Understanding the impact of host networking elements on traffic bursts

Erfan Sharafzadeh and Sepehr Abdous, *Johns Hopkins University;*
Soudeh Ghorbani, *Johns Hopkins University and Meta*

https://www.usenix.org/conference/nsdi23/presentation/sharafzadeh

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Understanding the impact of host networking elements on traffic bursts

Erfan Sharafzadeh[1], Sepehr Abdous[1], Soudeh Ghorbani[1,2]

[1]*Johns Hopkins University* , [2]*Meta*

Abstract

Conventional host networking features various traffic shaping layers (e.g., buffers, schedulers, and pacers) with complex interactions and wide implications for performance metrics. These interactions can lead to large bursts at various time scales. Understanding the nature of traffic bursts is important for optimal resource provisioning, congestion control, buffer sizing, and traffic prediction but is challenging due to the complexity and feature velocity in host networking.

We develop Valinor, a traffic measurement framework that consists of eBPF hooks and measurement modules in a programmable network. Valinor offers visibility into traffic burstiness over a wide span of timescales (nanosecond- to second-scale) at multiple vantage points. We deploy Valinor to analyze the burstiness of various classes of congestion control algorithms, qdiscs, Linux process scheduling, NIC packet scheduling, and hardware offloading. Our analysis counters the assumption that burstiness is primarily a function of the application layer and preserved by protocol stacks, and highlights the pronounced role of lower layers in the formation and suppression of bursts. We also show the limitations of canonical burst countermeasures (e.g., TCP pacing and qdisc scheduling) due to the intervening nature of segmentation offloading and fixed-function NIC scheduling. Finally, we demonstrate that, far from a universal invariant, burstiness varies significantly across host stacks. Our findings underscore the need for a measurement framework such as Valinor for regular burst analysis.

1 Introduction

Measurement studies show that traffic is bursty across a wide range of timescales in diverse contexts such as Ethernet LANs [38], WANs [56], data centers [25], and WWW traffic [21]. In particular, microsecond-scale congestion events, sometimes called *microbursts*, have been the focus of numerous measurement and control papers recently [13, 18, 19, 25, 33, 37, 72]. However, the modulating effect of host networking on traffic burstiness at various timescales is relatively less investigated. This paper addresses this gap. We ask *what causes the traffic to emerge from hosts in bursts?* Is burstiness an *scale-invariant* property of traffic, i.e., does the traffic retain its burstiness across a wide range of timescales, or do the microbursts become smooth at coarse timescales? Are canonical burst countermeasures such as TCP pacing and packet scheduling effective in curtailing bursts?

These questions have far-reaching implications for network performance and design. Controlling bursts at different timescales requires deploying mechanisms that operate at the corresponding pace. Microbursts, for instance, require real-time techniques with sub-RTT control loops, whereas bursts at longer timescales can be more effectively managed by resource provisioning techniques such as topology engineering and routing that take seconds to minutes to complete [71].

Unfortunately, studying the impact of host networking on bursts is complex. Take the Linux network stack as an example: the egress traffic that originates from the Linux kernel stack passes through many layers and optimizations before arriving at the wire. Transport protocol internals like initial window size, cumulative acknowledgments, queueing disciplines (qdiscs), driver rings, segmentation offloading, and hardware packet scheduler at the NIC all handle the traffic. All these elements and their complex interactions can play a role in forming or suppressing bursts at various timescales. These challenges are further compounded by the heterogeneity, scale, and the velocity of evolution in today's networks that constantly change in response to increasing demand and the rollout of new services [26, 46, 73].

To address this challenge, we build Valinor, a high-resolution traffic measurement framework that enables network operators to systematically and periodically dissect the elements of host networking, their impact on traffic burstiness in isolation, and importantly, their interactions with the emergent traffic patterns, all at different timescales. To ensure visibility into the impact of the software stack and the shape of the traffic on the wire through time, Valinor is composed of two main components: 1) An in-host timestamping framework (Valinor-H) based on eBPF that collects egress packet metadata nearly at the last stage of software stack processing. 2) An in-network packet timestamping framework (Valinor-N) that captures packet arrival timestamps in the programmable switch data plane immediately after the NIC, and sends the timestamp data to offline servers for collection, storage, and burstiness analysis.

Our analysis of the impact of host networking on the shape of traffic using Valinor reveals some surprising results. As an example, classical work paints a unifying and consistent picture of scale-invariant burstiness, i.e, they show the same degree of variability across a wide range of timescales in a variety of different network types [21, 38, 56]. It has also been established that this scale-invariant burstiness is primarily caused by *application layer* characteristics such as long-tailed flow size distributions and is "robust": it holds for a variety of transport protocols (e.g., TCP Reno, Vegas, and flow controlled UDP) and various network configurations [23, 53].

In contrast, our investigations paint a more nuanced and complex picture. We show that burstiness at various timescales varies significantly across host configurations (hardware configurations, transport protocols, scheduling, etc.). We also show the pronounced modulating effect of below application layer elements on bursts. This implies that, for the same heavy-tailed flow size distribution, the ultimate shape of traffic on the wire depends heavily on the host configuration such as the NIC scheduler. Plus, Valinor's analysis of newer reliable transport protocols (e.g., Homa [47], DCTCP [9], and BBR [17]) reveals the high degree of variability of burstiness for these protocols. As an example, BBR is less bursty not just at fine timescales (a result that is consistent with the literature [47, 52]) but also at coarse timescales. The latter finding (new to the best of our knowledge) implies that techniques such as topology engineering [71] and multi-timescale congestion control [66]—premised on the long-range burstiness of traffic—may yield limited performance improvements under these new protocols.

Finally, given the impact of some variants of transport protocols on bursts, we quantify the effectiveness of TCP pacing and active queue management paradigms such as CoDel [49] in qdiscs (software packet schedulers) in mitigating bursts. Our results show the pronounced impact of lower-layer functions (residing in the driver and NIC) on forming the ultimate shape of traffic on the wire relative to the higher-layer software operations of the TCP/IP stack and qdiscs. As an example, active queue management techniques such as CoDel and RED in the Linux kernel try to prevent the formation of large and lasting bursts. However, our results show that their impact is effectively erased by offloading (TSO, serialization, etc.) and the NIC scheduler. For example, while in isolation, the frequency of large 300 KB bursts under CoDel is 500 times lower than FIFO, this difference is barely visible on the wire after packets pass through the multi-queue NIC with segmentation offloading. Moreover, TCP pacing enforced in the qdiscs generates between 1.8×-19× larger bursts when NIC scheduler and offloading are in action compared to when in isolation.[1] This result indicates that the countermeasures for controlling bursts should be moved further down the packet processing pipeline at the end hosts.

Our results on the variability of burstiness (based on hardware configurations, transports, etc.)—combined with the ever-evolving workloads and features in today's networks—highlight the need for periodic traffic measurement and analysis. To facilitate this, we have released Valinor's sources and artifacts as open-source software.[2] We next introduce the mathematical notions developed for capturing bursts across time, present their practical implications in networks (§2), provide some background on host networking and the design space of burst measurement frameworks (§3), and present the design of Valinor (§4) before delving into our findings (§5).

2 Background: scale-invariant burstiness

Measurements of the Internet traffic show periods of sustained greater-than-average or lower-than-average traffic rates across a wide range of timescales [21, 24, 38, 53, 56]. This behavior, sometimes called *scaling* or *self-similarity*, has broad implications for performance. In this section, we first formalize the notion of self-similarity and re-introduce the Hurst exponent, a mathematical representation of self-similarity, before discussing the implications of self-similarity and characterizing bursts at fine timescales such as microbursts.

Self-similarity. Self-similarity is a notion pioneered by Benoit Mandelbrot [45] which refers to a phenomenon where a certain property of an object (such as an image or a time-series) is preserved with respect to scaling in space and/or time. If an object is self-similar, its parts, when magnified, resemble the shape of the whole [55].

More formally, let $(X_t)_{t\in\mathbb{Z}_+}$ be a timeseries, e.g., this time-series can represent a traffic trace measured at some fixed time granularity. The aggregated series $X_i^{(m)}$ is defined as

$$X_i^{(m)} = 1/m(X_{im-m+1} + \ldots + X_{im})$$

In other words, X_t is partitioned into blocks of size m, their values are averaged, and i denotes the index of these blocks.

Autocorrelation is a mathematical representation of the degree of similarity between a timeseries X_t and a time-shifted version of X_t over successive time intervals. It measures the relationship between the current value of a timeseries and its

[1] Despite making the traffic bursty and hard to manage, these low-level functions are essential for reducing the processing overhead and meeting the increasingly high link rates. For example, disabling TCP segmentation offload results in a 3× increase in CPU utilization, 71% lower throughput, and a 46% increase in median packet RTTs for a multi-flow *Iperf* test. Relatedly, disabling MQ results in a 4% decline in the throughput of the same workload.

[2] https://hopnets.github.io/valinor

future values. A strong positive autocorrelation for a traffic volume timeseries, for example, suggests that if the volume is high (i.e., higher than average) now, then it is likely to be also high in the next time slot, whereas a strong negative autocorrelation implies that a high-volume slot is likely to be followed by a low volume one.

Let $r(k)$ and $r^{(m)}(k)$ denote, respectively, the autocorrelation functions (ACFs) of X_t and $X_i^{(m)}$ where k is the time shift from the original timeseries. We say that X_t is *self-similar*, or more accurately *asymptotically second-order self-similar*, if these conditions hold:

$$r(k) \sim c \times k^{-\beta} \tag{1}$$

$$r^{(m)}(k) \sim r(k) \tag{2}$$

for large k and m, where $0 < \beta < 1$ and c is a constant, and $f(x) \sim g(x)$ as $x \to a$ means that $\lim_{x \to a} f(x)/g(x) = 1$ [66]. X_t is self-similar in the sense that its ACF $r(k)$ behaves hyperbolically with $\sum_{k=0}^{\infty} r(k) = \infty$ (Eq. 1). This property is also referred to as *long-range dependence*. Equation 2 implies that for self-similar timeseries, the autocorrelation structure is preserved with respect to time aggregation.

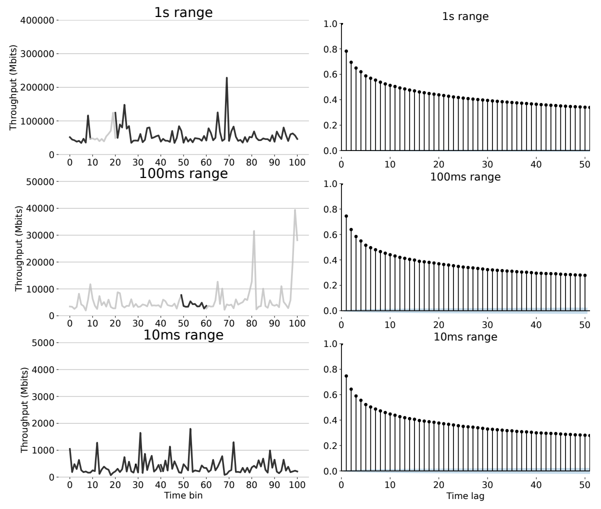
In networks, the traffic is called self-similar if the aggregated traffic over varying timescales remains bursty, regardless of the granularity of the timescale.

The Hurst exponent. Let $H = 1 - \beta/2$. H is called the *Hurst exponent.* The Hurst exponent, a number in the $(0, 1)$ range that is sometimes referred to as the *index of long-range dependence*, is a measure of the long-term memory of a timeseries. It characterizes the self-similarity and long-range dependence of the timeseries:

- $\underline{0.5 < H < 1}$ indicates a self-similar timeseries with long-term positive autocorrelations, i.e, a high value in the series (e.g., higher than average traffic volume) is likely to be followed by another high value. Plus, the values a long time into the future also tends to be high. It follows from Eq. (1) above that the closer H is to 1, the more long-range dependent X_t is. Conversely, H values closer to 0.5 show weaker long-range dependence.
- $\underline{H = 0.5}$ indicates a completely uncorrelated series.
- $\underline{0 < H < 0.5}$ indicates a *mean-reverting* timeseries, i.e., one with long-term switching between high and low values in adjacent pairs of time slots. That is, a single high value in the timeseries is likely to be followed by a low value.[3]

Various techniques (e.g., rescaled-range analysis and Periodogram [67]) exist for estimating H for an empirical dataset. Similar to the seminal work on Bellcore Ethernet traffic self-similarity [38], we use the rescaled-range, *R/S*, for the results presented in this paper. The details of this method are presented in Appendix §A.

[3]Note that the $0 < \beta < 1$ condition in the equations above is a requirement for self-similar, and not mean-reverting, series.



(a) Time-series **(b) Auto-correlation**

Figure 1: A self-similar timeseries with H=0.88.

Example: Figure 1a (the first row) shows a simulated scenario where 32 TCP connections generate a synthetic workload using Pareto flow size distribution with a mean of 4200 KB and $\alpha = 1.05$ and exponential arrivals that create 6 Gbps offered load. We plot the traffic rate (in Mbps) against time where time granularity is $1s$. A data point is the aggregated traffic volume over a $10ms$ interval. The second row of the same figure depicts the same traffic series where a randomly selected second interval in the first timeseries (the highlighted segment in the first row) is magnified by a factor of ten, resulting in a granularity of $100ms$ in the truncated timeseries. The last row similarly rescales a randomly selected slot by $10\times$. The figures show that this trace is self-similar: when traffic is aggregated over varying timescales, the aggregate traffic pattern remains bursty, regardless of the granularity of the timescale. This visual scaling is confirmed by the Hurst coefficient, $H = 0.88$, and the autocorrelation functions of the trace (Figure 1b) that show positive, slow (almost polynomial) decaying, and consistently shaped correlations across various timescales. Slow-decaying ACFs signify long-range dependence in a timeseries.

Practical implications of self-similarity. Self-similarity has broad implications on network design and performance, e.g., it is shown to lead to increased delay and loss [5, 6, 22, 42, 50, 53, 66]. We next discuss some of the key implications of self-similarity:

- **Queueing performance and buffer sizing.** Self-similarity greatly influences queueing performance. From a queueing theory standpoint, the defining characteristic of self-similarity is that the queue length distribution decays much more slowly than short-range-dependent traffic (polynomially vs. exponentially under short-range dependent traffic, e.g., Poisson processes) [66]. For strongly self-similar traffic, the mean queue length increases with the buffer size [54]. This implies that networks with strongly self-similar traffic should deploy small buffers to control the queueing delay.

- **Throughput and latency trade-off.** Prior work [53, 54] shows that jointly provisioning low delay and high throughput is adversely affected by self-similarity.
- **Traffic prediction and burst countermeasures.** The correlation structures present in self-similar traffic can be detected and exploited to predict future traffic over timescales larger than an RTT [66].[4] Traffic prediction at long timescales, in turn, is invaluable for designing the appropriate burst countermeasures. For instance, resource provisioning techniques with control loops larger than an RTT (e.g., multi-scale congestion control [66], re-routing, and topology rewiring [71]) enhance the performance of self-similar traffic.

Microbursts. Given their ubiquity and impact, in particular in data centers, microsecond-scale traffic surges, known as *microbursts* [13, 41, 72], have been the focus of many recent proposals [4, 19, 27, 37, 41, 70].

The intensity of a microburst has often been measured implicitly based on buffer utilization, or in more extreme cases, packet loss. Related work also quantifies microbursts as the number of packets from one flow that occupy a buffer at a time snapshot [33], the evolution of switch queue length over time [63], an uninterrupted sequence of packets with gaps of smaller than a threshold [35], and/or sequence size of larger than a threshold [68]. Using metrics that are independent of network queues allows us to perform universal measurements in the entire network, i.e., both at the hosts and the switches. Yet, measurement systems intending to quantify microbursts can leverage all the above definitions to provide a holistic view of burstiness behavior.

From the technical perspective, we define a burst as the cumulative sum of packet bytes whose inter-arrivals are smaller than a threshold τ. Setting the minimum value for τ initially depends on link speeds and MTUs. For example, in a fully utilized 40 Gbps link with MTU = 1500 bytes, packets arrive 300 ns apart. Therefore, an initial τ of 2-10$\times$ of this value is small enough to detect microbursts and large enough not to miss consecutive packets from flows. To ensure that τ is not affected by the network configuration and the internal characteristics of the workloads, we repeat our measurement with a wide range of values for τ.

3 Approaches to measuring traffic bursts

In this section, we provide a brief background on host networking and present the design space of burst measurement frameworks before discussing Valinor in §4.

[4] The prediction methods span diverse domains such as regression theory, neural networks, and estimation theory [66].

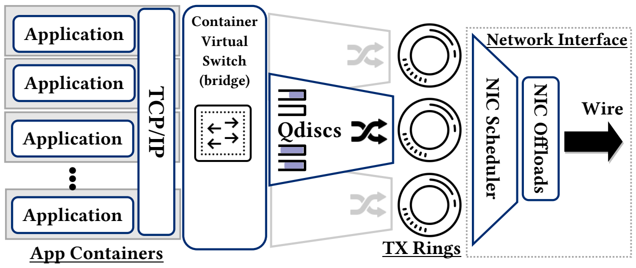


Figure 2: Conventional network processing stack architecture in a containerized Linux deployment.

3.1 Conventional host networking

Conventional network stacks consist of various processing layers glued together via several optimization techniques. In Linux, application data is passed to socket interfaces (buffering in the userspace), and then to the transport protocol processing (transport buffers, short queues [20]). Transport protocols populate *sk_buffs*,[5] a collection of data pointers and header information. After performing routing, *sk_buffs* eventually make their way towards interface *qdiscs*, the hierarchical packet schedulers in Linux. *qdiscs* operate in parallel on all CPU cores and forward the scheduled *sk_buffs* towards driver rings where another layer of buffering is performed before notifying the NIC [65]. Finally, with the conventional offloading features enabled, the NIC performs scatter/gather [58], segmentation, checksum, and sends the packets on the wire [16]. Figure 2 depicts an overview of the packet's path through the network processing pipeline in a Linux host.

3.2 Capturing timestamps

High-resolution timestamping is essential for burst analysis. Various techniques exist for capturing packet arrivals:

NIC timestamps. Hardware timestamping is available in all commodity NICs. This feature is supported by the Linux kernel via ancillary socket data. When a user requests timestamping through a socket option, the transmission timestamps are generated in the hardware before sending the packet on the wire and are eventually sent to the source socket. Therefore, the application is responsible for polling the error queue and reading the timestamps. Hardware timestamping supports most TCP and UDP connections, however, it suffers from two main shortcomings. First, if the operating system fails to poll the timestamp registers of the NIC in time, e.g., in higher packet rates, the timestamp will be overwritten by that of the next packet. Plus, modifying the network application to receive timestamps may impact the application's workload pattern, and thus must be performed with extra care.

[5] *sk_buff* stands for "Socket buffer" and is used to represent the socket data that eventually is shaped into the packet. *sk_buffs*, therefore, may contain a single or multiple packets.

Modifying networking stack software. To study realistic network traffic with higher arrival rates, hardware timestamping is not ideal due to the need to change the application internals and high overheads. An alternative solution is to directly capture the timestamps closer to the packet processing, e.g., the NIC driver, and either add the timestamps to the packet payload or re-route them to the userspace. Alas, accessing and modifying the packet data requires offloading features such as scatter-gather IO and Segmentation Offloading to be turned off. Additionally, timestamps that are at *sk_buff* granularity may not imitate the inter-packet gaps on the wire due to the intervention of lower layers.

eBPF hooks. eBPF offers a series of hooks inside the Linux kernel and the NIC driver that allows fast execution of arbitrary data plane logic. An eBPF program consists of a data plane and a control-plane code targeting a specific hook on the RX or TX path (XDP hook in the receive path of NIC driver and traffic control (*tc*) hook on the TX path of the qdisc subsystem are two examples). eBPF *tc* programs are registered to the kernel using the *tc* command and are executed inside a lightweight RISC virtual machine. eBPF also provides fast data structures that enable shared state between the kernel and the userspace. This allows us to perform burst measurements offline with any workload configuration without modifying the kernel source or packet payloads.

While eBPF relieves us from directly modifying the packet processing code in the kernel, it presents two shortcomings. First, eBPF, similar to the previous solution, works at *sk_buff* granularity since packet segmentation is almost always offloaded to the NIC. Therefore, the eBPF framework can only measure the gaps between larger chunks of data, not packets. Additionally, our measurements show that each eBPF invocation incurs up to 1 μs of delay, mostly due to memory accesses. While this overhead may be acceptable at the *sk_buff* granularity, the framework will lose its visibility into nanosecond-scale events. Ultimately, eBPF provides a convenient solution to plug into the network data path with minor interference. Making it a viable burstiness probing point on the egress path. We present the design and implementation of the Valinor eBPF framework, Valinor-H, in §4.1.

Timestamping in the switch data-plane. A holistic method to capture the behavior of all host networking components (including the NIC) is to perform measurements immediately after transmitting the packets on the wire, i.e., at the first network hop. Fortunately, the rise of programmable switch architectures with high-resolution timestamping enables capturing packet arrival timestamps and sending this data off the critical communication path for offline processing. This further ensures zero interference with the ongoing communication and the ability to track the entire egress host networking components. We describe the design of our in-network measurement system, Valinor-N, in §4.2.

Programmable NICs share many of the strengths of in-network measurements (e.g., timestamping close to the wire, low overhead, and no interference) but do not provide visibility into in-network queue occupancies. Plus, our experience with commodity DPUs [15] shows inconsistencies in the capabilities of existing devices. General-purpose SoC NICs [15] are either bound to their slow ARM CPUs or do not offer per-packet timestamping capabilities on their fast path. Due to these practical issues as well as the greater visibility that in-network measurements offer, alongside its host module, Valinor currently leverages programmable networks for capturing bursts on the wire.

4 Valinor measurement framework

For designing Valinor, we have three goals in mind:

1. Offering visibility into the host networking traffic, as well as the shape of the traffic on the wire.
2. Offering high-resolution timestamping of packet arrivals in line with the increasing link bandwidths and faster packet processing pipelines.
3. Providing insights on traffic shape and burstiness at different scales and time ranges.

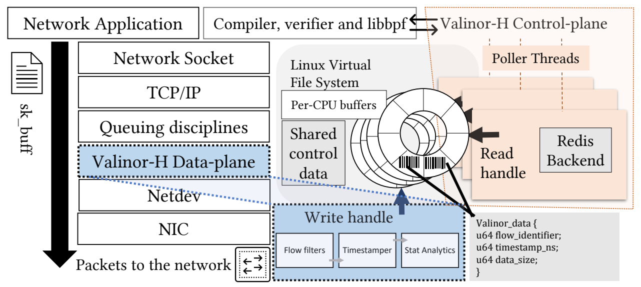
We design and implement Valinor, a measurement framework that consists of two main timestamping prongs to study packet arrivals from the host and network vantage points. First, we design Valinor-H to study the host's view of its egress traffic by choosing *tc* eBPF hooks. For capturing the external picture of traffic burstiness, we design Valinor-N, a timestamping module for programmable fabric.

4.1 Valinor-H: burst measurement in hosts

Valinor-H offers visibility into the impact of the software stack on traffic, immediately before the traffic is passed to the NIC. The insight into the characteristics of the traffic entering the hardware can help the design of the functions offloaded to the NIC. This becomes increasingly important as more and more functions migrate to the NIC, driven by the dire need to reduce software overhead.[6]

Figure 3 presents the design of our eBPF framework. Our framework consists of two separate programs. The data plane program follows a strict set of C-like instructions that are executed at the *tc* qdisc, every time a *sk_buff* arrives. We design

[6]As network speeds increase at a faster pace than CPU speeds, software overhead is increasingly the performance bottleneck [52]. This has motivated the offloading of various functions such as segmentation, serialization, scheduling, and even transport protocol processing to the NIC [11, 58, 64].



```
Valinor_data {
u64 flow_identifier;
u64 timestamp_ns;
u64 data_size;
}
```

Figure 3: The eBPF measurement framework's architecture. Valinor-h consists of a data plane and a control plane, communicating via lock-free ring buffers.

circular buffers capable of storing up to 2^{16} arbitrary data entries shared with the control plane. Then, the *write handle* determines the correct location for adding new timestamp entries and updates the data structure. In the control plane, we initialize the data plane and the circular buffer and start polling the buffer for new data. The data entries carry the *sk_buff* lengths as well as the flow hash and protocol header information. The *read handle*, retrieves the timestamp entries one by one and hands them to the Redis workers for persistent storage.

One challenge that arises when using a shared data structure is synchronization between the data plane and the control plane. This scenario generally needs locking mechanisms to prevent a race condition, however, the nature of the timestamping data, being strictly increasing, lifts this heavy burden. Therefore, in the control plane, Valinor-H only reads and increments its write handle if the timestamp value is larger than the previous value read. Another synchronization issue arises when multiple CPUs attempt to store packet metadata in the shared memory. Luckily, eBPF offers per-CPU structures to prevent race conditions in the data plane. The Valinor-H control plane uses separate threads to read from per-CPU buffers simultaneously.

With the in-host measurement framework, network operators can verify the operation of higher-level network processing layers on the transmission path of the sender hosts. Valinor-H, at this stage, can capture the ingress traffic into the NIC which includes the traffic egress from qdiscs, the transport layer, and the applications. To capture the traffic behavior in the core of the network, and on a per-packet granularity, we introduce Valinor-N in the following section.

4.2 Valinor-N: in-network burst measurement

Software-based measurements in the host stack are bound to the coarse-grained *sk_buff* arrivals and are implemented before NIC functions (i.e., ring schedulers and segmentation offloads). Hence, the captured traffic behavior might not match that of the wire. To fill this gap, we introduce the in-network variant of Valinor based on programmable switch data planes. Valinor-N consists of three pieces: 1) the switch component, 2) the collector data plane, and 3) the analysis component. Valinor-N is able to I) capture per-packet arrival timestamps with zero overhead outside the critical path, II) collect and store timestamp entries arriving at line rate, and III) perform various analyses on timestamp data to provide an in-depth image of the traffic burstiness at different scales.

Valinor Switch. The switch data plane program uses *mirroring* and *timestamping* functionalities available in the PISA architecture. For every packet that matches user-defined flow filters, Valinor-N appends the arrival timestamp, queuing delay, and the size of the original packet along with its layer 1-4 header information to a special IP packet with a pre-defined Valinor header. The packet is then sent to a collector server. The server machine, deployed outside the critical path of the communication between traffic endpoints, aggregates the timestamp information and performs the offline analysis.

Timestamp collection. The collector machine features a userspace packet processing framework based on DPDK that parses the arrived packets and stores the timestamp information along with flow metadata into an in-memory Redis [3] instance. Analysis of the timestamp data is then performed by querying the data store. Receiving timestamp packets at line rate and storing them in persistent storage poses several scalability challenges to the design of the collector component. To ensure that software can drain NIC buffers at line rate, we designate multiple worker threads to read and process the incoming packets. After parsing timestamp headers, the worker threads extract the timestamp data and send them to additional worker threads that are responsible for communicating with Redis. The stored metadata is then retrieved by the analysis framework to perform burst analysis using timestamps.

Valinor-N's Redis workers issue batched commands during idle periods to minimize interference with packet processing workers. We use Redis sorted sets to store timestamp entries sorted by arrival times since the packets that arrive at the collector may have a different order from the packets that arrive at the Valinor-N switch data plane. We use 1G *hugepages* and large memory pools to ensure that timestamp packets are not dropped at higher rates (Up to 40 Gbps in our testbed).

Offline timestamp processing. The last piece of Valinor's design is the offline timestamp analysis framework that queries the Redis data structures and performs analysis on timestamp data. Our framework is able to report various statistics on traffic burstiness by measuring the packet inter-arrivals. For example, in the next section, we report our findings on the scaling behavior, caused by various packet processing components in the sender machine. We

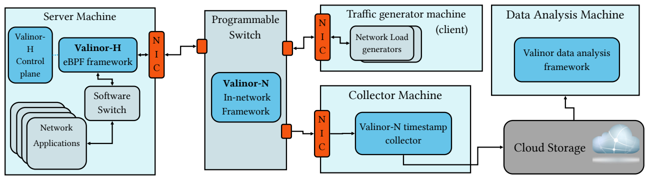


Figure 4: Deployment overview of Valinor framework.

report actual burst sizes in bytes, inter-arrival distributions, queueing delays, and various burstiness time series analyses. We implement the offline processing framework in Python.

5 Findings

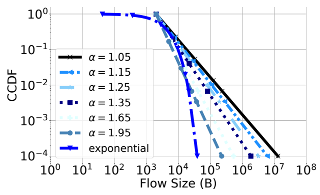
We deploy Valinor to analyze the burstiness of various workloads and configurations. Our results show:

- Host networking, largely overlooked in prior self-similarity studies, plays a major role in forming and suppressing bursts.
- Lower layers of the network processing stack (such as segmentation offloading and NIC scheduling) compromise the effectiveness of software-based traffic shaping and active queue management solutions.
- Software pacing has major limitations. For workloads with a mixture of short and large flows, lower layers of the network processing stack mask the impact of software-based traffic pacing. For workloads with very short flows, software pacing can blunt bursts but leads to a major increase in RTT and significant throughput reduction.
- NIC driver buffer sizing and process scheduling can reshape bursts.

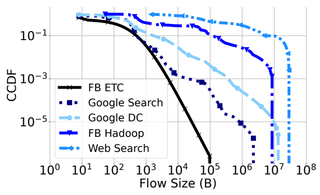
Experiment setup. Figure 4 demonstrates how Valinor framework components come together in a basic deployment. For evaluating Valinor, we use a wide range of workload distributions. We deploy Iperf instances alongside Homa's open-source load generator [47] inside Linux containers and configure the workload generators to simulate different trace-driven workload patterns including Facebook's ETC, Google search, aggregated Google data center, DCTCP's web search, and Facebook's intra-cluster and intra-rack Hadoop traces [9,12,47,60]. Unless stated otherwise, all application containers are connected via an *OVS* [2] virtual bridge to the external interface. Our testbed consists of servers featuring Intel Xeon E5-2620 v4 processors, 64 GB of memory, and Intel XL710 40G NICs. We connect the servers via a Wedge-100 Tofino switch running Valinor-N timestamping framework. We deploy Valinor-H on Linux kernel 5.17 with the latest version of *libbpf* and *iproute2* installed. The collector machine features

Setting	Default Value	Parameter Range
Transport	TCP cubic	cubic, reno, BBR, DCTCP, Homa
Qdisc	fq	fq, fq_codel, pfifo_fast, HHF, SFQ
Byte Queue Limit	Dynamic	[100B-10MB]
MTU	1500	1500, 9000
Process scheduler	CFS	CFS, FIFO, Microquanta

Table 1: Default system configuration and tested parameter ranges.



(a) Synthetic workload



(b) Trace-driven workload

Figure 5: Two sets of workloads used throughout the experiments. The figures show the complementary cumulative distribution functions (CCDFs) of flow sizes.

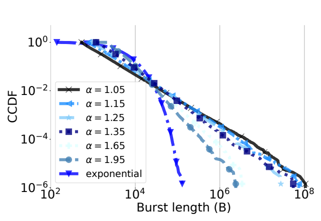
Valinor's userspace data plane based on *DPDK* v20. We disable idle states on all servers and set the frequency governor to *performance* to minimize the interference of power-saving features on networking performance. The default settings for the evaluated components are summarized in Table 1.

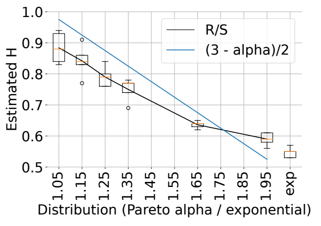
Finally, to calculate microburst lengths, since we use 40Gbps links, we set the burst inter-arrival threshold to 500ns for the presented results (see §2). Valinor also computes microburst lengths for other threshold settings (ranging from 5ns to 10μs). While the threshold setting impacts the size and quantity of observed bursts, we did not notice any difference in relative burstiness when comparing multiple cases.

5.1 Revisiting structural causality

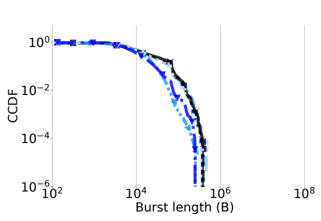
Where does traffic burstiness come from? Prior work [23, 53, 54] shows that the heavy-tailed property of the flow size distribution directly determines link-level traffic self-similarity, a phenomenon that is sometimes referred to as *structural causality*. Heavy-tailed flow size distributions are shown to be the sufficient condition for generating scale-invariant burstiness and the network stack is shown to play a negligible role in self-similarity [23, 53]. For instance, for traffic generated by TCP Reno for a heavy-tailed Pareto file size distribution with the shape parameter α, there exists an almost linear relation between H and α: the estimated H is close to $(3-\alpha)/2$.[7] Heavier tailed distributions (i.e., α close to 1) are more strongly self-similar (H closer to 1). The self-similarity of traffic with heavy-tailed flow sizes is in contrast to the lack

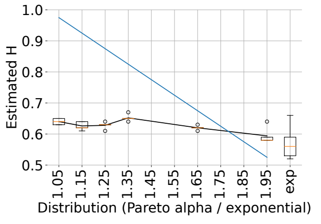
[7]The $H = (3-\alpha)/2$ relation shows the values of H predicted by a theoretical ON/OFF model in the idealized case corresponding to a fractional Gaussian noise process with independent traffic sources with constant ON/OFF amplitude [54]. This captures an ideal self-similar process.



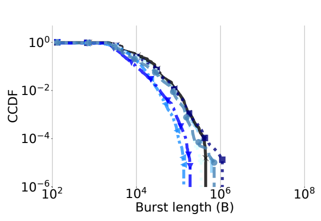


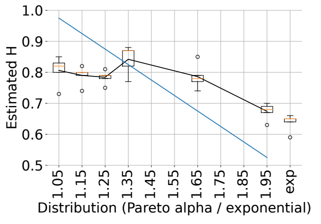
(a) Simulation microburst sizes **(b) Simulation Hurst exponents**





(c) In-host microburst sizes **(d) In-host Hurst exponents**





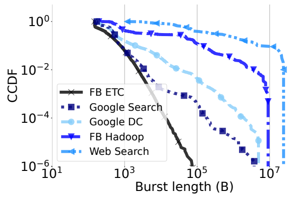
(e) In-network microburst sizes **(f) In-network Hurst exponents**

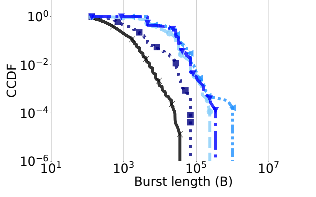
Figure 6: Microburst sizes and Hurst exponents of different synthetic workloads for simulated and testbed experiments. The interference of host networking elements is visible in the difference between the three scenarios.

of correlation structures for short-tailed flow size distributions such as an exponential distribution (H close to 0.5).

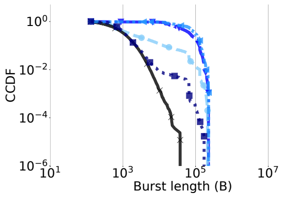
We first replicate this result using OMNET [1], an extensively used simulator [8, 14, 47], and observe an almost linear relation between α and H—consistent with the findings of prior work [53], the estimated H values closely track the $(3-\alpha)/2$ line. In a setup where the two simulated servers are connected via a network switch, we establish 32 long-running TCP connections and use Pareto and exponential flow size distributions (Figure 5a shows the flow size distributions). To achieve a target offered load of 6 Gbps, flows are initiated exponentially with a mean interarrival time of 87μs. We repeat each experiment five times. In the box and whisker plots, each box depicts the 1st and 3rd quartiles, the whiskers represent the upper and lower extremes, the circles are outlier points, and the orange dashes show the median Hurst estimates. Figure 6b shows that heavy-tailed flow size distributions generate self-similar traffic. Figure 6a shows that these distributions also result in larger microbursts with heavier tails.

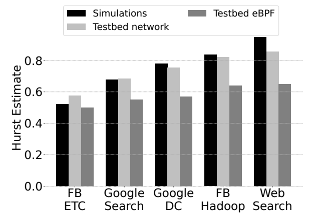
Next, we repeat the above scenario in a testbed, using Valinor to analyze burstiness after the software stack and on the wire. Using Valinor-H for in-host analysis, we observe that





(a) Simulation microburst sizes **(b) In-network microburst sizes**





(c) In-host microburst sizes **(d) Estim. Hurst exponents**

Figure 7: Self-similarity and microburst sizes vary across workloads and between testbed and simulation results. Positioned before the NIC, Valinor-H captures a smoother snapshot of traffic than in-network measurements.

the impact of the heavy-tailed distributions on self-similarity is barely visible at this stage with distributions with varying α parameters behaving similarly and close to a light-tail exponential distribution (Figure 6d), e.g., the software stack greatly diminishes the degree of self-similarity of heavy-tailed Pareto distribution with $\alpha = 1.05$ from $H = 0.88$ in the simulations (Figure 6b) to $H = 0.64$ at the eBPF hook (Figure 6d). We observe a similar effect on the microburst size distributions that are much more similar across different workloads and have shorter tails (Figure 6c).

We next use Valinor-N for analyzing traffic as observed on the wire. The patterns again change in interesting and non-uniform ways. Similar to in-host measurements, the in-network measurements indicate that the influence of flow size on self-similarity is lower than the simulated experiments, e.g., $H = 0.80$ and $H = 0.78$ for $\alpha = 1.05$ and $\alpha = 1.65$, respectively, on the wire in the testbed experiments compared to $H = 0.88$ and $H = 0.63$ for the same workloads in the simulated experiments (Figure 6f). The more amplified long-range burstiness in the network compared to in-host experiments is due to the intervention of driver and NIC functions (such as segmentation offloading scheduling) that reside below Valinor-H. We investigate the roles of these functions in §5.3. Figure 6e shows that the flow size distribution has a relatively subdued impact on the ultimate size of microbursts on the wire once the traffic traverses the host networking stack.

Summary: The shape of the traffic in the testbed experiments (in-network and in-host) is substantially different compared to the simulated experiments with identical setups. This suggests that host networking elements (e.g., qdiscs, process schedulers, and NIC schedulers, not modeled in common simulators) alter burstiness.

5.2 Impact of workloads

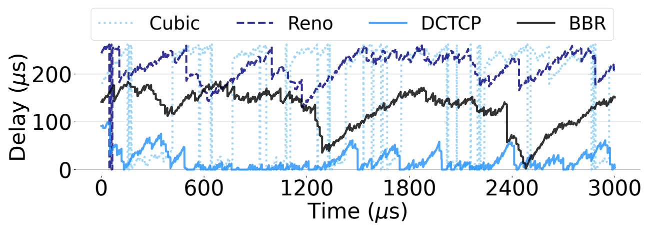
Next, we repeat the above experiments (using both the simulator and the testbed) by replaying the traces of five classes of workloads from [47]: (1) Facebook's ETC workload, (2) Google search workload, (3) Google's aggregated internal data center workload, (4) Facebook's Hadoop workload, (5) DCTCP's web search workload [9]. Figure 5b shows the flow size distributions of these traces. Similar to the previous experiments, in simulations, there exists a direct correlation between the flow size distributions, self-similarity, and the burst lengths (Figure 7). In the testbed, however, the difference in burst lengths starts to fade away as host networking components come into play. We also observe that the scaling behavior varies substantially across different workloads and between the simulated and testbed experiments. Hurst coefficients are larger for the more heavy-tailed distributions in the network but mostly homogeneous before reaching the driver. For example, the self-similarity estimates for the ETC workload (p99th flow size = 1.8 KB), the Google DC workload (p99th flow size = 31 KB), and the web search workload (p99th flow size = 27 MB) are 0.57, 0.75, and 0.85, respectively for in-network measurements and 0.50, 0.57, and 0.65, respectively for in-host measurements.

5.3 Sources and implications of burstiness

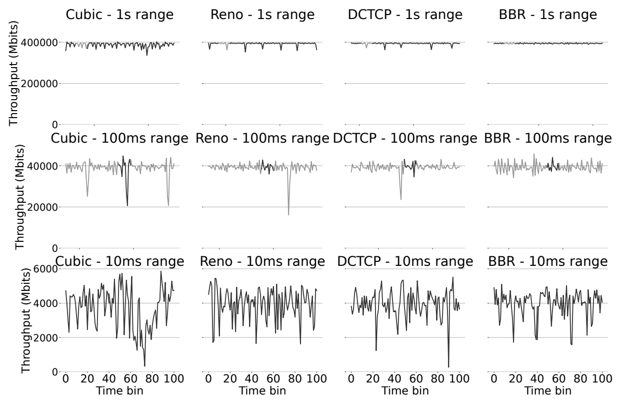
The previous section shows the aggregate impact of host networking elements on bursts. In this section, we measure the impact of each element, starting with the transport layer and moving to the elements that operate *below* the TCP/IP stack (e.g., qdiscs) and *in parallel* to it (e.g., the process scheduler).

5.3.1 Transports and congestion control

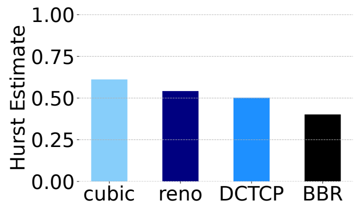
Starting with transports, we evaluate four TCP congestion control variants under a mixture of background traffic and a small-scale incast traffic pattern where two sender machines target one receiver. The background traffic consists of two *iperf* flows each taking 18Gbps of bottleneck link bandwidth. The incast traffic follows the map-reduce workload size distribution. For this experiment only, we run both the workload generators and the applications outside the container environment. Figure 8a shows how TCP Cubic [28], TCP Reno, DCTCP [9], and BBR [17] react to queue buildups in the network. Compared to Reno, TCP Cubic (the default congestion control setting in recent versions of Linux kernels) uses a more aggressive function for increasing its congestion window upon receiving acknowledgments. Therefore, it experiences larger queueing oscillations than Reno. BBR uses round-trip times to adjust its transmission window and varies its pacing rate to keep the in-flight bytes near its estimated bandwidth-delay product. Thus, it experiences a more steady queueing behavior while trying to keep the buffer half full.



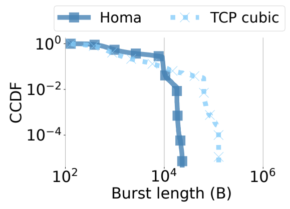
(a) Buffer occupancy under Incast



(b) Timeseries of packet arrivals



(c) H estimates for TCP variants



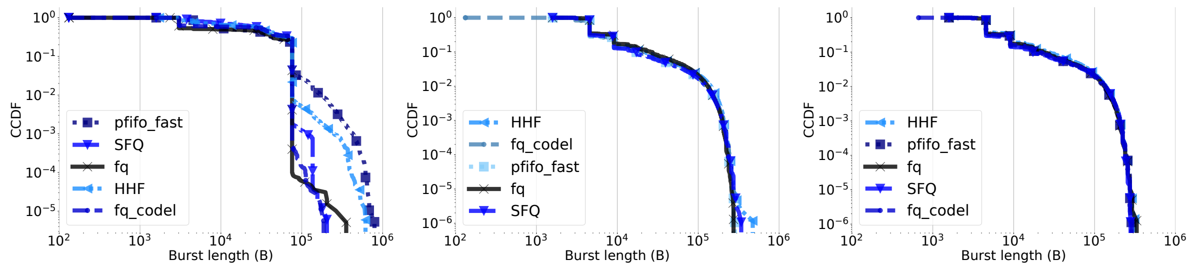
(d) Homa vs Cubic bursts

Figure 8: (a) Valinor captures the in-network buffer occupancy for different transport protocols. (b), (c) Timeseries and H coefficients show that burstiness (at both short and long timescales) varies significantly across transport protocols. (d) A receiver-driven transport, Homa, is less bursty than TCP Cubic.

Finally, DCTCP uses explicit congestion notifications from switches to maintain consistently low queuing.

Figure 8b presents the throughput timeseries of the four congestion control variants at different timescales followed by their Hurst exponent estimates in Figure 8c. With the help of pacing and RTT estimations, BBR is able to maintain a steady throughput and a non-bursty traffic shape, reflected by $H = 0.40$. On the other hand, Cubic's less conservative transmissions incur a self-similarity estimate of 0.60.

Finally, we deploy Homa's kernel module [47] as a representative implementation of receiver-driven transports in the Linux kernel. In receiver-driven transports, the destination initiates more packets by issuing *grant* control packets for the sending host. In our setup, Homa sends the first 90 KB of each flow unscheduled as an attempt to initiate the communication and retrieve the path's congestion status. The



(a) SQ w/out Offloading **(b) SQ w/ Offloading** **(c) MQ w/ Offloading**

Figure 9: Burst behavior of Linux queueing disciplines in the absence and presence of offloading (Offloading) and NIC scheduling (SQ=single-queue, MQ=multi-queue). Both MQ and TCP segmentation offload compromise the intended shape of software packet scheduling.

following packets are then scheduled using grants. Due to its limited implementation scope, the Homa module is not able to achieve line rate performance. Therefore, we limit our observation to the map-reduce workload tuned down to 6 Gbps offered load. Figure 8d presents the burst lengths for Homa and TCP Cubic as observed by Valinor-H eBPF framework. We observe that the p99 burst length under Homa is 9× lower than Cubic which might reflect two facts. First, unlike Cubic which sends up to 64 KB long data chunks, Homa's prepared *sk_buff* chunks are mostly as large as its MTU (9 KB in this experiment). This is also due to the fact that Homa kernel module is not making use of TSO because of certain Intel NIC limitations. Secondly, Homa uses pacing to keep the NIC fully saturated in its Linux implementation which further controls the spacing between its transmissions [52]. Combined, these factors result in Homa's less bursty behavior compared to TCP Cubic, not just at small timescales (Figure 8d) but also at large timescales ($H = 0.54$ for Homa *vs.* 0.62 for Cubic). However, we suspect a different behavior from Homa on different setups that can make use of NIC offloading.

5.3.2 Software switching

Linux leverages queueing disciplines (*qdiscs*) to enforce scheduling among segments originating from different applications in the system. If generic segmentation offload is not in use, *qdiscs* are the last software components to decide the order of data entities on NIC's FIFO rings. We study five representative queueing disciplines implemented in Linux:
1) Fair queue (fq) is the default scheduler in recent Linux kernels and is mainly used to enforce pacing on a per-flow (per socket) basis. The appropriate pace among flows is either explicitly enforced via socket options, or is determined by the TCP congestion control (e.g., BBR). By default, *fq* uses deficit round-robin with a default quantum of 3028 bytes to drain flow queues, with an initial quantum equalling TCP's initial 10-packet window.
2) fq_CoDel. The controlled delay (CoDel) algorithm, combined with fair queue, enforces CoDel on per-flow sub-queues. CoDel, a more recent AQM algorithm, uses packet sojourn time inside each flow queue to detect slow flows and prevents the queueing delay to exceed a user-specified target by dropping excess traffic.
3) Stochastic Fair Queueing (SFQ) extends flow-queuing with random-early marking/drop semantics with small default queue sizing to control the queueing delay. Similar to *fq*, it uses round-robin scheduling on per-flow sub-queues. SFQ uses a default deficit of one MTU.
4) pfifo_fast is a First-In First-Out priority queue. Higher priority packets are distinguished by their Type of Service (TOS) fields in IP headers which are set by upper layers.
5) Heavy Hitter Filter (HHF) attempts to identify and separate short flows from heavy hitters to prevent head-of-the-line blocking and increased delays for latency-sensitive flows. Such flows are given a higher deficit compared to heavy hitters in each transmission round.

We study qdiscs under three scenarios: First, to see the actual contribution of qdiscs to the traffic shape, we disable segmentation offload and serialization offload and limit the number of the transmit rings to one (single-queue). Segmentation is the process of breaking large *sk_buffs* into MTU-sized segments and is usually deferred to the last processing stages to reduce CPU utilization and improve flow performance. Segmentation offload can either be performed in the hardware (TCP Segmentation Offload or TSO) or just before passing the data to the hardware (Generic Segmentation Offload or GSO). Additionally, in a multi-queue architecture, the network stack communicates to the NIC via separate ring buffers pinned to each CPU core to reduce inter-core communication overheads and improve throughput. When enabled, a (reportedly, round-robin [65]) packet scheduler in the hardware will decide the order in which packets are drained from ring buffers.

Initially, we run 1000 Iperf instances spread across 200 containers, simulating the map-reduce workload on the single-queue server without offloading. Figure 9a demonstrates how, *in isolation*, per-flow queuing can significantly shorten the size of egress bursts. Techniques such *pfifo_fast*, and HHF use one large buffer containing packets from all egress flows, allowing multiple data segments of one flow to be enqueued simultaneously. On the other hand, per-flow queueing allows

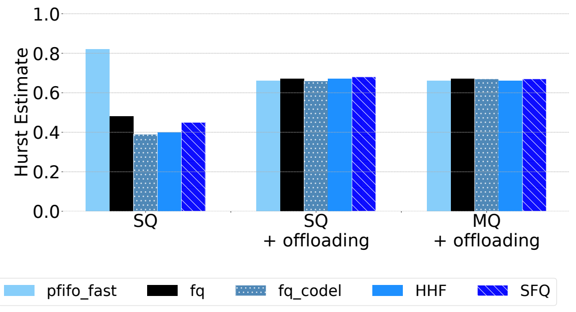


Figure 10: Hurst estimates for different queueing disciplines.

the scheduler to interleave among packets of different flows, primarily to maintain fairness and prevent head-of-the-line blocking [62].

To verify the impact of round-robin scheduling on blunting bursts, we repeat the same experiment with *fq* qdisc, increasing the per-flow deficit from one MTU (1514 bytes) to 16 MTUs, and observe a linear correlation between fq deficits and burst lengths. For example, the 90^{th} percentile of burst lengths under a deficit of 16 packets is increased to 25 KB from 13 KB under that of 8 MTUs (92% increase).

While qdisc is the last layer to perform packet scheduling in the software, the traffic ultimately often passes through segmentation offloading and NIC scheduling before reaching the wire. Is the impact of qdiscs on the wire preserved *after the interaction* with these lower layers? To investigate, we enable all the offloading features and perform our measurements again. Figure 9b demonstrates the impact of offloading segmentation and serialization on lengthening the egress bursts. With TSO at work, qdiscs no longer serve packets. Instead, they schedule between dynamically sized *sk_buffs*. Hardware offloading then helps increase the throughput by nearly 50% for all cases while moving the buffering to the hardware where large segments are broken into MTU-sized packets and sent on the wire. This significantly undermines qdisc's decisions on shaping the traffic. With offloading in action, the median burst sizes for *fq*, *fq_codel*, and *pfifo_fast* are, 132 KB, 127 KB, and 127 KB, respectively. While without offloading, these systems experienced a median burst length of 76 KB, 76 KB, and 172 KB[8], respectively.

To further ruffle the output of qdiscs, we enable the default multi-ring root qdisc which assigns a separate qdisc instance to each CPU core and enables the NIC scheduler to perform last-level scheduling on transmit rings (multi-queue architecture). Figure 9c presents the outcome. *With NIC scheduling and segmentation offloading at work, the shape of the qdisc's outgoing traffic is barely preserved on the wire.* That is because, NICs are equipped with internal round-robin schedulers to drain the software rings, further reducing the chances of creating long bursts. Finally, Figure 10 demonstrates the estimated Hurst exponents for the three scenarios. Without segmentation offloading, the degree of burstiness is considerably reduced (H < 0.5) for all but one case. Only *pfifo_fast* which does not offer any form of fair queueing suffers from heavier burstiness ($H = 0.8$) under the single-queue scenario.

Implications of disabling offloading and multi-ring scheduling. Apart from burstiness, both offloading and NIC scheduling have a profound impact on flow performance metrics. Our measurements demonstrate that disabling TCP segmentation offload for a workload consisting of 1000 same-size flows results in 71% decline in median flow throughput, 46% increase in median packet RTTs, and $3\times$ increase in sender CPU utilization. Therefore, disabling offloading, in order to enable software control is not always a viable option. Multi-queue NICs are also considered a quick solution with potential side effects. While enabling multi-queue reduces resource contention, they can increase response times and are usually fixed-function [65].

5.3.3 Software pacing

The above observations raise another important question on host networking design decisions. While many congestion control techniques [7, 17, 37, 47, 57] advocate for pacing in order to achieve accurate control over in-transit data, existing pacing implementation in the Linux kernel is deeply away from the wire, at fq qdisc. *Are qdiscs a suitable place for enforcing pacing?* To investigate, we repeat the map-reduce (M/R) workloads on the server with both offloading and NIC scheduling enabled and observe that *for workloads with large flows (intra-rack M/R), pacing doesn't have a significant impact on burstiness, and for those with short flows (intra-cluster M/R), pacing results in throughput reduction.* Overall, our results highlight the limitations of software pacing for data center workloads.

Concretely, we configure *fq* to pace 200 flows based on their fair share of bandwidth (200 Mbps), and gradually increase the portion of the flows that are counted as heavy hitters from 0% (no flow is paced) up to 100% (all flows are paced). Figure 11 compares the bursts for (a) workload with mostly large flows and (b) workload with a mix of small and large flows. In the former workload, we observe that while the impact of pacing ratio is less evident, pacing allows for better bandwidth allocation and the line rate is preserved for all rows. On the other hand, in the latter workload, the throughput is reduced by 22% under pacing. This is because short flows are not able to make up for the freed bandwidth that pacing creates. We also compare packet RTTs and find that pacing large heavy-hitters helps reduce median RTTs by two orders of magnitude as short flows experience less head-of-the-line blocking. This behavior changes in the intra-cluster workload as we do more pacing, as the increased RTT of paced flows drives the overall median RTT up by 160%. Further details on the theoretical analysis of burstiness under software pacing can be found in Appendix §B.

[8]pfifo_fast combined with offloading can exacerbate burstiness as both layers are prone to creating large, uncontrolled bursts.

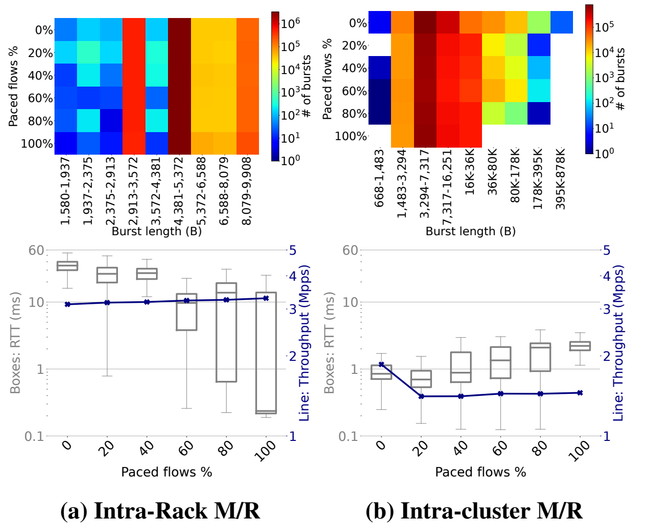


Figure 11: Software pacing is workload dependent. For work-loads consisting of large flows, its impact on smoothing bursts is unmade by lower layers. For workloads with both short and long flows, it reduces throughput.

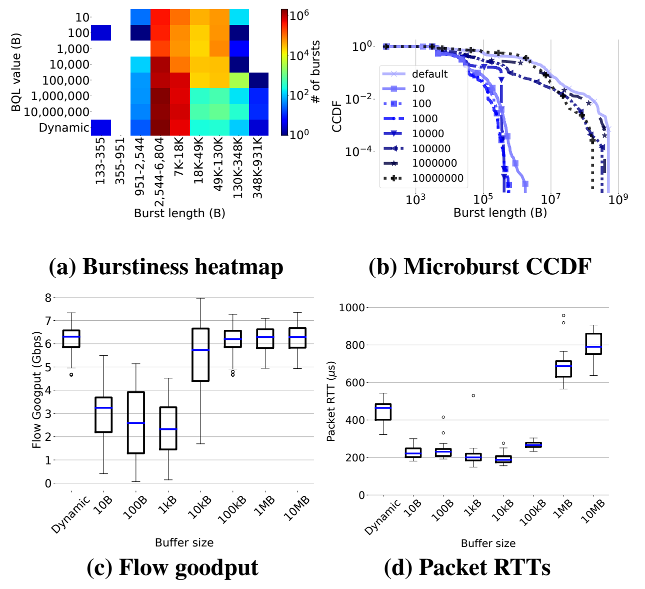


Figure 12: Larger BQL settings produce longer bursts. Also, the dynamic BQL algorithm presents a similar behavior to a large static ring size.

5.3.4 Byte Queue Limits

Linux kernel employs buffers at various stages of network stack processing to streamline the data movement among various components. The NIC driver queue is the last buffering stage before triggering the hardware. A fixed-size driver queue (a.k.a., TX ring) would ensure that the NIC can always find ready-to-send packets without communicating with the OS. However, due to the unpredictable size of packet buffers in the Linux kernel (ranging from 64B up to tens of kilobytes), the queueing time will considerably add to the overall RTT of packets. To prevent that, OS developers propose a dynamic bound on TX rings that adjusts the limit based on NIC's transmission rate and the availability of data in the TX rings [29]. To that end, after every transmission, BQL uses time intervals to check whether the NIC was starved in previous transmissions. If the NIC was not fully utilized during any interval while data was available at higher layers, the BQL algorithm increases the limit on the TX ring. Otherwise, if the NIC was fully busy, the BQL is decreased to reduce the queueing overheads. Enforcing smaller queue limits also ensures that the main queuing occurs at the qdisc-level where more advanced queuing disciplines can be employed.

Apart from Linux, NIC buffer sizing is also an important consideration for kernel-bypass runtimes that are less inclined to distribute TX processing among multiple ring buffers [36, 51]. Figure 12 demonstrates the impact of driver queue size on performance and burstiness. Intuitively, as we increase the size of the driver's buffer, we greatly increase the queueing time experienced by egress traffic, therefore, preventing the bursts of packets from arriving at the NIC. On the other hand, a larger driver queue is more prone to creating longer bursts as it is more susceptible to triggering segmentation offload (99th percentile burst length for 1 KB buffers and 1 MB buffers are 68 KB and 9 KB, respectively, but 99.99th lengths shift to 68 KB and 86 KB, respectively). The microburst length distributions in Figure 12b further suggest that the default dynamic buffer sizing algorithm tends to maintain larger ring buffers which leads to longer bursts.

5.3.5 Linux process scheduling

Apart from the network stack, the operating system features various internal components that might change the traffic shape. For example, Linux offers a range of process scheduling classes suited for various use cases:

Completely Fair Scheduler (CFS) is the default process scheduling class in Linux which aims at achieving fairness among active processes in the system while maintaining responsiveness for I/O-bound applications. when running a mix of compute-intensive and network-intensive workloads, CFS attempts to proportionally share the CPU among workloads leading to longer response times [39].

Real-time scheduler supports two policies: *Round-robin* and *First-In-First-Out (FIFO)* scheduling. Both policies give strict priority to I/O-bound applications (if configured properly). By default, the round-robin policy preempts high-priority processes every 100ms while the FIFO policy is non-preemptive. We also deploy Microquanta [46] a semi-real-time scheduling class with microsecond time precision.

Valinor's picture of traffic burstiness is consistently similar when the network application is running alone as Hurst estimates vary between 0.51 and 0.54 for all the schedulers. How-

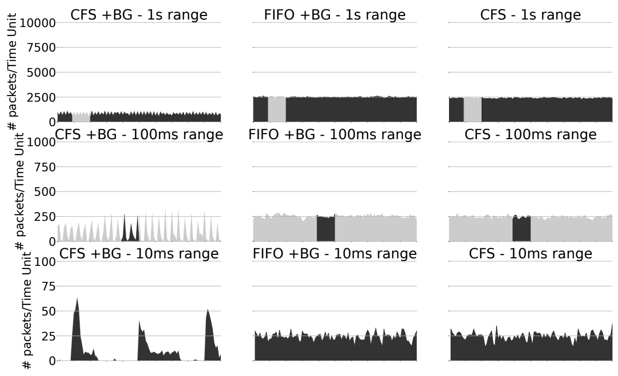


Figure 13: Impact of process scheduling on traffic bursts.

ever, when the background process is introduced, a course-grained process scheduler like CFS must enforce fair CPU time sharing, resulting in the leap of its H estimate to 0.74 while other schedulers are able to schedule the network application's threads in short timescales and result in smooth transmissions. To validate the self-similarity estimates, we plot the time series of CFS (running only the network workload), CFS+BG (running the network workload alongside background threads), and the real-time FIFO scheduler running both applications (FIFO+BG) in Figure 13. The self-similar nature of the CFS+BG scenario is noticeable in the leftmost column as CFS causes the network packets to be sent in larger chunks, causing intermittent but larger bursts at short time spans.

6 Related work

Traffic self-similarity. A large group of studies rely on quantifying bursts by using the notion of self-similarity in the time series [5, 24, 38, 50, 53–55] but overlook the role of host networking on shaping bursts. Valinor leverages the theoretical frameworks developed in these works to uncover *the impact of host networking on bursts.*

Detecting bursts. A growing number of proposals try to identify *what flows* are bursty [18, 19, 40, 48] but they cannot identify *why* those flows are bursty. Crucially, they cannot identify the elements on the traffic path that contribute to or blunt traffic burstiness. Frameworks such as BurstRadar [33] and SynDB [34] rely on buffer congestion or external triggers to capture packet arrivals, which prevents them from capturing long-range dependency patterns in host egress traffic.

Similar to Valinor, a few proposals study the causes of bursts. Some papers pinpoint transport protocol internals such as segmentation, slow start, bulk acknowledgments, and fast re-transmit as potential sources of bursts at the source level [10, 31, 69]. Another category of works study the impact of offloading techniques like segmentation offload on microbursts [35, 72]. Specifically, [35] investigates the impact of application behavior, operating system syscalls, and NIC offloading features on both sender and receiver hosts on burstiness and further show that burstiness imposed by TCP segmentation offload can marginally be controlled by configuring the kernel's maximum GSO size. Compared to these studies, Valinor has a broader scope; it studies the impact of various host elements (not just transport protocols), the effects of low-level offloading mechanisms on software scheduling and pacing, and bursts at various timescales (not just microsecond-scale). Finally, [25] introduces Millisampler, a host-centric burst characterization tool to study the impact of service placement on buffer contention and packet loss. Valinor uses its switch framework to detect synchronized flow arrivals at points of interest and unlike Millisampler which operates at *sk_buff* granularity, can attribute bursts at packet resolution. We believe that Valinor and Millisampler combined can assist data center network operators in accurately detecting the sources of bursty traffic at various timescales.

Burst control. A large and growing number of proposals [9, 27, 32, 37, 41, 43, 44, 59, 61, 62, 70] focus on *controlling* bursts, e.g., via rate-limiting at the switch [44], fine-grained pacing [61], and high-precision transport protocols [37, 41]. These studies are orthogonal to Valinor. Understanding the temporal properties of bursts and the causal mechanisms contributing to burstiness will benefit the design of effective burst control mechanisms.

7 Conclusions

We presented the design of Valinor, a burst measurement framework that consists of an in-host eBPF framework and an in-network timestamping module for programmable switches. Valinor can capture burstiness at different scales (ranging from nanoseconds to seconds). We use Valinor to demonstrate how host networking elements affect bursts. We show that the scaling behavior of traffic at long timescales and burstiness at fine timescales vary significantly across different host networking configurations (process schedulers, congestion control algorithms, single vs. multi-queue NICs, etc.) and across different classes of practical workloads. In particular, we show the impact of hardware-resident functions (e.g., NIC schedulers) that are largely overlooked in characterizing burstiness. This variability of burstiness and the implications of bursts on performance underscore the need for measurement systems to perform periodic burst analysis.

Acknowledgements

We would like to thank our shepherd, Srinivas Narayana, and the anonymous NSDI reviewers for their insightful feedback. We would also like to thank John Ousterhout for his input and feedback, and Xin Jin for his equipment support. This project was partially supported by an Intel Fast Forward award, a Facebook faculty research award, and NSF CNS grant 1910821.

References

[1] Omnet++ simulator. https://omnetpp.org/, 2022.

[2] Open vswitch. http://openvswitch.org/, 2022.

[3] Redis: an open source, in-memory data structure store. https://redis.io, 2022.

[4] ABDOUS, S., SHARAFZADEH, E., AND GHORBANI, S. Burst-tolerant datacenter networks with Vertigo. In *CoNEXT* (2021).

[5] ADAS, A., AND MUKHERJEE, A. On resource management and qos guarantees for long range dependent traffic. In *INFOCOM* (1995).

[6] ADDIE, R. G., ZUKERMAN, M., AND NEAME, T. Fractal traffic: measurements, modelling and performance evaluation. In *INFOCOM* (1995).

[7] AGGARWAL, A., SAVAGE, S., AND ANDERSON, T. Understanding the performance of TCP pacing. In *INFOCOM* (2000).

[8] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: distributed congestion-aware load balancing for datacenters. In *SIGCOMM* (2014).

[9] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).

[10] ALLMAN, M., AND BLANTON, E. Notes on burst mitigation for transport protocols. *SIGCOMM CCR* (2005).

[11] ARASHLOO, M. T., LAVROV, A., GHOBADI, M., REXFORD, J., WALKER, D., AND WENTZLAFF, D. Enabling programmable transport protocols in high-speed NICs. In *NSDI* (2020).

[12] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *SIGMETRICS* (2012).

[13] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *IMC* (2010).

[14] BESTA, M., SCHNEIDER, M., KONIECZNY, M., CYNK, K., HENRIKSSON, E., GIROLAMO, S. D., SINGLA, A., AND HOEFLER, T. FatPaths: Routing in supercomputers and data centers when shortest paths fall short. In *SC* (2020).

[15] BURSTEIN, I. Nvidia data center processing unit (DPU) architecture. In *IEEE HCS* (2021).

[16] CAI, Q., CHAUDHARY, S., VUPPALAPATI, M., HWANG, J., AND AGARWAL, R. Understanding host network stack overheads. In *SIGCOMM* (2021).

[17] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. BBR: congestion-based congestion control. *ACM Queue* (2016).

[18] CHEN, X., FEIBISH, S. L., KORAL, Y., REXFORD, J., AND ROTTENSTREICH, O. Catching the microburst culprits with snappy. In *SelfDN* (2018).

[19] CHEN, X., FEIBISH, S. L., KORAL, Y., REXFORD, J., ROTTENSTREICH, O., MONETTI, S. A., AND WANG, T.-Y. Fine-grained queue measurement in the data plane. In *CoNEXT* (2019).

[20] CORBET, J. TCP small queues. https://lwn.net/Articles/507065/, 2012.

[21] CROVELLA, M. E., AND BESTAVROS, A. Self-similarity in World Wide Web traffic: evidence and possible causes. *ToN* (1997).

[22] DUFFIELD, N. G., AND O'CONNELL, N. Large deviations and overflow probabilities for the general single-server queue, with applications. In *Mathematical Proceedings of the Cambridge Philosophical Society* (1995).

[23] FELDMANN, A., GILBERT, A. C., HUANG, P., AND WILLINGER, W. Dynamics of ip traffic: A study of the role of variability and the impact of control. In *SIGCOMM* (1999).

[24] GARRETT, M. W., AND WILLINGER, W. Analysis, modeling and generation of self-similar VBR video traffic. *SIGCOMM CCR* (1994).

[25] GHABASHNEH, E., ZHAO, Y., LUMEZANU, C., SPRING, N., SUNDARESAN, S., AND RAO, S. A microscopic view of bursts, buffer contention, and loss in data centers. In *IMC* (2022).

[26] GOVINDAN, R., MINEI, I., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Evolve or die: high-availability design principles drawn from googles network infrastructure. In *SIGCOMM* (2016).

[27] GOYAL, SHAH, ZHAO, NIKOLAIDIS, AND OTHERS. Backpressure flow control. In *NSDI* (2022).

[28] HA, S., RHEE, I., AND XU, L. CUBIC: a new TCP-friendly high-speed TCP variant. *SOSR* (2008).

[29] HERBERT, T. bql: Byte Queue Limits. https://lwn.net/Articles/469652/, 2011.

[30] HURST H. E. Long-Term storage capacity of reservoirs. *Trans. of the American Soc. of Civil Eng.* (1951).

[31] JIANG, H., AND DOVROLIS, C. Source-level IP packet bursts. In *IMC* (2003).

[32] JIN, P., GUO, J., XIAO, Y., SHI, R., NIU, Y., LIU, F., QIAN, C., AND WANG, Y. PostMan: Rapidly mitigating bursty traffic by offloading packet processing. In *SoCC* (2019).

[33] JOSHI, R., QU, T., CHAN, M. C., LEONG, B., AND LOO, B. T. BurstRadar: Practical real-time microburst monitoring for datacenter networks. In *APSys* (2018).

[34] KANNAN, P. G., BUDHDEV, N., JOSHI, R., AND CHAN, M. C. Debugging transient faults in data centers using synchronized network-wide packet histories. In *NSDI* (2021).

[35] KAPOOR, R., SNOEREN, A. C., VOELKER, G. M., AND PORTER, G. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *CoNEXT* (2013).

[36] KAUFMANN, A., STAMLER, T., PETER, S., SHARMA, N. K., KRISHNAMURTHY, A., AND ANDERSON, T. TAS: TCP acceleration as an OS service. In *EuroSys* (2019).

[37] KUMAR, G., DUKKIPATI, N., JANG, K., WASSEL, H. M. G., WU, X., MONTAZERI, B., WANG, Y., SPRINGBORN, K., ALFELD, C., RYAN, M., WETHERALL, D., AND VAHDAT, A. Swift: delay is simple and effective for congestion control in the datacenter. In *SIGCOMM* (2020).

[38] LELAND, W. E. On the self-similar nature of Ethernet traffic (extended version). *ToN* (1994).

[39] LI, J., SHARMA, N. K., PORTS, D. R. K., AND GRIBBLE, S. D. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *SoCC* (2014).

[40] LI, Y., MIAO, R., KIM, C., AND YU, M. FlowRadar: a better NetFlow for data centers. In *NSDI* (2016).

[41] LI, Y., MIAO, R., LIU, H. H., ZHUANG, Y., FENG, F., TANG, L., CAO, Z., ZHANG, M., KELLY, F., ALIZADEH, M., AND YU, M. HPCC: high precision congestion control. In *SIGCOMM* (2019).

[42] LIKHANOV, N., TSYBAKOV, B., AND GEORGANAS, N. D. Analysis of an atm buffer with self-similar (" fractal") input traffic. In *INFOCOM* (1995).

[43] LIM, H., BAI, W., ZHU, Y., JUNG, Y., AND HAN, D. Towards timeout-less transport in commodity datacenter networks. In *EuroSys* (2021).

[44] LIU, K., TIAN, C., WANG, Q., ZHENG, H., YU, P., SUN, W., XU, Y., MENG, K., HAN, L., FU, J., DOU, W., AND CHEN, G. Floodgate: taming incast in datacenter networks. In *CoNEXT* (2021).

[45] MANDELBROT, B. B. Self-Affine Fractals and Fractal Dimension. *Physica Scripta* (1985).

[46] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., KIDD, N., KONONOV, R., KUMAR, G., MAUER, C., MUSICK, E., OLSON, L., RUBOW, E., RYAN, M., SPRINGBORN, K., TURNER, P., VALANCIUS, V., WANG, X., AND VAHDAT, A. Snap: A Microkernel Approach to Host Networking. In *SOSP* (2019).

[47] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM* (2018).

[48] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Trumpet: Timely and precise triggers in data centers. In *SIGCOMM* (2016).

[49] NICHOLS, K., AND JACOBSON, V. Controlling Queue Delay: A modern AQM is just one piece of the solution to bufferbloat. *ACM Queue* (2012).

[50] NORROS, I. A storage model with self-similar input. *Queueing systems* (1994).

[51] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI* (2019).

[52] OUSTERHOUT, J. A Linux kernel implementation of the Homa transport protocol. In *ATC* (2021).

[53] PARK, K., KIM, G., AND CROVELLA, M. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *ICNP* (1996).

[54] PARK, K., KIM, G., AND CROVELLA, M. E. Effect of traffic self-similarity on network performance. In *Performance and Control of Network Systems* (1997).

[55] PARK, K., AND WILLINGER, W. Self-similar network traffic: An overview. *Self-Similar Network Traffic and Performance Evaluation* (2000).

[56] PAXSON, V., AND FLOYD, S. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM ToN* (1995).

[57] PRAKASH, P., DIXIT, A., HU, Y. C., AND KOMPELLA, R. The TCP outcast problem: exposing unfairness in data center networks. In *NSDI* (2012).

[58] Raghavan, D., Levis, P., Zaharia, M., and Zhang, I. Breakfast of champions: towards zero-copy serialization with NIC scatter-gather. In *HotOS* (2021).

[59] Rezaei, H., and Vamanan, B. Superways: A datacenter topology for incast-heavy workloads. In *WWW* (2021).

[60] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).

[61] Saeed, A., Dukkipati, N., Valancius, V., The Lam, V., Contavalli, C., and Vahdat, A. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM* (2017).

[62] Sanaee, A., Shahinfar, F., Antichi, G., and Stephens, B. E. Backdraft: a lossless virtual switch that prevents the slow receiver problem. In *NSDI* (2022).

[63] Shan, D., Ren, F., Cheng, P., Shu, R., and Guo, C. Micro-burst in data centers: observations, analysis, and mitigations. In *IEEE ICMP* (2018).

[64] Stephens, B., Akella, A., and Swift, M. Loom: Flexible and Efficient NIC Packet Scheduling. In *NSDI* (2019).

[65] Stephens, B., Singhvi, A., Akella, A., and Swift, M. Titan: Fair packet scheduling for commodity multiqueue NICs. In *ATC* (2017).

[66] Tuan, T., and Park, K. Multiple time scale congestion control for self-similar network traffic. *Performance Evaluation* (1999).

[67] Weron, R. Estimating long-range dependence: finite sample properties and confidence intervals. *Physica A: Statistical Mechanics and its Applications* (2002).

[68] Woodruff, J., Moore, A. W., and Zilberman, N. Measuring burstiness in data center applications. In *ACM BS* (2019).

[69] Wu-Chun Feng, Tinnakornsrisuphap, P., and Philip, I. On the burstiness of the TCP congestion-control mechanism in a distributed computing system. In *ICDCS* (2000).

[70] Yan, S., Wang, X., Zheng, X., Xia, Y., Liu, D., and Deng, W. ACC: automatic ECN tuning for high-speed datacenter networks. In *SIGCOMM* (2021).

[71] Zhang, M., Zhang, J., Wang, R., Govindan, R., Mogul, J. C., and Vahdat, A. Gemini: Practical Reconfigurable Datacenter Networks with Topology and Traffic Engineering. *arXiv cs.NI 2110.08374* (2021).

[72] Zhang, Q., Liu, V., Zeng, H., and Krishnamurthy, A. High-resolution measurement of data center microbursts. In *IMC* (2017).

[73] Zhou, Y., Zhang, Y., Yu, M., Wang, G., Cao, D., Sung, E., and Wong, S. Evolvable Network Telemetry at Facebook. In *NSDI* (2022).

A Rescaled-range analysis for estimating H

For a weak stationary stochastic process $X = (X_t : t = 0,1,2,...,N)$, the mean-adjusted series is defined as Y, $Y_t = X_t - m$ where m is the empirical mean of the process X. Let Z denote the cumulative deviate of Y where $Z_t = \sum_{t=1}^{N} Y_t$. We also define m_t as the cumulative mean of the series X through time t.

The rescaled range of X is denoted by

$$(R/S)_t = \frac{R_t}{S_t}, t \in \{0,1,2,...,N\} \quad (3)$$

where the Range series R is defined as

$$R_t = Max(Z_1, Z_2, ..., Z_N) - Min(Z_1, Z_2, ..., Z_N), \\ t \in \{0,1,2,...,N\} \quad (4)$$

and the standard deviation series S is defined as

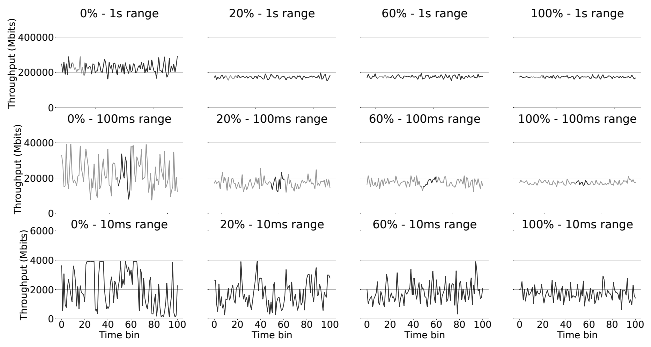
$$S_t = \sqrt{\frac{1}{t}\sum_{i=1}^{t}(X_i - m_t)^2}, t \in \{0,1,2,...,N\} \quad (5)$$

According to [30], R/S scales with the power law of t. Therefore, to estimate H, the slope of the least-squares linear regression of R/S over t in a log-log scale is used. The resulting exponent is in the 0-1 range and a value between 0.5 to 1 indicates low to strong long-range dependence (self-similarity), respectively. In other words, an H estimate close to one indicates a strong desire to maintain the previous trend or more burstiness. As the H estimate nears 0.5, the time series becomes indistinguishable from random noise, and a value close to zero signifies the traffic's aim at reverting to its mean value.

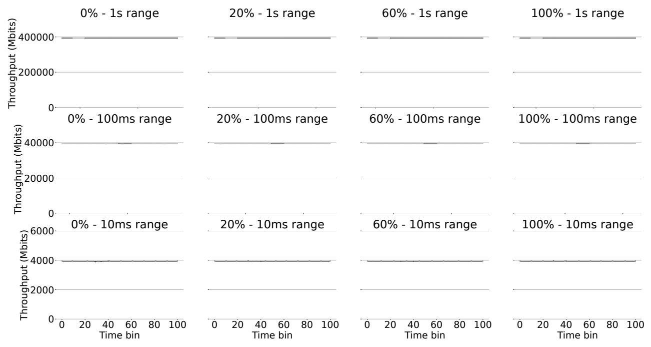
B Theoretical analysis of software pacing under different workloads

We presented the size of per-flow bursts for explicit software pacing in §5.3.3. To further verify our findings using the notion of self-similarity, we first plot the time-series of packet arrivals in 1s, 100ms, and 10ms time scales in Figure 14 for both the intra-cluster (Figure 14a) and intra-rack (Figure 14b) workloads. One can notice the gradual decay of burstiness in all time scales as higher degrees of pacing are enforced to the intra-cluster workload. On the other hand, we can observe that the intra-rack traffic follows a non-bursty, steady trend in all time scales regardless of pacing.

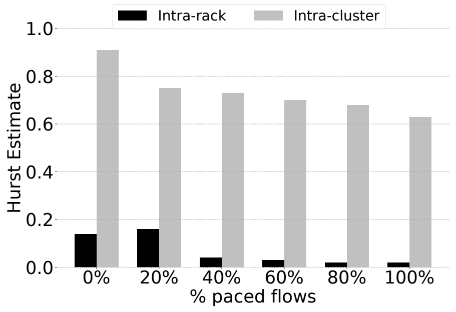
Next, we calculate the Hurst exponents for the intra-rack and intra-cluster workloads. According to Figure 14c, self-similarity in the latter workload follows the degree of pacing (i.e., percentage of the paced flows) where 100% pacing results in 31% reduction in the Hurst estimate compared to the no-pacing case. For example, the Hurst estimates are 0.91, 0.73, and 0.63 for 0%, 40%, and 100% pacing ratios, respectively. However, under the intra-rack workload pacing seems



(a) Intra-cluster time series



(b) Intra-rack time series

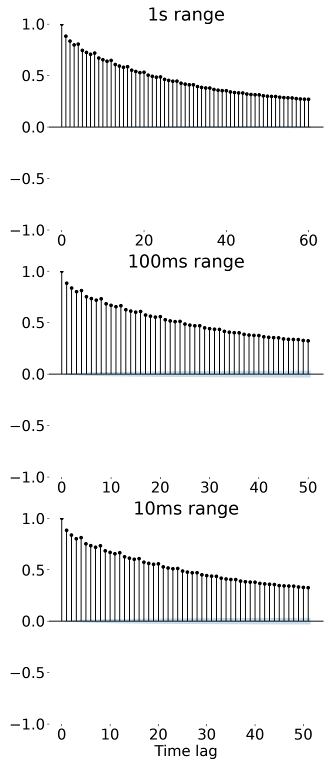


(c) Hurst exponent estimations

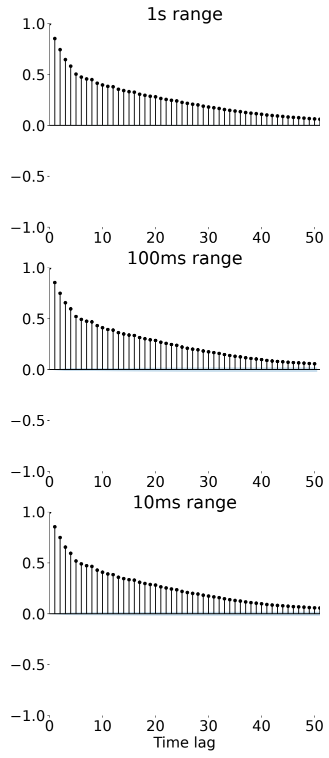
Figure 14: Time-series graphs and Hurst exponents for the software pacing experiments presenting the traffic behavior at three time ranges.

to have little to no effect as the egress traffic follows a mean-reverting behavior during 1-second time ranges (H < 0.20 for all the cases).

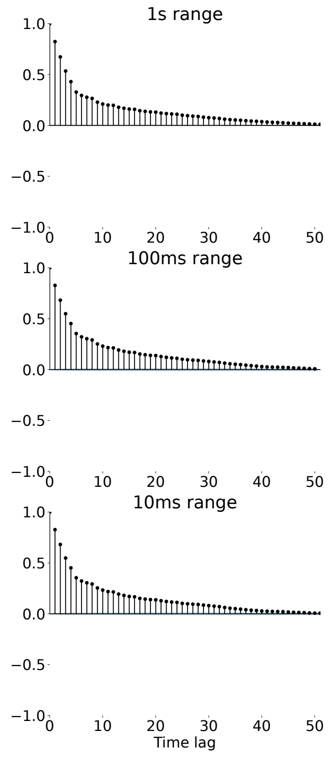
Finally, Figure 15 presents the corresponding auto-correlation functions (ACFs) for the above time series. While the cycling trend of bars between positive and negative correlations suggests a strong mean-reverting behavior for the intra-rack workload (Figures 15e-15h), the intra-cluster ACF features a slow-decaying, strong positive correlations across time lags, suggesting strong self-similarity (Figures 15a-15d). As we increase the pacing ratio (from 0% gradually to 100%), the correlations start to decline.



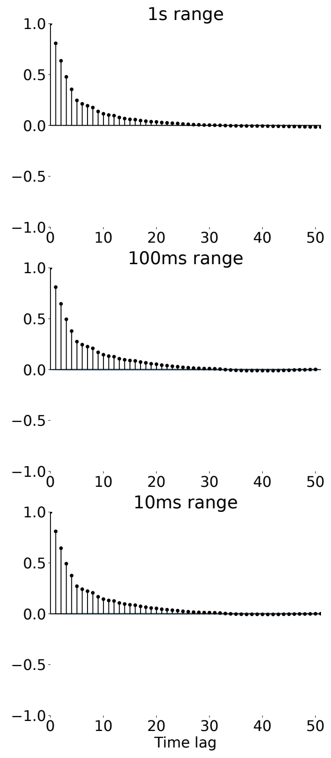
(a) Intra-cluster 0% paced



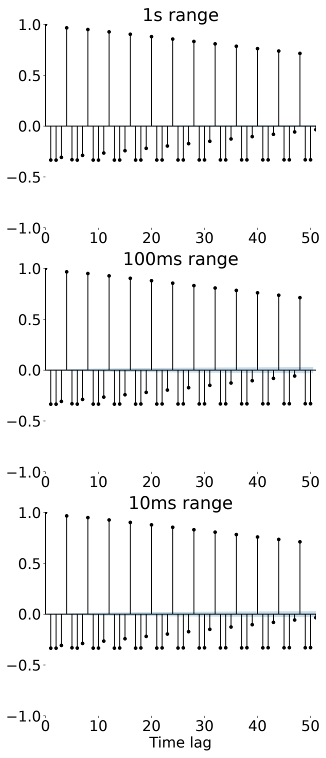
(b) Intra-cluster 20% paced



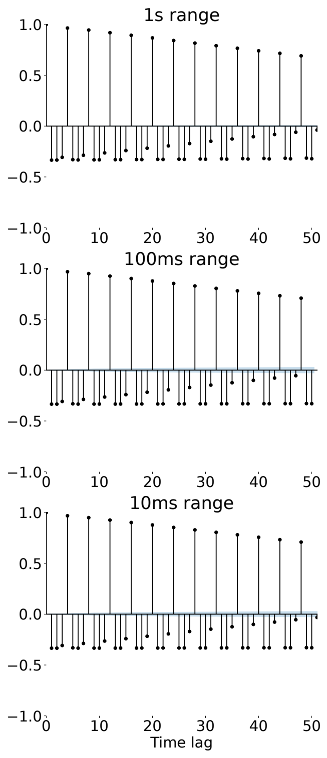
(c) Intra-cluster 60% paced



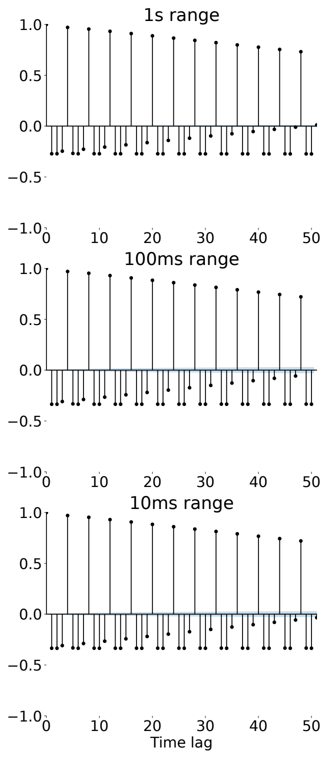
(d) Intra-cluster 100% paced



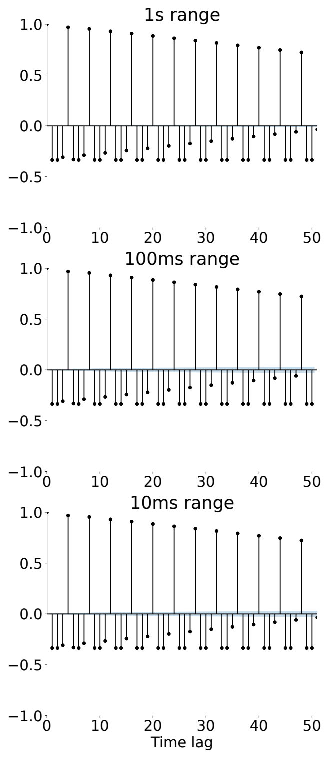
(e) Intra-rack 0% paced



(f) Intra-rack 20% paced



(g) Intra-rack 60% paced



(h) Intra-rack 100% paced

Figure 15: Comparing the auto-correlation functions (ACFs) for two workloads when we increase the ratio of paced flows from 0% to 100%. For the intra-cluster workload, enforcing pacing on flows can significantly reduce the self-similarities. For the intra-rack workload, the correlations between consecutive time lags oscillate between positive and negative numbers, signifying the mean-reverting nature of the workload irrespective of the pacing.