used for 5 seconds to generate interference. This ensures there is a traffic stream for most of the time. After this, a new value is sampled according to the distribution, which is then used to generate traffic again. To ensure experiments are carried out on a number of Iperf3 interference bandwidths, experiments will need to be carried out multiple times or over a longer duration.

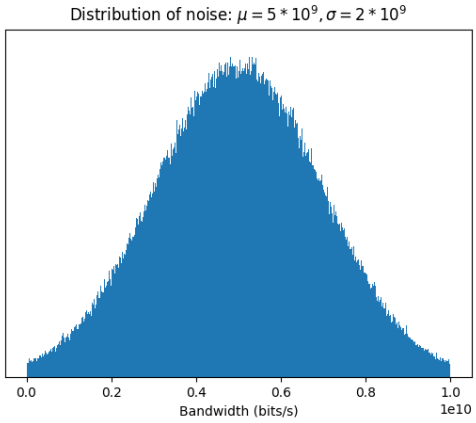

Figure 2: Distribution of network noise generated during the experiment in bits/s: $\mu = 5 \times 10^9$, $\sigma = 2 \times 10^9$. Values outside the viable range (i.e. lower than 0 or higher than the maximum link speed of 10Gb/s) are regenerated, therefore the actual distribution varies slightly.

4.3 Traffic Shaping

All experiments use the same set of traffic shaping profiles, which are shown in Table 2. The interference traffic is generated as described in Section 4.2, and is present in all profiles except the "no interference" profile. The token bucket and priority queue settings are applied to the switch ports connected to all nodes, with the priority queue settings having high and low priority for different ports.

Experiment Runtime (µs)	**Description**
No Interference	No changes
Normally Distributed Interference	Normally distributed interference traffic on node interfaces
Token Bucket	Token bucket on node interfaces in addition to interference
Priority Queue	Priority queue alternating between high and low priority on node interfaces in addition to interference
Token Bucket & Priority Queue	Combination of the above

Table 2: Experiment profiles with relation to traffic shaping and interference. Experiments are executed on each of the above.

Priority queues on-switch work on a port-by-port basis. Therefore, traffic emerging from the node cannot be subjected to different QoS parameters. External traffic sources transmitting to the same node or port *can* be subjected to different QoS parameters, however. Therefore, the DSCP priorities are to be configured on the source ports of the traffic. In this case, two configurations are chosen: Equal priority and low/high priority. The first uses equal (or no) DSCP values for the traffic, while the second uses traffic class 1 (low) and 4 (high) to grant priority to alternating nodes. Drop probability is omitted here since it is superseded by traffic class. All queues used are Weighted Round Robin (WRR) queues, as having strict priority queues would prevent lower priority traffic from being transmitted entirely. The priority queues are applied to switch ports in an alternating fashion, such that hosts which transmit interference traffic between them have different priorities. In the experiment topologies that will be described in Sections 4.5, 4.6 and 4.7, the interference marked between nodes has one node on higher priority than the other if unequal priority is used. Similarly, priority queues on-host are realized using OVS-DPDK. Here, flows dictating the traffic behaviour are given different DSCP traffic classes, as mentioned previously, also in an alternating fashion. This ensures the experiments can be setup without manual intervention.

Traffic shaping using a single bucket is configured with a Committed Burst Size (CBS) of 100 k and a Committed Information Rate (CIR) of 1000 M. This allows for the usage of most of the available bandwidth, thus reducing the impact the bandwidth reduction itself has on performance, though it is still throttled to a small degree. This type of shaping facilitates micro bursts through the CBS, which

should allow for applications that send data infrequently to be hardly impacted by the token bucket, as their average used bandwidth is lower. The bursts may result in small temporary bottlenecks however, which is expected to impact the performance of the applications. Token buckets are applied to switch ports corresponding to the nodes, and police only *outgoing* traffic from the node (i.e. *incoming* traffic from the perspective of the switch). This restriction with relation to policing direction also applies to the interfaces used by OVS-DPDK, which are used to apply a per-container token bucket. These use the same CBS and CIR as the interfaces of the physical switch such that their results can be compared.

Finally, a hybrid traffic shaping approach using both the physical switch as the OVS-DPDK backed on-host switch was used. These use either an on-host token bucket and an on-switch priority queue or vice versa. The configurations of either are unaltered compared to the description given previously, allowing for the effect of shared responsibility to be observed as well. This way more fine-grained control may be possible by using on-host shaping, while on-switch shaping could reduce load on the host by taking over traffic shaping tasks which benefit less from this locality.

Traffic shaping using the dual-bucket setup as described in Section 2.2.2 would allow the full link speed to be utilized as normal without any baseline throttling. However, since traffic exceeding the CIR is marked using lower DSCP values to accomplish this – increasing the probability packets are dropped – this would effectively cause an overlap between the token bucket and priority queue. Therefore, only the single bucket traffic shaping is used.

4.4 Distributed Applications

The distributed applications used in experiments fall in the following categories: Key-Value Store, Big Data, and High Performance Computing (HPC). This creates a representative baseline covering different types of distributed applications, which are typically found in cloud environments nowadays [47, 48, 49]. The respective applications used are MongoDB [50], Apache Spark [51] and the Message Passing Interface (MPI) [52]. The benchmarks on these applications will measure both performance relative to a baseline without traffic shaping or even without interference traffic, as well as the "consistency" of performance. "Consistent" performance is defined as a low variability in results, whereas an "inconsistent" performance is defined as a high variability in results. In this section, for each of the experiments, the applications will be described in further detail.

4.4.1 Key-Value Store

To represent a distributed Key-Value Store, MongoDB [50] was chosen. As the name suggests, it allows for storing data based on key-value pairs, which may be nested similarly to how JSON is structured. Additionally, it supports replication between different MongoDB instances on different nodes, which should allow for fluctuations in network congestion to impact its performance. MongoDB uses a primary node as an interface for data-modifying operations, which in turn get replicated to the secondary nodes asynchronously [53]. While read operations on the primary node are always consistent, due to the asynchronous replication, read operations on the secondary nodes are eventually consistent.

Since replication requires network communication, a primary node suffering from network congestion could impact the delay at which secondary nodes get updated. The replication could be impacted further by putting the primary node under load, increasing the network congestion. Apart from this, the maximum attainable performance regarding insert and update operations on the primary node should be impacted by a congested network as well, due to a reduced amount of data being able to reach the primary node in a given time.

4.4.2 Big Data

Apache Spark [51] is the representative system for Big Data workloads. It specifically allows for the execution of applications in a Map-Reduce fashion. A classic example of this is a word frequency counter, where "Map" denotes the counting of word frequencies on a slice of data on each node, and "Reduce" denotes the aggregation of results (i.e. counted words) across nodes. Apache Spark is able to use the Hadoop Distributed File System (HDFS) [54] which is a robust file system capable of handling very

large datasets across multiple heterogeneous nodes, and is resilient to hardware failure. HDFS is part of the Hadoop project [55], which is the predecessor of Apache Spark, capable of serving similar Big Data workloads. Apache Spark additionally allows for streaming, interactive queries and machine learning workloads [56]. Next to this, combining Apache Spark and HDFS allows for using data locality by executing computation tasks near (or on) nodes where the data is present, instead of moving data to the computation.

4.4.3 High Performance Computing

To represent High Performance Computing (HPC) applications, the Message Passing Interface (MPI) [52] library is utilised. It is used in distributed applications, and supports the deployment of computation across a multitude of heterogeneous nodes through the use of either `rsh` or `ssh`. The number of desired processes, as well as the specific hosts to use in running an MPI program can be specified when starting the program. MPI will then automatically start processes on the hosts using the `rsh/ssh` commands. MPI provides its own "wrapper compiler" in order to compile applications that use MPI, supporting the use of C, C++ and FORTRAN [57]. Next to this, it is able to handle communication between processes on different nodes on a one-to-one, one-to-many, many-to-one and many-to-many basis, using SEND, BCAST, REDUCE and ALLTOALL calls respectively [58].

4.5 Key-Value Store

The Key-Value Store experiment uses a synchronizing MongoDB cluster with one primary and two backup nodes. The backup nodes replicate the data from the primary node. The MongoDB cluster can be reached from the node executing the benchmark through a user-managed switch, as is shown in Figure 3. On-switch traffic shaping is implemented here. Between node pairs, bidirectional, normally distributed interference traffic is generated according to the specification of Section 4.2. Both MongoDB, the benchmark and the Iperf3 noise generation use the OVS-DPDK backed Docker overlay network, which spans the network through the user-managed switch. Next to this, all nodes have out-of-band communication for Docker network overlay events, such as containers joining or leaving.

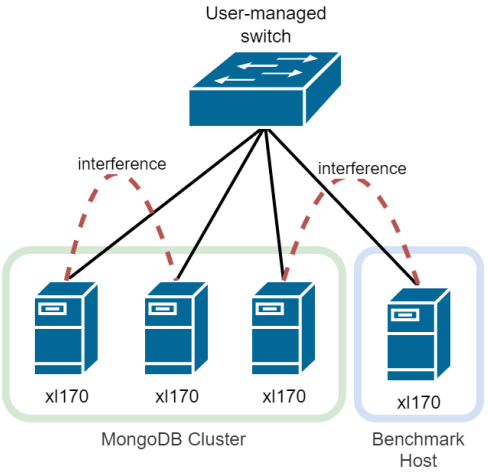

Figure 3: Topology used for the Key-Value Store experiment. The interference simulates noisy neighbours utilizing the same network, and is run bidirectionally between nodes.

In order to evaluate the performance of the MongoDB Key-Value Store cluster, the MongoDB YCSB benchmark [59] is used. This benchmark contains 6 predefined experiments [60], shown in Table 3, which cover a range of use cases using varying relative amounts of read, update and write operations. All 6 experiments are used in the evaluation of the cluster. The experiment execution settings were altered to use a record count of 1,000,000 as well as an operations count of 1,000,000. This would allow for multiple normally sampled interference bandwidths to occur over the duration of the experiment.

Workload	Description	Example
A	Update heavy workload	Recent actions being recorded in a session store
B	Read mostly workload	Tagging Photos. Tags can be added (update), but reading tags is the most occurring operation
C	Read only	Caches of user profiles which are created elsewhere
D	Read latest workload	News, where people want to read the latest, e.g. user status
E	Short ranges	Forum threads, where each scan retrieves all comments for the thread, using e.g. a thread id
F	Read-modify-write workload	E.g. a database that allows for the recording of user data and activity i.e. reading, writing and modifying data.

Table 3: Workloads defined in the YCSB Benchmark Suite, covering a range of use cases [60]. Each benchmark is run using a record and operations count of 1,000,000, using a modified version of YCSB which outputs more detailed results.

The default output of the YCSB suite only contains average, minimum and maximum latencies, as well as 95^{th} and 99^{th} percentiles. As this inadequately showed the spread of latencies, the impact of traffic shaping was not always apparent. The YCSB source was therefore altered to allow for the collection of not only the previously mentioned fields, but also the 25^{th}, 50^{th}, 75^{th} and 99.9^{th} percentiles. Next to this, the box plot whiskers, as well as any possible outliers were added to the output. This represented the spread of latencies resulting from the experiments more closely, therefore allowing for a more accurate display and comparison of the experiment results. The modified YCSB sources will be provided in the git repository accompanying this thesis, and further elaboration regarding this setup will be provided in Appendix B

4.6 Big Data

In order to measure the effect of traffic shaping on a Big Data workload, Apache Spark [51] and the HiBench [61] "sparkbench" benchmark were used. This benchmark comprises workloads from web search, machine learning and analytical query domains, and contains "micro" benchmarks, which are based on the example applications provided with Apache Spark. As the full suite of benchmarks at the default `tiny` dataset size setting takes over 2 hours to complete, preliminary tests were done over 10 runs in order to determine a suitable benchmark candidate to run a larger experiment on. Other than this selected benchmark, the remaining benchmarks were not used in further evaluations, as the evaluation of the full HiBench benchmark suite on large datasets would take up an unfeasible amount of time. Each benchmark used the default settings for resource usage: Access to all cores & 4GB of memory. The experiment topology consists of four Spark workers with interference traffic between themselves – generated as described in Section 4.2 – as well as a separate master node which also houses the `namenode`, as is shown in Figure 4.

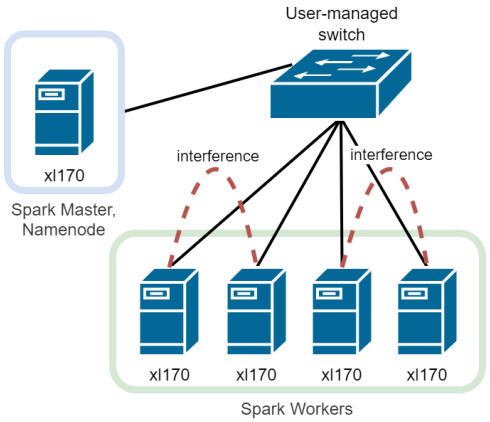

Figure 4: Topology used for the Big Data experiment.

In the preliminary experiments, the Terasort benchmark showed promise due to its reconfigurability with regard to data size, as well as due to its degree of use of communication channels. This makes it more susceptible to changes in bandwidth and latency, which should allow any effect of traffic shaping to be visible through application performance. The benchmark settings for Terasort were modified to use a dataset size of 300,000,000, which is approximately the `huge` built-in dataset size, and it was run 10 times. Within the experiment infrastructure, this was the largest available size that resulted in a stable system, as larger sizes caused occasional namenode problems. This was likely due to the shared-resource nature of containers as well as very limited availability of local storage.

4.7 High Performance Computing

The setup for the High Performance Computing experiment consists of a cluster of four OpenMPI nodes with bidirectional interference traffic between them – as described in Section 4.2 – which are connected through a user-managed switch, as is illustrated in Figure 5. The Docker containers on the MPI nodes need to be run using privileged mode, as well as in the host IPC namespace, to ensure that the MPI send & recv calls will run without error. This seems to be due to the way MPI accesses memory in these calls. Next to this, each MPI node will need to be able to reach each other MPI node through SSH using public/private key pairs.

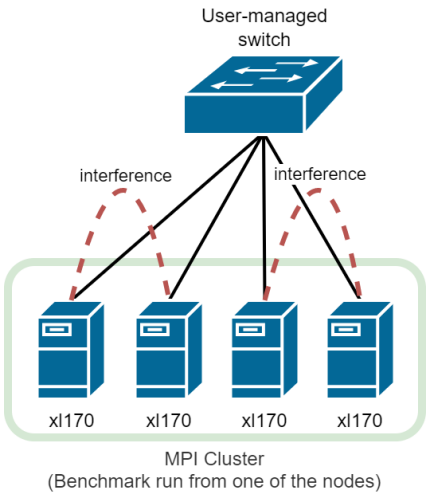

Figure 5: Topology used for the High Performance Computing experiment.

The performance of the cluster of MPI nodes was tested using the HPC Challenge (HPCC) benchmark suite [62], which consists of a set of standardized benchmarks as well as latency and bandwidth related

benchmarks [63], as is shown in Table 4. These benchmarks cover a range of workloads such that multiple facets of the performance of the system may be evaluated. The benchmark suite is run for 100 times for all traffic shaping settings described in Section 4.3, using the "base run" settings. This prohibits the modification of source code, which was not needed in order to run HPCC on the Docker-based MPI cluster. Each node was configured through the `hostfile` to have eight slots, with a maximum of ten. No memory constraints were imposed.

Benchmark	**Benchmark Focus**
HPL [64]	Floating point execution rate for solving a system of linear equations.
DGEMM [65]	Floating point execution rate for double precision real matrix-matrix multiplication.
STREAM [66]	Sustainable memory bandwidth (in GB/s).
PTRANS [67]	Rate of transfer for large arrays of data from multiprocessor's memory.
RandomAccess [68]	Rate of random updates of memory.
FFT [69]	Floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT).
Latency/Bandwidth (Based on `b_eff` [70])	Latency and bandwidth of network communication using basic MPI routines. The measurement is done during non-simultaneous and simultaneous communication and therefore it covers two extreme levels of contention that might occur in a real application: no contention and contention caused by each process communicating with a randomly chosen neighbour in parallel.

Table 4: The measurement focus of workloads [71] contained within the HPCC benchmark suite [62]. The suite covers different aspects of HPC performance such that a reliable measure of system performance may be obtained. Each benchmark is run for 100 passes in the "base run" config, where no altered source code is allowed.

5 Results

The execution of the experiments as described in Section 4 yielded a large amount of results for each experiment. These will be used to find out what "lessons can be learned from benchmarking distributed applications under the influence of traffic shaping". In this Section a selection of relevant or interesting results will be discussed for each experiment. First, the results for the Key-Value Store experiment using YCSB to benchmark MongoDB will be discussed. Second, the results for the Big Data experiment which used Apache Spark and the HiBench benchmark will be examined. Finally, the results of the High Performance Computing experiment conducted using HPCC on MPI will be considered. The complete results for all experiments will be included in Appendices C, D and E for completion.

5.1 Key-Value Store

The results that were obtained from the Key-Value Store experiments fell into roughly three categories. The first category encompasses the general improvement of the latency consistency when traffic shaping techniques are applied, with the exception of on-switch token buckets which slightly reduce consistency and have high tail latency. The second category concerns the fact that traffic shaping techniques have little effect on the latency consistency, with the exception of on-switch token buckets which make it slightly worse. The third category is similar, with generally little influence from the traffic shaping techniques, though in this case the on-switch token bucket has a large negative impact on the latency consistency. In this Section, for each of the categories some of the results will be highlighted. The full results for the experiments can be found in Appendix C.

5.1.1 General Latency Improvement

An example of the first category – showing general improvement over the baseline with interference traffic – is shown in Figure 6. All traffic shaping techniques reduce the spread of operation latencies measured, with the exception of the on-switch token bucket. This is seen in the results of `READ` operations across workloads `B`, `C` and `D`, which are `READ`-heavy workloads. Their median latency is below 200 µs, whereas the median latency of the `READ` or `SCAN` operations in other workloads is much higher, even when comparing the "no interference" baseline latencies. Presumably this is due to the fact that their `READ` operations are larger, which reduces the relative effect the traffic-shaping techniques might have.

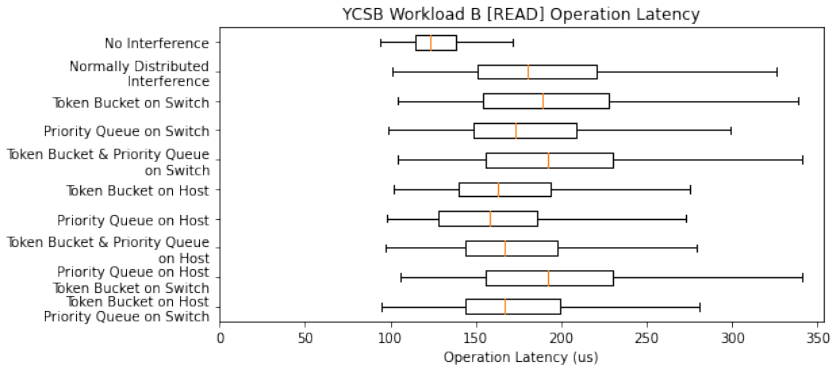

Figure 6: Read latency during the execution of YCSB Workload B (Max whisker: ±1.5 IQR). Traffic shaping measures generally improve latency, but on-switch token buckets seem to have little effect.

While the spread of the latencies depicts the on-switch token bucket as equivalent or slightly worse compared to the normally distributed interference without traffic shaping, the tail latencies show a very large difference. As Table 5 shows, these tail latencies are two to three orders of magnitude larger. Presumably this is due to the fact that in this case the token bucket is shared between tenants (i.e. the benchmark and the interference), causing heavy contention. On-host token buckets also negatively impact the tail latency, however, achieving a P99.9 twice as high as the experiment without any traffic

shaping present. This is slightly improved through the addition of priority queues, which across all experiments improve tail latencies. For this `READ` operation workload in particular, the on-host priority queue outperforms the other options in both consistency and tail latency.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	246	360	551
Normally Distributed Interference	440	633	937
Token Bucket on Switch	430	**205695**	**211199**
Priority Queue on Switch	389	536	759
Token Bucket & Priority Queue on Switch	429	**205823**	**211071**
Token Bucket on Host	394	856	1947
Priority Queue on Host	318	449	681
Token Bucket & Priority Queue on Host	391	653	1708
Priority Queue on Host, Token Bucket on Switch	428	**205695**	**210687**
Token Bucket on Host, Priority Queue on Switch	392	626	1610

Table 5: Tail latencies of the Read operation in YCSB Workload B with shaping present are generally slightly above or below that of interference without shaping, though on-switch token buckets incur a tail latency far larger than any other traffic shaping measure.

Conclusion 1. In Key-Value Store workloads, smaller read operations benefit from priority queues in both latency consistency and tail latency.

Conclusion 2. In Key-Value Store workloads, the latency consistency of smaller read operations may benefit from on-host token buckets, though at the cost of a higher tail latency. On-switch token buckets provide worse consistency and have tail latencies two to three orders of magnitude higher.

5.1.2 No Latency Improvement & Large Tail Latency

An example of the second category, in which traffic shaping has little effect on latency consistency in general, is shown in Figure 7. Here the on-switch token bucket once again slightly reduces consistency compared to the baseline interference without any traffic shaping present. This pattern can be observed in the `READ` operation of workloads `A` and `F`, the `READ-MODIFY-WRITE` operation of workload `F`, and the `SCAN` and `INSERT` operations of workload `E`. These workloads generally take more time than those of the first category shown in Section 5.1.1, and are more focused on updating, inserting or modifying records instead of primarily reading them. This shift in focus might also be the reason that the "no interference" baseline latency is higher in these workloads, leaving little room for improvement due to traffic shaping.

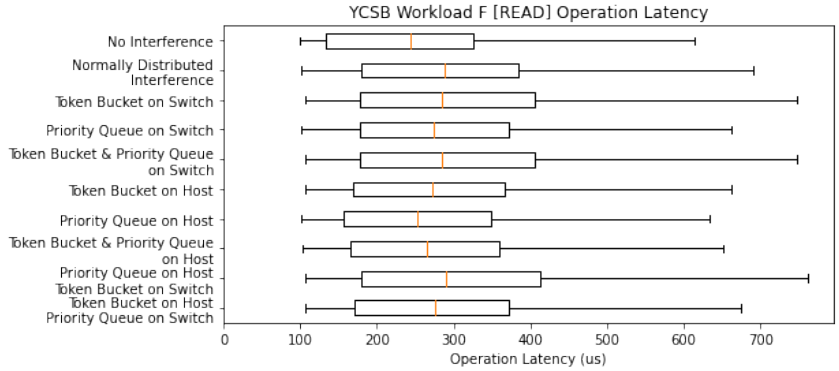

Figure 7: Read latency during the execution of YCSB Workload F (Max whisker: ±1.5 IQR). Traffic shaping measures generally have little effect on latency, with on-switch token buckets appearing to make latency slightly less consistent.

Similarly to the `READ`-focused workloads, the tail latencies for traffic shaping measures using on-switch token buckets are two to three orders of magnitude larger than the other solutions, as Table 6 shows. Contrary to those `READ`-focused workloads, however, any other traffic shaping measure is equivalent or improves tail latencies compared to the baseline with interference traffic. This is again improved by the addition of priority queues, except in the case where the token bucket is implemented on-host and the priority queue on-switch.

The `INSERT` operation of workload `E` is a special case since its on-switch token bucket tail latencies differ from the others, as they are more in line with that of the other traffic shaping techniques. In the case of the on-host token bucket, it even achieves the lowest tail latency of all traffic shaping settings, though the on-host priority queue still provides the most consistent latency. This might be because workload `E` uses (short) ranges in its operations. As inserting a range likely takes longer than a single insert or update, the latency consistency may be relatively less impacted between traffic shaping settings. The on-host token bucket is presumably better due to the same reason, as the larger size of ranges benefits from per-tenant throttling as this results in a fairer division of bandwidth.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	398	470	665
Normally Distributed Interference	523	927	1621
Token Bucket on Switch	627	**205055**	**209791**
Priority Queue on Switch	467	651	1111
Token Bucket & Priority Queue on Switch	629	**205183**	**210687**
Token Bucket on Host	518	911	1524
Priority Queue on Host	426	559	846
Token Bucket & Priority Queue on Host	469	765	1378
Priority Queue on Host, Token Bucket on Switch	634	**205183**	**210175**
Token Bucket on Host, Priority Queue on Switch	497	785	1628

Table 6: Tail latencies of the Read operation in YCSB Workload F with shaping present are generally similar to that of interference without shaping, and better when priority queues are used, though usage of on-switch token buckets increases latency at the tail end.

Conclusion 3. In Key-Value Store workloads, read operations in update-heavy workloads benefit from traffic shaping – especially priority queues – mostly by reducing tail latency, though general consistency is hardly impacted. This does not hold for on-switch token buckets which increase tail latencies by two to three orders of magnitude.

Conclusion 4. In Key-Value Store workloads, insert operations inserting (small) ranges – i.e. consecutive records of a sorted attribute – benefit from on-host token buckets to reduce tail latency and exhibit a small impact on consistency and tail latency when using on-switch token buckets.

5.1.3 Large Decrease in Latency Consistency

The third category generally shows little impact on latency consistency by introducing the traffic shaping techniques, with the exception of the on-switch token bucket. The latter negatively impacts the consistency, as is shown in Figure 8. This kind of behaviour is observed in the `UPDATE` operations of workloads `A`, `B` and `F`, as well as the `INSERT` operation of workload `D`. As this `UPDATE`-heavy workload generally has a very low operation latency, the negative impact of the on-switch token bucket is relatively large compared to the previous two categories. Still small improvements may be achieved through the use of the on-host priority queue.

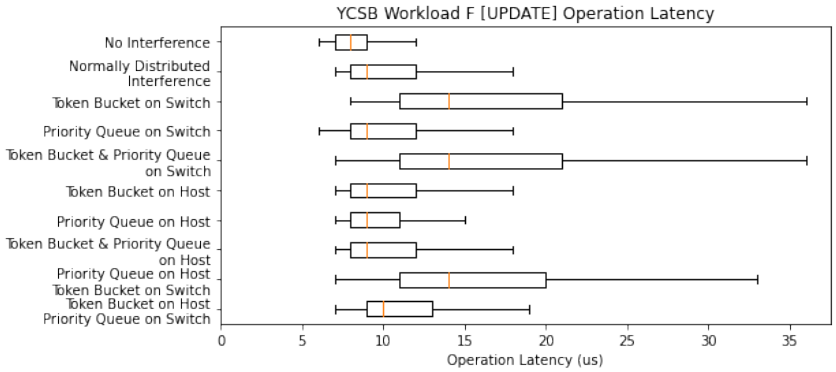

Figure 8: Update latency during the execution of YCSB Workload F (Max whisker: ±1.5 IQR). Traffic shaping measures generally have little effect on latency, with the exception of the on-host priority queue. On-switch token buckets greatly reduce the consistency of latency.

Compared to the other two categories, tail latencies hardly increase through the use of traffic shaping techniques. Though the on-switch token buckets generally show an increase in tail latency over the interference without traffic shaping, this increase is comparatively small, as is shown in Table 7. Given the structure of the experiment, it seems likely that the `UPDATE` operations hardly gets impacted by the on-switch token bucket, since this meters incoming traffic from the perspective of the switch, i.e. traffic sent by the nodes. While interference traffic is bidirectional, the sending part of the benchmark has no contention with it due to this reason. Presumably only the confirmation response from the MongoDB node has a possibility of getting delayed due to contention, which would explain the lack of large tail latencies for the on-switch token bucket configurations. The overall increase in latency – both in the tail and consistency – could then be explained by a flat overhead possibly introduced by the on-switch token bucket.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	18	29	44
Normally Distributed Interference	24	33	56
Token Bucket on Switch	32	58	90
Priority Queue on Switch	23	30	52
Token Bucket & Priority Queue on Switch	32	58	89
Token Bucket on Host	25	34	63
Priority Queue on Host	23	30	50
Token Bucket & Priority Queue on Host	24	33	62
Priority Queue on Host, Token Bucket on Switch	32	58	89
Token Bucket on Host, Priority Queue on Switch	25	34	63

Table 7: Tail latencies of the Update operation of YCSB Workload F with shaping present are generally similar to that of interference without shaping, though on-switch token buckets generally have slightly worse tail latency by comparison.

Conclusion 5. In Key-Value Store workloads, insert operations – which are short in the YCSB workloads – are disproportionally impacted by the on-switch token bucket in latency consistency. This is likely because the on-switch token bucket may introduce a flat overhead.

5.2 Big Data

After running the full HiBench benchmark suite on the `tiny` dataset, the Terasort benchmark was run using a larger dataset, as was described in Section 4.6. This was done because not all benchmarks are equally dependent on the underlying network, and the effect of traffic shaping on an application hardly using the network would be very difficult to show over the noise of background interference traffic. Terasort, by comparison, is relatively dependent on the underlying network.

The results of the HiBench Terasort benchmark, as shown in Figure 9, show that an increase in throughput appears to directly translate to a decrease in duration. In the case of a throughput-constrained application like Terasort this makes intuitive sense. It also shows that the on-switch token bucket has a negative impact on performance, though it retains similar consistency to the results achieved on interference traffic without traffic shaping. This is expected as all communication between nodes (i.e. the benchmark traffic as well as the interference traffic) gets throttled by the token bucket, increasing contention. Adding a priority queue to this token bucket, however, results in a decrease in performance consistency, especially so when it concerns an on-switch priority queue. This decrease in consistency is greater than that would be expected when compared to the slight decrease in consistency resulting from adding only the (on-switch) priority queue.

Presumably the cause of this is that the difference in priority between nodes causes some benchmarks to sometimes send to nodes at a higher priority than the interference traffic, leading to better measured performance. On the other hand, the opposite may occur, where the interference traffic has higher priority. These differences are then exacerbated by adding the token bucket, which (in the presence of lower interference) may lead to higher tails in performance, as well as lower tails when higher interference is present.

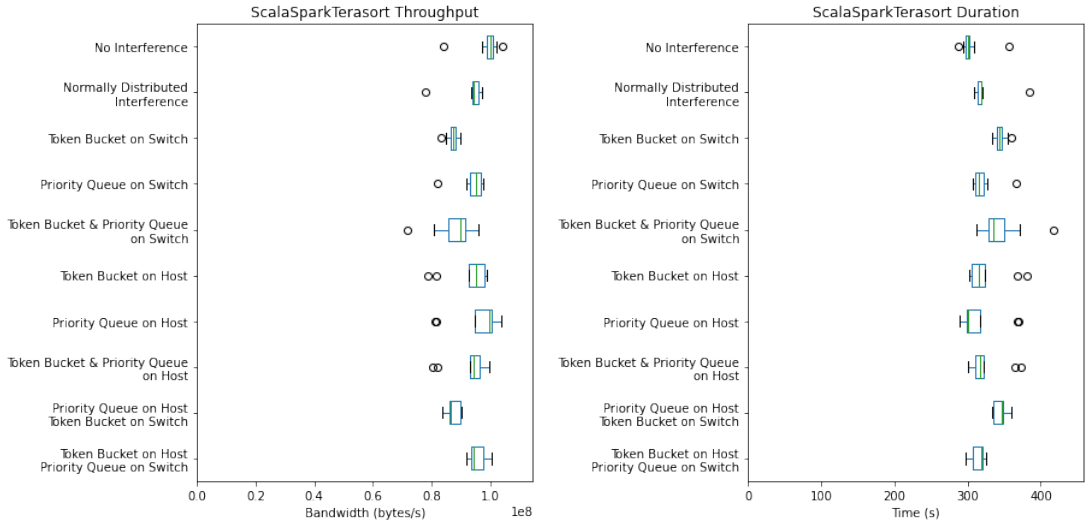

Figure 9: Throughput (left) and Duration (right) of the Terasort experiment under the effects of different traffic shaping settings (Max whisker: ±1.5 IQR). Only the combination of on-switch priority queue and token bucket shows a decrease in consistency.

Some of the other HiBench benchmarks – amongst which is the Wordcount benchmark – exhibited hardly any difference under influence of traffic shaping measures. Compared to the Terasort benchmark, the Wordcount benchmark is relatively independent of the underlying network due to the reduced communication requirements between nodes. Since these results did not exhibit any interesting interactions with the traffic shaping, they were excluded from this section. However, the results on the `tiny` dataset, as well as Wordcount results on a larger dataset can be seen in the Big Data results included in Appendix D.

Conclusion 6. In Big Data workloads, the throughput related performance loss caused by the introduction of the on-switch token bucket may increase variability caused by the on-switch priority queue. This appears to happen in cases where any node may connect to any other node, with not all nodes having equal traffic shaping settings.

5.3 High Performance Computing

The High Performance Computing experiment used the HPCC benchmark suite to benchmark MPI performance, as was previously mentioned in Section 4.7. It consists of many sub-benchmarks, each of which has their corresponding results. These results fall into roughly four categories, which are as follows:

- Performance of Token Bucket and Priority Queue combinations
- Negative Impact of the On-Switch Token Bucket
- Token Bucket Improvement
- Minimal Effect

In this Section, for each category some results will be highlighted and discussed. The complete results for the HPCC experiment, including the remaining few results that fall outside these categories, will be included in Appendix E.

5.3.1 Performance of Token Bucket and Priority Queue combinations

In multiple HPCC sub-benchmarks the combination of a token bucket and priority queue results in better performance when compared to the baseline, where only the normally distributed interference without any traffic shaping is present. This performance increase presents itself in either increased consistency, decreased consistency with higher performance tails, or equivalent consistency at a higher performance

level. However, this improvement is often not seen across all combinations of token bucket and priority queue locations, i.e. both on-host, both on-switch or split up either way. In some cases, the performance of the token bucket and priority queue combination even gets worse. This Section will highlight some of these combinations.

The first combination concerns the on-host priority queue and the on-switch token bucket. As Figure 10 shows, both the average and maximum ping-pong latencies are very consistent when the on-switch token bucket is used, which is further improved upon by the addition of a priority queue. Using the same configuration with an on-host token bucket results in worse performance, however. Similar behaviour can be seen when measuring the average bandwidth, though here the influence from the token bucket is far less. Both of these are measured in dedicated latency and bandwidth-measuring benchmarks, which do not measure any other aspects of the system.

The latency results in particular are surprising since they directly contradict those measured in YCSB on MongoDB in Section 5.1. Perhaps this is caused by the fact that the ping-pong latency is measured between all nodes, while the YCSB benchmark communicates with a single MongoDB entrypoint. As this spreads the load, the impact from contention at the token bucket might be smaller. Another point of difference is that YCSB measures the entire operation latency, not merely the network latency.

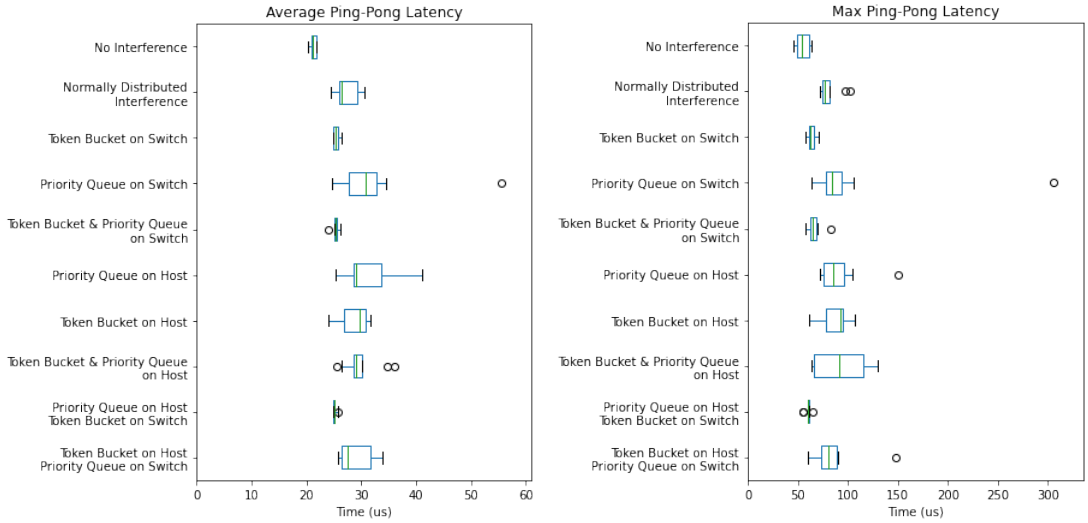

Figure 10: The on-switch token bucket improves consistency over no-shaping interference, while adding a priority queue makes it worse. Combining the two increases consistency further, while such improvements are not observed when using the on-host token bucket. The maximum whisker is ±1.5 IQR.

Since this HPCC ping-pong benchmark has negligible operation cost on the nodes, as well as minimal required IP-packet size – the packet size is 8 byte – the chance for contention to occur due to the token bucket is simply lower. Iperf3, by comparison, has a packet size of 8 KB for TCP and 1470 byte for UDP [72]. With standard Ethernet frames having a maximum size of 1500 byte by default [73] (no 9000 byte jumbo frames are used), the TCP packets sent by Iperf3 will have to be split up into multiple large frames. All Iperf3 traffic is therefore far larger than the traffic generated in the ping-pong experiment. This might lead it to be able to pass the token bucket whereas the Iperf3 frames would be dropped, when the token bucket nears depletion.

Due to the fact that the Ethernet links have a maximum bandwidth by default, this behaviour should also be observed when the token bucket is absent, which it is not. As is shown in the bandwidth benchmark results in Figure 11 there is only a small difference in consistency between the presence and absence of the on-host switch. This could indicate that the Iperf3 traffic is allowed to fully saturate the Ethernet link, leaving less chance for the smaller ping-pong packets to pass, increasing their latency due to head-of-line blocking.

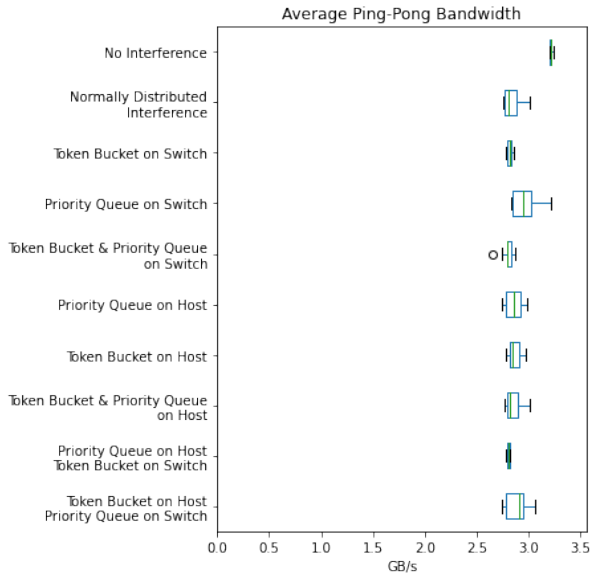

Figure 11: The on-switch token bucket improves consistency slightly over no-shaping interference, while the on-switch priority queue makes it worse. Combining the on-switch token bucket with an on-host priority queue improves consistency slightly. The maximum whisker is $\pm 1.5\,\mathrm{IQR}$.

Conclusion 7. In HPC workloads, very small IP-packets or Ethernet frames may benefit from on-switch (i.e. shared) token buckets when competing traffic consists of large Ethernet frames.

The second combination concerns the increase in performance at the cost of consistency when both the on-switch priority queue and token bucket are used. As Figure 12 shows, the throughput increases by adding the on-switch token bucket. It even manages to approach the performance of the benchmark without any interference present when the on-switch priority queue is added, which allows higher throughput peaks though at a far lower consistency. This behaviour is not seen in any other HPCC benchmark, however.

It seems plausible that the StarSTREAM "Add" benchmark benefits from the same perks as the latency benchmark previously did, in that the packet sizes are small enough to be allowed past the token bucket. The overhead added by on-host processing – which did not occur in the latency benchmark – might indicate that this HPCC benchmark is more susceptible to other forms of resource contention (e.g. CPU), which is outside the scope of this thesis. The decrease in consistency by adding the on-switch token bucket might then be explained by the difference in priority of the node, as it varies between nodes. Traffic from higher priority nodes would naturally be less likely to be dropped than that from lower priority nodes, and given the fact that a star topology is used, multiple priority combinations will be met. In the experiment, the MPI node on which the HPCC benchmark was started had a higher priority, which could explain the higher peaks.

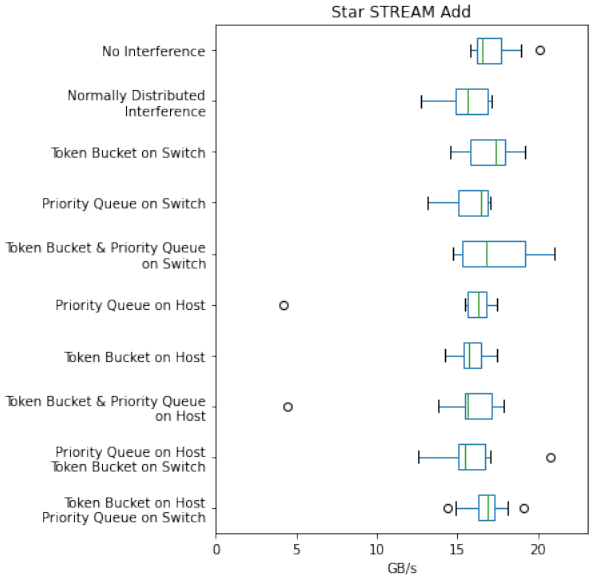

Figure 12: On-switch token bucket, perhaps with the addition of an on-host priority queue performs better for the StarSTREAM "Add" benchmark only. The maximum whisker is ±1.5 IQR.

A third combination is one where combining the on-switch token bucket and on-switch priority queue results in *worse* performance, contrary to that seen in the StarSTREAM "Add" benchmark. As is shown in Figure 13, this is once again likely due to the on-switch token bucket. Given the fact that the HPCC "Single" benchmarks run on a (randomly selected) single node, hardly any network influence should be possible. Perhaps the measured latency inconsistency is due to the transferring of data from the main HPCC node to the selected benchmark node at the start and end of the experiment, though presumably this would not have this large of an influence. This behaviour is seen in both the Single FFT and RandomAccess benchmarks. While these results are surprising, since the "Single" benchmarks are inherently non-distributed, further analysis of these results will be omitted.

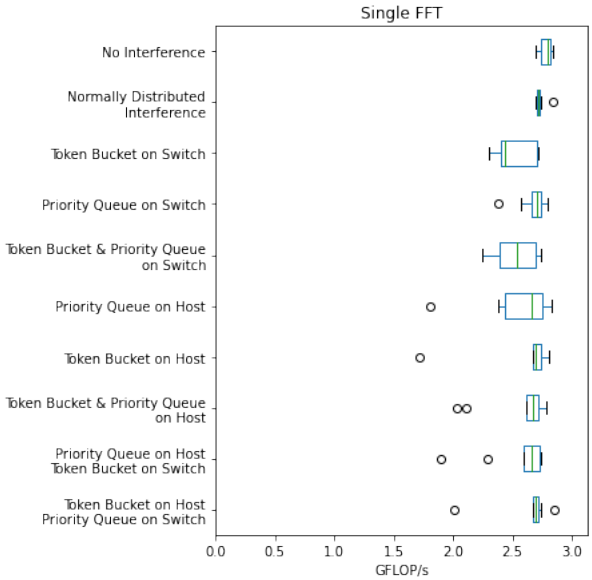

Figure 13: The on-switch token bucket, both by itself and combined with the on-switch priority queue, performs worse on "Single" FFT and RandomAccess. Since these execute on a single node, there should not be a large reliance on the underlying network. The maximum whisker is ±1.5 IQR.

5.3.2 Negative Impact of the On-Switch Token Bucket

Some of the HPCC experiments showed severe impact from the addition of the on-switch token bucket, to the point where in some cases there are orders-of-magnitude differences compared to other results. In Figure 14 two such examples can be seen. In the left case, the difference is relatively small, allowing the results of the other experiments to still be visible. The MPI FFT benchmark shown on the right seems very sensitive to the on-switch token bucket in particular, showing far worse performance.

This large difference is surprising since the addition of the on-switch token bucket showed an improvement of performance with regard to latency, as was previously shown in Figure 10. It has also been shown in Figure 11 that the consistency of the bandwidth is improved by the addition of the on-switch token bucket. It is possible that the ping-pong bandwidth shown there suffers less than the "Naturally Ordered Ring Bandwidth" benchmark of Figure 14, and that the MPI FFT benchmark is particularly sensitive to reductions in available bandwidth. Considering the previous results, it seems exceedingly unlikely that high tail latency would be the cause in this case.

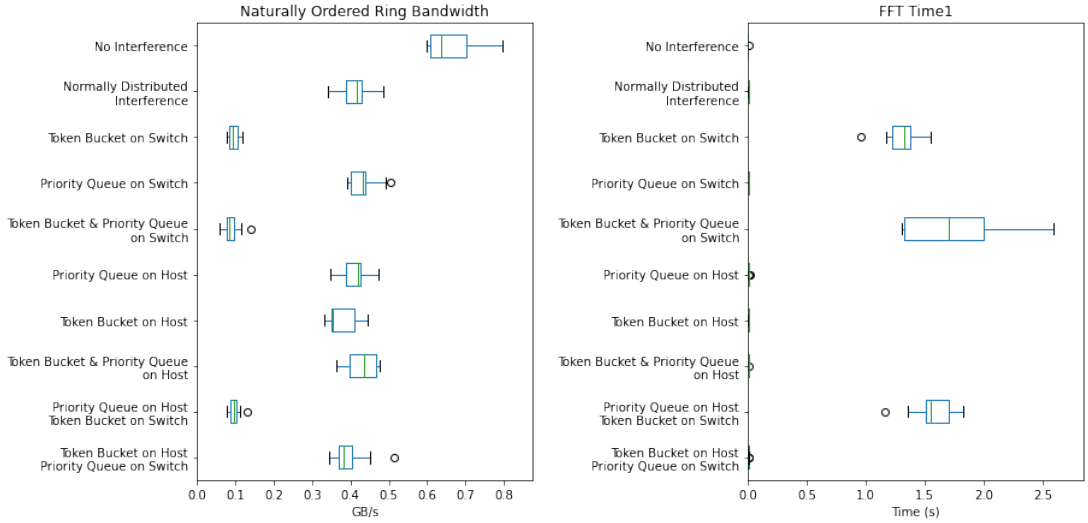

Figure 14: Left: Relatively minor impact of the on-switch token bucket, Right: very large impact due to the on-switch token bucket. This behaviour is surprising as it is a stark contrast to the improvement in performance shown in Figure 10. The maximum whisker is $\pm 1.5\,\mathrm{IQR}$.

Conclusion 8. In HPC workloads, bandwidth-dependent applications have their performance reduced due to bandwidth contention when an on-switch token bucket is implemented. If bandwidth management is required, usage of an on-host token bucket for more granular control is recommended instead.

5.3.3 Token Bucket Improvement

Some of the HPCC benchmarks perform considerably more consistent once a token bucket gets added. The HPCC StarSTREAM benchmark in particular shows improvement on "Copy" and "Scale" when an on-switch token bucket gets introduced. Unlike the previous results where the on-switch token bucket caused either a gain or loss of performance, in the case of the HPCC HPL benchmark both the on-switch and on-host token bucket have a positive impact, as Figure 15 shows. The addition of priority queues in this case mostly causes a reduction in consistency.

HPL solves a dense linear system [64] and claims to be scalable with respect to computation and communication volume [74]. This latter claim comes with the assumption that there are direct point-to-point links between processors, which have a communication time roughly linearly dependent on the number of items communicated. This is not the case in this experiment due to the use of a switched network, where no point-to-point connections between nodes exist. Taking this into account, it appears

that the token buckets increase the reliability (i.e. consistent latency and available bandwidth) of the network, while the priority queues likely interfere with the assumed linear nature of communication, or at least do nothing to improve it. This would make sense, as the traffic from different nodes or containers in the experiment is not treated equally when using priority queues, so a decrease in consistency would be strange.

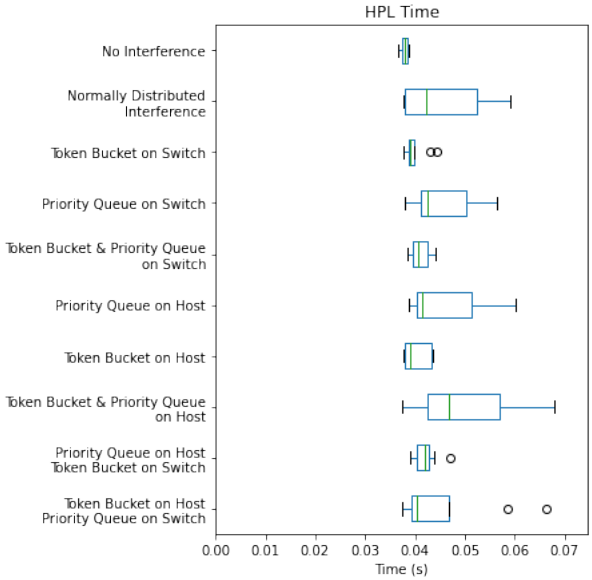

Figure 15: The introduction of a token bucket consistently improves upon the performance achieved without any traffic shaping, unlike priority queues, which approximately match it. The maximum whisker is $\pm 1.5\,\mathrm{IQR}$.

Conclusion 9. In HPC workloads, the applications with stricter assumptions regarding network bandwidth and/or latency are likely to benefit from traffic shaping, token buckets in particular.

5.3.4 Minimal Effect

Of the HPCC benchmarks, some results showed hardly any effect when a form of traffic shaping was added, as well as minimal to no effect in response to the presence of interference traffic. An example of this is shown in Figure 16. Unsurprisingly this occurs on most of the HPCC benchmarks executed on just a single node, as seen on the left, since there is very little to no dependency on the network in this case. Due to the fact that all computation takes place on a single node, the only communication that is needed is between the main MPI node and the node picked to execute the benchmark.

The right shows the results of the Star FFT benchmark, which does get executed on multiple nodes. This benchmark is different from the previously shown MPI FFT benchmark which was heavily affected by the on-switch token bucket, though HPCC does not explain the differences. The results of this Star FFT benchmark are surprising because a star topology supposedly creates a dependency on the underlying network. This effect was previously seen in Figure 12, where a token bucket had a positive influence on results. Presumably the Star FFT benchmark is not as reliant on the underlying network and instead has its performance dominated more by on-host processing. This would also explain the minimal influence the addition interference traffic has.

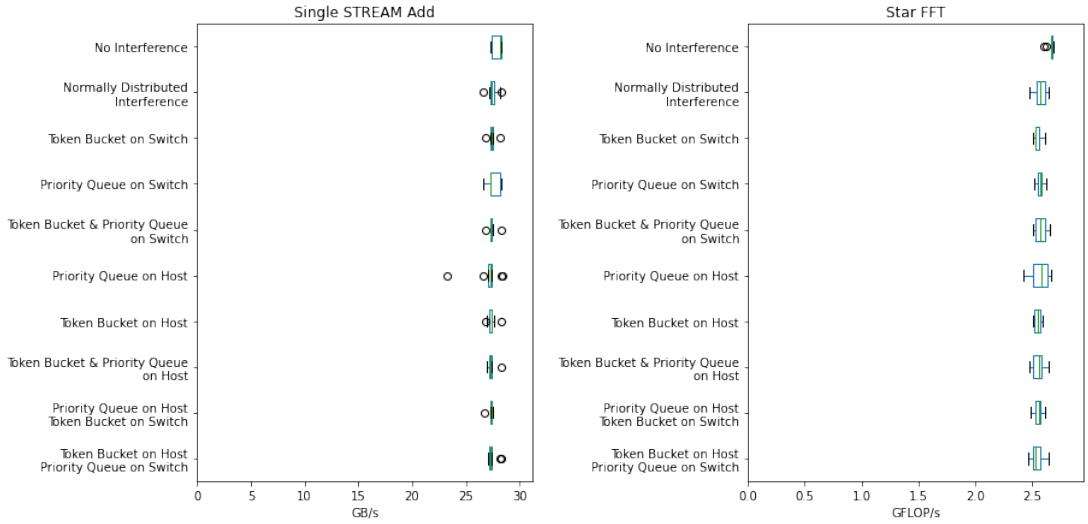

Figure 16: Some benchmarks show hardly any effect due to traffic shaping. The "Single" benchmark on the left executes on a single node, while the "Star FFT" benchmark on the right executes on multiple nodes connected in a star pattern. The maximum whisker is $\pm 1.5\,\mathrm{IQR}$.

6 Practical Implications

Taking these results into account, there are some practical implications that are to be considered when experimenting, benchmarking or deploying in shared cloud environments. In all cases it should be noted that part of the distributed application performance is at the mercy of the cloud providers, since the traffic shaping policies for the shared network are not set in stone and may therefore be subject to change. Even after careful evaluation, it may be possible that after some time a different solution would provide better performance, everything else being equal.

1. Benchmark the to-be-deployed application. As the response of distributed applications to the addition of traffic shaping techniques varies based on both the application and traffic shaping techniques used, which the variety in Conclusions show, no one-size-fits-all solution is possible. Therefore, it is advisable to benchmark the application in the desired environment on a smaller scale, comparing the results to a (non-shared) controlled environment to quantify the impact. Additionally, if the provider of the shared cloud environment allows for some control over the intermittent traffic shaping, add this as a tunable parameter in the benchmark. Finally, different cloud providers will have differences in their traffic shaping policies. Should project constraints allow it, it is therefore worth comparing between providers as well.

2. On-switch Token Buckets negatively impact tail latencies of many applications. This is mentioned in Conclusions 2, 3 and 5. It happens primarily when implemented in a way where contention may occur, which sharply increases tail latency compared to a baseline without traffic shaping. Therefore, if control of bandwidth is needed, the use of an on-host token bucket is advisable. This also allows for more granular, per-container control, and impacts tail latencies to a smaller degree, as mentioned in Conclusion 8. As mentioned in Conclusion 1, in both cases the introduction of a priority queue may alleviate some of this impact, though this does not hold for all types of applications.

3. Applications with small IP packets may benefit from Token Buckets. If the distributed application sends small IP packets or Ethernet frames – compared to other traffic on the shared links – the use of a token bucket may reduce tail latency when compared to a baseline without traffic shaping. In cases where tail latency is not reduced, the increase is relatively small. This effect is observed in both on-host and on-switch token buckets in Conclusions 2, 4 and 7. Note that this introduces a dependency on the network traffic of other tenants, which can generally not be controlled.

4. Consider the assumptions about the network. In the general case, applications with strict assumptions about the network benefit from traffic shaping techniques like priority queues and token buckets, which is mentioned Conclusion 9. If assumptions are made about the topology of the network (e.g. star or mesh topology), it needs to be considered that this may influence the performance of the application when traffic shaping is added, as is mentioned in Conclusion 6.

5. Design experiments taking cloud variability into account. Both the use of token buckets and priority queues – regardless of whether they are on-host or on-switch – may exacerbate the increase of variability caused by contention. While some variance may be tolerated when getting a rough estimate of viable cloud providers and traffic shaping settings, when repeatability and representative results are required, additional measures need to be taken. While some of these recommendations have been made in the past [12], in the context of the acquired results they deserve repeating. In the case of repeatability, when running experiments in shared cloud environments, the presence of traffic shaping may require additional repeats of experiments to improve reliability. In the case of representative results, specific experiments showing the change in performance in the presence of traffic shaping – such as those seen in shared cloud environments – may be necessary. In both cases it may be beneficial to specifically benchmark the network latency and bandwidth over time – alongside other sources of contention – and provide this information in addition to the acquired results.

7 Conclusion

In this thesis, the effects of traffic shaping on the performance of distributed applications were discussed. This is a facet of performance influencing factors often overseen when evaluating the performance of distributed applications, be they in real-world deployment or in research with cloud-based experiments. First the existing traffic shaping mechanisms – Priority Queue and Token Bucket – were explained. Then, their occurrence in real-world cloud environments was determined, which gave rise to an infrastructure with both on-host virtualized switches and physical network switches, both of which were capable of traffic shaping. Several experiments were then run on distributed applications covering a range of applications generally found in cloud environments: Key-Value Store, Big Data and High Performance Computing (HPC). These were implemented using MongoDB, Apache Spark and MPI respectively. All of them were subjected to various configurations of traffic shaping, specifically token buckets and priority queues present on combinations of the on-host and on the physical network switch.

The experiments showed that the performance of the distributed applications was affected to different degrees depending on the type of application, its network usage and the traffic shaping techniques – including their locations – that were used. The on-switch token bucket often had a negative influence on performance when compared to a baseline without traffic shaping, though sometimes it resulted in an improvement. Its performance decrease manifested itself primarily in an increase in tail latency, with a reduction in consistency sometimes exacerbated by the addition of a priority queue, though the priority queue by itself generally had a positive influence on the consistency of performance. On the other hand, the on-host traffic shaping techniques enforced by OVS-DPDK showed smaller (negative) effect when compared to this baseline, especially when considering the fact that the on-host vSwitch requires pinned cores for active polling. In all cases it needs to be noted that the size and frequency of traffic, as well as its sources and destinations change the response of the application to the introduction of traffic shaping.

Given the possible combinations of traffic priority and token bucket sizes, it seems likely that there is an optimal configuration of measures that would minimize the effect on specific underlying applications. However, traffic shaping measures requiring tuning on a case-by-case basis would not fit with the general purpose usage that is seen in public cloud environments, as a rebalancing of traffic shaping measures is not feasible with every change in tenants' usage. Not to mention that the specific settings for one application would likely make the performance of a different application worse, i.e. there is no free lunch. It is therefore important to benchmark the application with enough repeats to account for the possible increase in variance due to traffic shaping, and to compare different cloud environments as their traffic shaping settings likely differ.

Future work may include avenues that were left out-of-scope due to time constraints, such as multiple classes of traffic priority and multiple sizes of token buckets, including combinations, or more types of applications benchmarked in greater detail. In addition to this, the reason why specific network packet characteristics lead to a positive or negative influence compared to baseline results may be explored. Finally the incorporation of traffic processing executed on an FPGA instead of on pinned CPU cores may lead to interesting results – also when applying various traffic shaping rules. A preliminary experiment on FPGA-based traffic processing combined with OVS-DPDK has been included in Appendix A.

References

[1] R. Shea, F. Wang, H. Wang, and J. Liu, "A deep investigation into network performance in virtual machine based cloud environments," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pp. 1285–1293, 2014.

[2] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proc. VLDB Endow.*, vol. 3, p. 460–471, sep 2010.

[3] J. Ericson, M. Mohammadian, and F. Santana, "Analysis of performance variability in public cloud computing," in *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pp. 308–314, 2017.

[4] Google Cloud, "Configuring a vm with higher bandwidth." https://cloud.google.com/compute/docs/networking/configure-vm-with-high-bandwidth-configuration, Jan 2022.

[5] J. Solon and M. Flax, "Amazon ec2 instance network bandwidth." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html, Sep 2021.

[6] Google Cloud Platform, "Egress bandwidth." https://cloud.google.com/compute/docs/network-bandwidth#vm-out, Jan 2022.

[7] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea, "Chatty tenants and the cloud network sharing problem," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, (Lombard, IL), pp. 171–184, USENIX Association, Apr. 2013.

[8] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, (New York, NY, USA), p. 51–62, Association for Computing Machinery, 2009.

[9] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, (New York, NY, USA), p. 242–253, Association for Computing Machinery, 2011.

[10] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, (New York, NY, USA), p. 435–448, Association for Computing Machinery, 2015.

[11] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, 2011.

[12] A. Uta, A. Custura, D. Duplyakin, I. Jimenez, J. Rellermeyer, C. Maltzahn, R. Ricci, and A. Iosup, "Is big data performance reproducible in modern cloud networks?," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, (Santa Clara, CA), pp. 513–527, USENIX Association, Feb. 2020.

[13] L. Bulej, V. Horký, P. Tuma, F. Farquet, and A. Prokopec, "Duet benchmarking: Improving measurement accuracy in the cloud," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, (New York, NY, USA), p. 100–107, Association for Computing Machinery, 2020.

[14] A. Abedi and T. Brecht, "Conducting repeatable experiments in highly variable cloud computing environments," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, (New York, NY, USA), p. 287–292, Association for Computing Machinery, 2017.

[15] A. Campbell, G. Coulson, and D. Hutchison, "A quality of service architecture," *Computer communication review*, vol. 24, no. 2, pp. 6–27, 1994.

[16] D. J. Heinanen and D. R. Guerin, "A Single Rate Three Color Marker." RFC 2697, Sept. 1999.

[17] D. J. Heinanen and D. R. Guerin, "A Two Rate Three Color Marker." RFC 2698, Sept. 1999.

[18] S.-W. Moon and K. Shin, "Implementing traffic shaping and link scheduling on a high-performance server," in *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, pp. 216–225, 2001.

[19] DPDK Project, "Data plane development kit." https://www.dpdk.org/, Nov 2021.

[20] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 1–14, July 2019.

[21] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[22] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vSwitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 117–130, USENIX Association, May 2015.

[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, p. 69–74, mar 2008.

[24] B. Pfaff, J. Pettit, and J. Tourrilhes, "ovs-fields – protocol header fields in openflow and open vswitch." http://www.openvswitch.org/support/dist-docs/ovs-fields.7.pdf, 2021.

[25] Open Networking Foundation, "Openflow switch specification – version 1.5.1 (protocol version 0x06)." https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, Mar 2015.

[26] N. Pitaev, M. Falkner, A. Leivadeas, and I. Lambadaris, "Characterizing the performance of concurrent virtualized network functions with ovs-dpdk, fd.io vpp and sr-iov," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, (New York, NY, USA), p. 285–292, Association for Computing Machinery, 2018.

[27] Cisco, "Defining qos queues." https://www.cisco.com/assets/sol/sb/Switches_Emulators_v2_2_015/help/nk_configuring_quality_service16.html, Jan 2010.

[28] Dell EMC Inc., "User's configuration guide - dell emc networking n-series n1100-on, n1500, n2000, n2100-on, n3000, n3100-on, and n4000 switches," Jun 2017.

[29] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable traffic shaping at end hosts," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, (New York, NY, USA), p. 404–417, Association for Computing Machinery, 2017.

[30] M.-F. Homg, W.-T. Lee, K.-R. Lee, and Y.-H. Kuo, "An adaptive approach to weighted fair queue with qos enhanced on ip network," in *Proceedings of IEEE Region 10 International Conference on Electrical and Electronic Technology. TENCON 2001 (Cat. No.01CH37239)*, vol. 1, pp. 181–186 vol.1, 2001.

[31] A. Parekh and R. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 344–357, 1993.

[32] D. B. Grossman, "New Terminology and Clarifications for Diffserv." RFC 3260, Apr. 2002.

[33] W. Weiss, D. J. Heinanen, F. Baker, and J. T. Wroclawski, "Assured Forwarding PHB Group." RFC 2597, June 1999.

[34] L. Liechti, P. Gouveia, J. Neves, P. Kropf, M. Matos, and V. Schiavoni, "Thunderstorm: A tool to evaluate dynamic network topologies on distributed systems," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pp. 241–24109, 2019.

[35] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella, "Ac/dc tcp: Virtual congestion control enforcement for datacenter networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, (New York, NY, USA), p. 244–257, Association for Computing Machinery, 2016.

[36] Y. Yang, H. Jiang, Y. Wu, Y. Lv, X. Li, and G. Xie, "C2QoS: Cpu-cycle based network qos strategy in vswitch of public cloud," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 438–444, 2021.

[37] M. Kogias, S. Mallon, and E. Bugnion, "Lancet: A self-correcting latency measuring tool," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 881–896, USENIX Association, July 2019.

[38] S. Guruprasad, R. Ricci, and J. Lepreau, "Integrated network experimentation using simulation and emulation," in *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pp. 204–212, 2005.

[39] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang, "When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, (New York, NY, USA), p. 621–637, Association for Computing Machinery, 2021.

[40] T. Monjalon, "Dpdk supported nics." https://core.dpdk.org/supported/nics/, Jan 2022.

[41] CloudLab, "The cloudlab manual - hardware." https://docs.cloudlab.us/hardware.html, Apr 2021.

[42] HashiCorp, "Consul by hashicorp." https://www.consul.io/, May 2022.

[43] Y. Ahmed. https://www.intel.com/content/www/us/en/developer/articles/technical/using-docker-containers-with-open-vswitch-and-dpdk-on-ubuntu-1710.html, 2018.

[44] M. Dalton, D. Schultz, A. Arefin, A. Docauer, A. Gupta, B. M. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. Adriaens, J. L. Alpert, J. Ai, J. Olson, K. P. DeCabooter, M. A. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018*, 2018.

[45] G. Shetty, "Ovn-docker-overlay-driver." https://github.com/shettyg/ovn-docker/blob/master/ovn-docker-overlay-driver, Nov 2015.

[46] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu, "Iperf - the ultimate speed test tool for tcp, udp and sctp." https://iperf.fr/.

[47] IBM Cloud Team, "Top 7 most common uses of cloud computing." https://www.ibm.com/cloud/blog/top-7-most-common-uses-of-cloud-computing, Jul 2020.

[48] Microsoft, "What is cloud computing? a beginner's guide: Microsoft azure." https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/#uses, 2022.

[49] R. Bala, B. Gill, D. Smith, D. Wright, and K. Ji, "Magic quadrant for cloud infrastructure and platform services." https://www.gartner.com/doc/reprints?id=1-271OE4VR&ct=210802&st=sb, Jul 2021.

[50] MongoDB Inc., "Mongodb documentation." https://docs.mongodb.com/manual/introduction/, 2021.

[51] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, p. 56–65, oct 2016.

[52] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[53] MongoDB Inc., "Replication." https://docs.mongodb.com/manual/replication/, 2021.

[54] D. Borthakur, "Hdfs architecture guide." https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, Oct 2020.

[55] The Apache Software Foundation, "Hdfs architecture." https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, Jun 2021.

[56] Apache Spark, "Spark overview." https://spark.apache.org/docs/3.2.1/, 2022.

[57] The Open MPI Project. https://www.open-mpi.org/faq/?category=mpi-apps#general-build, May 2019.

[58] The Open MPI Project, "Open mpi v4.1.2 documentation." https://www.open-mpi.org/doc/current, Nov 2021.

[59] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), p. 143–154, Association for Computing Machinery, 2010.

[60] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Ycsb workloads." https://github.com/brianfrankcooper/YCSB/tree/master/workloads, Sep 2019.

[61] S. Huang, J. Huang, Y. Liu, and J. Dai, "Hibench : A representative and comprehensive hadoop benchmark suite," in *Intel Asia-Pacific Research and Development*, 2012.

[62] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi, "Introduction to the hpc challenge benchmark suite," tech. rep., Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2005.

[63] P. Luszczek and J. J. Dongarra, "Hpc challenge benchmark." https://icl.utk.edu/hpcc/index.html, 2012.

[64] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers." https://www.netlib.org/benchmark/hpl/, Dec 2018.

[65] J. Dongarra, I. Duff, J. D. Croz, and S. Hammarling, "Lapack: Linear algebra package - dgemm." https://www.netlib.org/lapack/explore-html/d1/d54/group__double__blas__level3_gaeda3cbd99c8fb834a60a6412878226e1.html, Feb 1989.

[66] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[67] D. Bailey, M. Baker, M. Berry, J. Dongarra, V. Getov, I. Glendinning, C. Grassl, T. Haupt, T. Hey, R. Hockney, and et al., "Parkbench matrix kernel benchmarks." https://www.netlib.org/parkbench/html/matrix-kernels.html, May 1996.

[68] D. Koester and B. Lucas, "Randomaccess." https://icl.utk.edu/projectsfiles/hpcc/RandomAccess/, 2009.

[69] D. Takahashi, "Ffte: A fast fourier transform package." http://www.ffte.jp/, 2020.

[70] G. Schulz and R. Rabenseifner, "Effective bandwidth (b_eff) benchmark." https://fs.hlrs.de/projects/par/mpi//b_eff/, Feb 2016.

[71] P. Luszczek and J. J. Dongarra, "Hpcc benchmark suite measurements." https://icl.utk.edu/hpcc/faq/index.html#90, Mar 2016.

[72] V. Gueant, "iperf = iperf3 and iperf2 user documentation." https://iperf.fr/iperf-doc.php.

[73] M. Duckett, J. Moisand, T. Anschutz, D. Kourkouzelis, and P. Arberg, "Accommodating a Maximum Transit Unit/Maximum Receive Unit (MTU/MRU) Greater Than 1492 in the Point-to-Point Protocol over Ethernet (PPPoE)." RFC 4638, Sept. 2006.

[74] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "Hpl scalability analysis." https://www.netlib.org/benchmark/hpl/scalability.html, Dec 2018.

[75] C. Neely, G. Brebner, and "yanz-xlnx", "Xilinx/open-nic: Amd opennic project overview." https://github.com/Xilinx/open-nic, 2022.

[76] Open vSwitch, "Open vswitch with dpdk." https://docs.openvswitch.org/en/latest/intro/install/dpdk/, 2016.

Appendix A Preliminary FPGA Experiment

In the experiments of this thesis, OVS-DPDK was used as a kernel-bypass network processor. However, OVS-DPDK mainly prevents traffic from going through the kernel network stack twice (once in the container, once in the host), and instead makes it just once. It does this through active polling of the network output of the containers. It still needs to process the packet in this process, setting correct headers and moving the data around. This can be improved upon by introducing a dedicated FPGA for this task.

The use of the FPGA requires some changes to the kernel network stack in the containers, as these now need to directly interface with the FPGA. However, this will free up some of the cores that were previously used for active polling since most of the heavy lifting of packet processing will be handled by the FPGA instead. This way, the addition of the FPGA will result in more available performance on the system, as well as a better performing network interface, capable of far higher bandwidths.

Xilinx provided software for offloading DPDK processing to the FPGA (Xilinx Alveo U250), which builds upon the AMD OpenNIC Project [75]. Note that Vivado, which is used for programming the FPGA, may not have the correct board data files. These were retrieved from the Xilinx Board Store[1] The OpenNIC project allows for the use of the FPGA for processing network traffic, directly connecting through its 100Gb/s interface to the network. An altered version of DPDK is used to directly interface with the card through the VFIO driver. To do this, the driver used to communicate with the FPGA device over the PCIe bus is altered using a `dpdk-devbind.py` script included with DPDK.

After generating the bitstream file, programming the FPGA and reloading the PCIe link (or rebooting), the FPGA is in principle ready to process traffic. However, some changes are still needed. First, the FPGA needs a separate kernel module to be loaded, which adds the connected network interfaces of the FPGA to the kernel networking stack. This way, regular applications can already benefit from the use of the FPGA. Additionally, the PCI memory needs to be written to, to add queue information as well as configure the CMAC links. This is further described in the AMD OpenNic Project [75] and its sub-projects.

In preliminary results the baseline throughput achieved was $\pm 37 \times 10^6$ packets/s per interface, while using two CPU cores to generate the required load, one per interface. Note that the `pktgen` (included with DPDK) version used would only allow one pinned CPU core per Rx and Tx queue per interface, in addition to one core used for `pktgen` itself. Other assigned cores would be put to sleep due to there being "nothing to do". This load generation is usually carried out by conventional applications (e.g. key-value stores, web servers, etc.). As the interfaces of the FPGA are indirectly exposed to the applications, no intermediate processing through active polling is required. This changes somewhat when a virtualized setup is used, as now the active polling is needed to move packets from the container's network stack to the FPGA's network processing. No further comparative experiments were executed, though this is certainly an interesting avenue for future research, especially when the addition of traffic shaping policies to the FPGA processing is considered.

[1] https://github.com/Xilinx/XilinxBoardStore/tree/2019.2.2/boards/Xilinx/au250

Appendix B Docker Overlay Network with OVS-DPDK

The basis of each experiment is the Docker overlay network, backed by Open vSwitch [22] (OVS) on each host. This vSwitch uses the Data Plane Development Kit to achieve kernel-bypass network processing, resulting in OVS-DPDK [19]. To combine the two, the steps describing how to compile OVS with DPDK support provided by Open vSwitch [76] were followed. This also contains the steps needed to get the virtual switch up and running. This virtual network lays on top of the physical network, allowing containers across nodes to communicate. In this Appendix, the setup of this overlay network (on a CloudLab [20] disk image) is described. In it, the following software versions are used:

- OVS-vSwitchd (Open vSwitch) 2.16.2
- DPDK 20.11.0
- YCSB 0.17.0, modified to provide more elaborate output
- HPCC 1.5.0
- HiBench 7.1.1
- MongoDB 4.4.12
- OpenMPI 4.0.3
- Spark[2] 3.0.0
- Iperf3[3] 3.7
- Docker 20.10.7
- Docker-Compose 2.2.3
- Consul 1.0.6

All software running in containers will be launched using `docker-compose`. The corresponding `yml` files will be provided in a git repository[4] accompanying this thesis. In this repository the necessary commands and scripts for configuring the traffic shaping on the network will be added, as well as any scripts and cronjobs for setting up the system. Lastly, it will contain the instructions for running the experiments, `docker-compose` files, as well as the scripts to generate figures and tables from the experiment results. Note that YCSB needs to be compiled with the provided files in order to get the more elaborate output.

Before any containers can be run, however, first the Docker overlay network will need to be configured, as described in the instructions provided by Intel [43]. Note that the version of OVS-DPDK used differs (amongst others), as the one used in the experiments is compiled using the two previously mentioned versions for OVS and DPDK. The IP addresses used in the instructions can be changed. In the experiments executed in this thesis, two separate networks are used, as mentioned in Section 4.1. All IP addresses therefore are within the same subnet, with primary nodes located on node0 as it is always present. The only exception is the `ovs` docker overlay network, which uses a dedicated IP range. Since all containers communicate using DNS, this range is arbitrary.

When Consul has been configured on the main node, it may be started as a service using the service files included in the accompanying git repository. After that, the Docker service file needs to be altered on all nodes so it points to the node where Consul is present. An example service file has been included in the accompanying git repository.

In order to get the `ovn-docker-overlay-driver`[5] to run within the python3 environment, some changes are needed. On line 87, before `.strip('"')`, insert `.detach()`. Similarly, on line 269, insert `.detach()` after `ret`. The overlay driver may now be run on all nodes using the service provided in the accompanying git repository.

[2] https://github.com/Marcel-Jan/docker-hadoop-spark.git
[3] https://github.com/JasperAH/iperf3-python.git
[4] https://github.com/JasperAH/ts-ds-resources
[5] https://github.com/shettyg/ovn-docker/blob/master/ovn-docker-overlay-driver

It is advisable to give the node on which the north and south bridge connections of OVN have been set a static IP, or to configure DNS to resolve it. Its IP is needed during setup to configure the OVN bridge. The `setup.sh` file contains commands executed at boot in order to automatically configure nodes as well as OVN. Note that the `CENTRAL_IP` here points to the node previously mentioned.

With everything configured and started, the network can be created and tested using the following commands:

```
docker network create -d openvswitch --subnet=192.168.10.0/24 ovs
docker run -it --net=ovs --rm --name=i3-server -p 5201:5201 networkstatic/iperf3 -s
docker run --rm -it --net=ovs --name=i3-client networkstatic/iperf3 -c i3-server
```

Note that if the network is not working, the provided scripts to configure flows for IP and ARP traffic (with or without traffic shaping present) may need to be used. Lastly, if a container is to use the overlay network, it must be started with the `--net=ovs` option, or must specify `networks: ovs` (for each container), as well as `networks: ovs: external: true` in its `docker-compose.yml` file. Examples are given in the accompanying git repository.

Appendix C Complete Key-Value Store Results

In this appendix, the results for the Key-Value Store experiment using YCSB on MongoDB will be shown. MongoDB [50] is a distributed key-value store, which (as the name suggests) allows for storing data based on key-value pairs. This may be nested similar to JSON. A primary node is used as an interface for data-modifying operations, with additional support for data synchronization between secondary nodes in a cluster. This synchronization occurs asynchronously. The YCSB Benchmark suite [59] benchmarks MongoDB on six different workloads ranging from update-heavy, to read-only workloads. They cover various use cases of the key-value store and should result in a representative evaluation of the system. Each experiment uses 1,000,000 operations and records, with maximum box plot whiskers of ± 1.5 IQR.

The experiment was executed on the CloudLab [20] infrastructure, using the xl170 nodes shown in Table 8. These use Mellanox ConnectX-4 NICs, which are supported by DPDK [40], which – combined with Open vSwitch – is used as an on-host kernel bypass network processor. They are connected to a user-managed Mellanox MSN2410-BB2F switch using 10Gb/s Ethernet links.

Node: xl170	
CPU	10-core Intel E5-2640v4 (2.4GHz)
RAM	64GB ECC DDR4-2400 (4x 16GB)
Disk	Intel DC S3520 480 GB 6G SATA SSD
NIC	Two Dual-port Mellanox ConnectX-4 25 GB NIC (PCIe v3.0, 8 lanes)

Table 8: Specifications of the xl170 node [41] on CloudLab [20].

C.1 Workload A: Upload Heavy Workload

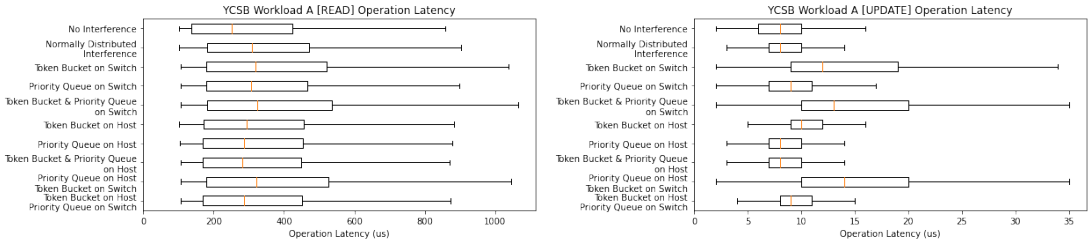

Figure 17: Left: Read latency of YCSB Workload A. Traffic shaping measures generally have little effect, but the on-switch token bucket slightly reduces consistency. Right: Update latency of YCSB Workload A. Traffic shaping measures have hardly any positive effect and are likely to result in worse or less consistent performance. The on-switch token bucket is by far the worst outlier here.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	763	1138	1716
Normally Distributed Interference	843	1209	1821
Token Bucket on Switch	1125	**205567**	**210559**
Priority Queue on Switch	843	1223	1870
Token Bucket & Priority Queue on Switch	1144	**205567**	**210559**
Token Bucket on Host	819	1184	1816
Priority Queue on Host	825	1202	1806
Token Bucket & Priority Queue on Host	812	1179	1761
Priority Queue on Host, Token Bucket on Switch	1135	**205567**	**210303**
Token Bucket on Host, Priority Queue on Switch	821	1202	1868

Table 9: Tail latencies of the read operation of YCSB Workload A. The on-switch token bucket achieves very high tail latencies.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	21	30	61
Normally Distributed Interference	23	30	60
Token Bucket on Switch	33	59	109
Priority Queue on Switch	22	30	65
Token Bucket & Priority Queue on Switch	36	60	98
Token Bucket on Host	24	31	64
Priority Queue on Host	22	31	62
Token Bucket & Priority Queue on Host	22	30	61
Priority Queue on Host, Token Bucket on Switch	36	59	95
Token Bucket on Host, Priority Queue on Switch	23	31	64

Table 10: Tail latencies of the update operation of YCSB Workload A. The on-switch token bucket has slightly higher tails, but generally results are in line with the no interference results.

C.2 Workload B: Read Mostly Workload

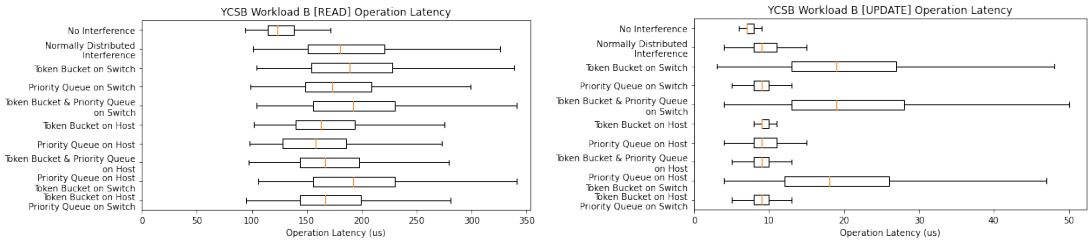

Figure 18: Left: Read latency of YCSB Workload B. Traffic shaping measures generally improve latency, but the on-switch token bucket has little effect. Right: Update latency of YCSB Workload B. Traffic shaping measures may increase consistency, though the on-switch token bucket achieves far worse performance.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	246	360	551
Normally Distributed Interference	440	633	937
Token Bucket on Switch	430	**205695**	**211199**
Priority Queue on Switch	389	536	759
Token Bucket & Priority Queue on Switch	429	**205823**	**211071**
Token Bucket on Host	394	856	1947
Priority Queue on Host	318	449	681
Token Bucket & Priority Queue on Host	391	653	1708
Priority Queue on Host, Token Bucket on Switch	428	**205695**	**210687**
Token Bucket on Host, Priority Queue on Switch	392	626	1610

Table 11: Tail latencies of the read operation of YCSB Workload B. The on-switch token bucket achieves very high tail latencies. The priority queue improves tail latencies over the interference traffic.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	16	33	78
Normally Distributed Interference	22	36	71
Token Bucket on Switch	45	73	150
Priority Queue on Switch	20	36	93
Token Bucket & Priority Queue on Switch	45	72	142
Token Bucket on Host	23	46	79
Priority Queue on Host	21	37	89
Token Bucket & Priority Queue on Host	23	43	91
Priority Queue on Host, Token Bucket on Switch	44	69	136
Token Bucket on Host, Priority Queue on Switch	20	38	86

Table 12: Tail latencies of the update operation of YCSB Workload B. Tail latencies are generally in line with the no interference latencies, though on-host token buckets achieve slightly higher tails still.

C.3 Workload C: Read Only Workload

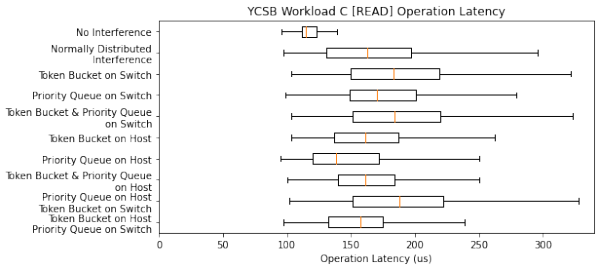

Figure 19: Read latency of YCSB Workload C. Traffic shaping measures generally improve latency, but the on-switch token bucket has little effect and slightly decreases consistency.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	143	174	427
Normally Distributed Interference	289	510	1064
Token Bucket on Switch	341	**205695**	**211071**
Priority Queue on Switch	284	479	624
Token Bucket & Priority Queue on Switch	340	**205823**	**210815**
Token Bucket on Host	329	650	1491
Priority Queue on Host	232	495	2009
Token Bucket & Priority Queue on Host	258	588	1667
Priority Queue on Host, Token Bucket on Switch	347	**205823**	**211327**
Token Bucket on Host, Priority Queue on Switch	222	388	601

Table 13: Tail latencies of the read operation of YCSB Workload C. The on-switch token bucket achieves very high tail latencies. The on-switch token bucket improves tail latencies slightly.

C.4 Workload D: Read Latest Workload

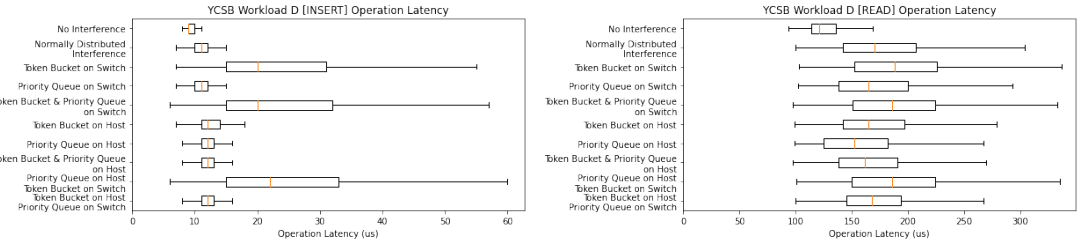

Figure 20: Left: Insert latency of YCSB Workload D. Traffic shaping measures have little effect on consistency, though the on-switch token bucket achieves far worse performance. Right: Read latency of YCSB Workload D. Traffic shaping measures generally improve latency, but the on-switch token bucket has little effect and slightly decreases consistency.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	205	287	485
Normally Distributed Interference	378	688	1513
Token Bucket on Switch	400	**205695**	**210559**
Priority Queue on Switch	376	535	759
Token Bucket & Priority Queue on Switch	432	**205823**	**211071**
Token Bucket on Host	419	644	1096
Priority Queue on Host	304	528	1716
Token Bucket & Priority Queue on Host	377	585	1714
Priority Queue on Host, Token Bucket on Switch	436	**205823**	**211327**
Token Bucket on Host, Priority Queue on Switch	344	559	868

Table 14: Tail latencies of the read operation of YCSB Workload D. The on-switch token bucket achieves far worse tail latencies, though the on-switch priority queue improves them slightly.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	18	30	123
Normally Distributed Interference	26	39	129
Token Bucket on Switch	46	76	206
Priority Queue on Switch	26	37	115
Token Bucket & Priority Queue on Switch	47	78	166
Token Bucket on Host	27	38	107
Priority Queue on Host	27	43	165
Token Bucket & Priority Queue on Host	27	43	100
Priority Queue on Host, Token Bucket on Switch	48	79	198
Token Bucket on Host, Priority Queue on Switch	27	41	122

Table 15: Tail latencies of the insert operation of YCSB Workload D. The on-switch token bucket and on-host priority queue slightly increase tail latencies while the other measures generally have little effect.

C.5 Workload E: Short Ranges Workload

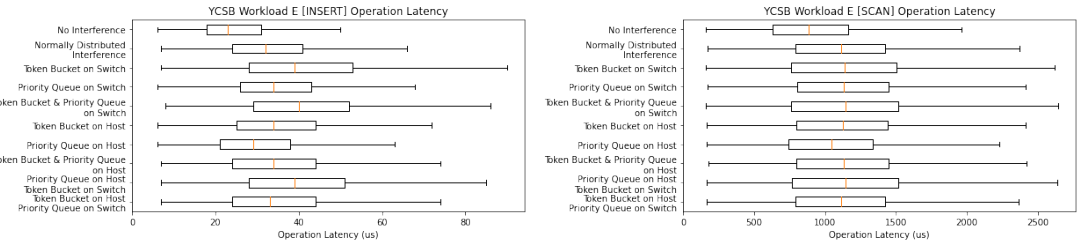

Figure 21: Left: Insert latency of YCSB Workload E. Traffic shaping measures may slightly decrease consistency, though the on-switch token bucket reduces consistency to a slightly greater degree. Right: Scan latency of YCSB Workload E. Traffic shaping measures generally have little effect, but the on-switch token bucket slightly reduces consistency.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	43	60	171
Normally Distributed Interference	55	73	216
Token Bucket on Switch	69	92	223
Priority Queue on Switch	57	79	213
Token Bucket & Priority Queue on Switch	68	94	210
Token Bucket on Host	60	79	157
Priority Queue on Host	51	75	172
Token Bucket & Priority Queue on Host	60	82	196
Priority Queue on Host, Token Bucket on Switch	68	92	212
Token Bucket on Host, Priority Queue on Switch	59	78	170

Table 16: Tail latencies of the insert operation of YCSB Workload E. On-host traffic shaping improve tail latencies slightly.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	1448	1680	2507
Normally Distributed Interference	1933	2809	3633
Token Bucket on Switch	3641	**71743**	**222975**
Priority Queue on Switch	1961	2833	3657
Token Bucket & Priority Queue on Switch	6407	**75775**	**223359**
Token Bucket on Host	1906	2871	3875
Priority Queue on Host	1788	2573	3523
Token Bucket & Priority Queue on Host	1966	3055	3949
Priority Queue on Host, Token Bucket on Switch	13063	81087	**223103**
Token Bucket on Host, Priority Queue on Switch	1936	2889	3753

Table 17: Tail latencies of the scan operation of YCSB Workload E. The on-switch token bucket achieves very high tail latencies. Other traffic shaping measures have little effect.

C.6 Workload F: Read-Modify-Write Workload

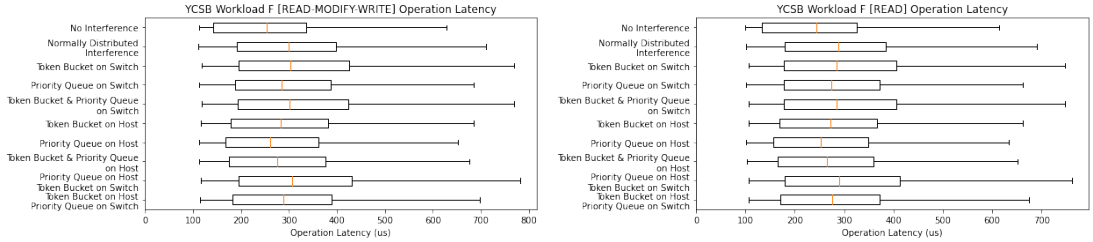

Figure 22: Left: Read-modify-write latency of YCSB Workload F. Right: Read latency of YCSB Workload F. In both cases, the traffic shaping measures may very slightly improve consistency, though they generally have little effect. The on-switch token bucket will reduce consistency slightly, however.

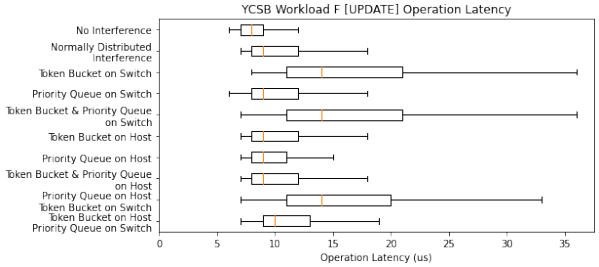

Figure 23: Update latency of YCSB Workload F. Traffic shaping measures generally have little effect, though the on-host token bucket may improve consistency slightly. The on-switch token bucket on the other hand makes the performance far worse.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	398	470	665
Normally Distributed Interference	523	927	1621
Token Bucket on Switch	627	**205055**	**209791**
Priority Queue on Switch	467	651	1111
Token Bucket & Priority Queue on Switch	629	**205183**	**210687**
Token Bucket on Host	518	911	1524
Priority Queue on Host	426	559	846
Token Bucket & Priority Queue on Host	469	765	1378
Priority Queue on Host, Token Bucket on Switch	634	**205183**	**210175**
Token Bucket on Host, Priority Queue on Switch	497	785	1628

Table 18: Tail latencies of the read operation of YCSB Workload F. Priority queues slightly improve tail latencies, but on-switch token buckets have far worse tail latencies by comparison.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	414	488	686
Normally Distributed Interference	539	951	1645
Token Bucket on Switch	655	**205055**	**210175**
Priority Queue on Switch	483	665	1150
Token Bucket & Priority Queue on Switch	656	**205183**	**210815**
Token Bucket on Host	537	937	1579
Priority Queue on Host	441	574	866
Token Bucket & Priority Queue on Host	487	780	1429
Priority Queue on Host, Token Bucket on Switch	661	**205183**	**209919**
Token Bucket on Host, Priority Queue on Switch	514	806	1672

Table 19: Tail latencies of the read-modify-write operations of YCSB Workload F. The use of priority queues slightly improves tail latencies, though the addition of an on-host token bucket makes tail latencies far worse.

Experiment Runtime (µs)	**P95**	**P99**	**P99.9**
No Interference	18	29	44
Normally Distributed Interference	24	33	56
Token Bucket on Switch	32	58	90
Priority Queue on Switch	23	30	52
Token Bucket & Priority Queue on Switch	32	58	89
Token Bucket on Host	25	34	63
Priority Queue on Host	23	30	50
Token Bucket & Priority Queue on Host	24	33	62
Priority Queue on Host, Token Bucket on Switch	32	58	89
Token Bucket on Host, Priority Queue on Switch	25	34	63

Table 20: Tail latencies of the update operation of YCSB Workload F. Traffic shaping measures generally have little effect on the tail latencies, though the on-switch token bucket slightly increases them.

Appendix D Complete Big Data Results

In this appendix, the results for the Big Data experiment using Apache Spark and HiBench will be shown. Apache Spark [51] is a system for executing Big Data workloads such as a word frequency counter, implemented as a Map-Reduce workload. This kind of workload is split into a "Map" phase (counting words) and a "Reduce" phase (aggregation of results). Apache Spark is able to use the Hadoop Distributed File System (HDFS) [54] which is a robust file system capable of handling very large datasets across multiple heterogeneous nodes, and is resilient to hardware failure. This also allows Apache Spark to use data locality to improve performance. The HiBench Benchmark suite [61] comprises workloads from web search, machine learning and analytical query domains. The benchmark was first run on the `tiny` dataset, after which the Terasort and (partially) Wordcount "micro" benchmarks were run. The full benchmark was only run using no interference, with interference and on-switch traffic shaping measures, to get an indication on what applications might show a difference in results. The Wordcount benchmark used a data size of 500,000,000,000 and no experiments with token bucket and priority queue on separate hosts were executed for this benchmark. The Terasort results have been included in Section 5.2. The benchmarks use default settings, are allowed access to all CPU cores and 4GB of memory, and were executed using 10 runs. The box plots have maximum whiskers of ± 1.5 IQR.

The experiment was executed on the CloudLab [20] infrastructure, using the xl170 nodes shown in Table 21. These use Mellanox ConnectX-4 NICs, which are supported by DPDK [40], which – combined with Open vSwitch – is used as an on-host kernel bypass network processor. They are connected to a user-managed Mellanox MSN2410-BB2F switch using 10Gb/s Ethernet links.

Node: xl170	
CPU	10-core Intel E5-2640v4 (2.4GHz)
RAM	64GB ECC DDR4-2400 (4x 16GB)
Disk	Intel DC S3520 480 GB 6G SATA SSD
NIC	Two Dual-port Mellanox ConnectX-4 25 GB NIC (PCIe v3.0, 8 lanes)

Table 21: Specifications of the xl170 node [41] on CloudLab [20].

D.1 Result on `tiny` dataset

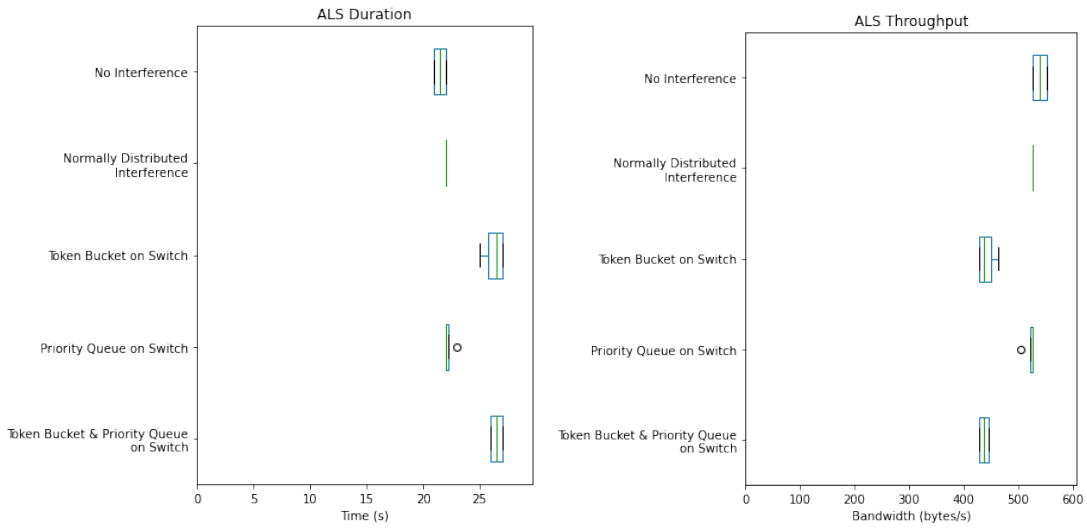

Figure 24: Left: The duration of the HiBench ALS benchmark. Right: The throughput of the HiBench ALS benchmark. The token bucket has a negative influence on performance, reducing throughput and therefore increasing duration.

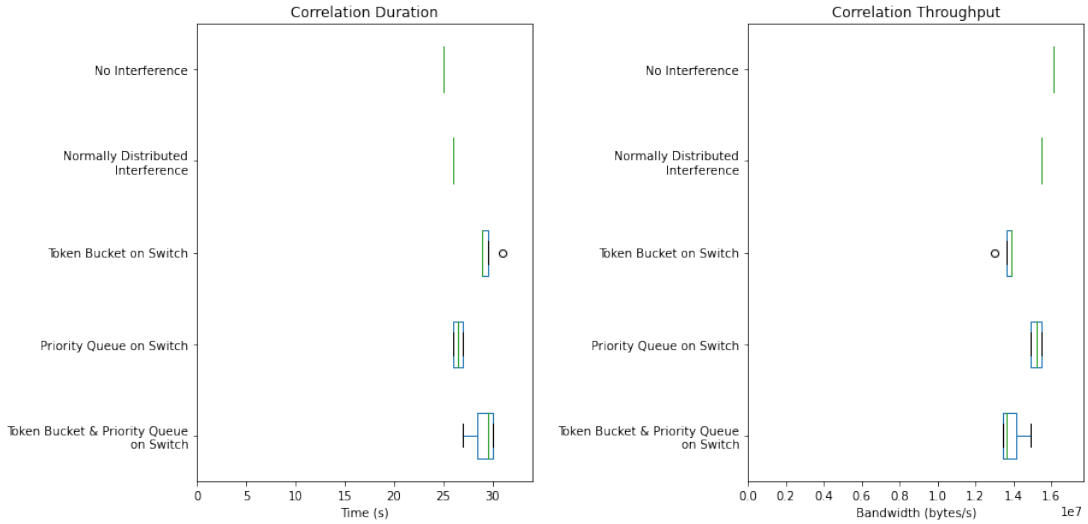

Figure 25: Left: The duration of the HiBench Correlation benchmark. Right: The throughput of the HiBench Correlation benchmark. The on-switch token bucket appears to have a slight negative influence on throughput (and therefore, duration). The difference is too small to be conclusive with the current data, which was based on 10 runs.

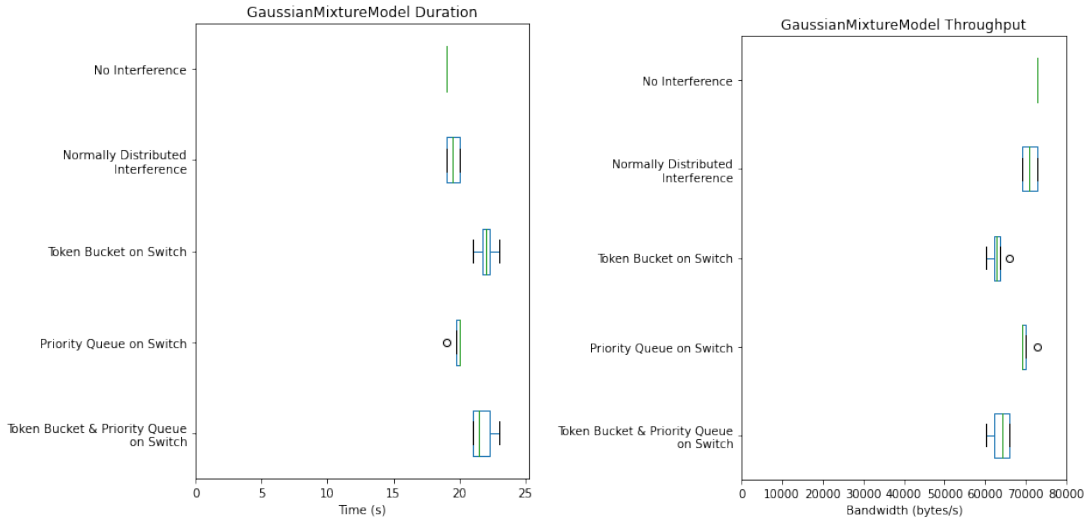

Figure 26: Left: The duration of the HiBench Gaussian Mixture Model benchmark. Right: The throughput of the HiBench Gaussian Mixture Model benchmark. This benchmark is very slightly affected by the on-host token bucket, leading to a reduction in performance and consistency.

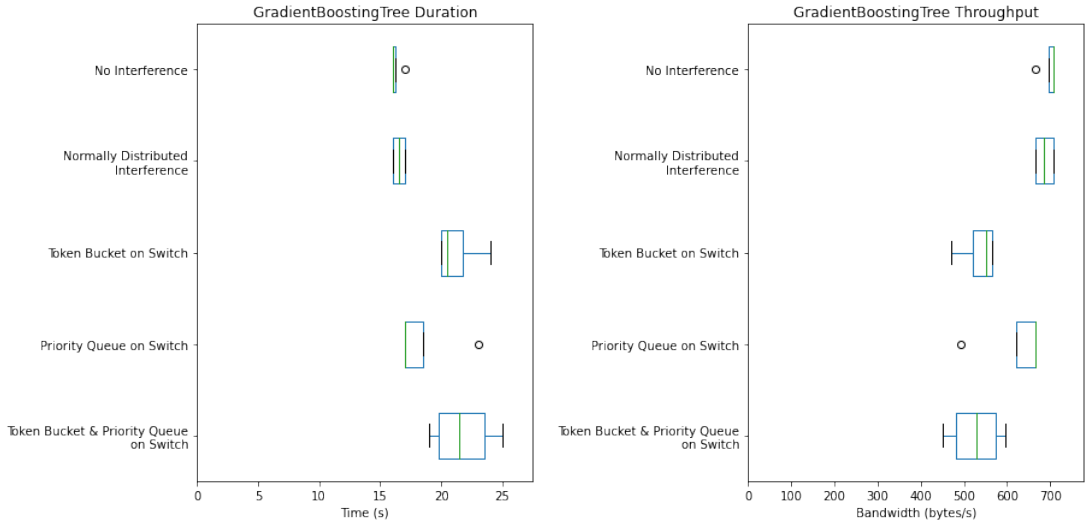

Figure 27: Left: The duration of the HiBench Gradient Boosting Tree benchmark. Right: The throughput of the HiBench Gradient Boosting Tree benchmark. The addition of the on-switch token bucket decreases consistency when compared to the results achieved using only the normally distributed interference.

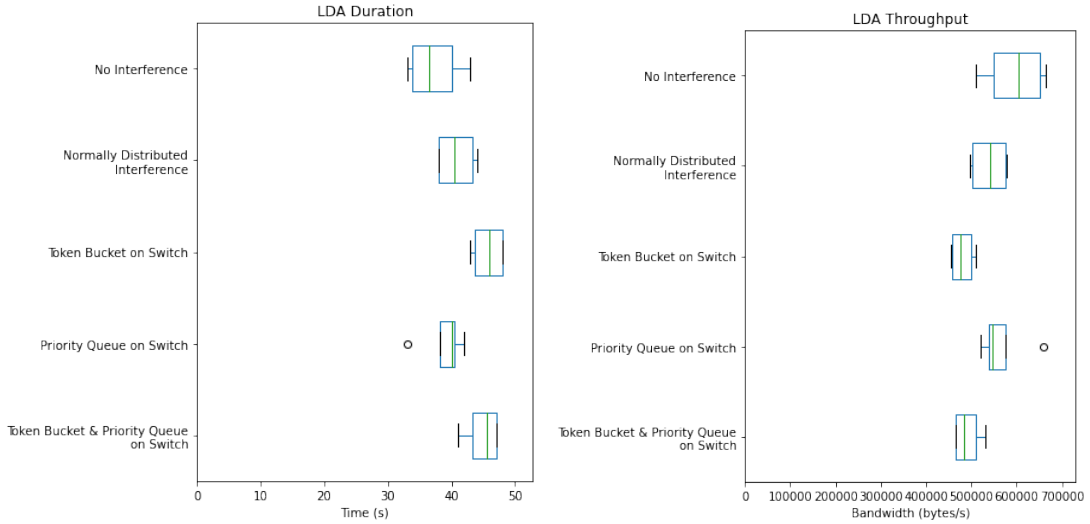

Figure 28: Left: The duration of the HiBench LDA benchmark. Right: The throughput of the HiBench LDA benchmark. The difference in benchmark results is hardly conclusive given the fact that there have only been 10 runs. Given the fact that the results without any interference are quite inconsistent, the distance between them and the other results are too small to be conclusive.

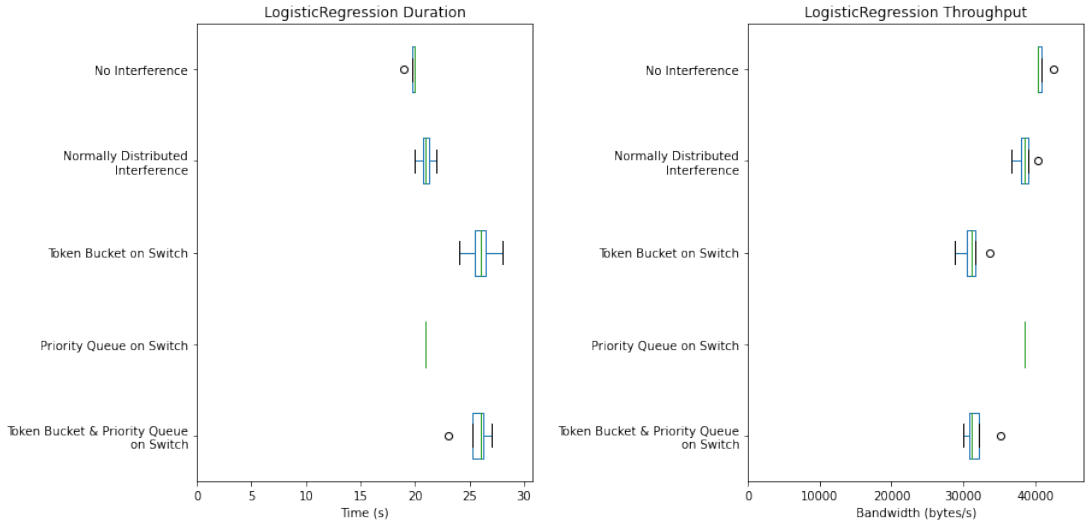

Figure 29: Left: The duration of the HiBench Logistic Regression benchmark. Right: The throughput of the HiBench Logistic Regression benchmark. The addition of the on-switch token bucket reduces throughput, which increases duration. The difference in consistency, when compared to the results achieved using only the normally distributed interference, is negligible.

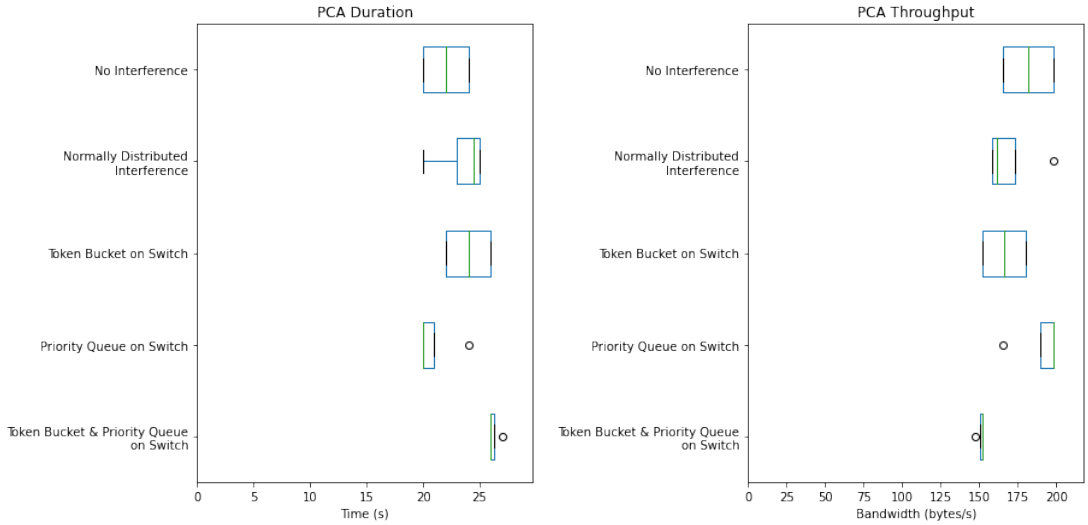

Figure 30: Left: The duration of the HiBench PCA benchmark. Right: The throughput of the HiBench PCA benchmark. The experiments show little difference in their results, with the exception of the priority queue (possibly combined with the token bucket). The former increases consistency towards the lower end of the result range, while the latter increases consistency towards the higher end of the result range, when compared to the normally distributed interference.

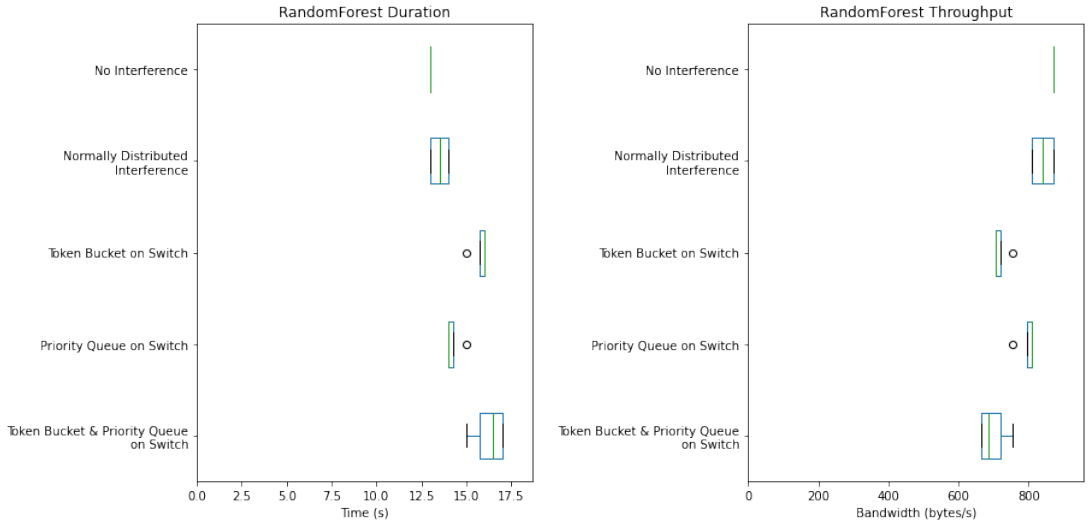

Figure 31: Left: The duration of the HiBench Random Forest benchmark. Right: The throughput of the HiBench Random Forest benchmark. The addition of the on-switch token bucket slightly reduces performance. This effect gets partially reduced – through decreasing consistency – by adding a token bucket.

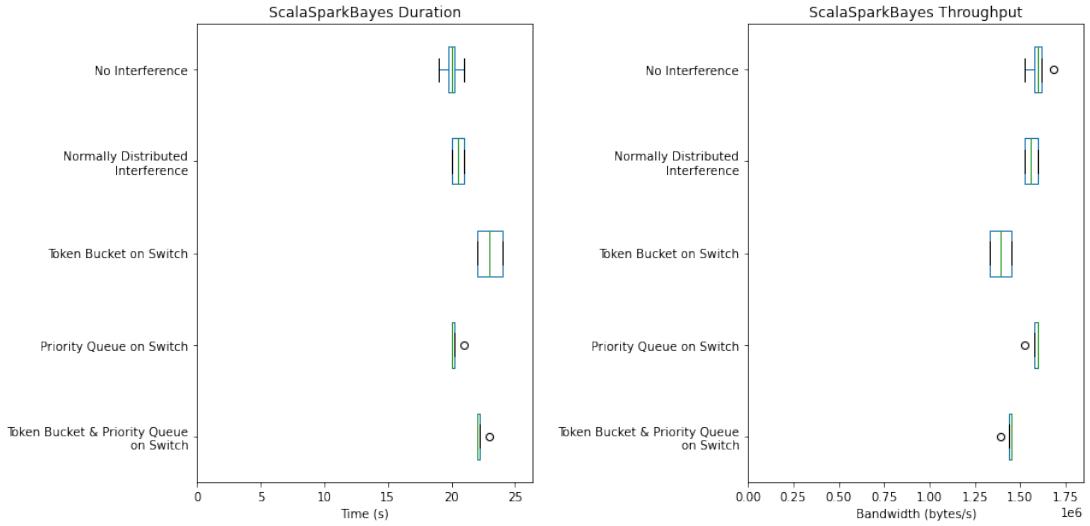

Figure 32: Left: The duration of the HiBench Scala Spark Bayes benchmark. Right: The throughput of the HiBench Scala Spark Bayes benchmark. The on-switch token bucket appears to have a minor negative influence on performance, which gets slightly alleviated by the addition of the priority queue.

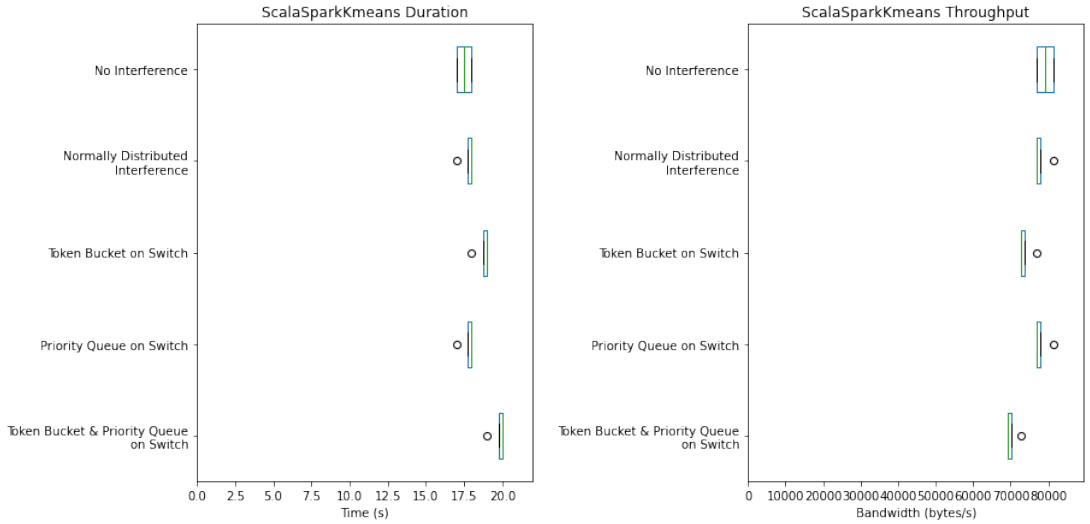

Figure 33: Left: The duration of the HiBench Scala Spark K-means benchmark. Right: The throughput of the HiBench Scala Spark K-means benchmark. With the number of runs, the results hardly differ, though the use of both the priority queue and token bucket appears to have a minor negative influence on performance.

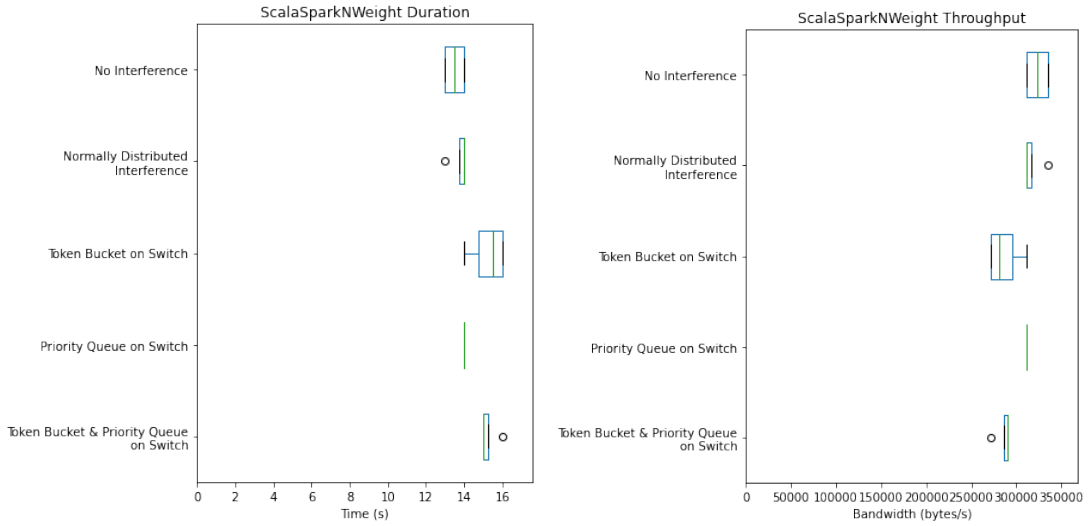

Figure 34: Left: The duration of the HiBench Scala Spark N-weight benchmark. Right: The throughput of the HiBench Scala Spark N-weight benchmark. The on-switch token bucket appears to have a negative influence on performance consistency, which is slightly improved by the addition of the priority queue.

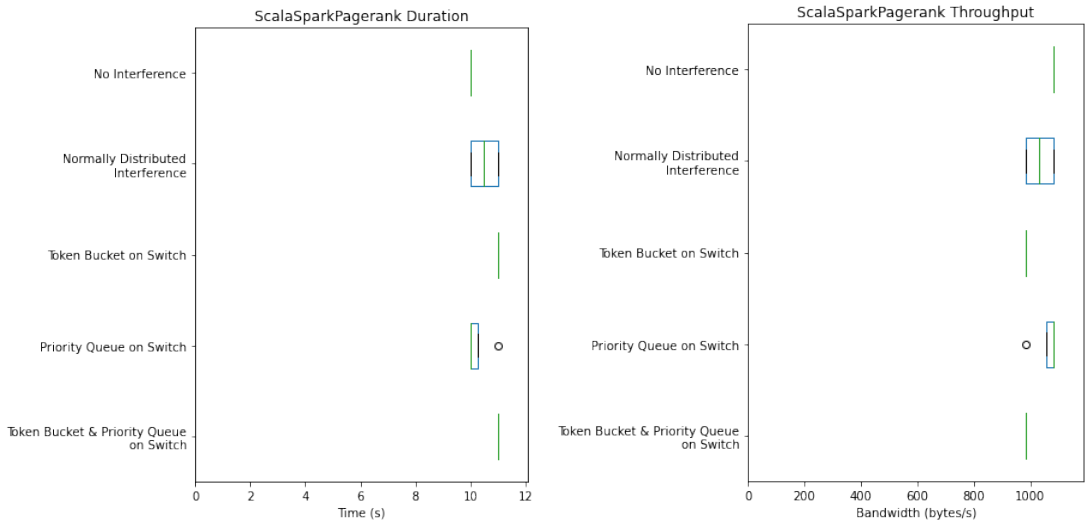

Figure 35: Left: The duration of the HiBench Scala Spark PageRank benchmark. Right: The throughput of the HiBench Scala Spark PageRank benchmark. The results achieved using the on-switch token bucket appear very consistent, with their performance being close to that achieved by the normally distributed interference experiment.

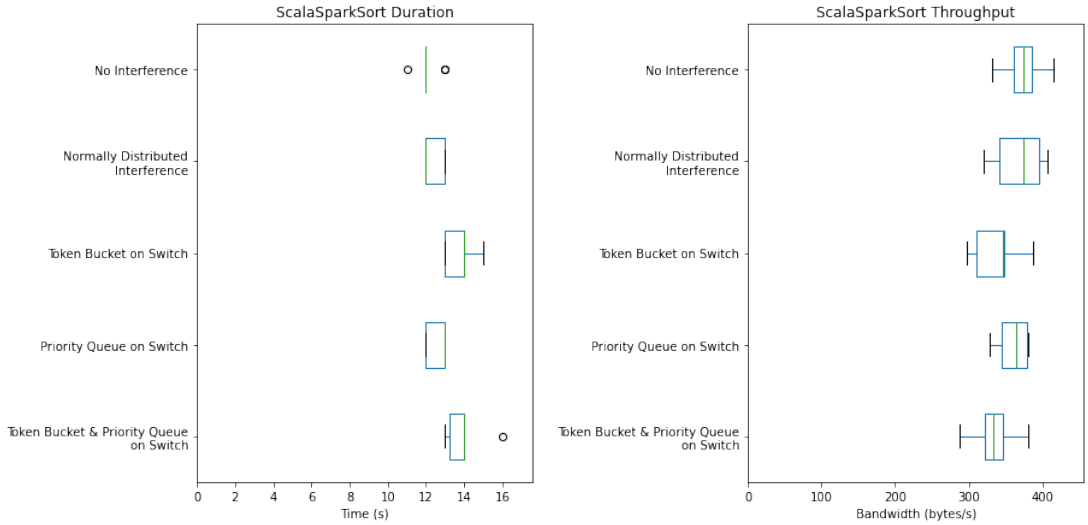

Figure 36: Left: The duration of the HiBench Scala Spark Sort benchmark. Right: The throughput of the HiBench Scala Spark Sort benchmark. Interestingly the results show a slight difference between the duration and throughput in this benchmark. This is likely because of the lack of throughput (i.e. hardly any reliance on the network). Given the inconsistency of results, however, the difference between the results seems very small.

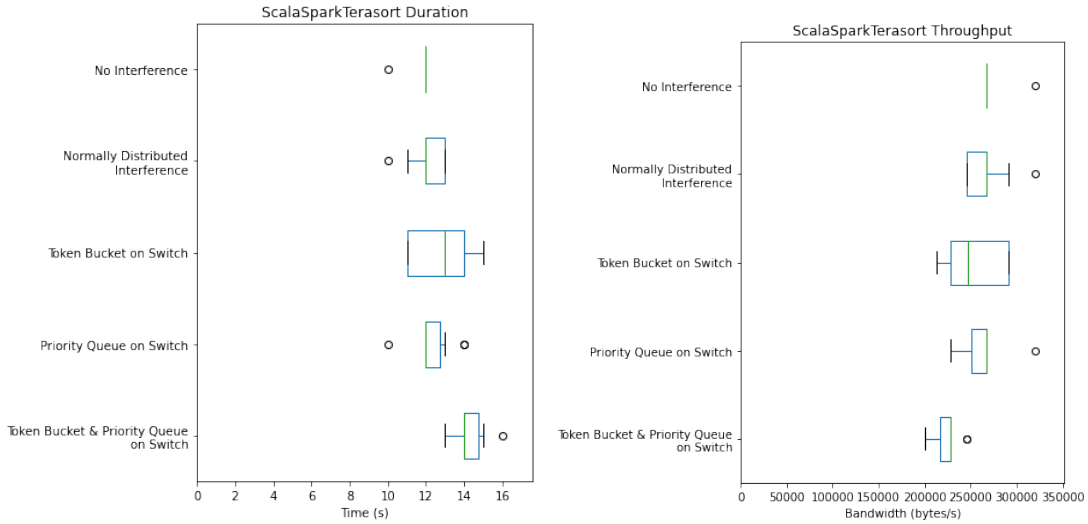

Figure 37: Left: The duration of the HiBench Scala Spark Terasort benchmark. Right: The throughput of the HiBench Scala Spark Terasort benchmark. Contrary to the Sort benchmark, Terasort does appear to rely heavily on the network, showing a decrease in performance as more traffic shaping measures are added.

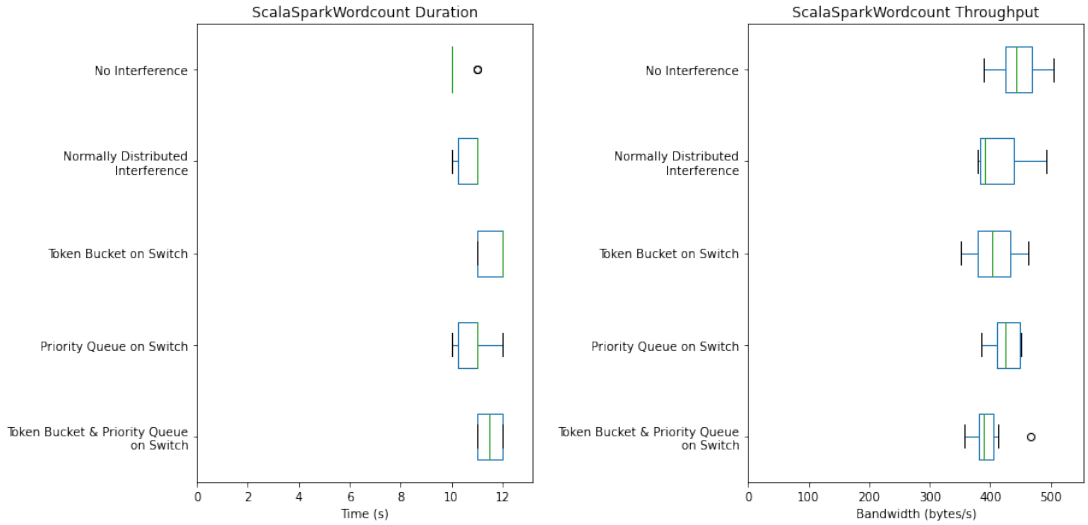

Figure 38: Left: The duration of the HiBench Scala Spark Wordcount benchmark. Right: The throughput of the HiBench Scala Spark Wordcount benchmark. This benchmark shows some influence from the addition of traffic shaping, when compared to the normally distributed interference experiment. It is not very dependent on the network, however.

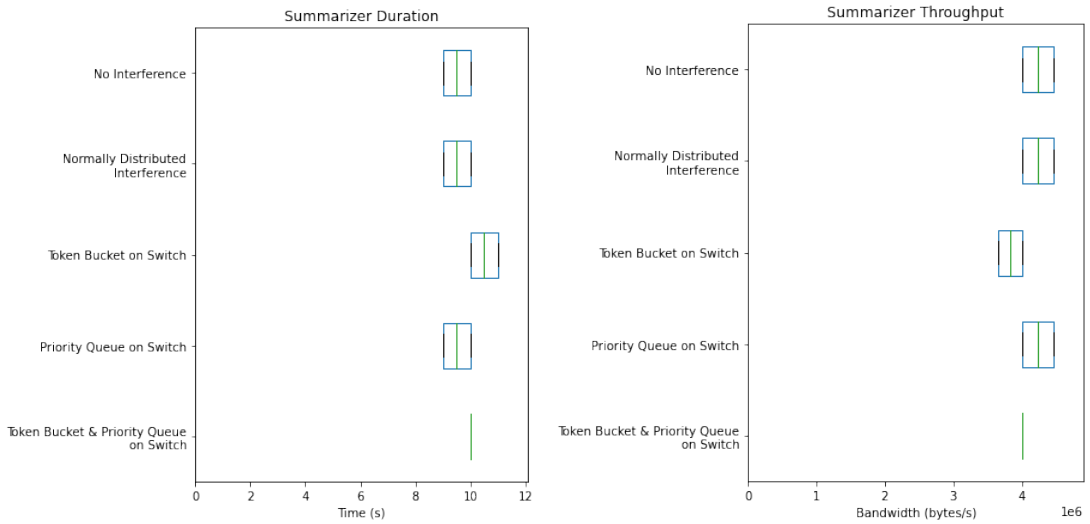

Figure 39: Left: The duration of the HiBench Summarizer benchmark. Right: The throughput of the HiBench Summarizer benchmark. This benchmark appears to have only a minor influence from the addition of the on-switch token bucket, though the difference seems very minor.

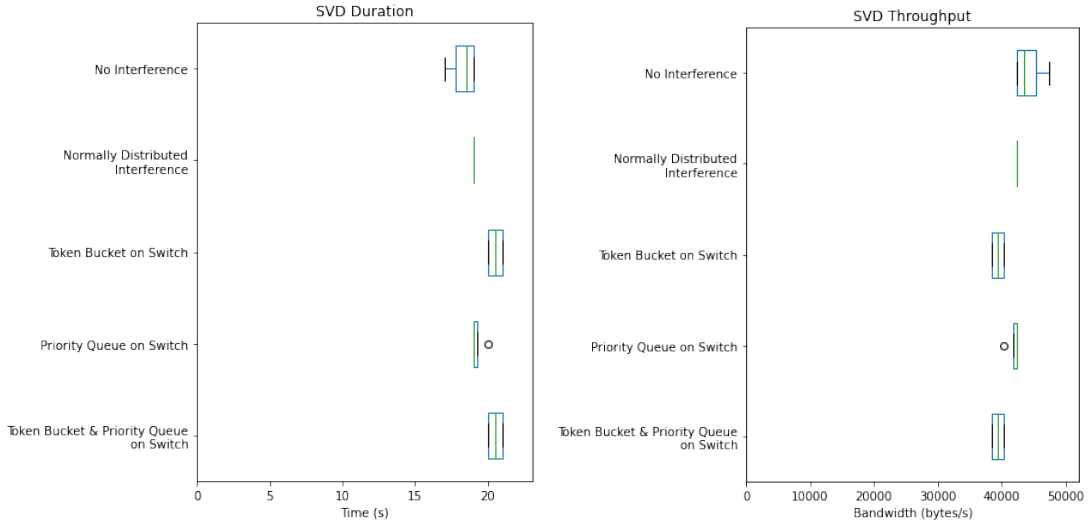

Figure 40: Left: The duration of the HiBench SVD benchmark. Right: The throughput of the HiBench SVD benchmark. While the addition of interference traffic reduces performance when compared to the baseline, the addition of traffic shaping seems to have little additional effect.

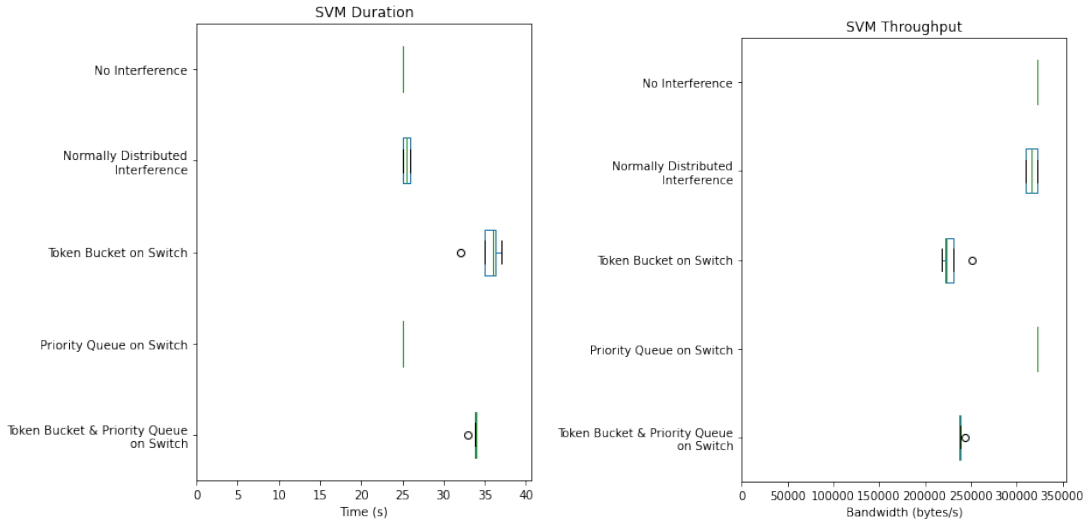

Figure 41: Left: The duration of the HiBench SVM benchmark. Right: The throughput of the HiBench SVM benchmark. In this benchmark, the addition of the on-switch token bucket appears to have a large negative influence on performance when compared to the normally distributed interference, though consistency does not change much.

D.2 Wordcount results

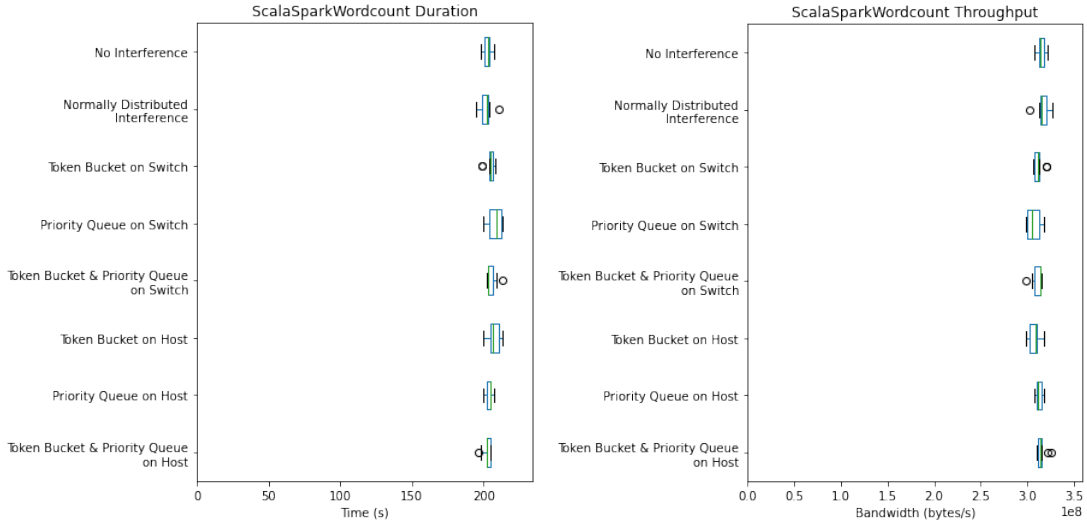

Figure 42: Left: The duration of the HiBench Wordcount benchmark. Right: The throughput of the HiBench Wordcount benchmark. Both results are related (i.e. a higher throughput leads to a lower duration). Little influence is seen from the addition of traffic shaping measures, suggesting that the wordcount micro benchmark is not very reliant on the underlying network.

D.3 Terasort results

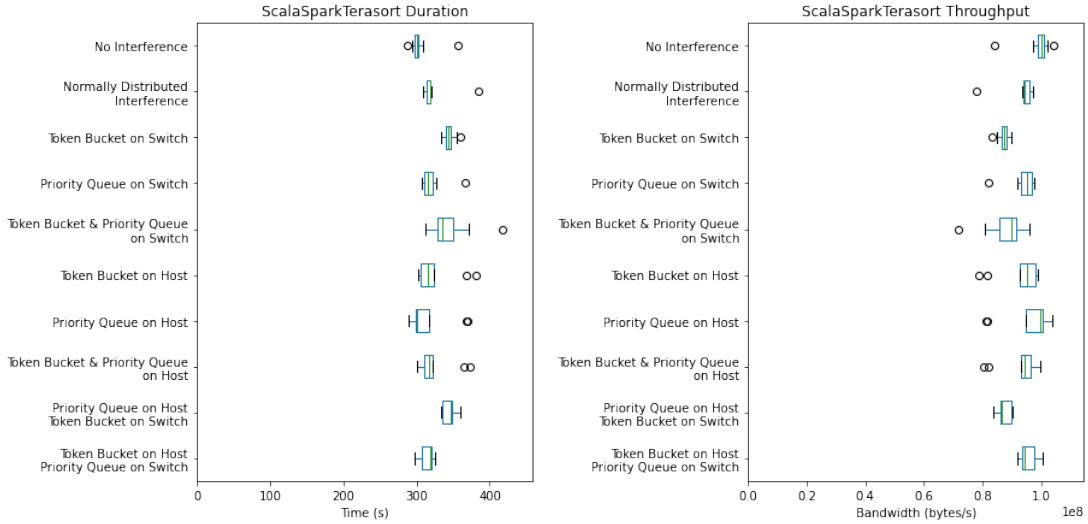

Figure 43: Throughput (left) and Duration (right) of the Terasort experiment under the effects of different traffic shaping settings. Only the combination of on-switch priority queue and token bucket shows a decrease in consistency.

Appendix E Complete High Performance Computing Results

In this appendix, the results for the High Performance Computing experiment utilizing HPCC and MPI will be shown. MPI (Message Passing Interface) [52] is a library used in distributed applications, allowing for the deployment of compute across multiple heterogeneous systems. When starting an MPI job, the number of processes and specific hosts may be specified. MPI will then automatically start these processes and execute the job. The HPCC Benchmark suite [62] consists of a set of standardized benchmarks, and latency and bandwidth related benchmarks. These cover a range of workloads which allow for the evaluation of the performance of a system across multiple facets. The benchmark is run 100 times using "base run" settings, which prohibit the modification of source code. Each node is configured to have eight slots with a maximum of ten using a `hostfile`, with no imposed memory constraints. The box plots have maximum whiskers of $\pm 1.5\,\mathrm{IQR}$.

The experiment was executed on the CloudLab [20] infrastructure, using the xl170 nodes shown in Table 22. These use Mellanox ConnectX-4 NICs, which are supported by DPDK [40], which – combined with Open vSwitch – is used as an on-host kernel bypass network processor. They are connected to a user-managed Mellanox MSN2410-BB2F switch using 10Gb/s Ethernet links.

Node: xl170	
CPU	10-core Intel E5-2640v4 (2.4GHz)
RAM	64GB ECC DDR4-2400 (4x 16GB)
Disk	Intel DC S3520 480 GB 6G SATA SSD
NIC	Two Dual-port Mellanox ConnectX-4 25 GB NIC (PCIe v3.0, 8 lanes)

Table 22: Specifications of the xl170 node [41] on CloudLab [20].

E.1 Negative Token Bucket Influence

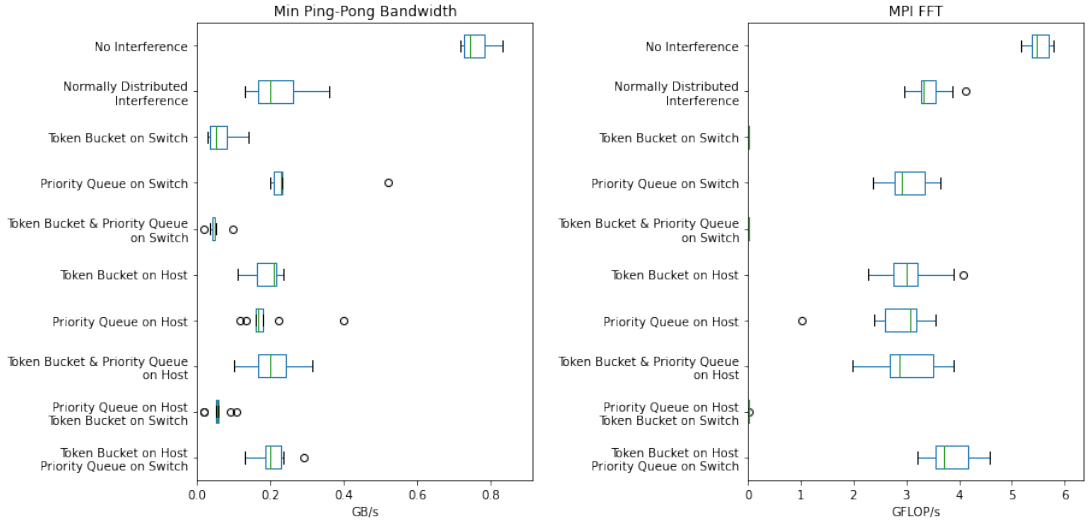

Figure 44: Left: The result of the HPCC Minimum Ping-Pong Bandwidth benchmark. Results with on-switch token buckets show slightly worse performance when compared to the baseline and other results. Right: The result of the HPCC MPI FFT benchmark. Results with the on-switch token buckets are far worse than the other results.

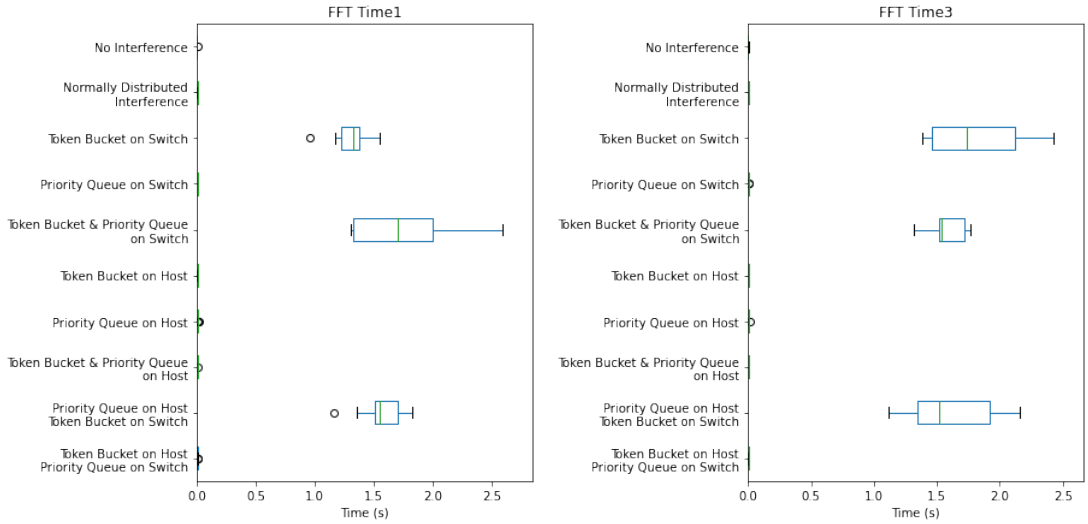

Figure 45: Left: The results of the HPCC FFT Time1 benchmark. Right: The results of the HPCC FFT Time3 benchmark. Both benchmarks appear to be partial measurements of the FFT benchmark, and both show a large degradation in performance and consistency when the on-switch token bucket is added, when comparing to the other results.

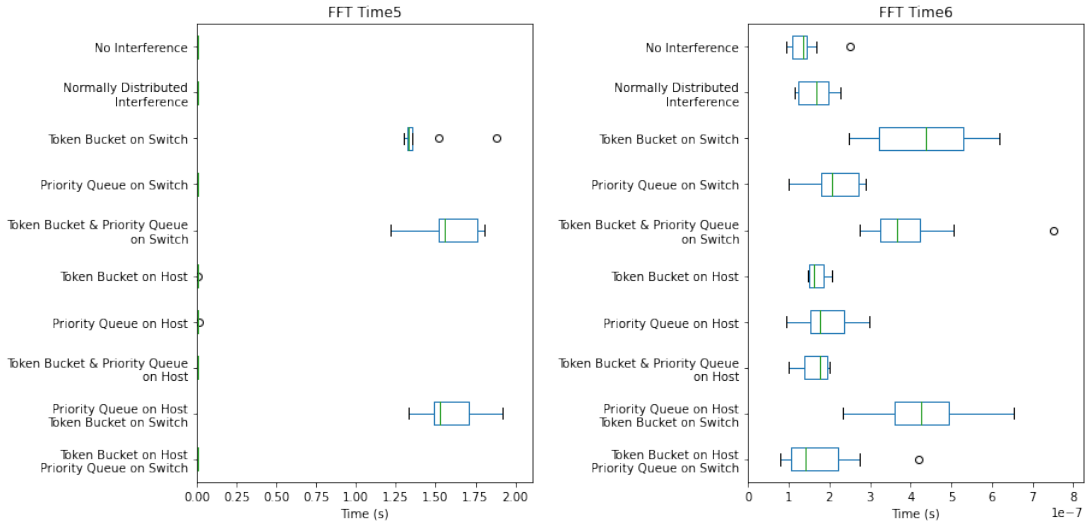

Figure 46: Left: The results of the HPCC FFT Time5 benchmark, which like the Time1 and Time3 benchmarks, shows a large decrease in performance when the on-switch token bucket is added. The addition of a priority queue furthermore decreases consistency. Right: The results of the HPCC FFT Time6 benchmark. Its results are far closer than the other FFT partial results, though the addition of the on-switch token bucket still has a negative impact on performance when compared to the baseline.

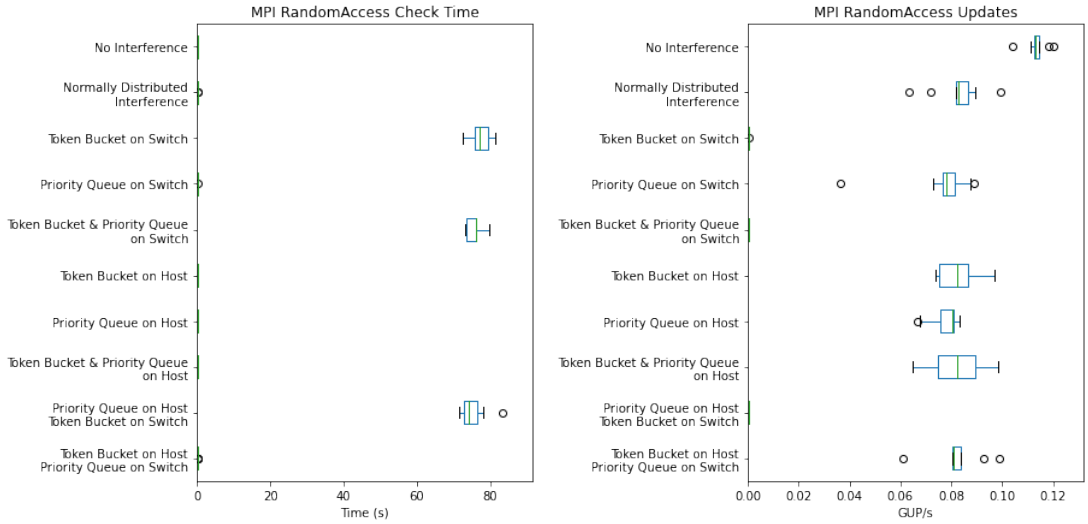

Figure 47: Left: The results of the HPCC MPI RandomAccess Check Time benchmark. Right: The results of the HPCC MPI RandomAccess Updates benchmark. Both results show very large discrepancies upon the addition of the on-switch token bucket.

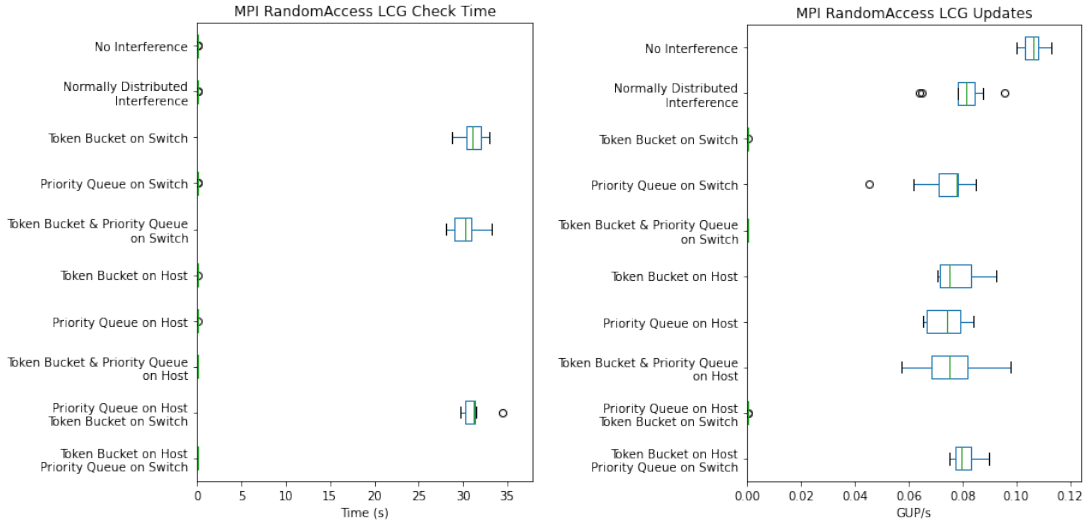

Figure 48: Left: The results of the HPCC MPI RandomAccess LCG Check Time benchmark. Right: The results of the HPCC MPI RandomAccess LCG Updates benchmark. Similar to the previous results, these benchmarks show that the addition of the on-switch token bucket has a negative impact on performance, when compared to the baseline where only the normally distributed interference is present.

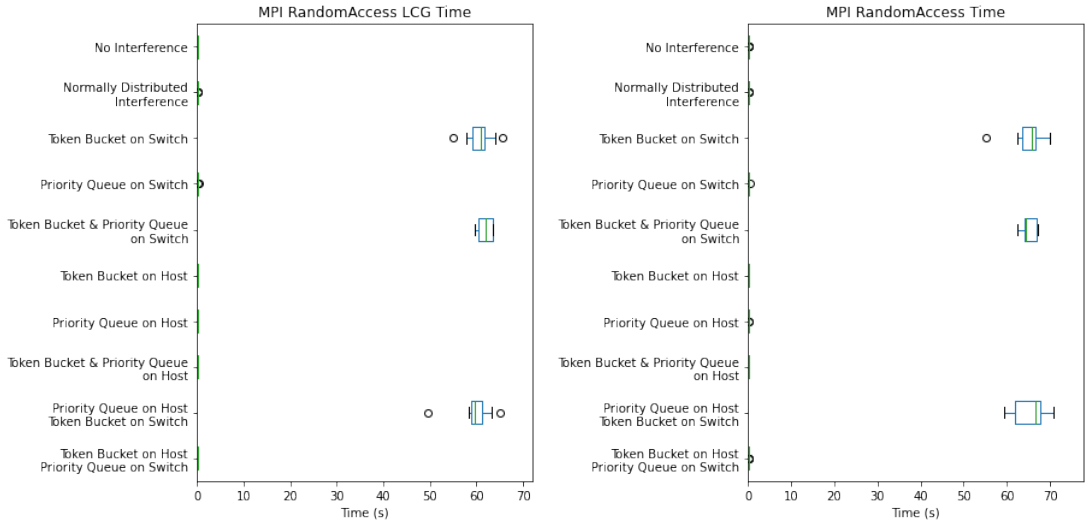

Figure 49: Left: The results of the HPCC MPI RandomAccess LCG Time benchmark. Right: The results of the HPCC MPI RandomAccess Time benchmark. The required time for each benchmark greatly increases when the on-switch token bucket is added.

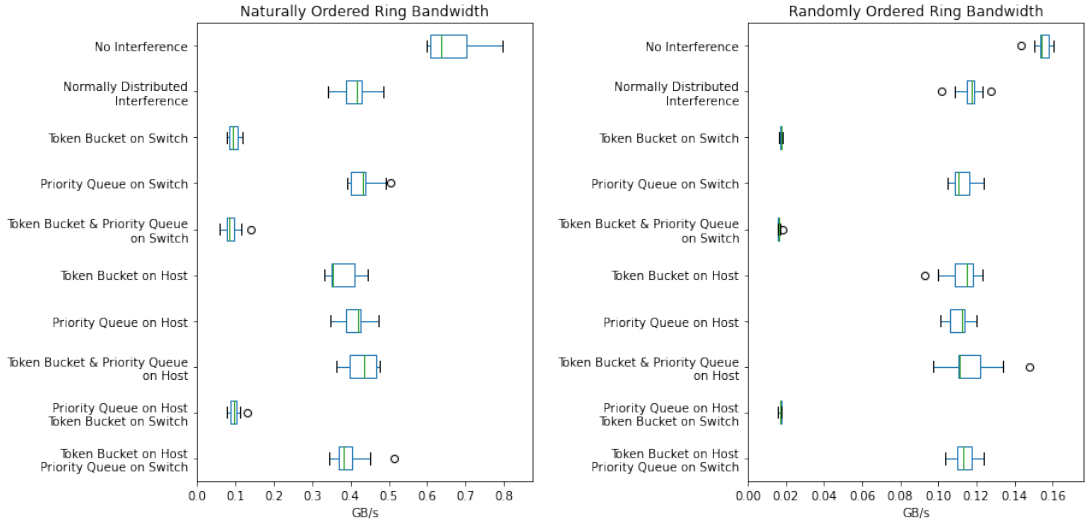

Figure 50: Left: The results of the HPCC Naturally Ordered Ring Bandwidth benchmark. Right: The results of the HPCC Randomly Ordered Ring Bandwidth benchmark. Both benchmarks specifically measure bandwidth, thus it is unsurprising that the token bucket negatively impacts performance, especially when the contention from the interference traffic is added.

E.2 No Difference in Results

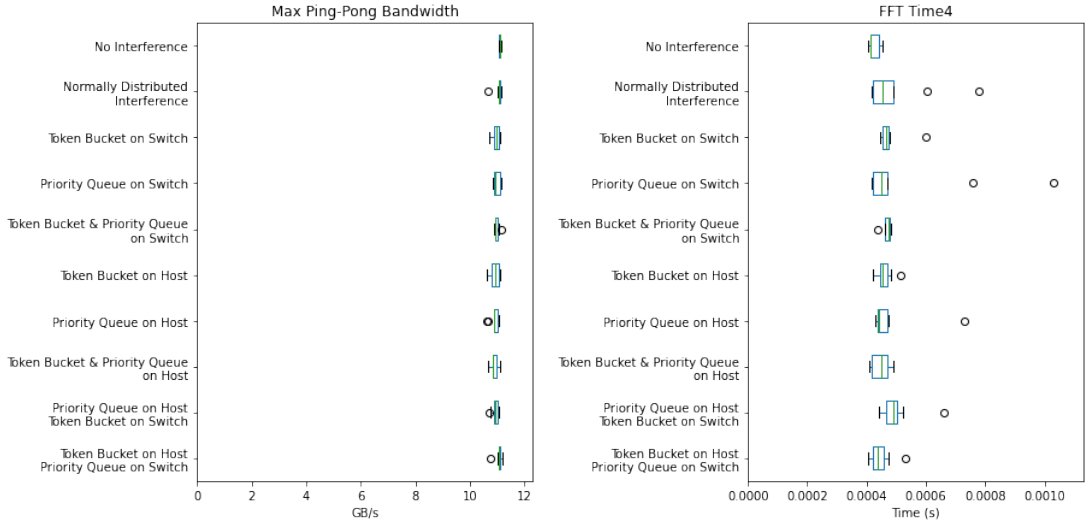

Figure 51: Left: The results of the HPCC Max Ping-Pong Bandwidth benchmark. Right: The results of the HPCC FFT Time4 benchmark. While the FFT Time4 benchmark does show some outliers, neither of the results show large differences between the baseline and the addition of traffic shaping.

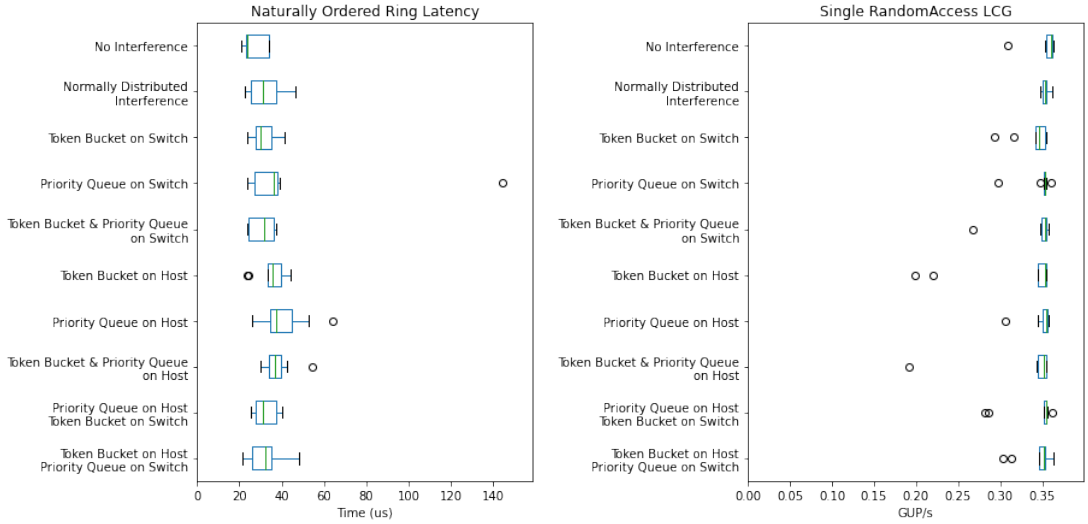

Figure 52: Left: The results of the HPCC Naturally Ordered Ring Latency benchmark. Right: The results of the HPCC Single RandomAccess LCG benchmark. The former shows some small differences in consistency, while the second shows some outliers. However, in general hardly any differences are observed between the results.

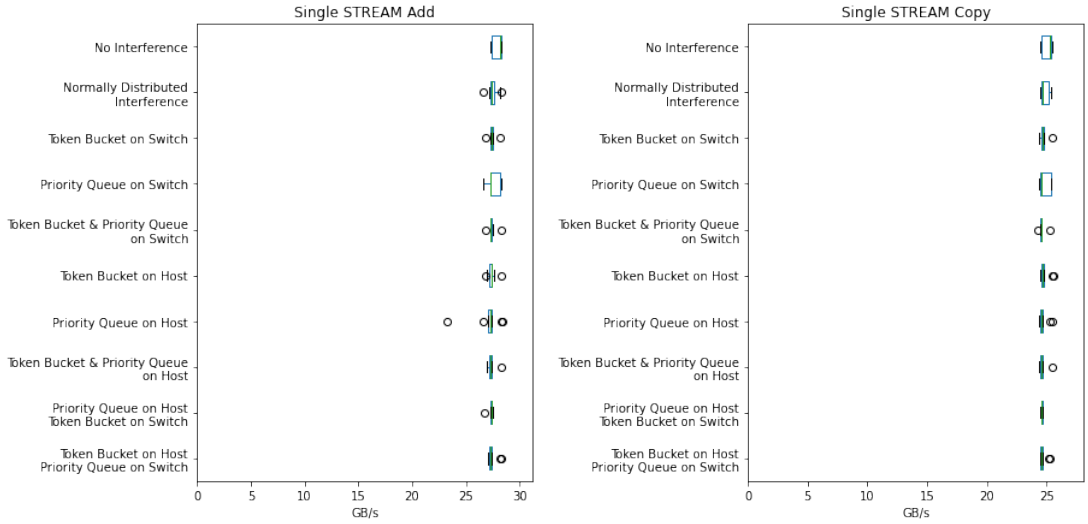

Figure 53: Left: The results of the HPCC Single STREAM Add benchmark. Right: The results of the HPCC Single STREAM Copy benchmark. Neither of the benchmarks show any large differences between any results.

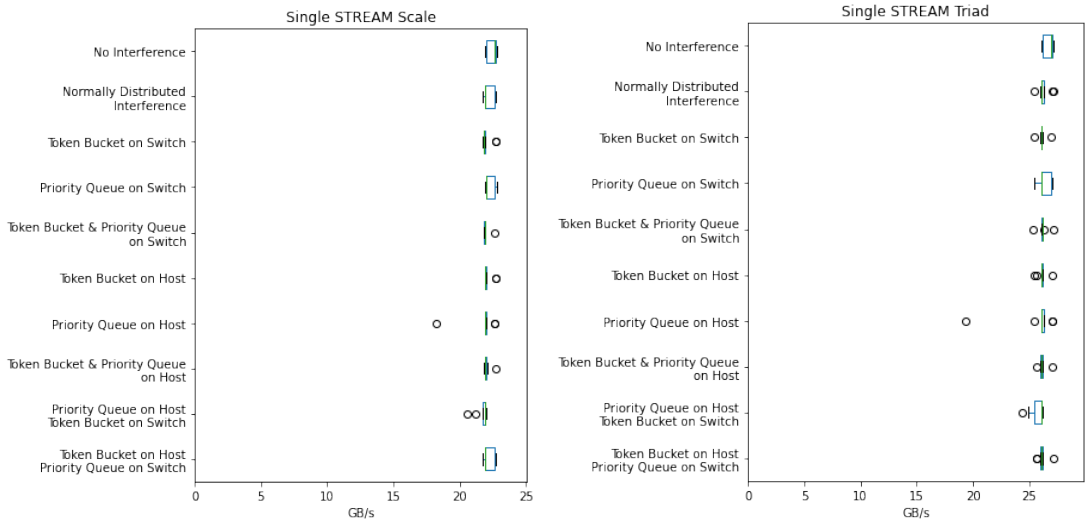

Figure 54: Left: The results of the HPCC Single STREAM Scale benchmark. Right: The results of the HPCC Single STREAM Triad benchmark. While there are small differences in consistency, there are hardly any distinguishable differences between the results.

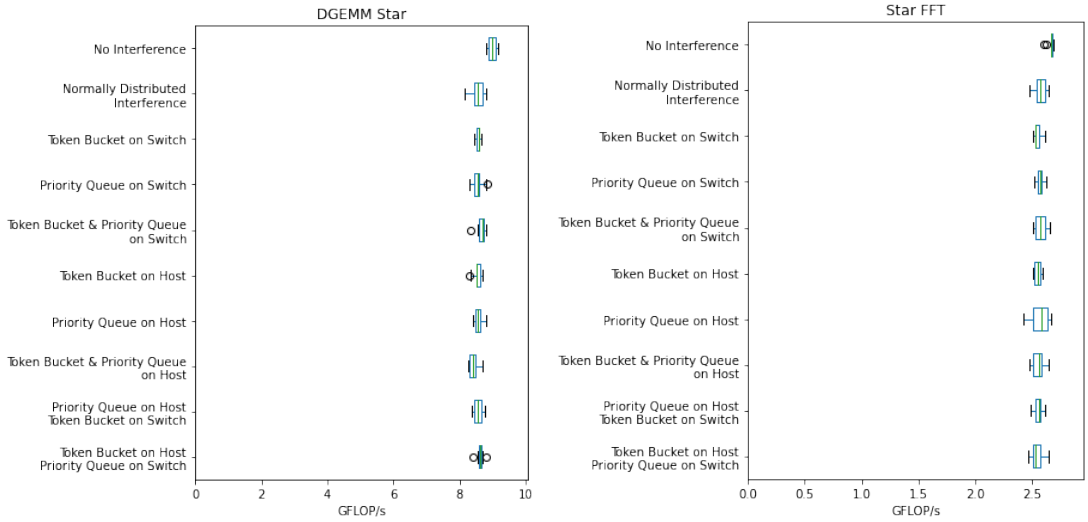

Figure 55: Left: The results of the HPCC DGEMM Star benchmark. Right: The results of the HPCC Star FFT benchmark. While both show a slight decrease in throughput once the interference traffic is added, none of the traffic shaping techniques cause any additional difference in performance.

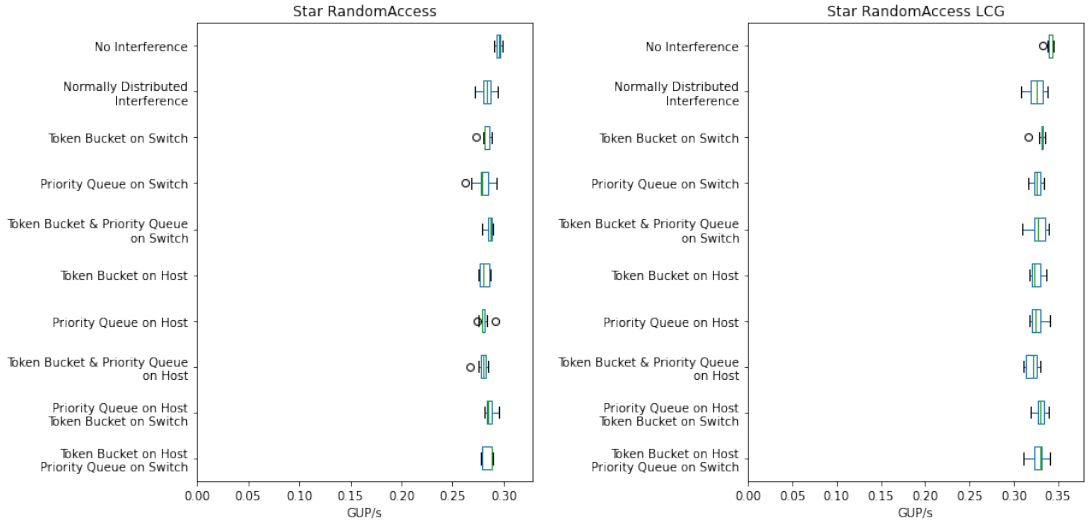

Figure 56: Left: The results of the HPCC Star RandomAccess benchmark. Right: The results of the HPCC Star RandomAccess LCG benchmark. Neither show any notable differences between the results, apart from the no-interference baseline achieving slightly better performance.

E.3 Priority Queue Improvement

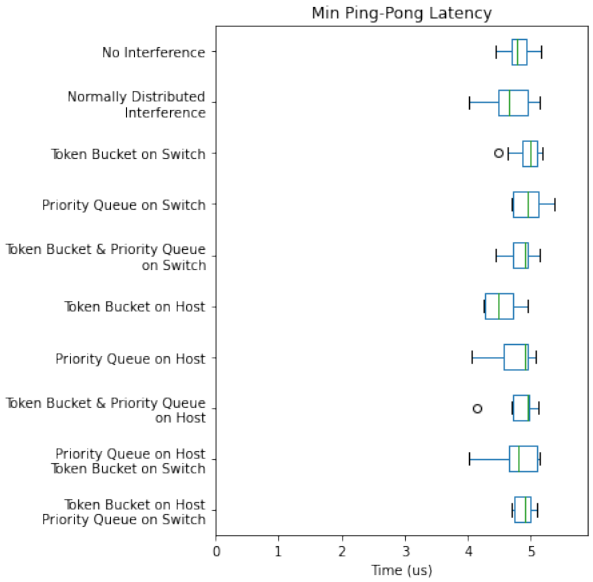

Figure 57: The results of the HPCC MPI Ping-Pong Latency benchmark. The on-host priority queue decreases consistency, however, its tails are towards the lower end of latency, leading to a slight increase in performance.

E.4 Token Bucket Improvement

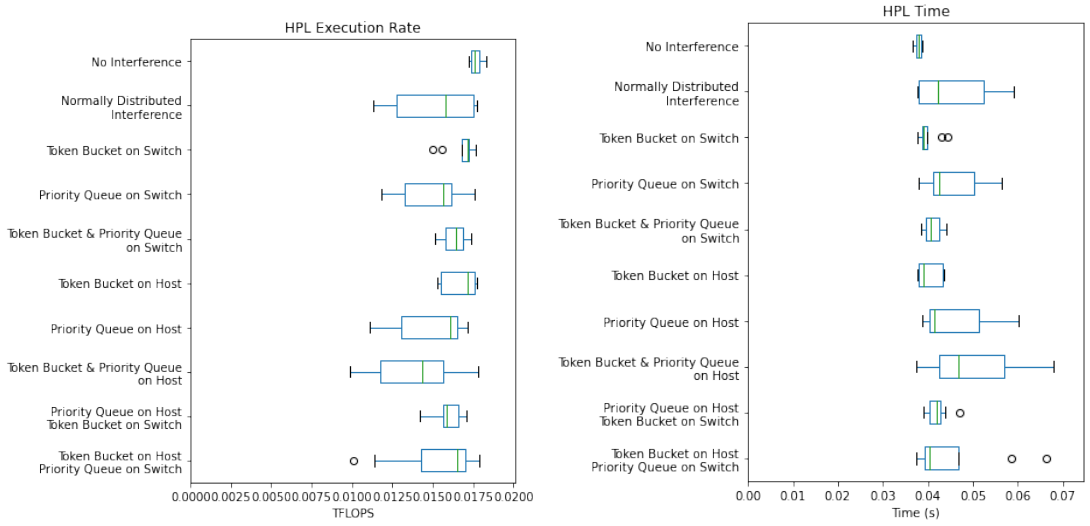

Figure 58: Left: The results of the HPCC HPL Execution Rate benchmark. Right: The results of the HPCC HPL Time benchmark. Both benchmarks show increased consistency as well as greater performance when a token bucket is added, when compared to results without a token bucket present. This is irrespective of the location of the token bucket.

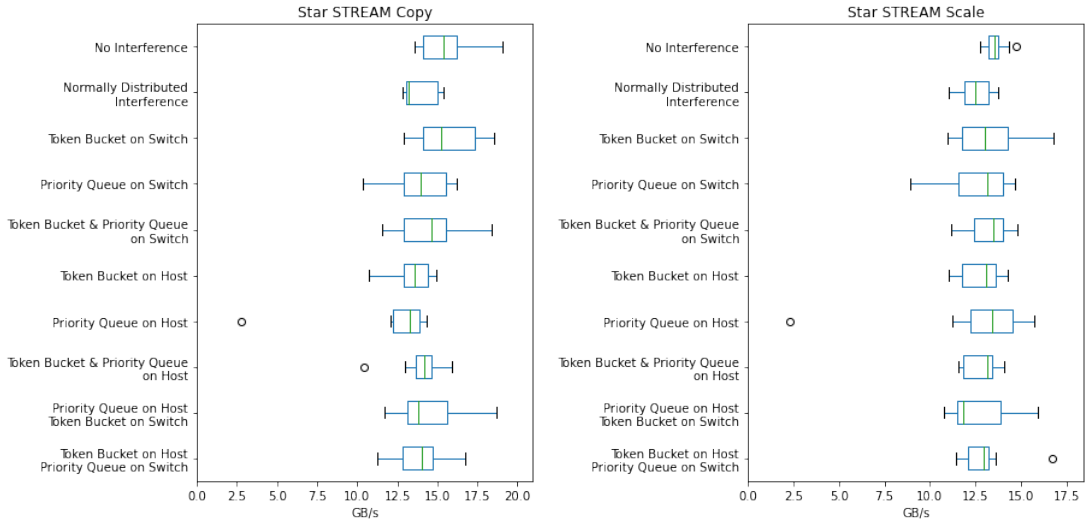

Figure 59: Left: The results of the HPCC Star STREAM Copy benchmark. Right: The results of the HPCC Star STREAM Scale benchmark. While the improvement due to the addition of a token bucket are not as pronounced, the on-host token bucket in particular causes a decrease of performance consistency, with peaks towards the higher end of throughput, thus increasing performance. This effect is still present – though less pronounced – when a priority queue is added.

E.5 Location-dependent performance with both Token Bucket and Priority Queue

When using both the priority queue and token bucket, performance may vary on an application-by-application basis depending on whether these traffic shaping techniques are present on-host or on-switch. This section shows varying results based on these differences.

E.5.1 On-Host Priority Queue & On-Switch Token Bucket

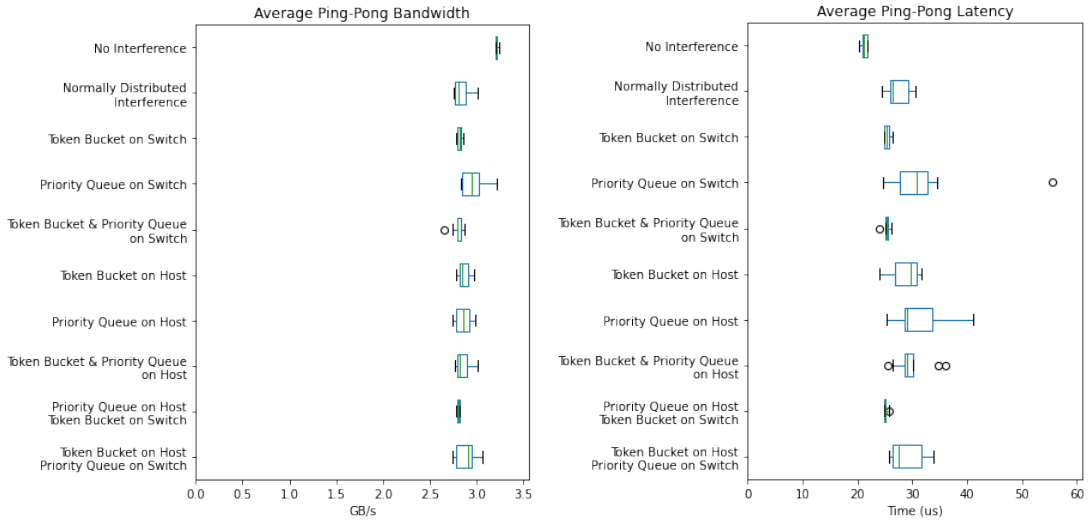

Figure 60: Left: The results of the HPCC Average Ping-Pong Bandwidth benchmark. Right: The results of the HPCC Average Ping-Pong Latency benchmark. When the on-switch token bucket is added, especially when combined with a priority queue, the performance is more consistent when compared to the normally distributed interference.

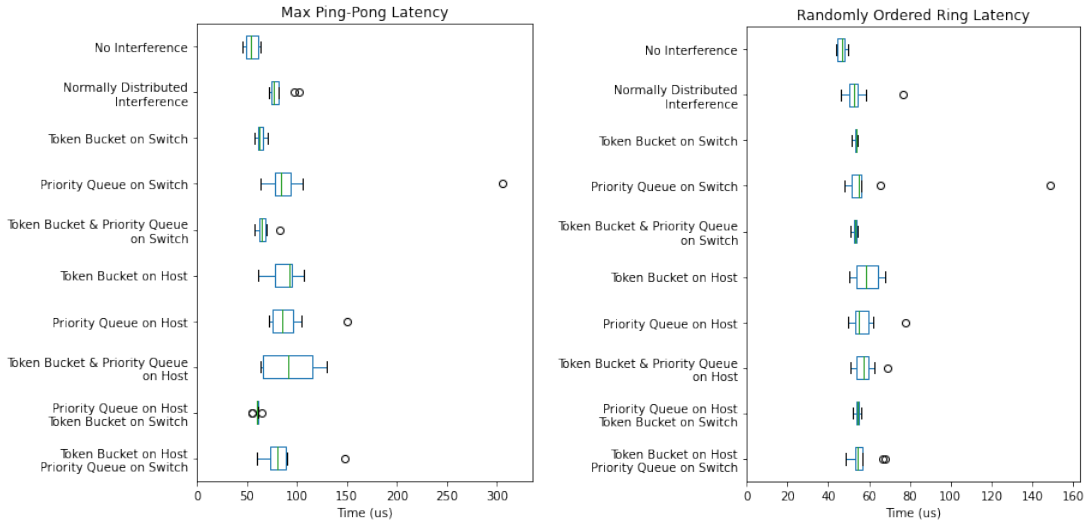

Figure 61: Left: The results of the HPCC Max Ping-Pong Latency benchmark. Right: The results of the HPCC Randomly Ordered Ring Latency benchmark. Both results show far greater consistency when the on-switch token bucket and on-host priority queue are combined, though this is in large part due to the effect of the token bucket.

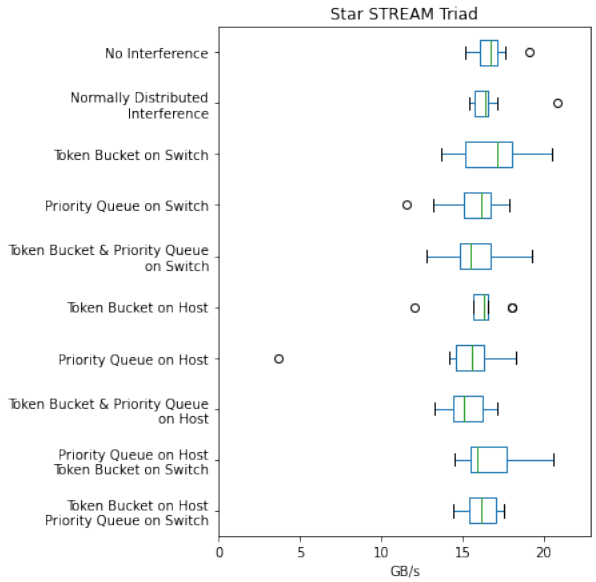

Figure 62: The results of the HPCC Star STREAM Triad benchmark show that the addition of the on-switch token bucket decreases consistency but increases performance, with a slightly higher bottom tail when the on-host priority queue is added.

E.5.2 On-Host Token Bucket & On-Switch Priority Queue

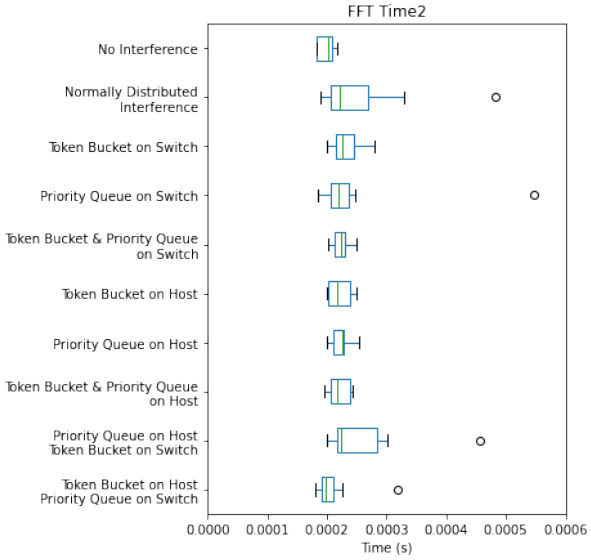

Figure 63: The results of the HPCC FFT Time2 benchmark show that the combination of the on-host token bucket and the on-switch priority queue slightly increase performance at similar consistency when compared to results utilizing other combinations of traffic shaping techniques. It seems to improve over the normally distributed interface baseline.

E.5.3 On-Host Token Bucket & Priority Queue

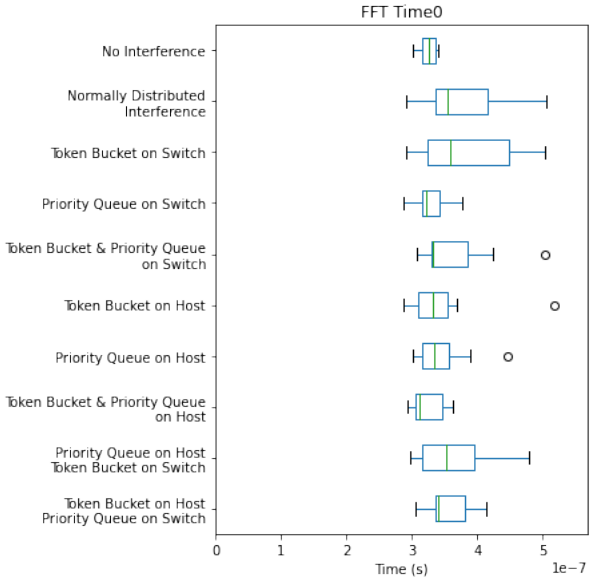

Figure 64: The results of the HPCC FFT Time0 benchmark show that the on-switch priority queue, as well as all on-host traffic shaping measures have a positive impact on performance, with the combination of the on-host measures resulting in slightly improved performance yet. Combining them with the on-switch priority queue does not have this effect, however.

E.5.4 On-Switch Token Bucket & Priority Queue

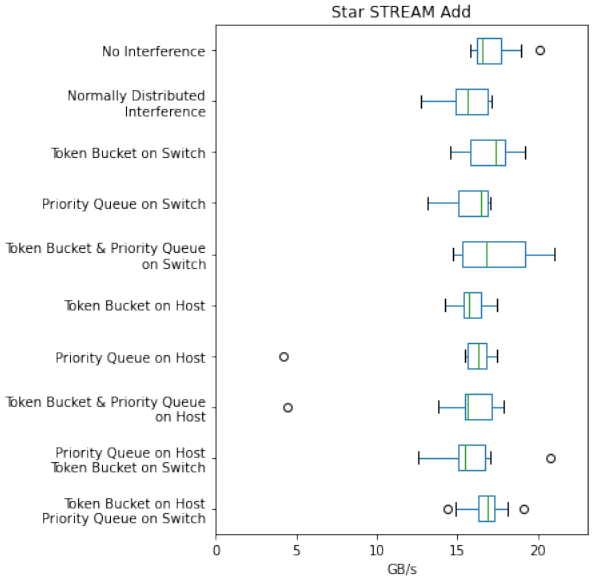

Figure 65: The results of the HPCC Star STREAM Add benchmark show that the on-switch token bucket, especially when combined with the on-switch priority queue, has a sizeable positive impact on throughput, though at a reduced consistency. Interestingly the addition of the on-host priority queue does not have this effect.

E.5.5 On-Switch Token Bucket & Priority Queue with *Worse* Performance

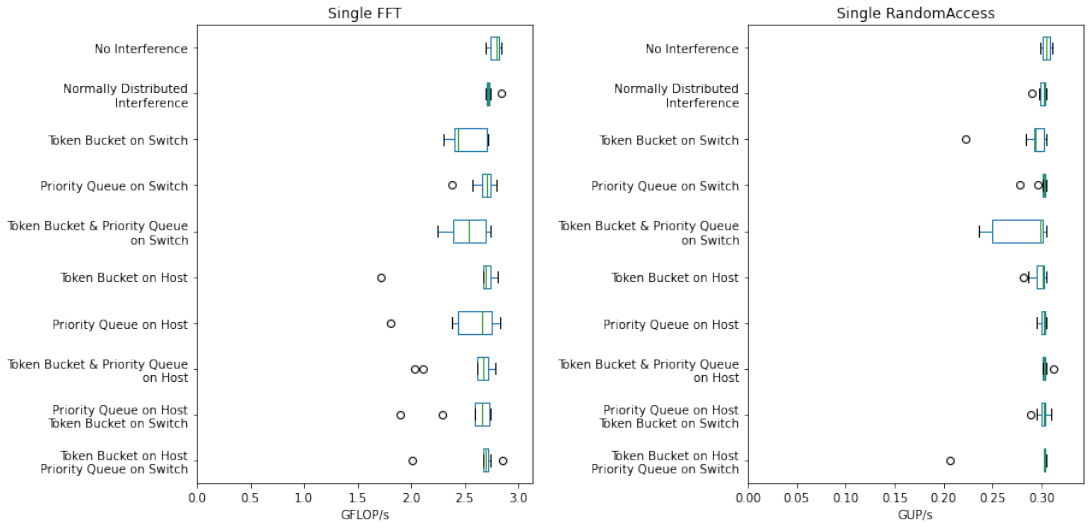

Figure 66: Left: The results of the HPCC Single FFT benchmark. Right: The results of the HPCC Single RandomAccess benchmark. Surprisingly, both these "Single" benchmarks show a negative influence from the addition of the on-switch token bucket and priority queue. Given the fact that this type of benchmark is run on a single (albeit randomly selected) node, the only cause for this change in performance seems to be the initial transfer of data as well as the response.

E.6 Uncategorised Results

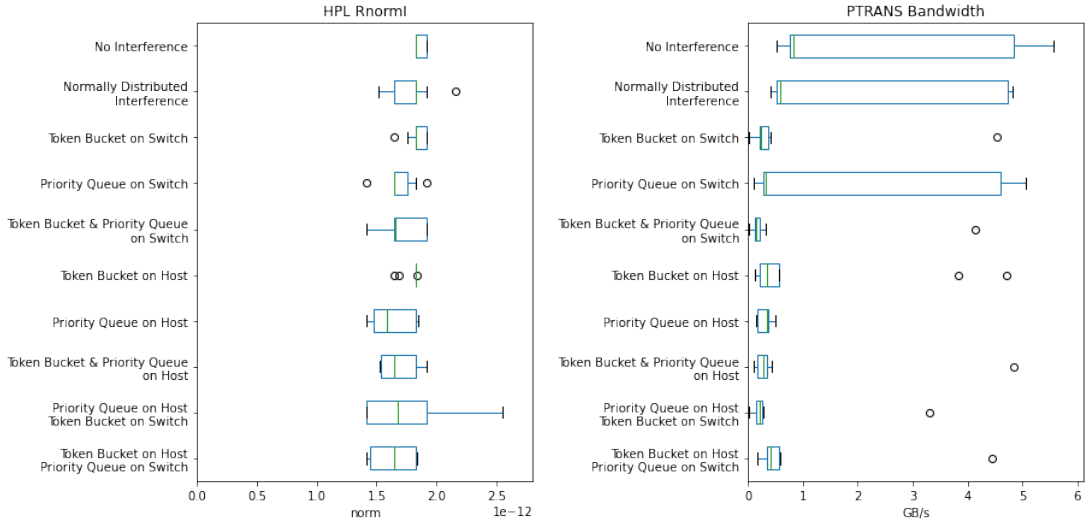

Figure 67: Left: The results of the HPCC HPL RnormI benchmark. The results show various consistencies seemingly irrespective of the traffic shaping measures. Right: The results of the HPCC PTRANS Bandwidth benchmark. Nearly all results show similar performance, except for the on-switch token bucket, which appears to be more in line with the baseline results.

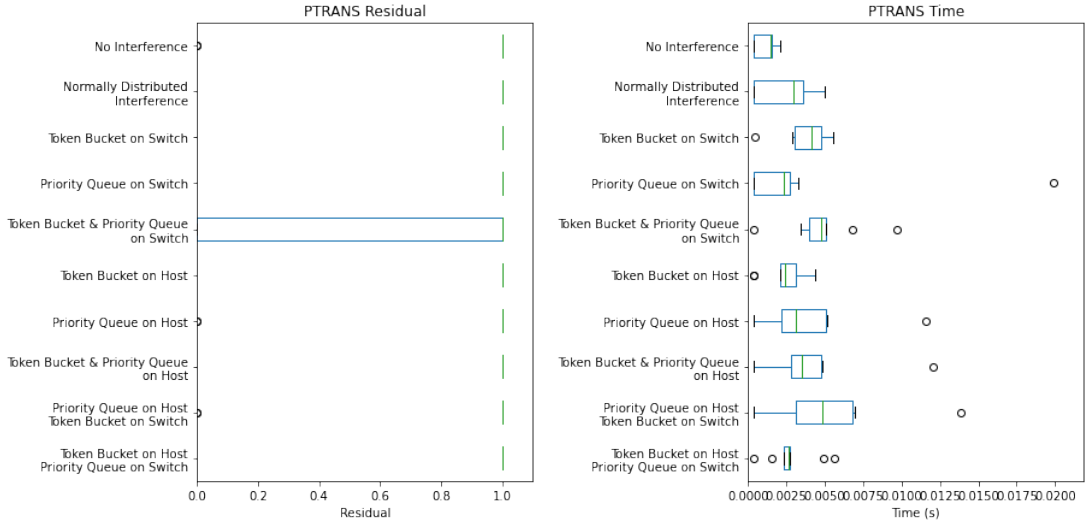

Figure 68: Left: The results of the HPCC PTRANS Residual benchmark. The residuals appear to be either 0 or 1, with the latter appearing more often. Right: The results of the HPCC PTRANS Time benchmark. The on-switch priority queue appears to perform most in line with the baseline results, while the others perform less consistent and slightly worse across the board, with various outliers.

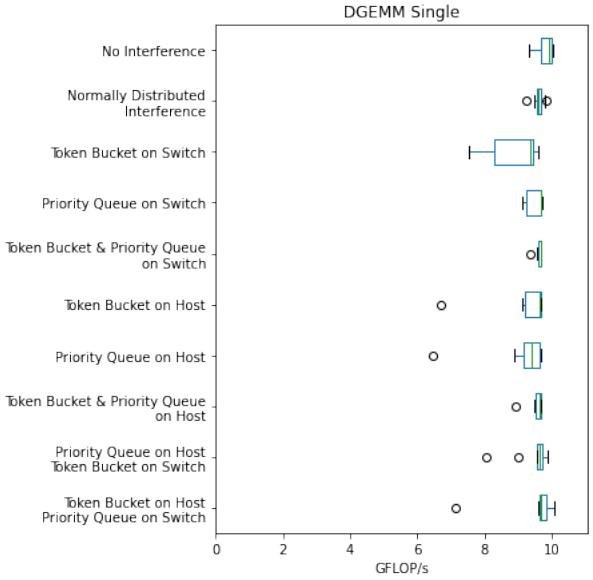

Figure 69: The results of the HPCC DGEMM Single benchmark show that the on-switch token bucket reduces performance, which is surprising as the "Single"-type benchmarks are run on a single node.