# Delay Sensitivity-driven Congestion Mitigation for HPC Systems

Archit Patke
University of Illinois at Urbana-Champaign

Saurabh Jha
University of Illinois at Urbana-Champaign

Haoran Qiu
University of Illinois at Urbana-Champaign

Jim Brandt
Sandia National Laboratories

Ann Gentile
Sandia National Laboratories

Joe Greenseid
Cray/Hewlett Packard Enterprise

Zbigniew T. Kalbarczyk
University of Illinois at Urbana-Champaign

Ravishankar K. Iyer
University of Illinois at Urbana-Champaign

## ABSTRACT

Modern high-performance computing (HPC) systems concurrently execute multiple distributed applications that contend for the high-speed network leading to congestion. Consequently, application runtime variability and suboptimal system utilization are observed in production systems. To address these problems, we propose Netscope, a congestion mitigation framework based on a novel *delay sensitivity* metric that quantifies the impact of congestion on application runtime. Netscope uses delay sensitivity estimates to drive a congestion mitigation mechanism to selectively throttle applications that are less susceptible to congestion. We evaluate Netscope on two Cray Aries systems, including a production supercomputer, on common scientific applications. Our evaluation shows that Netscope has a low training cost and accurately estimates the impact of congestion on application runtime with a correlation between 0.7 and 0.9. Moreover, Netscope reduces application tail runtime increase by up to 16.3× while improving the median system utility by 12%.

## CCS CONCEPTS

• **Networks** → *Data center networks*; **Network resources allocation**; *Application layer protocols*; **Network performance modeling**; *Network performance analysis*; **Network measurement**.

## KEYWORDS

congestion, interconnect, high-speed networks, application-aware, high-performance computing

**ACM Reference Format:**
Archit Patke, Saurabh Jha, Haoran Qiu, Jim Brandt, Ann Gentile, Joe Greenseid, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2021. Delay Sensitivity-driven Congestion Mitigation for HPC Systems. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3460362


*ICS '21, June 14–17, 2021, Virtual Event, USA*

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460362

## 1 INTRODUCTION

Modern high-performance computing (HPC) systems concurrently execute multiple distributed applications that contend for the high-speed network leading to congestion. To limit contention in the network, several congestion control (CC) mechanisms that restrict application traffic injection into the network have been proposed [1–6]. However, application runtime variability (i.e., variable job completion times for the exact same input parameters) and suboptimal system utilization (as measured by increased node hours to completion) continue to be a problem in production systems [7, 8]. For example, runtime variability of up to 100% has been observed in a 512 node MILC application [9], which projects to several million node hours of wasted compute time (based on [10]).

**Approach.** We introduce a new metric *delay sensitivity* of an application to quantify the impact of congestion on runtime. Delay sensitivity is defined as the gradient of application runtime with respect to network congestion (measured as message delivery time). The delay sensitivity of an application is provided as a parameter to an AIMD [11] (Additive Increase Multiplicative Decrease) controller to quantify the extent of throttling a low delay sensitivity application, while implicitly rewarding the high delay sensitivity application dynamically. The mechanism was tested on Cori (a NERSC supercomputer) with common scientific applications (on over 100 nodes), and was shown to reduce runtime variability by 7-15× compared to Cray static rate control and DCQCN [12] (state-of-the-art in RDMA networks), while incurring a performance overhead of less than 5%. Our approach outperforms alternative CC mechanisms for the following reasons:

(a) Other CC mechanisms treat applications as a blackbox and are unable to be as responsive as our approach.
(b) Delay sensitivity is determined offline per application and used as prior knowledge to augment the mitigation mechanism.
(c) Delay sensitivity is calculated using a multi-dimensional dataset consisting of network performance counters, network topology, application node placement, and application

runtime; thereby increasing estimation accuracy. The multi-dimensional dataset captures the complex interplay between computation and communication.

**Contributions.** In this paper, we present Netscope, a delay sensitivity-based congestion mitigation framework. Our major contributions are:

(a) Design and implementation of Netscope that consists of: (i) delay sensitivity estimation from large-scale network monitoring datasets using inter-linked probabilistic regression models, and (ii) congestion mitigation using an AIMD controller whose parameters are set using delay sensitivity estimates. A low-overhead wrapper for MPI communication primitives enables traffic rate control and network counter sampling to support congestion mitigation.
(b) Demonstration of Netscope on Voltrino (a testbed system at Sandia National Laboratories) and Cori (a large-scale production system at NERSC) with commonly used applications such as MILC [9], AMG [13], and LAMMPS [14].

**Results.** Evaluation of Netscope shows that:

(a) *The congestion mitigation mechanism significantly reduces runtime variability and improves system utilization.* In our evaluation, the tail of runtime increase with Netscope is up to 16.3× and 11.8× lower than Cray static rate control and DCQCN rate control [12] respectively. Additionally, we show that Netscope is tolerant to delay sensitivity estimation errors. In worst-case error estimation, Netscope's performance is no worse than that of DCQCN.
(b) *Delay sensitivity can accurately estimate the impact of congestion on application runtime.* Correlation between application runtime estimated using delay sensitivity and actual runtime is between 0.7 and 0.9 for common scientific applications under synthetically generated congestion and natural system congestion found in production settings.
(c) *Delay sensitivity estimation has a low training cost.* Delay sensitivity can be estimated for a new application within thirty runs in production environments and five runs with synthetically generated congestion.

## 2 MOTIVATION

Applications running in HPC systems [10] have diverse communication characteristics due to complex interplay of computation and communication. Consequently, effects of network congestion on runtime are variable for every application. Such variability can be captured by the *delay sensitivity* metric, defined as the gradient between application runtime and network congestion (measured in terms of message delivery time). The design of Netscope is inspired by two insights associated with (1) estimating delay sensitivity, and (2) integrating delay sensitivity with the CC mechanism as described below:

**Insight 1: Network performance counters[1] and application runtime can be used to estimate application delay sensitivity.** To understand whether network performance counters (collected from network switches [15]) are useful for estimating application delay sensitivity, we characterize the relation between counters and the application runtime variability. Fig. 1 shows the runtime for MILC and GPCNeT under low (0.25% percent time stalled[2]) and high (7.5% percent time stalled) congestion, as measured from stall performance counters. The figure shows that the response to congestion in the system (that is captured through network counters) varies for each application. For instance, MILC has a 60% increase in runtime, while GPCNeT's runtime marginally increases by 2.6% for the same difference in measured congestion Any application whose runtime degrades more with respect to the same congestion difference has a higher delay sensitivity (as the gradient is higher). Thus, using both network performance counters and runtime we conclude that MILC has a higher delay sensitivity than GPCNeT. In §4.1 we will expand on this insight to provide an exact quantification of delay sensitivity.

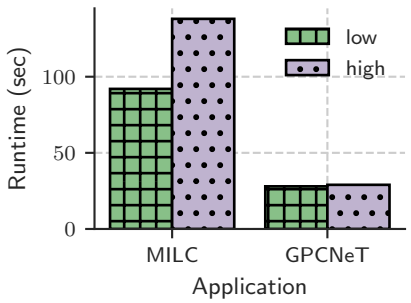



**Figure 1: Variable impact of congestion on MILC and GPCNeT runtime as captured by low and high network stall counters**

**Insight 2: Delay sensitivity can be used to augment congestion mitigation to reduce application runtime variability.** Congestion signals used in CC mechanisms such as ECN [16], QCN [17], and network delay [5, 6] do not address the source of congestion directly. Consequently, during the operation of the CC mechanism, the non-congestion-contributing applications are penalized along with the culprit that is responsible for the congestion. Moreover, if the application that was not responsible for congestion is also delay sensitive, it's performance degradation due to congestion mitigation can be severe. To test whether augmenting the CC mechanism can alleviate this problem, we run three different algorithms (Cray static rate control [18], DCQCN [12], and DCQCN augmented with application delay sensitivities) on a Cray Aries [18] HPC system (16 switches). Our test workload consists of a highly delay sensitive application (MILC, isolated runtime 26 sec) and a non-delay sensitive application (GPCNeT, isolated runtime 75 sec) that run simultaneously. For the test setup, we penalize GPCNeT five times more than MILC, as the goal is to understand whether augmentation with delay sensitivity aids the CC mechanism. Overall (see Table 1), we show that prior knowledge of delay sensitivity may be useful for reducing runtime variability and therefore could be augmented in the CC mechanism (described in §5).

[1]Network performance counters such as link stall times and traffic can be collected at switches via vendor-provided mechanisms.

| | Cray | DCQCN | Augmented-DCQCN |
|---|---|---|---|
| MILC (Isolated: 26 sec) | 56 sec (115%) | 41 sec (58%) | 28 sec (7%) |
| GPCNeT (Isolated: 75 sec) | 75 sec (0%) | 77 sec (2.7%) | 77 sec (2.7%) |

**Table 1: Measured percentage increase in application runtime with respect to isolated runtime for three CC mechanisms.**

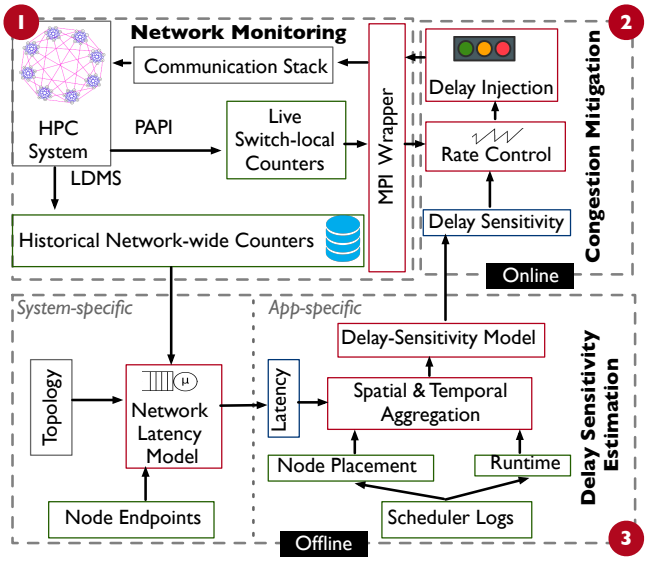



**Figure 2: Overview of Netscope**

## 3 NETSCOPE OVERVIEW

Fig. 2 depicts the architecture of Netscope, our proposed delay sensitivity-driven congestion mitigation framework for high-performance computing systems. The framework consists of two major components: delay sensitivity estimation and congestion mitigation driven by the inferred delay sensitivity, discussed below.

**Network Monitoring (1).** HPC systems expose network telemetry data of each switch as network performance counters. Netscope uses network telemetry data to drive (i) delay sensitivity estimation; and (ii) congestion mitigation. For delay sensitivity estimation, the counters are collected and aggregated across the network by the Lightweight Distributed Metric Service (LDMS) [15] to provide a coherent global network snapshot. For the congestion mitigation mechanism, each node periodically samples local switch counters by using the Performance API (PAPI) [19].[3]

Our test HPC systems use the Cray Aries network technology [18] with a low-diameter Dragonfly topology [20]. The Dragonfly topology partitions the compute nodes and switches into fully connected "electrical groups," [18] in which each group operates as a high-radix virtual switch. To mitigate congestion in the Dragonfly network, each node performs static rate control and uses adaptive routing to distribute traffic across network paths [20].

**Delay Sensitivity Estimation (3).** Delay sensitivity estimation is driven by three inter-linked probabilistic regression models: (i) the *network latency model* (described in §4.2), which estimates network latency (i.e., the time required to send a quanta [4] of data[5]) under varying levels of congestion by using network performance counters (indicative of queuing delays), network topology, and compute-node endpoints; (ii) *spatial & temporal aggregation* (described in §4.1), which estimates expected message delivery time by aggregating inferred network latency spatially (i.e., across multiple communication paths) and temporally (i.e., across the application's lifespan); and (iii) the *delay sensitivity model* (described in §4.1), which estimates application delay sensitivity using inferred message delivery times and corresponding application runtimes. The network latency model is system-specific, i.e., it needs to be retrained for each system, whereas the delay sensitivity model and spatial & temporal aggregation are application-specific, i.e., they need to be retrained for every application configuration.

To train the latency model, we use round-trip time measurements obtained via a specialized `pingpong` application under diverse congestion conditions and representative compute node endpoints. To train the application model, we use application runtimes collected from application runs under synthetically generated or natural system congestion. After model training, delay sensitivity is determined for each application and can be provided as prior knowledge to augment the congestion mitigation mechanism.

**Congestion Mitigation (2).** The congestion mitigation mechanism uses a dynamic traffic rate limiter based on an additive increase multiplicative decrease (AIMD) [11] controller whose parameters are set using the estimated delay sensitivity. An AIMD controller exponentially throttles application traffic (multiplicative decrease) when congestion is sensed in the network, and linearly increases traffic (additive increase) in the absence of detected congestion. Netscope enhances the state-of-the-art AIMD mechanism[6] by making the multiplicative decrease step in the AIMD controller dependent on the application's delay sensitivity, to minimize the penalty (i.e., traffic throttling) on applications that are more sensitive to congestion.

## 4 DELAY SENSITIVITY ESTIMATION

This section describes the estimation of delay sensitivity, which includes application-specific and system-specific models.

### 4.1 Application-specific Models

Here we present application-specific models that we used to derive the application delay sensitivity, which encapsulates the relationship between network congestion and application runtime as given in definition 4.1. To make our methodology of estimating delay sensitivity extensible to a large class of applications, we developed a model that assumes no knowledge of application communication patterns. Our model relies solely on network topology, application node placement, and network performance counters to infer delay sensitivity.

*Definition 4.1. Delay sensitivity* ($c_a$) is the gradient between application runtime ($T_a$) and expected value of message delivery times

[2]Percent Time Stalled is a proxy for measuring congestion in credit-based flow control networks that we describe in §4.2

[3]We use PAPI because timely mitigation depends on quick access to network performance counters, which is hard to achieve via LDMS, as it aggregates performance counters on the file system.

[4]Note that an application message would comprise of multiple data quanta

[5]In Aries, the smallest quantum of data is a flit whose size is ~6 bytes.

[6]For example, DCQCN uses AIMD congestion control with a fixed additive increase and multiplicative decrease step for all applications.

($M_a$).

$$c_a = \frac{\Delta T_a}{\Delta M_a} \tag{1}$$

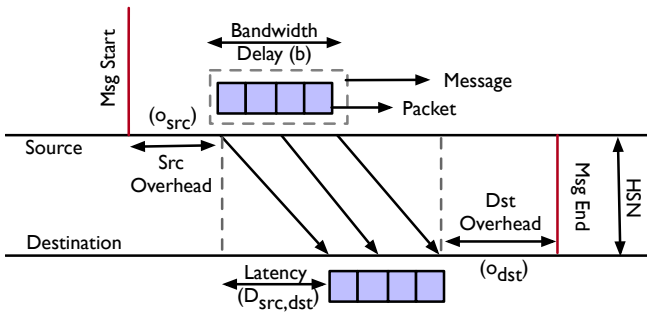



**Figure 3: Breakdown of total message delivery time**

Our key assumption is that delay sensitivity does not vary across the application's lifespan due to their iterative communication patterns (discussed in §7). As delay sensitivity is constant, the gradient of application runtime with expected message delivery time ($M_a$) is constant, and their interdependence can be modeled by a linear regressor given by (2).

$$T_a = c_a M_a + k_a \tag{2}$$

Thus, our quest for estimating delay sensitivity reduces to estimating the slope between expected message delivery times and application runtime. Application runtime can be trivially obtained from scheduler logs. However, estimating expected message delivery times is more involved, and we devote the rest of this section to describing our approach.

**Abstracting Message Delivery Time.** HPC applications typically rely on sending of messages between two endpoints using a communication library such as the Message Passing Interface. The various delays involved in the process of sending/receiving messages are shown in Fig. 3 (inspired by the logP model [21]). A message consists of several packets that are transmitted sequentially across the network from the source to the destination. The total message delivery time for a message ($M_{src,dst}(t)$) that is initiated at the source at time $t$ can be broadly categorized into the following components:

(a) End-host Overheads ($o_{src}, o_{dst}$): Housekeeping tasks need to be performed at the source (before the packet is sent out) and at the destination (after the packet has been received). These tasks include packet header processing, interface-level processing, and OS scheduling of the packet onto the network interface card (NIC). The total time associated with these tasks are represented by $o_{src}$ and $o_{dst}$ for the source and destination, respectively.

(b) Latency ($D$): The time needed for the first quanta of data to reach the destination is the latency ($D$). Note that latency differs from round trip time that includes both forward and reverse path latency.

(c) Bandwidth Delay ($b$): The time required to push all the remaining packets onto the network link is the bandwidth delay. Given the homogeneity of the network, the bandwidth delays at the source and destination are typically equal. Bandwidth delay is given by the ratio of message size to NIC bandwidth ($\frac{\text{msg size}}{\text{bandwidth}}$).

Therefore, the total message delivery time for a message from source $src$ to destination $dst$ initiated at time $t$ is:

$$M_{src,dst}(t) = o_{src} + D_{src,dst} + b + o_{dst} \tag{3}$$

Variations in end-host overheads can be considered noise when the delay introduced by network congestion is the predominant factor in runtime variation. Unlike the end-host overheads, bandwidth delay and latency can vary significantly depending on the application and network congestion conditions, which we can estimate through network counters. Bandwidth delay can be estimated from the application's message size and injection bandwidth at the network interface card. However, network latency is not directly observable in the system. Therefore, we infer the network latency metric through performance counters, such as link stall times[7]. Note that $D$ is the one-way latency, which differs from the round-trip time that can be computed directly from the network performance counters. Also, note that one-way latency cannot be computed simply by halving round-trip time measurements because the return path congestion may be significantly different from the forward path congestion. If we did not consider reverse-path congestion, our model would yield poor estimation of delay sensitivity. We will use the network latency model to compute $D$, as we discuss in §4.2.

Moreover, because of performance and cost constraints, there are challenges associated with collecting network performance counters for each message individually. In particular, on our system, there is a significant difference between the measurement granularity (1 second) and the message delivery times (on the order of microseconds). Hence, we apply (3) for all messages sent in a measurement interval.

The expected message delivery time is dependent on node placement and on variation of congestion over space (spatial characteristics) and time (temporal variation).

**Accounting for Spatial Characteristics.** We estimate expected time to deliver a message between any two application nodes by modeling spatial characteristics (i.e., application-to-node mapping and congestion variation over network links). As we have no prior information regarding application communication patterns, we assume that any two nodes are equally likely to communicate. However, when prior knowledge is available, we can deduce which node pairs are more likely to communicate, and use that to improve our estimations. In practice, we find that without prior knowledge of communication characteristics, we can robustly correlate estimated and actual application runtime, as shown in §6. Under the no-prior-knowledge assumption, the expected message delivery time for any node-to-node communication at time $t$ (denoted by $M_s(t)$) for an application with $N$ nodes is given by:

$$M_s(t) = \frac{\sum_{(src,dst)\in N} M_{src,dst}(t)}{\binom{|N|}{2}} \tag{4}$$

**Accounting for Temporal Variation.** We model temporal variation (i.e. congestion variation over time) by estimating the expected message delivery time across measurement intervals. The

[7] *Stall time* in credit-based flow control networks refers to the time spent waiting to transmit data due to the unavailability of egress buffer space.

expected message delivery time across the application's lifespan ($M_a(t)$) is estimated using the expected spatial message delivery time ($M_s(t)$), as given by (5):

$$M_a = \frac{\sum_{t=0}^{T_a} M_s(t)}{T_a}, \tag{5}$$

where $T_a$ is the total application runtime.

## 4.2 System-specific Model

In this section, we present a model for estimating the network latency ($D$), i.e., the time required to deliver the first quanta of data after the start of message transmission. We first present the high-level system abstraction, followed by a model for estimating the latency for a single link and its extension to a set of paths.

**System Abstraction.** In our system abstraction, we represent each hop between the source and destination nodes as a link. In a high-speed interconnect, examples of such links are connections between the processor, NIC, and network tiles/ports on intermediary switches. The interconnect is lossless and uses a credit-based hop-by-hop flow control mechanism [22]. To transmit data, the sender requires credits from the receiver, which are calculated based on queue occupancy. When there are insufficient credits available to transfer the waiting data, the data transmission stalls, and the data will be buffered in the link buffer until sufficient credits become available. The process of waiting in the queue can cause head-of-line blocking, which can lead to cascading backpressure in adjacent links and formation of congestion trees across the interconnect [3].

The time quanta each link waits to send data is referred to as a *stall*. Network counters record the total time each link is stalled ($T_s$). Using these link stall times, we compute the derived metric for each link, Percent Time Stalled ($P_{Ts}$).

$$P_{T_s} = 100 \times \frac{T_s}{T_m}, \tag{6}$$

where $T_s$ is the total stall duration during the measurement interval and $T_m$ is the measurement interval.

**Single-Link Latency Estimation.** To estimate the latency for a single link, we break it down into the following components:

(a) *Processing delay* is the time the switch needs to process packet headers and move the packet along the queue before further routing.
(b) *Transmission delay* is the time needed to push all of the packet's bits onto the wire.
(c) *Queuing delay* is the time that the data to be transmitted spends in the link buffers awaiting transmission.
(d) *Propagation delay* is the time needed for data to transit the wire, which depends on the number of hops needed to route the message from 'src' to 'dst' and wire length.

Both the processing and the propagation delays are dependent on the transmission media latency and the packet header processing overhead. When modeling a single link, we assume that these delays are constant. Queuing delays dominate over transmission delays because we consider only a small quanta of data and the time required to transmit the quanta into the link is negligible compared to queuing delays.

For the sake of modeling, we represent whether the link buffers are serviced or freed by a random probabilistic process. At each time unit, the link state can be either stalled or non-stalled. During the non-stalled state, the buffer is emptied and the data are allowed to be transmitted. Thus, the link state at each time unit can be represented by a Bernoulli random variable with parameter $p$ that takes value 1 if the link is in a stalled state and 0 otherwise. To infer the total waiting time, we need to compute the expected number of consecutive time units for which the packet waits in the link buffer. An appropriate discrete distribution to model the above-mentioned constraints is a Geometric distribution.

$$P[T = t] = (1-p)p^{t-1} \tag{7}$$

The expected waiting time is given by Equation (8).

$$E[T] = \sum_{t=1}^{\infty} t(1-p)p^{t-1} = \frac{1}{1-p} \tag{8}$$

To estimate the parameter $p$, we use the maximum likelihood estimation (MLE) [23]. The log-likelihood of measuring $T_s$ units of stall in the measurement interval $T_m$ is given by Equation (9).

$$\begin{aligned} log(\mathcal{L}) &= log(p^{T_s}(1-p)^{T_m - T_s}) \\ &= T_s log(p) + (T_m - T_s) log(1-p) \end{aligned} \tag{9}$$

The optimal parameter value can be obtained by setting the gradient of $log(\mathcal{L})$ to zero. Then the value of $p$ that maximizes the log-likelihood of observing the given stalls is given by Equation (10).

$$p = \frac{T_s}{T_m} = \frac{P_{Ts}}{100} \tag{10}$$

The expected queuing delay for the link ($Q_l$) is given by:

$$Q_l = E[T] = \frac{100}{100 - P_{Ts}} \tag{11}$$

**Extension to Multiple Paths.** To estimate the network latency using link-level queuing delays ($Q_l$), we use a linear regression model (refer to (12). Let the possible paths (i.e., a set of links) between two nodes $n_1$ and $n_2$ be represented by the sets $S^*$. Each set $S_i$ that belongs to $S^*$ is composed of a set of links that make up a unique path between the two nodes. Note that this path comprises all the intermediary links that include connections between processors, NICs, and network tiles on switches.

$$D_{n_1,n_2} = \sum_{S_i \in S^*} k_s \sum_{l \in S_i} k_{i_1} Q_l + k_{i_2}, \tag{12}$$

where $S^*$ is the set of all links on all possible paths as constrained by the routing algorithm [20] between 2 nodes $n_1$ and $n_2$. The constants $k_i$ are estimated using the least sum of squares fit.

# 5 CONGESTION MITIGATION

Netscope's online congestion mitigation consists of the following major components:

(a) The *congestion signal*, which is an estimation of the congestion experienced by packets. It is calculated from network performance counters sampled at the NIC.
(b) The *delay injection module*, which provides a mechanism for controlling congestion at the source by introducing probabilistic delays between consecutive message send requests.

(c) *Rate control*, which uses the delay injection module to dynamically change the traffic injection rate based upon the congestion signal and estimated delay sensitivity.

**Congestion Signal.** We choose to use stalls experienced when injecting traffic from the NIC into the high-speed network (HSN) as the signal for congestion. While network latency (described in §4.2) would be an ideal congestion signal (as it can estimate the impact of congestion on application runtime), such a metric is difficult to calculate during online mitigation as we have access to network performance counters on local switches *only*. However, our network latency model requires a global snapshot of system-wide collected network counters. In lieu of network latency, we use stalls experienced by the application traffic injected from the NIC to the HSN (referred to as *nic2hsn*) as it captures both local congestion (i.e., backpressure applied by the NIC) and outside congestion (i.e., backpressure imposed by the network). We collected the *nic2hsn* counters by using the PAPI [19] interface with the sampling frequency set at one second.

**Delay Injection Module.** As no direct traffic rate-throttling method is available on the compute nodes, we introduce delays between consecutive message requests sent out by every application process. The delay is the time delta between traffic injection at the line rate and traffic injection at the desired reduced rate:

$$\Delta T_{inj} = \frac{R_{line} - R_{req}}{B_{nic}}, \tag{13}$$

where $\Delta T_{inj}$ is the delay injected between messages, $R_{line}$ is the line rate at which messages are sent out, $R_{req}$ is the required reduced rate, and $B_{nic}$ is the NIC bandwidth.

It is possible that $\Delta T_{inj}$ may be on the order of nanoseconds; however, Linux only supports process sleep with microsecond resolution. To overcome this challenge, we use probabilistic delays. For example, if the delay that must be introduced is 0.3$\mu$s, we introduce a 1$\mu$s delay with a probability of 0.3. The probability is simulated using a uniform random number generator from the Linux standard library. We achieve the following using an MPI wrapper as shown in Fig. 2.

**Rate Control.** While there are a plethora of rate-control algorithms based on Additive Increase Multiplicative Decrease (AIMD) rate control [11], we decided to adopt the rate-control algorithm first proposed in DCTCP [4] for general-purpose datacenters and later used in DCQCN [12] for RDMA over converged Ethernet (RoCE) networks. We selected the DCQCN rate control algorithm because it has been widely adopted in production networks because of its inherent simplicity and consequent ease of deployment. The sender maintains an estimate of stalled traffic called $\alpha$, which is updated every measurement window using the following update:

$$\alpha \leftarrow (1 - g)\alpha + g \cdot nic2hsn, \tag{14}$$

where $g \in (0, 1)$ is the weight given to past samples as opposed to past measurements. Consequently, the traffic rate injected by application $a$ at time $t$ (i.e., $R_a$) is updated as follows:

$$R_a \leftarrow \begin{cases} R_a(1 - \alpha/(2 + c_a)), & \text{if } \alpha > 0 \\ R_a + 1, & \text{otherwise,} \end{cases} \tag{15}$$

where $c_a \geq 0$ is the delay sensitivity of the application $a$. Thus, $\alpha$ and $c_a$ together control the sending rate. A higher value of $\alpha$ (high congestion) throttles the traffic rate more aggressively, while a higher value of $c_a$ (more sensitive to network congestion) prevents the traffic rate from being reduced more than necessary. Note that $c_a$ is a term introduced by us in the multiplicative decrease step. We leverage and extend the existing fairness and convergence properties of the AIMD rate controller to ensure that the modified AIMD controller also supports them [24].

# 6 EVALUATION

In this section, we evaluate the effectiveness of delay sensitivity estimation and demonstrate the performance gain obtained using our congestion mitigation. We performed the evaluation on two Cray Aries [18] HPC systems:

(a) *Voltrino* consists of a chassis with 16 switches and 240 network links.
(b) *Cori* is a production-scale supercomputer located at *NERSC* comprising 34 electrical groups and 130,560 network links.

We used these systems to evaluate our models in testbed and production settings. For example, Voltrino is a testbed system that we used to conduct multiple application runs and rigorously validate our models. We used Cori to validate our methodology in production settings.

Our test application set consists of a mix of benchmarks and real codes as described below:

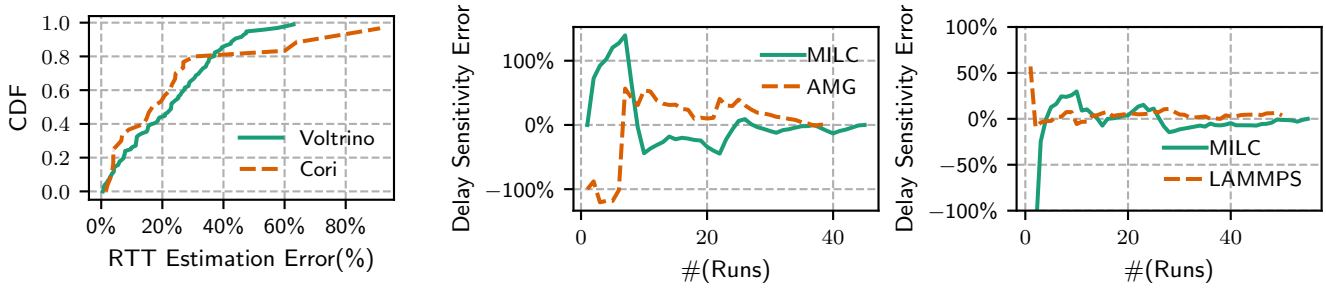
(a) *GPCNeT* is a congestion-creation benchmark suite that captures four congestion patterns (all-to-all, RMA[8] incast, point-to-point incast, and RMA broadcast) commonly found in HPC workloads [25].
(b) *MILC* is used to study quantum chromodynamics, the theory of strong interactions of subatomic physics. It performs four-dimensional lattice communication, and its performance limiters include network latency [9].
(c) *LAMMPS* is a molecular dynamics program focused on materials modeling [14]. Performance limiters are dependent on problem type and scale. They may include compute, memory bandwidth, network bandwidth, and network latency [26].
(d) *AMG* is a parallel algebraic multigrid solver for linear systems [13]. Performance limiters include memory-access and network latency [26].

Application a is an open-source congestion benchmark suite. We chose applications (b–e) because (i) they are among the top 10 most commonly used applications on NERSC production systems, and (ii) previous work has shown that these applications are very susceptible to congestion-induced performance degradation [8].

## 6.1 Evaluating Application Delay Sensitivity

**Training the Network Latency Model.** The network latency model uses round-trip time (RTT) measurements recorded using a `pingpong` application to generate training labels, and network performance counters and node endpoints as inputs. We use RTT measurements because it is challenging to estimate one-way network latency directly in the absence of perfect time synchronization (which is infeasible in practice). We executed the `pingpong` application multiple times (approximately forty) on two randomly

[8] *RMA* stands for *remote memory access.*




**(a) Distribution of percentage error in RTT estimation for Voltrino and Cori**

**(b) Delay sensitivity convergence with increase in production application runs**

**(c) Delay sensitivity convergence with increase in synthetically congested application runs**

**Figure 4: Evaluating convergence and accuracy of delay sensitivity**

| Application | Nodes | System | Congestion Source | Placement Decision |
|---|---|---|---|---|
| MILC-small | 4 | Voltrino | GPCNeT | Random |
| LAMMPS-small | 4 | Voltrino | GPCNeT | Random |
| AMG-small | 4 | Voltrino | GPCNeT | Random |
| MILC | 384 | Cori | System Congestion | Slurm Scheduler |
| LAMMPS | 108 | Cori | GPCNeT | Slurm Scheduler |
| AMG | 768 | Cori | System Congestion | Slurm Scheduler |

**Table 2: List of applications (and their configurations) used for delay sensitivity estimation**

| Application | Mean Runtime (sec) | Runtime Fit Correlation | Delay Sensitivity |
|---|---|---|---|
| MILC-small | 220 | 0.71 | 6.67 |
| LAMMPS-small | 23 | 0.81 | 0.47 |
| AMG-small | 45 | N/A | 0 |
| MILC | 848 | 0.73 | 45 |
| LAMMPS | 43 | 0.91 | 26 |
| AMG | 459 | 0.70 | 12.5 |

**Table 3: Delay sensitivity estimated for different applications**

chosen nodes in the presence of congestion created using GPCNeT. During the pingpong application training runs, the percent time stalled measurements (obtained using network counters) across all links in the system ranged between 0% and 99%. Each pingpong run involved 10,000 consecutive message exchanges (without any compute), and the average RTT for each message exchange was recorded. A large number of messages were sent to compensate for the difference in network latency (2–100 $\mu$s) and granularity of measurements (1 s). To train the model, we sum the forward and reverse path latencies that are calculated using Equation (12) to estimate the round-trip time. Thus, we can calculate the constants ($k_i$) using the least sum of squares fit with training data (i.e. RTT measurements obtained from the pingpong application).

**Accuracy of Latency Model.** We evaluate the accuracy of the latency model, described in §4.2, which is used to estimate network latency by using performance counters. The model was trained and validated on *Voltrino* and *Cori*. Fig. 4a shows the percentage error in estimating RTT for the two systems. For both of them, 75% of the RTT measurements were estimated within an error bound of 30%. Moreover, we observed statistically high Pearson correlation coefficients [27] of 0.71 and 0.67 between the measured and the estimated round-trip time for Voltrino and Cori, respectively. We also show in §6.2 how errors in delay sensitivity estimation (that may propagate from errors in latency model) affect congestion mitigation. *The latency model remains accurate under large variation in congestion and system scale.*

**Accuracy of Estimating Delay Sensitivity.** We have designed a comprehensive test suite for validating delay sensitivity ($c_a$) estimation (from Equation (2)) that uses the configurations described in Table 2. For Voltrino, a smaller single-user testbed system, we use synthetically generated congestion to create runtime variation. However, on larger, multi-user systems like Cori, we find sufficient natural runtime variation due to network contention between different applications.

To test whether delay sensitivity estimation is sensitive to application node placement, we randomly varied the application-to-node mapping for the above-mentioned applications on the smaller-scale systems. For Cori, the node placement was determined by the Slurm scheduling policy [28]. Applications on Cori were run over a year to ensure sufficient diversity in application-to-node mapping and system congestion.

Table 3 provides the delay sensitivity values for our test suite. We found that expected runtime and actual runtime have a strong correlation between 0.7 and 0.9 for all applications in the test suite except for AMG-small. Correlation cannot be calculated for AMG-small as it has a negligible variance in runtime even under the presence of congestion.

The estimated delay sensitivity could vary between zero (implying negligible increase in application runtime due to congestion; e.g., AMG-small) to a high value (implying a significant increase in application runtime due to congestion; e.g., MILC). The delay sensitivity is calculated with respect to the pingpong application, i.e., the pingpong application has a delay sensitivity equal to one. The proposed delay sensitivity metric is robust as it captures the relationship between application size and congestion. In our experiments, we observed that increasing the application size increased the delay sensitivity, thereby confirming our intuition that applications with larger sizes have larger number of communication paths,

and thus link delays are more likely to stall the entire iteration. For example, delay sensitivity increases by a factor of 6.7× (from 6.67 to 45) for MILC when the number of nodes increases from 4 to 384. A similar trend of increase in delay sensitivity with increasing application size was observed for LAMMPS and AMG. In addition, as expected, we observed that applications that are only latency-bound have higher delay sensitivity than applications that have multiple performance limiters, such as compute and memory. For example, MILC, which is solely latency-bound, has a higher delay sensitivity than LAMMPS, which is also limited by memory and compute. *We find that the delay sensitivity estimation is accurate for a wide range of applications, scales, congestion, and placements.*

**Training Requirements for Delay Sensitivity Estimation.** We quantify the cost of training our models to accurately estimate the delay sensitivity ($\hat{c}_a$)in terms of the number of application runs. Fig. 4b shows the error in estimating the delay sensitivity with an increasing number of production training runs for MILC and AMG. The error in delay sensitivity at the $i^{\text{th}}$ run is given by $\frac{c_a^i - \hat{c}_a}{\hat{c}_a} \times 100$. For both applications, we observe that the delay sensitivity fluctuates significantly at the beginning of the training, but stabilizes to within 10% of the final estimated delay sensitivity ($\hat{c}_a$) after ~25 runs. However, we can further reduce the training time significantly by exposing the application to both extremes of congestion (i.e., no congestion and high congestion), thus providing early benefits to the user. As we show in Fig. 4c, the training time is reduced by 5× (from 25 runs to 5 runs) in the case of MILC-small running under synthetic congestion. Similarly, for LAMMPS delay sensitivity converges quickly under 5 runs as well. *Our model converges fast, especially under synthetic congestion.*

## 6.2 Evaluating Congestion Mitigation

We compared Netscope's delay sensitivity-based congestion mitigation mechanism (described in §5) to DCQCN rate control (deployed in RDMA networks) and the static rate control mechanism in Cray. In these experiments, we ran three applications—MILC, LAMMPS, and AMG—under a range of congestion conditions induced via GPCNeT. These applications were selected as they cover an entire spectrum of delay sensitivity, from high (MILC: 6.67), to medium (LAMMPS: 0.47), to low (AMG: 0). These applications are described at the beginning of §6. Varying levels of congestion were generated by randomizing node allocations (i.e., application to node mapping) and GPCNeT parameter tuning (refer to Section 4.5 in [25]). The applications were run on Voltrino. We *only* run two applications at a time because mitigation is performed individually for each application, and is dependent only on network congestion (created by the workload) and not on the number of applications. Overall, we executed 200 runs to compare each application with every congestion mitigation mechanism separately. We also evaluate these applications with larger node counts on Cori.

**Reduced Runtime Variability.** To evaluate the efficacy of the CC mechanisms, we calculated the *runtime increase*, defined as the ratio of runtime to isolated runtime ($\frac{\text{runtime}}{\text{isolated runtime}}$). Isolated runtime is obtained in the absence of external congestion, i.e., when no other applications are executing on the system. Fig. 5 shows the barplot of the tail (i.e., the 99th percentile) of the runtime increase for the three applications (MILC, LAMMPS, and AMG) across three CC mechanisms. We found that Netscope outperforms both Cray and DCQCN in reducing the tail of the runtime increase for each application. For example, the tail of the runtime increase for MILC with Netscope is 1.11× , which is 11% higher than that of isolated runtime. In contrast, the tail of the runtime increase with DCQCN and Cray static control was found to be 2.3× (130% increase) and 2.8× (180% increase). Thus, compared to Cray rate control and DCQCN, Netscope provides a 16.3× (180% to 11%) and 11.8× (130% to 11%) reduction in the tail of the runtime increase. Netscope is significantly better than DCQCN and Cray in the following respects. (i) Compared to Cray static rate control, Netscope prevents extreme congestion cases because it dynamically adjusts traffic rates (throttles) based on measured congestion. (ii) Compared to DCQCN, Netscope penalizes the delay sensitive application less aggressively than GPCNeT as the delay sensitive application has a smaller multiplicative decrease factor (described in §5). We also found that benefits of Netscope improve with increase in application delay sensitivity. For example, Netscope reduces the tail of the runtime increase in MILC (delay sensitivity: 6.67) by 16.3× (180% to 11%) compared to 5× (25% to 5%) in LAMMPS (delay sensitivity: 0.47). Netscope provides greater benefit for MILC than for LAMMPS and AMG, as effects of congestion and incorrect penalization are exacerbated in applications with higher delay sensitivity.

| | Latency ($\mu$s) | | Traffic (Flits/sec) | |
|---|---|---|---|---|
| | Median | 90%ile | Median | 90%ile |
| Cray | 10.1 | 52 | 2.9e7 | 5.4e7 |
| DCQCN | 3.4 | 11.9 | 2.1e7 | 5.1e7 |
| Netscope | 9.7 | 12.2 | 4.2e7 | 5.2e7 |

**Table 4: Comparing traffic and latency for Cray, Netscope, and DCQCN.**

**Increased System Utilization.** To compare the performance of CC mechanisms, decreased runtime variability is not sufficient as it may come at the cost of system utilization. For example, reducing the MILC runtime increase may come at the cost of increasing runtime for GPCNeT, thereby lowering system utilization. Hence, we also measured the overall system utilization of the CC mechanisms in terms of the node seconds required to complete the workload. Our comparison metric is the increase in nodeseconds, defined as the ratio of the nodeseconds required to complete the workload, and the sum of the nodeseconds required to complete all of the applications in the workload in isolation. Fig. 6 shows the tail (i.e., the 99th percentile) of the increase in node seconds required to run the workloads using the three congestion mitigation mechanisms. For example, with MILC, we found that the increases in nodeseconds for Netscope, Cray, and DCQCN were 1.02× , 1.32× , and 1.24× . *Overall, across all applications, we found that the reduction in runtime increase offered by Netscope does not adversely reduce system utilization in the way that isolation-based QoS mechanisms [29] do.*

**Impact on Traffic and Congestion.** Table 4 shows the network latency (i.e., the time required to deliver a flit) and traffic for the three rate-control algorithms across all experiments. We found that DCQCN has a lower network latency than either Cray or Netscope (by 2.9× ) at the 50th percentile. Thus, DCQCN is significantly better at reducing network congestion than either Netscope or Cray.

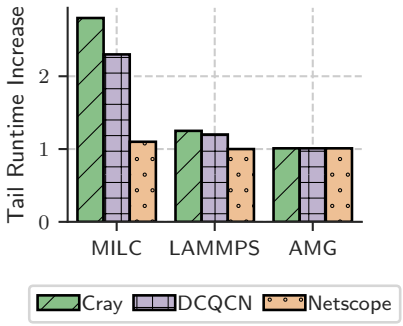



**Figure 5: Comparing runtime increase across Cray, DCQCN, and Netscope**

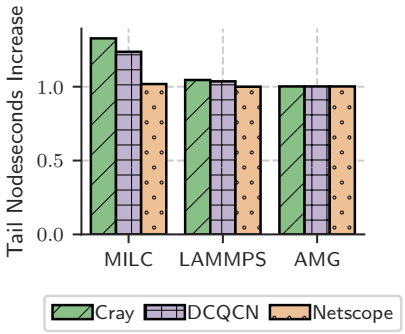



**Figure 6: Comparing system utilization across Cray, DCQCN, and Netscope**

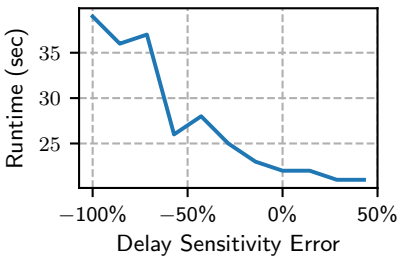



**Figure 7: Impact of error in estimating delay sensitivity on MILC runtime**

However, solely attempting to lower congestion (i.e., minimizing queuing at link buffers) may adversely affect delay sensitive applications, leading to worse system utilization as discussed above. Unsurprisingly, static rate control, which does not adapt to varying levels of congestion, performs worse at the 90th percentile, as seen for the Cray rate-control mechanism. *Overall, we found that Netscope outperforms both DCQCN and the Cray static rate control mechanism at reducing application runtime variation and improving system throughput.*

**Impact of Training Errors on Congestion Mitigation.** Here, we evaluate the impact on congestion mitigation efficacy of inaccurately estimating delay sensitivity. We define the error in estimating delay sensitivity as $\frac{c_a^{err}-\hat{c}_a}{\hat{c}_a} \times 100$, where $c_a^{err}$ is the erroneous delay sensitivity estimate and $\hat{c}_a$ is the correct delay sensitivity. Such errors could be introduced because of insufficient training or errors in measurement data. A negative error in estimated delay sensitivity, i.e., one that causes the estimated value to be lower than the true value, may increase application runtime. The reason is that a delay sensitive application would have a more aggressive throttling during the rate control step, leading to severe penalization. At the same time, a positive error in estimating delay sensitivity is unfavorable, as it leads to less aggressive throttling and slower congestion mitigation. Fig. 7 shows the runtime for MILC (when MILC and GPCNeT are running) with the error introduced in the delay sensitivity estimation for MILC. We found that the runtime increase is close to isolated runtime when the error in estimating delay sensitivity is close to 0%. As the negative error increases, the MILC runtime increases. In particular, at an error of -100% in estimating delay sensitivity, Netscope's performance is no worse than that of DCQCN. *If there is a negative error in estimated delay sensitivity, Netscope's performance worsens (but does not become worse than DCQCN's). However, at the same time, we also want to minimize positive estimation error, to ensure timely congestion mitigation.*

**Testing Netscope in Production Systems.** We tested Netscope on Cori with scaled-up versions of applications, i.e., MILC (108 nodes, isolated runtime 149 s), AMG (108 nodes, isolated runtime 21 s), LAMMPS (108 nodes, isolated runtime 14 s), and GPCNeT (108 nodes, isolated runtime 70 s). We collected a total of 30 runs across all applications on Cori. GPCNeT was used as the primary source for inducing congestion, and placement was decided by the Slurm job scheduler.

| Application | Cray | DCQCN | Netscope |
|---|---|---|---|
| MILC | 1.3 | 1.2 | 1.02 |
| AMG | 1.3 | 1.1 | 1.03 |
| LAMMPS | 1.7 | 1.3 | 1.10 |

**Table 5: Comparing runtime increase of CC mechanisms on Cori**

Table 5 summarizes the average runtime increase for each of the three applications for the CC mechanisms. Recall that we defined *runtime increase* as the ratio of runtime to isolated runtime ($\frac{\text{runtime}}{\text{isolated runtime}}$). The maximum runtime increase observed with Netscope is 1.1× (i.e., less than 10% deviation from isolated runtime). Compared to Cray and DCQCN, Netscope can offer between 7× and 15× improvement in reduction in average runtime increase. For example, in the case of LAMMPS, with Netscope the runtime increase is 1.1× (a 10% increase) whereas the Cray runtime increase is 1.7× (a 70% increase). Thus, Netscope offers a 7× (70% to 10%) improvement. Moreover, we expect that Netscope's performance benefits will increase with large-scale applications because their theoretical peak congestion is much higher than the congestion observed in our experiments. For example, the authors of [25] found that the peak congestion latency that can be induced in a Cray Aries supercomputer can be as high as 6000$\mu$s, whereas we observed 100$\mu$s in our experiments. *Despite the constraints introduced in production, we find that the performance benefits of Netscope continue to hold with larger application sizes.*

**Overhead of Netscope.** Although Netscope increases the congestion-free runtime for the tested applications by up to 2%, the benefits of Netscope are realized in the presence of congestion. Without Netscope, congestion could cause runtime variation as high as 200% (see Fig. 5). Netscope's overhead is negligible in terms of memory (~0.8KB) and computation.

## 7 DISCUSSION

**Can Netscope apply to other HPC network technologies?** Application performance variation has been observed across other

HPC network technologies, such as Infiniband [25], Cray Gemini [7], and Cray Slingshot [30]. Netscope can readily adapt to such network technologies because architectural features such as availability of network counters, adaptive routing, credit stall flow control, and use of MPI communication primitives are universal in most HPC systems and applications. Therefore, the system and application model (described in §4.2) and control mechanism (described in §5) remain the same.

We expect Netscope to provide additional performance improvements over the mechanisms of Slingshot and Infiniband networks, because the rate controllers in these networks do not characterize the delay sensitivity of the applications. Thus, the highly delay sensitive application may be penalized more than necessary, leading to worse system utilization.

**Does Netscope need to be retrained with variation in application scale?** The delay sensitivity metric is dependent on the application node count. Therefore, Netscope needs to be retrained for a variant of the application that uses a different node count. However, there exist numerous HPC applications that are executed with fixed node count. For example, our analysis of jobs on Cori run over two weeks reveal that MILC was run with only 5 distinct node counts. The impact of errors in estimated delay sensitivity that can be introduced by different application node counts has been discussed in §6. In our future work, we will develop models to characterize the relationship between delay sensitivity and node count.

**How does Netscope handle multiple congestors with high delay sensitivity?** If multiple applications act as sources of congestion while also being highly delay-sensitive, the control mechanism of Netscope will converge more slowly, leading to a lag in congestion mitigation. However, as the control mechanism has a multiplicative decrease step, congestion is mitigated exponentially, and the time required to mitigate congestion only grows logarithmically. Moreover, we have been unable to find congestion-inducing workloads for which multiple applications have high delay sensitivity.

**Can simulation be used to reduce the cost of estimating delay sensitivity?** Indeed, simulation of applications on a network simulator (such as [31, 32]) can be used for delay sensitivity estimation to further reduce the number of application runs required on a real system. However, simulations do not accurately mimic system behavior [33, 34]. Delay sensitivity obtained from simulation could be used as a prior to bias the estimation of delay sensitivity on the real system and further accelerate training.

**Is constant delay sensitivity a reasonable assumption?** Recall from §4.1, delay sensitivity is assumed to be constant throughout the application lifespan. To validate this assumption, we characterize traffic injection of two applications with non-zero delay sensitivity: MILC and LAMMPS. Fig. 8 shows the variation in cumulative traffic injection for the two applications. We observed iterative communication behavior across the run for both applications. An iterative communication behaviour ensures that network characteristics of applications do not significantly vary across time and delay sensitivity is constant. Moreover, such iterative behaviour has also been demonstrated for multiple other commonly used HPC applications [35–37]. As a part of future work, we could model delay sensitivity as a distribution dependent on application progress in an execution.

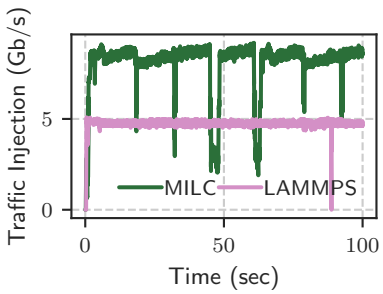



**Figure 8: Variation in cumulative per node traffic injection across the application's lifespan for MILC and LAMMPS**

**How does Netscope work across varying input sets?** Netscope is well suited for applications such as weather forecasting (WRF) and ML training/inference that have mostly identical communication patterns despite variation in input datasets. Fig. 7 shows how an error in estimating delay sensitivity that may be introduced because of changing input sets can impact the performance of Netscope. Even in the worst case, Netscope performs no worse than other approaches (i.e., DCQCN and Cray Aries).

## 8 RELATED WORK

**Modeling the impact of congestion on applications.** Several researchers have studied the effects of network congestion on the performance of applications. [8, 38] has proposed blackbox machine learning models to estimate application runtime from network counters. [7] has proposed a congestion region-based approach to characterize congestion in torus networks. [39] proposed inferring latency requirements of applications using delay injection. However, these approaches do not provide application characteristics that can be consumed by the CC algorithm.

**Rate Control for Congestion Mitigation.** Traditionally, Infiniband congestion mitigation [40] has been the defacto standard for CC in credit-based flow control networks. In addition, researchers have proposed additional schemes [1–3] to mitigate congestion for HPC systems. Moreover, there are a large number of application-oblivious (e.g., [4, 5, 12]) and application-aware (e.g., [41–43]) CC mechanisms proposed for cloud datacenters. However, the application-oblivious CC mechanisms end up incorrectly penalizing applications whereas the application-aware CC mechanisms require complicated parameter tuning that is difficult to achieve without an automated framework. Cray Slingshot [44] proposes to alleviate congestion by selectively targeting the true sources of congestion using network features only. The use of programmable switches has also been proposed to provide explicit traffic feedback for congestion control [45, 46].

**In-network Mechanisms.** [47] proposed a mechanism to support routing dependent on a per-application basis in Cray Aries networks. However, the primary goal of the work was to reduce self-congestion created by the application as opposed to interference between applications. Our proposed delay sensitivity metric

can also be used to assign priority classes to provide quality-of-service. For example, we can use the delay sensitivity metric to assign priority classes in Cray Slingshot [44]. In addition, there has been a large body of work on using different packet scheduling policies [48–50] at the switches to minimize the impact of congestion on applications that do not contribute to congestion. However, these policies have yet to be deployed on real switch hardware.

## 9 CONCLUSION AND FUTURE WORK

We propose Netscope, a delay sensitivity-based framework for congestion mitigation. We comprehensively evaluate Netscope on two Cray Aries systems (including a production supercomputer) and commonly used scientific applications. Netscope is shown to reduce the application tail runtime variability on a Cray Aries testbed system by 16.3× while increasing median system utilization by 12%.

There are several open questions and research challenges regarding Netscope (discussed in §7). As a part of future work, we will extend Netscope to use delay sensitivity for driving in-network mechanisms such as QoS classes and programmable packet scheduling.

## 10 ACKNOWLEDGEMENTS

We thank the reviewers for their valuable comments that improved the paper. We appreciate J. Applequist, K. Saboo, and K. Chung for their insightful comments on the early drafts of this manuscript. This research was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under award No. 2015-02674; by the National Science Foundation (NSF) under grant No. 2029049; by Sandia National Laboratories[9] under contract No. 1951381; and by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR), a research collaboration that is part of the IBM AI Horizon Network. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, SNL, HPE, IBM, DOE or the United States Government. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. Saurabh Jha is supported by a 2020 IBM PhD fellowship. We would also like to thank Larry Kaplan and Eric Roman for fruitful discussions and suggestions.

## REFERENCES

[1] N. Jiang, L. Dennison, and W. J. Dally, "Network endpoint congestion control for fine-grained communication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807600

[2] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu, "Congestion avoidance on manycore high performance computing systems," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 121–132. [Online]. Available: https://doi.org/10.1145/2304576.2304594

[3] J. Escudero-Sahuquillo, E. G. Gran, P. J. Garcia, J. Flich, T. Skeie, O. Lysne, F. J. Quiles, and J. Duato, "Efficient and cost-effective hybrid congestion control for hpc interconnection networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 107–119, 2015.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, p. 63–74, Aug. 2010. [Online]. Available: https://doi.org/10.1145/1851275.1851192

[5] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 537–550. [Online]. Available: https://doi.org/10.1145/2785956.2787510

[6] G. Kumar, N. Dukkipati, K. Jang, H. M. G. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat, "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 514–528. [Online]. Available: https://doi.org/10.1145/3387514.3406591

[7] S. Jha, A. Patke, J. Brandt, A. Gentile, B. Lim, M. Showerman, G. Bauer, L. Kaplan, Z. Kalbarczyk, W. Kramer, and R. Iyer, "Measuring congestion in high-performance datacenter interconnects," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 37–57. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/jha

[8] A. Bhatele, J. J. Thiagarajan, T. Groves, R. Anirudh, S. A. Smith, B. Cook, and D. K. Lowenthal, "The case of performance variability on dragonfly-based systems," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 896–905.

[9] "The MIMD Lattice Computation (MILC) Collaboration," http://www.physics.utah.edu/~detar/milc/.

[10] "NERSC Workload Analysis," https://portal.nersc.gov/project/mpccc/baustin/NERSC_2014_Workload_Analysis_v1.1.pdf.

[11] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN Systems*, vol. 17, no. 1, pp. 1–14, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0169755289900196

[12] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 523–536. [Online]. Available: https://doi.org/10.1145/2785956.2787484

[13] "AMG," https://github.com/LLNL/AMG.

[14] "LAMMPS Molecular Dynamics Simulator," https://lammps.sandia.gov/.

[15] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 154–165.

[16] K. Ramakrishnan, S. Floyd, and D. Black, "Rfc3168: The addition of explicit congestion notification (ecn) to ip," 2001.

[17] R. Pan, B. Prabhakar, and A. Laxmikantha, "Qcn: Quantized congestion notification," *IEEE802*, vol. 1, pp. 52–83, 2007.

[18] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray xc series network," *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[19] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710. Citeseer, 1999.

[20] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *2008 International Symposium on Computer Architecture*, 2008, pp. 77–88.

[21] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: Towards a realistic model of parallel computation," *SIGPLAN Not.*, vol. 28, no. 7, p. 1–12, Jul. 1993. [Online]. Available: https://doi.org/10.1145/173284.155333

[22] N. Kung and R. Morris, "Credit-based flow control for atm networks," *IEEE Network*, vol. 9, no. 2, pp. 40–48, 1995.

[23] I. J. Myung, "Tutorial on maximum likelihood estimation," *Journal of Mathematical Psychology*, vol. 47, no. 1, pp. 90–100, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022249602000287

[24] Y. Yang and S. Lam, "General aimd congestion control," in *Proceedings 2000 International Conference on Network Protocols*, 2000, pp. 187–198.

[25] S. Chunduri, T. Groves, P. Mendygral, B. Austin, J. Balma, K. Kandalla, K. Kumaran, G. Lockwood, S. Parker, S. Warren, N. Wichmann, and N. Wright, "Gpcnet: Designing a benchmark suite for inducing and measuring contention

[9] Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

in hpc networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356215

[26] "CORAL-2 Benchmarks," https://asc.llnl.gov/coral-2-benchmarks.

[27] J. Benesty, J. Chen, Y. Huang, and I. Cohen, *Pearson Correlation Coefficient*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–4. [Online]. Available: https://doi.org/10.1007/978-3-642-00296-0_5

[28] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[29] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The ibm blue gene/q interconnection network and message unit," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–10.

[30] D. D. Sensi, S. D. Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An in-depth analysis of the slingshot interconnect," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020, pp. 1–14. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SC41405.2020.00039

[31] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale, "Evaluating hpc networks via simulation of parallel workloads," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 154–165.

[32] G. Zheng, G. Kakulapati, and L. Kale, "Bigsim: a parallel simulator for performance prediction of extremely large parallel machines," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004, pp. 78–.

[33] S. Rampfl, "Network simulation and its limitations," in *Proceeding zum seminar future internet (FI), Innovative Internet Technologien und Mobilkommunikation (IITM) und autonomous communication networks (ACN)*, vol. 57. Citeseer, 2013.

[34] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USA: USENIX Association, 2010, p. 365–378.

[35] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the performance of an algebraic multigrid cycle on hpc platforms," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 172–181. [Online]. Available: https://doi.org/10.1145/1995896.1995924

[36] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. D. Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari, and R. M. Wentzcovitch, "QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials," *Journal of Physics: Condensed Matter*, vol. 21, no. 39, p. 395502, sep 2009. [Online]. Available: https://doi.org/10.1088/0953-8984/21/39/395502

[37] S. Maintz and M. Wetzstein, "Strategies to accelerate vasp with gpus using open acc," in *Proceedings of the Cray User Group*, 2018.

[38] A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, and L. V. Kale, "Identifying the culprits behind network congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 113–122.

[39] J. C. Mogul and R. R. Kompella, "Inferring the network latency requirements of cloud tenants," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mogul

[40] E. G. Gran, M. Eimot, S.-A. Reinemo, T. Skeie, O. Lysne, L. P. Huse, and G. Shainer, "First experiences with congestion control in infiniband hardware," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[41] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, p. 115–126, Aug. 2012. [Online]. Available: https://doi.org/10.1145/2377677.2377709

[42] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, p. 127–138, Aug. 2012. [Online]. Available: https://doi.org/10.1145/2377677.2377710

[43] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can JUMP them!" in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/grosvenor

[44] D. D. Sensi, S. D. Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An in-depth analysis of the slingshot interconnect," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020, pp. 1–14. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SC41405.2020.00039

[45] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "Hpcc: High precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 44–58. [Online]. Available: https://doi.org/10.1145/3341302.3342085

[46] N. Dukkipati, "Rate control protocol (rcp): Congestion control to make flows complete quickly," Ph.D. dissertation, Stanford, CA, USA, 2008, aAI3292347.

[47] D. De Sensi, S. Di Girolamo, and T. Hoefler, "Mitigating network noise on dragonfly networks through application-aware routing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356196

[48] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 44–57. [Online]. Available: https://doi.org/10.1145/2934872.2934899

[49] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Pfabric: Minimal near-optimal datacenter transport," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 435–446. [Online]. Available: https://doi.org/10.1145/2486001.2486031

[50] J. Y.-T. Leung, "A New Algorithm for Scheduling Periodic, Real-time Tasks," *Algorithmica*, vol. 4, no. 1-4, p. 209, 1989.